

The L^AT_EX3 Sources

The L^AT_EX Project*

Released 2024-05-08

Abstract

This is the typset sources for the `expl3` programming environment; see the matching `interface3` PDF for the API reference manual. The `expl3` modules set up a naming scheme for L^AT_EX commands, which allow the L^AT_EX programmer to systematically name functions and variables, and specify the argument types of functions.

The T_EX and ϵ -T_EX primitives are all given a new name according to these conventions. However, in the main direct use of the primitives is not required or encouraged: the `expl3` modules define an independent low-level L^AT_EX3 programming language.

The `expl3` modules are designed to be loaded on top of L^AT_EX 2 ϵ . With an up-to-date L^AT_EX 2 ϵ kernel, this material is loaded as part of the format. The fundamental programming code can also be loaded with other T_EX formats, subject to restrictions on the full range of functionality.

*E-mail: latex-team@latex-project.org

Contents

I	Introduction	1
1	Introduction to <code>expl3</code> and this document	2
1.1	Naming functions and variables	2
1.1.1	Scratch variables	5
1.1.2	Terminological inexactitude	5
1.2	Documentation conventions	5
1.3	Formal language conventions which apply generally	7
1.4	<code>TeX</code> concepts not supported by <code>L^AT_EX3</code>	7
II	Bootstrapping	8
2	The <code>l3bootstrap</code> module: Bootstrap code	9
2.1	Using the <code>L^AT_EX3</code> modules	9
3	The <code>l3names</code> module: Namespace for primitives	11
3.1	Setting up the <code>L^AT_EX3</code> programming language	11
III	Programming Flow	12
4	The <code>l3basics</code> module: Basic definitions	13
4.1	No operation functions	13
4.2	Grouping material	13
4.3	Control sequences and functions	14
4.3.1	Defining functions	14
4.3.2	Defining new functions using parameter text	15
4.3.3	Defining new functions using the signature	17
4.3.4	Copying control sequences	19
4.3.5	Deleting control sequences	20
4.3.6	Showing control sequences	20
4.3.7	Converting to and from control sequences	21
4.4	Analysing control sequences	22
4.5	Using or removing tokens and arguments	23
4.5.1	Selecting tokens from delimited arguments	26
4.6	Predicates and conditionals	27
4.6.1	Tests on control sequences	28
4.6.2	Primitive conditionals	28
4.7	Starting a paragraph	30
4.8	Debugging support	30

5	The <code>l3expan</code> module: Argument expansion	31
5.1	Defining new variants	31
5.2	Methods for defining variants	32
5.3	Introducing the variants	34
5.4	Manipulating the first argument	35
5.5	Manipulating two arguments	37
5.6	Manipulating three arguments	37
5.7	Unbraced expansion	39
5.8	Preventing expansion	40
5.9	Controlled expansion	41
5.10	Internal functions	43
6	The <code>l3sort</code> module: Sorting functions	44
6.1	Controlling sorting	44
7	The <code>l3tl-analysis</code> module: Analysing token lists	46
8	The <code>l3regex</code> module: Regular expressions in \TeX	47
8.1	Syntax of regular expressions	48
8.1.1	Regular expression examples	48
8.1.2	Characters in regular expressions	49
8.1.3	Characters classes	49
8.1.4	Structure: alternatives, groups, repetitions	50
8.1.5	Matching exact tokens	51
8.1.6	Miscellaneous	53
8.2	Syntax of the replacement text	53
8.3	Pre-compiling regular expressions	55
8.4	Matching	56
8.5	Submatch extraction	57
8.6	Replacement	58
8.7	Scratch regular expressions	60
8.8	Bugs, misfeatures, future work, and other possibilities	60
9	The <code>l3prg</code> module: Control structures	63
9.1	Defining a set of conditional functions	63
9.2	The boolean data type	65
9.2.1	Constant and scratch booleans	67
9.3	Boolean expressions	68
9.4	Logical loops	70
9.5	Producing multiple copies	71
9.6	Detecting \TeX 's mode	71
9.7	Primitive conditionals	72
9.8	Nestable recursions and mappings	72
9.8.1	Simple mappings	73
9.9	Internal programming functions	73

10 The <code>l3sys</code> module: System/runtime functions	74
10.1 The name of the job	74
10.2 Date and time	74
10.3 Engine	75
10.4 Output format	76
10.5 Platform	76
10.6 Random numbers	76
10.7 Access to the shell	77
10.8 System queries	78
10.9 Loading configuration data	79
10.9.1 Final settings	79
11 The <code>l3msg</code> module: Messages	80
11.1 Creating new messages	80
11.2 Customizable information for message modules	81
11.3 Contextual information for messages	82
11.4 Issuing messages	83
11.4.1 Messages for showing material	87
11.4.2 Expandable error messages	87
11.5 Redirecting messages	88
12 The <code>l3file</code> module: File and I/O operations	90
12.1 Input–output stream management	90
12.1.1 Reading from files	92
12.1.2 Reading from the terminal	96
12.1.3 Writing to files	96
12.1.4 Wrapping lines in output	98
12.1.5 Constant input–output streams, and variables	99
12.1.6 Primitive conditionals	99
12.2 File operations	99
12.2.1 Basic file operations	99
12.2.2 Information about files and file contents	100
12.2.3 Accessing file contents	103
13 The <code>l3luatex</code> module: Lua\TeX-specific functions	105
13.1 Breaking out to Lua	105
13.2 Lua interfaces	106
14 The <code>l3legacy</code> module: Interfaces to legacy concepts	108
IV Data types	109

15 The <code>l3tl</code> module: Token lists	110
15.1 Creating and initialising token list variables	110
15.2 Adding data to token list variables	111
15.3 Token list conditionals	112
15.3.1 Testing the first token	114
15.4 Working with token lists as a whole	115
15.4.1 Using token lists	115
15.4.2 Counting and reversing token lists	116
15.4.3 Viewing token lists	117
15.5 Manipulating items in token lists	118
15.5.1 Mapping over token lists	118
15.5.2 Head and tail of token lists	119
15.5.3 Items and ranges in token lists	121
15.5.4 Sorting token lists	123
15.6 Manipulating tokens in token lists	123
15.6.1 Replacing tokens	123
15.6.2 Reassigning category codes	124
15.7 Constant token lists	125
15.8 Scratch token lists	126
16 The <code>l3tl-build</code> module: Piecewise <code>tl</code> constructions	127
16.1 Constructing <code><tl var></code> by accumulation	127
17 The <code>l3str</code> module: Strings	129
17.1 Creating and initialising string variables	130
17.2 Adding data to string variables	131
17.3 String conditionals	131
17.4 Mapping over strings	133
17.5 Working with the content of strings	135
17.6 Modifying string variables	138
17.7 String manipulation	139
17.8 Viewing strings	140
17.9 Constant strings	141
17.10 Scratch strings	141
18 The <code>l3str-convert</code> module: String encoding conversions	142
18.1 Encoding and escaping schemes	142
18.2 Conversion functions	144
18.3 Conversion by expansion (for PDF contexts)	144
18.4 Possibilities, and things to do	144
19 The <code>l3quark</code> module: Quarks and scan marks	146
19.1 Quarks	146
19.2 Defining quarks	147
19.3 Quark tests	147
19.4 Recursion	148
19.4.1 An example of recursion with quarks	149
19.5 Scan marks	150

20 The <code>l3seq</code> module: Sequences and stacks	151
20.1 Creating and initialising sequences	151
20.2 Appending data to sequences	153
20.3 Recovering items from sequences	153
20.4 Recovering values from sequences with branching	155
20.5 Modifying sequences	156
20.6 Sequence conditionals	157
20.7 Mapping over sequences	157
20.8 Using the content of sequences directly	160
20.9 Sequences as stacks	161
20.10 Sequences as sets	162
20.11 Constant and scratch sequences	163
20.12 Viewing sequences	164
21 The <code>l3int</code> module: Integers	165
21.1 Integer expressions	165
21.2 Creating and initialising integers	168
21.3 Setting and incrementing integers	169
21.4 Using integers	170
21.5 Integer expression conditionals	170
21.6 Integer expression loops	172
21.7 Integer step functions	174
21.8 Formatting integers	175
21.9 Converting from other formats to integers	176
21.10 Random integers	177
21.11 Viewing integers	177
21.12 Constant integers	178
21.13 Scratch integers	178
21.14 Direct number expansion	179
21.15 Primitive conditionals	179
22 The <code>l3flag</code> module: Expandable flags	181
22.1 Setting up flags	181
22.2 Expandable flag commands	182
23 The <code>l3clist</code> module: Comma separated lists	184
23.1 Creating and initialising comma lists	185
23.2 Adding data to comma lists	186
23.3 Modifying comma lists	187
23.4 Comma list conditionals	188
23.5 Mapping over comma lists	188
23.6 Using the content of comma lists directly	191
23.7 Comma lists as stacks	192
23.8 Using a single item	193
23.9 Viewing comma lists	193
23.10 Constant and scratch comma lists	194

24 The <code>l3token</code> module: Token manipulation	195
24.1 Creating character tokens	196
24.2 Manipulating and interrogating character tokens	197
24.3 Generic tokens	200
24.4 Converting tokens	201
24.5 Token conditionals	201
24.6 Peeking ahead at the next token	205
24.7 Description of all possible tokens	210
25 The <code>l3prop</code> module: Property lists	213
25.1 Creating and initialising property lists	214
25.2 Adding and updating property list entries	216
25.3 Recovering values from property lists	217
25.4 Modifying property lists	218
25.5 Property list conditionals	218
25.6 Recovering values from property lists with branching	219
25.7 Mapping over property lists	220
25.8 Viewing property lists	221
25.9 Scratch property lists	222
25.10 Constants	222
26 The <code>l3skip</code> module: Dimensions and skips	223
26.1 Creating and initialising <code>dim</code> variables	223
26.2 Setting <code>dim</code> variables	224
26.3 Utilities for dimension calculations	224
26.4 Dimension expression conditionals	225
26.5 Dimension expression loops	227
26.6 Dimension step functions	228
26.7 Using <code>dim</code> expressions and variables	229
26.8 Viewing <code>dim</code> variables	231
26.9 Constant dimensions	232
26.10 Scratch dimensions	232
26.11 Creating and initialising <code>skip</code> variables	232
26.12 Setting <code>skip</code> variables	233
26.13 Skip expression conditionals	234
26.14 Using <code>skip</code> expressions and variables	234
26.15 Viewing <code>skip</code> variables	234
26.16 Constant skips	235
26.17 Scratch skips	235
26.18 Inserting skips into the output	235
26.19 Creating and initialising <code>muskip</code> variables	236
26.20 Setting <code>muskip</code> variables	236
26.21 Using <code>muskip</code> expressions and variables	237
26.22 Viewing <code>muskip</code> variables	237
26.23 Constant muskips	238
26.24 Scratch muskips	238
26.25 Primitive conditional	238

27 The <code>l3keys</code> module: Key–value interfaces	239
27.1 Creating keys	240
27.2 Sub-dividing keys	245
27.3 Choice and multiple choice keys	246
27.4 Key usage scope	248
27.5 Setting keys	248
27.6 Handling of unknown keys	249
27.7 Selective key setting	250
27.8 Digesting keys	251
27.9 Utility functions for keys	251
27.10 Low-level interface for parsing key–val lists	252
28 The <code>l3intarray</code> module: Fast global integer arrays	255
28.1 Creating and initialising integer array variables	255
28.2 Adding data to integer arrays	256
28.3 Counting entries in integer arrays	256
28.4 Using a single entry	256
28.5 Integer array conditional	256
28.6 Viewing integer arrays	256
28.7 Implementation notes	257
29 The <code>l3fp</code> module: Floating points	258
29.1 Creating and initialising floating point variables	260
29.2 Setting floating point variables	260
29.3 Using floating points	261
29.4 Floating point conditionals	262
29.5 Floating point expression loops	264
29.6 Symbolic expressions	266
29.7 User-defined functions	268
29.8 Some useful constants, and scratch variables	269
29.9 Scratch variables	269
29.10 Floating point exceptions	270
29.11 Viewing floating points	271
29.12 Floating point expressions	271
29.12.1 Input of floating point numbers	271
29.12.2 Precedence of operators	272
29.12.3 Operations	273
29.13 Disclaimer and roadmap	280
30 The <code>l3fpararray</code> module: Fast global floating point arrays	283
30.1 Creating and initialising floating point array variables	283
30.2 Adding data to floating point arrays	283
30.3 Counting entries in floating point arrays	284
30.4 Using a single entry	284
30.5 Floating point array conditional	284
31 The <code>l3bitset</code> module: Bitsets	285
31.1 Creating bitsets	286
31.2 Setting and unsetting bits	287
31.3 Using bitsets	287

32 The <code>l3cctab</code> module: Category code tables	289
32.1 Creating and initialising category code tables	289
32.2 Using category code tables	290
32.3 Category code table conditionals	290
32.4 Constant and scratch category code tables	290
V Text manipulation	292
33 The <code>l3unicode</code> module: Unicode support functions	293
34 The <code>l3text</code> module: Text processing	296
34.1 Expanding text	296
34.2 Case changing	297
34.3 Removing formatting from text	299
34.4 Control variables	299
34.5 Mapping to graphemes	300
VI Typesetting	301
35 The <code>l3box</code> module: Boxes	302
35.1 Creating and initialising boxes	302
35.2 Using boxes	303
35.3 Measuring and setting box dimensions	304
35.4 Box conditionals	305
35.5 The last box inserted	305
35.6 Constant boxes	305
35.7 Scratch boxes	305
35.8 Viewing box contents	306
35.9 Boxes and color	306
35.10 Horizontal mode boxes	306
35.11 Vertical mode boxes	307
35.12 Using boxes efficiently	309
35.13 Affine transformations	310
35.14 Viewing part of a box	313
35.15 Primitive box conditionals	314
36 The <code>l3coffins</code> module: Coffin code layer	315
36.1 Creating and initialising coffins	315
36.2 Setting coffin content and poles	316
36.3 Coffin affine transformations	317
36.4 Joining and using coffins	318
36.5 Measuring coffins	318
36.6 Coffin diagnostics	319
36.7 Constants and variables	320

37 The <code>l3color</code> module: Color support	321
37.1 Color in boxes	321
37.2 Color models	321
37.3 Color expressions	323
37.4 Named colors	324
37.5 Selecting colors	324
37.6 Colors for fills and strokes	325
37.6.1 Coloring math mode material	325
37.7 Multiple color models	325
37.8 Exporting color specifications	326
37.9 Creating new color models	327
37.9.1 Color profiles	328
38 The <code>l3pdf</code> module: Core PDF support	329
38.1 Objects	329
38.1.1 Named objects	329
38.1.2 Indexed objects	330
38.1.3 General functions	330
38.2 Version	331
38.3 Page (media) size	331
38.4 Compression	331
38.5 Destinations	332
VII Implementation	333
39 <code>l3bootstrap</code> implementation	334
39.1 The <code>\pdfstrcmp</code> primitive in $X_{\text{q}}\text{TeX}$	334
39.2 Loading support Lua code	334
39.3 Engine requirements	335
39.4 The $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}3$ code environment	336
40 <code>l3names</code> implementation	338
41 <code>l3kernel-functions</code>: kernel-reserved functions	364
41.1 Internal <code>l3debug</code> kernel functions	364
41.2 Internal kernel functions	365
41.3 Kernel backend functions	372
42 <code>l3basics</code> implementation	374
42.1 Renaming some $\text{T}_{\text{E}}\text{X}$ primitives (again)	374
42.2 Defining some constants	376
42.3 Defining functions	376
42.4 Selecting tokens	377
42.5 Gobbling tokens from input	380
42.6 Debugging and patching later definitions	380
42.7 Conditional processing and definitions	381
42.8 Dissecting a control sequence	387
42.9 Exist or free	389
42.10 Preliminaries for new functions	392

42.11	Defining new functions	393
42.12	Copying definitions	395
42.13	Undefining functions	396
42.14	Generating parameter text from argument count	396
42.15	Defining functions from a given number of arguments	397
42.16	Using the signature to define functions	398
42.17	Checking control sequence equality	401
42.18	Diagnostic functions	401
42.19	Decomposing a macro definition	403
42.20	Doing nothing functions	404
42.21	Breaking out of mapping functions	404
42.22	Starting a paragraph	404
43	l3expan implementation	406
43.1	General expansion	406
43.2	Hand-tuned definitions	410
43.3	Last-unbraced versions	413
43.4	Preventing expansion	415
43.5	Controlled expansion	415
43.6	Defining function variants	416
43.7	Definitions with the automated technique	426
43.8	Held-over variant generation	427
44	l3sort implementation	429
44.1	Variables	429
44.2	Finding available <code>\toks</code> registers	430
44.3	Protected user commands	432
44.4	Merge sort	434
44.5	Expandable sorting	437
44.6	Messages	442
45	l3tl-analysis implementation	445
45.1	Internal functions	445
45.2	Internal format	445
45.3	Variables and helper functions	446
45.4	Plan of attack	448
45.5	Disabling active characters	449
45.6	First pass	450
45.7	Second pass	455
45.8	Mapping through the analysis	458
45.9	Showing the results	459
45.10	Peeking ahead	462
45.11	Messages	469

46	Regex implementation	470
46.1	Plan of attack	470
46.2	Helpers	471
46.2.1	Constants and variables	474
46.2.2	Testing characters	474
46.2.3	Internal auxiliaries	475
46.2.4	Character property tests	478
46.2.5	Simple character escape	480
46.3	Compiling	486
46.3.1	Variables used when compiling	487
46.3.2	Generic helpers used when compiling	488
46.3.3	Mode	489
46.3.4	Framework	491
46.3.5	Quantifiers	494
46.3.6	Raw characters	497
46.3.7	Character properties	499
46.3.8	Anchoring and simple assertions	500
46.3.9	Character classes	500
46.3.10	Groups and alternations	504
46.3.11	Catcodes and csnames	506
46.3.12	Raw token lists with <code>\u</code>	510
46.3.13	Other	514
46.3.14	Showing regexes	514
46.4	Building	521
46.4.1	Variables used while building	521
46.4.2	Framework	522
46.4.3	Helpers for building an NFA	525
46.4.4	Building classes	526
46.4.5	Building groups	528
46.4.6	Others	532
46.5	Matching	534
46.5.1	Variables used when matching	534
46.5.2	Matching: framework	537
46.5.3	Using states of the NFA	540
46.5.4	Actions when matching	541
46.6	Replacement	543
46.6.1	Variables and helpers used in replacement	543
46.6.2	Query and brace balance	545
46.6.3	Framework	546
46.6.4	Submatches	549
46.6.5	Csnames in replacement	551
46.6.6	Characters in replacement	552
46.6.7	An error	556
46.7	User functions	556
46.7.1	Variables and helpers for user functions	560
46.7.2	Matching	561
46.7.3	Extracting submatches	562
46.7.4	Replacement	567
46.7.5	Peeking ahead	570
46.8	Messages	576

46.9	Code for tracing	582
47	l3prg implementation	584
47.1	Primitive conditionals	584
47.2	Defining a set of conditional functions	584
47.3	The boolean data type	584
47.4	Internal auxiliaries	586
47.5	Boolean expressions	588
47.6	Logical loops	592
47.7	Producing multiple copies	594
47.8	Detecting T _E X's mode	595
47.9	Internal programming functions	596
48	l3sys implementation	598
48.1	Kernel code	598
48.1.1	Detecting the engine	598
48.1.2	Platform	601
48.1.3	Configurations	601
48.1.4	Access to the shell	604
48.2	Dynamic (every job) code	606
48.2.1	The name of the job	606
48.2.2	Time and date	607
48.2.3	Random numbers	608
48.2.4	Access to the shell	609
48.3	System queries	609
48.3.1	Held over from l3file	611
48.4	Last-minute code	611
48.4.1	Detecting the output	611
48.4.2	Configurations	612
49	l3msg implementation	614
49.1	Internal auxiliaries	614
49.2	Creating messages	614
49.3	Messages: support functions and text	616
49.4	Showing messages: low level mechanism	617
49.5	Displaying messages	619
49.6	Kernel-specific functions	628
49.7	Internal messages	629
49.8	Expandable errors	636
49.9	Message formatting	637

50 l3file implementation	638
50.1 Input operations	638
50.1.1 Variables and constants	638
50.1.2 Stream management	639
50.1.3 Reading input	642
50.2 Output operations	645
50.2.1 Variables and constants	645
50.2.2 Internal auxiliaries	646
50.3 Stream management	647
50.3.1 Deferred writing	649
50.3.2 Immediate writing	650
50.3.3 Special characters for writing	651
50.3.4 Hard-wrapping lines to a character count	651
50.4 File operations	660
50.4.1 Internal auxiliaries	662
50.5 GetIdInfo	678
50.6 Checking the version of kernel dependencies	679
50.7 Messages	681
50.8 Functions delayed from earlier modules	681
51 l3luatex implementation	683
51.1 Breaking out to Lua	683
51.2 Messages	684
51.3 Lua functions for internal use	684
51.4 Preserving iniTeX Lua data for runs	690
52 l3legacy implementation	692
53 l3tl implementation	694
53.1 Functions	694
53.2 Constant token lists	696
53.3 Adding to token list variables	696
53.4 Internal quarks and quark-query functions	699
53.5 Reassigning token list category codes	700
53.6 Modifying token list variables	703
53.7 Token list conditionals	707
53.8 Mapping over token lists	712
53.9 Using token lists	714
53.10 Working with the contents of token lists	714
53.11 The first token from a token list	717
53.12 Token by token changes	722
53.13 Using a single item	724
53.14 Viewing token lists	727
53.15 Internal scan marks	729
53.16 Scratch token lists	729
54 l3tl-build implementation	730

55	l3str implementation	734
55.1	Internal auxiliaries	734
55.2	Creating and setting string variables	735
55.3	Modifying string variables	736
55.4	String comparisons	737
55.5	Mapping over strings	741
55.6	Accessing specific characters in a string	743
55.7	Counting characters	747
55.8	The first character in a string	749
55.9	String manipulation	750
55.10	Viewing strings	753
56	l3str-convert implementation	754
56.1	Helpers	754
56.1.1	Variables and constants	754
56.2	String conditionals	756
56.3	Conversions	757
56.3.1	Producing one byte or character	757
56.3.2	Mapping functions for conversions	758
56.3.3	Error-reporting during conversion	759
56.3.4	Framework for conversions	760
56.3.5	Byte unescape and escape	764
56.3.6	Native strings	765
56.3.7	clist	766
56.3.8	8-bit encodings	766
56.4	Messages	769
56.5	Escaping definitions	770
56.5.1	Unescape methods	771
56.5.2	Escape methods	775
56.6	Encoding definitions	777
56.6.1	UTF-8 support	777
56.6.2	UTF-16 support	782
56.6.3	UTF-32 support	787
56.7	PDF names and strings by expansion	790
56.7.1	ISO 8859 support	791
57	l3quark implementation	808
57.1	Quarks	808
57.2	Scan marks	816

58 l3seq implementation	818
58.1 Allocation and initialisation	819
58.2 Appending data to either end	822
58.3 Modifying sequences	823
58.4 Sequence conditionals	827
58.5 Recovering data from sequences	829
58.6 Mapping over sequences	833
58.7 Using sequences	838
58.8 Sequence stacks	839
58.9 Viewing sequences	839
58.10 Scratch sequences	840
59 l3int implementation	841
59.1 Integer expressions	842
59.2 Creating and initialising integers	844
59.3 Setting and incrementing integers	846
59.4 Using integers	847
59.5 Integer expression conditionals	847
59.6 Integer expression loops	851
59.7 Integer step functions	852
59.8 Formatting integers	854
59.9 Converting from other formats to integers	860
59.10 Viewing integer	862
59.11 Random integers	863
59.12 Constant integers	863
59.13 Scratch integers	864
59.14 Integers for earlier modules	864
60 l3flag implementation	865
60.1 Protected flag commands	865
60.2 Expandable flag commands	866
60.3 Old n-type flag commands	867
61 l3clist implementation	869
61.1 Removing spaces around items	870
61.2 Allocation and initialisation	871
61.3 Adding data to comma lists	873
61.4 Comma lists as stacks	874
61.5 Modifying comma lists	876
61.6 Comma list conditionals	879
61.7 Mapping over comma lists	880
61.8 Using comma lists	884
61.9 Using a single item	886
61.10 Viewing comma lists	888
61.11 Scratch comma lists	889

62	l3token implementation	890
62.1	Internal auxiliaries	890
62.2	Manipulating and interrogating character tokens	890
62.3	Creating character tokens	893
62.4	Generic tokens	896
62.5	Token conditionals	898
62.6	Peeking ahead at the next token	908
63	l3prop implementation	915
63.1	Internal auxiliaries	916
63.2	Structure of a property list	917
63.3	Allocation and initialisation	919
63.4	Accessing data in property lists	926
63.5	Removing data from property lists	929
63.6	Adding data to property lists	932
63.7	Property list conditionals	934
63.8	Mapping over property lists	936
63.9	Uses of mapping over property lists	938
63.10	Viewing property lists	939
64	l3skip implementation	943
64.1	Length primitives renamed	943
64.2	Internal auxiliaries	943
64.3	Creating and initialising <code>dim</code> variables	943
64.4	Setting <code>dim</code> variables	944
64.5	Utilities for dimension calculations	945
64.6	Dimension expression conditionals	946
64.7	Dimension expression loops	948
64.8	Dimension step functions	949
64.9	Using <code>dim</code> expressions and variables	951
64.10	Conversion of <code>dim</code> to other units	952
64.11	Viewing <code>dim</code> variables	957
64.12	Constant dimensions	957
64.13	Scratch dimensions	957
64.14	Creating and initialising <code>skip</code> variables	957
64.15	Setting <code>skip</code> variables	959
64.16	Skip expression conditionals	959
64.17	Using <code>skip</code> expressions and variables	960
64.18	Inserting skips into the output	960
64.19	Viewing <code>skip</code> variables	961
64.20	Constant skips	961
64.21	Scratch skips	961
64.22	Creating and initialising <code>muskip</code> variables	961
64.23	Setting <code>muskip</code> variables	962
64.24	Using <code>muskip</code> expressions and variables	963
64.25	Viewing <code>muskip</code> variables	963
64.26	Constant muskips	964
64.27	Scratch muskips	964

65 l3keys implementation	965
65.1 Low-level interface	965
65.2 Constants and variables	972
65.2.1 Internal auxiliaries	974
65.3 The key defining mechanism	975
65.4 Turning properties into actions	977
65.5 Creating key properties	984
65.6 Setting keys	990
65.7 Utilities	999
65.8 Messages	1002
66 l3intarray implementation	1004
66.1 Lua implementation	1004
66.1.1 Allocating arrays	1004
66.1.2 Array items	1007
66.1.3 Working with contents of integer arrays	1009
66.2 Font dimension based implementation	1010
66.2.1 Allocating arrays	1011
66.2.2 Array items	1012
66.2.3 Working with contents of integer arrays	1014
66.3 Common parts	1016
67 l3fp implementation	1017
68 l3fp-aux implementation	1018
68.1 Access to primitives	1018
68.2 Internal representation	1018
68.3 Using arguments and semicolons	1019
68.4 Constants, and structure of floating points	1020
68.5 Overflow, underflow, and exact zero	1023
68.6 Expanding after a floating point number	1023
68.7 Other floating point types	1024
68.8 Packing digits	1027
68.9 Decimate (dividing by a power of 10)	1030
68.10 Functions for use within primitive conditional branches	1032
68.11 Integer floating points	1033
68.12 Small integer floating points	1034
68.13 Fast string comparison	1035
68.14 Name of a function from its l3fp-parse name	1035
68.15 Messages	1035
69 l3fp-traps implementation	1036
69.1 Flags	1036
69.2 Traps	1036
69.3 Errors	1040
69.4 Messages	1040
70 l3fp-round implementation	1042
70.1 Rounding tools	1042
70.2 The round function	1046

71 l3fp-parse implementation	1051
71.1 Work plan	1051
71.1.1 Storing results	1052
71.1.2 Precedence and infix operators	1053
71.1.3 Prefix operators, parentheses, and functions	1056
71.1.4 Numbers and reading tokens one by one	1057
71.2 Main auxiliary functions	1059
71.3 Helpers	1060
71.4 Parsing one number	1061
71.4.1 Numbers: trimming leading zeros	1067
71.4.2 Number: small significand	1068
71.4.3 Number: large significand	1070
71.4.4 Number: beyond 16 digits, rounding	1072
71.4.5 Number: finding the exponent	1075
71.5 Constants, functions and prefix operators	1078
71.5.1 Prefix operators	1078
71.5.2 Constants	1081
71.5.3 Functions	1082
71.6 Main functions	1083
71.7 Infix operators	1085
71.7.1 Closing parentheses and commas	1086
71.7.2 Usual infix operators	1088
71.7.3 Juxtaposition	1089
71.7.4 Multi-character cases	1089
71.7.5 Ternary operator	1090
71.7.6 Comparisons	1090
71.8 Tools for functions	1092
71.9 Messages	1095
72 l3fp-assign implementation	1096
72.1 Assigning values	1096
72.2 Updating values	1097
72.3 Showing values	1097
72.4 Some useful constants and scratch variables	1099
73 l3fp-logic implementation	1100
73.1 Syntax of internal functions	1100
73.2 Tests	1100
73.3 Comparison	1101
73.4 Floating point expression loops	1104
73.5 Extrema	1107
73.6 Boolean operations	1109
73.7 Ternary operator	1110

74	l3fp-basics implementation	1112
74.1	Addition and subtraction	1112
74.1.1	Sign, exponent, and special numbers	1113
74.1.2	Absolute addition	1115
74.1.3	Absolute subtraction	1117
74.2	Multiplication	1121
74.2.1	Signs, and special numbers	1121
74.2.2	Absolute multiplication	1123
74.3	Division	1125
74.3.1	Signs, and special numbers	1125
74.3.2	Work plan	1126
74.3.3	Implementing the significand division	1129
74.4	Square root	1134
74.5	About the sign and exponent	1141
74.6	Operations on tuples	1142
75	l3fp-extended implementation	1144
75.1	Description of fixed point numbers	1144
75.2	Helpers for numbers with extended precision	1145
75.3	Multiplying a fixed point number by a short one	1146
75.4	Dividing a fixed point number by a small integer	1146
75.5	Adding and subtracting fixed points	1147
75.6	Multiplying fixed points	1148
75.7	Combining product and sum of fixed points	1149
75.8	Extended-precision floating point numbers	1152
75.9	Dividing extended-precision numbers	1154
75.10	Inverse square root of extended precision numbers	1158
75.11	Converting from fixed point to floating point	1160
76	l3fp-expo implementation	1162
76.1	Logarithm	1162
76.1.1	Work plan	1162
76.1.2	Some constants	1163
76.1.3	Sign, exponent, and special numbers	1163
76.1.4	Absolute ln	1163
76.2	Exponential	1171
76.2.1	Sign, exponent, and special numbers	1171
76.3	Power	1175
76.4	Factorial	1181

77 l3fp-trig implementation	1184
77.1 Direct trigonometric functions	1185
77.1.1 Filtering special cases	1185
77.1.2 Distinguishing small and large arguments	1188
77.1.3 Small arguments	1189
77.1.4 Argument reduction in degrees	1189
77.1.5 Argument reduction in radians	1190
77.1.6 Computing the power series	1198
77.2 Inverse trigonometric functions	1200
77.2.1 Arctangent and arccotangent	1201
77.2.2 Arcsine and arccosine	1206
77.2.3 Arccosecant and arcsecant	1208
78 l3fp-convert implementation	1210
78.1 Dealing with tuples	1210
78.2 Trimming trailing zeros	1210
78.3 Scientific notation	1211
78.4 Decimal representation	1212
78.5 Token list representation	1214
78.6 Formatting	1215
78.7 Convert to dimension or integer	1215
78.8 Convert from a dimension	1216
78.9 Use and eval	1217
78.10 Convert an array of floating points to a comma list	1218
79 l3fp-random implementation	1220
79.1 Engine support	1220
79.2 Random floating point	1223
79.3 Random integer	1224
80 l3fp-types implementation	1229
80.1 Support for types	1229
80.2 Dispatch according to the type	1229
81 l3fp-symbolic implementation	1232
81.1 Misc	1232
81.2 Building blocks for expressions	1232
81.3 Expanding after a symbolic expression	1233
81.4 Applying infix operators to expressions	1234
81.5 Applying prefix functions to expressions	1235
81.6 Conversions	1236
81.7 Identifiers	1237
81.8 Declaring variables and assigning values	1238
81.9 Messages	1241
81.10 Road-map	1241
82 l3fp-functions implementation	1242
82.1 Declaring functions	1242
82.2 Defining functions by their expression	1243

83	l3parray implementation	1246
83.1	Allocating arrays	1246
83.2	Array items	1247
84	l3bitset implementation	1251
84.1	Messages	1256
85	l3ctab implementation	1257
85.1	Variables	1257
85.2	Allocating category code tables	1258
85.3	Saving category code tables	1259
85.4	Using category code tables	1260
85.5	Category code table conditionals	1265
85.6	Constant category code tables	1267
85.7	Messages	1268
86	l3unicode implementation	1270
86.1	User functions	1270
86.2	Data loader	1274
87	l3text implementation	1285
87.1	Internal auxiliaries	1285
87.2	Utilities	1286
87.3	Codepoint utilities	1289
87.4	Configuration variables	1292
87.5	Expansion to formatted text	1293
88	l3text-case implementation	1302
88.1	Case changing	1302
89	l3text-map implementation	1337
89.1	Mapping to text	1337
90	l3text-purify implementation	1345
90.1	Purifying text	1345
90.2	Accent and letter-like data for purifying text	1351
91	l3box implementation	1358
91.1	Support code	1358
91.2	Creating and initialising boxes	1358
91.3	Measuring and setting box dimensions	1359
91.4	Using boxes	1360
91.5	Box conditionals	1361
91.6	The last box inserted	1361
91.7	Constant boxes	1361
91.8	Scratch boxes	1362
91.9	Viewing box contents	1362
91.10	Horizontal mode boxes	1363
91.11	Vertical mode boxes	1365
91.12	Affine transformations	1368
91.13	Viewing part of a box	1377

92 l3coffins implementation	1380
92.1 Coffins: data structures and general variables	1380
92.2 Basic coffin functions	1381
92.3 Measuring coffins	1387
92.4 Coffins: handle and pole management	1387
92.5 Coffins: calculation of pole intersections	1391
92.6 Affine transformations	1393
92.7 Aligning and typesetting of coffins	1401
92.8 Coffin diagnostics	1406
92.9 Messages	1412
93 l3color implementation	1413
93.1 Basics	1413
93.2 Predefined color names	1414
93.3 Setup	1415
93.4 Utility functions	1415
93.5 Model conversion	1416
93.6 Color expressions	1417
93.7 Selecting colors (and color models)	1426
93.8 Math color	1428
93.9 Fill and stroke color	1431
93.10 Defining named colors	1431
93.11 Exporting colors	1434
93.12 Additional color models	1436
93.13 Applying profiles	1451
93.14 Diagnostics	1451
93.15 Messages	1452
94 l3pdf implementation	1456
94.1 Compression	1456
94.2 Objects	1457
94.3 Version	1461
94.4 Page size	1462
94.5 Destinations	1463
94.6 PDF Page size (media box)	1463
95 l3deprecation implementation	1465
95.1 Patching definitions to deprecate	1465
95.2 Deprecated l3basics functions	1467
95.3 Deprecated l3file functions	1467
95.4 Deprecated l3keys functions	1467
95.5 Deprecated l3msg functions	1468
95.6 Deprecated l3pdf functions	1468
95.7 Deprecated l3prg functions	1469
95.8 Deprecated l3str functions	1469
95.9 Deprecated l3seq functions	1470
95.10 Deprecated l3sys functions	1470
95.11 Deprecated l3text functions	1471
95.12 Deprecated l3tl functions	1471
95.13 Deprecated l3token functions	1472

95.14 Deprecated l3prop functions	1474
96 l3debug implementation	1475
Index	1499

Part I
Introduction

Chapter 1

Introduction to `expl3` and this document

This document is intended to act as a comprehensive reference manual for the `expl3` language. A general guide to the `LATEX3` programming language is found in [expl3.pdf](#).

1.1 Naming functions and variables

`LATEX3` does not use `@` as a “letter” for defining internal macros. Instead, the symbols `_` and `:` are used in internal macro names to provide structure. The name of each *function* is divided into logical units using `_`, while `:` separates the *name* of the function from the *argument specifier* (“arg-spec”). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all `expl3` function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end `:`. Most functions take one or more arguments, and use the following argument specifiers:

N and n These mean *no manipulation*, of a single token for `N` and of a set of tokens given in braces for `n`. Both pass the argument through exactly as given. Usually, if you use a single token for an `n` argument, all will be well.

c This means *csname*, and indicates that the argument will be turned into a `csname` before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`. All macros that appear in the argument are expanded. An internal error will occur if the result of expansion inside a `c`-type argument is not a series of character tokens.

V and v These mean *value of variable*. The `V` and `v` specifiers are used to get the content of a variable without needing to worry about the underlying `TEX` structure containing the data. A `V` argument will be a single token (similar to `N`), for example

`\foo:V \MyVariable`; on the other hand, using `v` a csname is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.

- o This means *expansion once*. In general, the `V` and `v` specifiers are favoured over `o` for recovering stored information. However, `o` is useful for correctly processing information with delimited arguments.
- x The `x` specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The \TeX `\edef` primitive carries out this type of expansion. Functions which feature an `x`-type argument are *not* expandable.
- e The `e` specifier is in many respects identical to `x`, but uses `\expanded` primitive. Parameter character (usually `#`) in the argument need not be doubled. Functions which feature an `e`-type argument may be expandable.
- f The `f` specifier stands for *full expansion*, and in contrast to `x` stops at the first non-expandable token (reading the argument from left to right) without trying to expand it. If this token is a *space token*, it is gobbled, and thus won't be part of the resulting argument. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_my_a_tl { A }
\tl_set:Nn \l_my_b_tl { B }
\tl_set:Nf \l_my_a_tl { \l_my_a_tl \l_my_b_tl }
```

will leave `\l_my_a_tl` with the content `A\l_my_b_tl`, as `A` cannot be expanded and so terminates expansion before `\l_my_b_tl` is considered.

- T and F** For logic tests, there are the branch specifiers `T` (*true*) and `F` (*false*). Both specifiers treat the input in the same way as `n` (no change), but make the logic much easier to see.
- p The letter `p` indicates \TeX *parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w Finally, there is the `w` specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some specified string).
- D The `D` stands for **Do not use**. All of the \TeX primitives are initially `\let` to a `D` name, and some are then given a second name. These functions have no standardized syntax, they are engine dependent and their name can change without warning, thus their use is *strongly discouraged* in package code: programmers should instead use the interfaces documented in [interface3.pdf](#).

Notice that the argument specifier describes how the argument is processed prior to being passed to the underlying function. For example, `\foo:c` will take its argument, convert it to a control sequence and pass it to `\foo:N`.

Variables are named in a similar manner to functions, but begin with a single letter to define the type of variable:

- c Constant: global parameters whose value should not be changed.

g Parameters whose value should only be set globally.

l Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module¹ name and then a descriptive part. Variables end with a short identifier to show the variable type:

bitset a set of bits (a string made up of a series of 0 and 1 tokens that are accessed by position).

clist Comma separated list.

dim “Rigid” lengths.

fp Floating-point values;

int Integer-valued count register.

muskip “Rubber” lengths for use in mathematics.

skip “Rubber” lengths.

str String variables: contain character data.

tl Token list variables: placeholder for a token list.

Applying V-type or v-type expansion to variables of one of the above types is supported, while it is not supported for the following variable types:

bool Either true or false.

box Box register.

coffin A “box with handles” — a higher-level data type for carrying out **box** alignment operations.

flag Non-negative integer that can be incremented expandably.

fparray Fixed-size array of floating point values.

intarray Fixed-size array of integers.

ior/iow An input or output stream, for reading from or writing to, respectively.

prop Property list: analogue of dictionary or associative arrays in other languages.

regex Regular expression.

seq “Sequence”: a data type used to implement lists (with access at both ends) and stacks.

¹The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the **int** module contains some scratch variables called `\l_tmpa_int`, `\l_tmpb_int`, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in `\l_int_tmpa_int` would be very unreadable.

1.1.1 Scratch variables

Modules focussed on variable usage typically provide four scratch variables, two local and two global, with names of the form `\langle scope \rangle_tmpa_⟨ type ⟩/\langle scope \rangle_tmpb_⟨ type ⟩`. These are never used by the core code. The nature of \TeX grouping means that as with any other scratch variable, these should only be set and used with no intervening third-party code.

1.1.2 Terminological inexactitude

A word of warning. In this document, and others referring to the `expl3` programming modules, we often refer to “variables” and “functions” as if they were actual constructs from a real programming language. In truth, \TeX is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a “function” with no arguments and a “token list variable” are almost the same.² On the other hand, some “variables” are actually registers that must be initialised and their values set and retrieved with specific functions.

The conventions of the `expl3` code are designed to clearly separate the ideas of “macros that contain data” and “macros that contain code”, and a consistent wrapper is applied to all forms of “data” whether they be macros or actually registers. This means that sometimes we will use phrases like “the function returns a value”, when actually we just mean “the macro expands to something”. Similarly, the term “execute” might be used in place of “expand” or it might refer to the more specific case of “processing in \TeX ’s stomach” (if you are familiar with the \TeX book parlance).

If in doubt, please ask; chances are we’ve been hasty in writing certain definitions and need to be told to tighten up our terminology.

1.2 Documentation conventions

This document is typeset with the experimental `l3doc` class; several conventions are used to help describe the features of the code. A number of conventions are used here to make the documentation clearer.

Each group of related functions is given in a box. For a function with a “user” name, this might read:

```
\ExplSyntaxOn \ExplSyntaxOn ... \ExplSyntaxOff
\ExplSyntaxOff
```

The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use `_` and `:` in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed *variants* of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

² \TeX nically, functions with no arguments are `\long` while token list variables are not.

`\seq_new:N` `\seq_new:N` $\langle sequence \rangle$

`\seq_new:c`

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, $\langle sequence \rangle$ indicates that `\seq_new:N` expects the name of a sequence. From the argument specifier, `\seq_new:c` also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

Fully expandable functions Some functions are fully expandable, which allows them to be used within an `x`-type or `e`-type argument (in plain \TeX terms, inside an `\edef` or `\expanded`), as well as within an `f`-type argument. These fully expandable functions are indicated in the documentation by a star:

`\cs_to_str:N` \star `\cs_to_str:N` $\langle cs \rangle$

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a $\langle cs \rangle$, shorthand for a $\langle control\ sequence \rangle$.

Restricted expandable functions A few functions are fully expandable but cannot be fully expanded within an `f`-type argument. In this case a hollow star is used to indicate this:

`\seq_map_function:NN` \star `\seq_map_function:NN` $\langle seq \rangle$ $\langle function \rangle$

Conditional functions Conditional (`if`) functions are normally defined in three variants, with `T`, `F` and `TF` argument specifiers. This allows them to be used for different “true”/“false” branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

`\sys_if_engine_xetex:TF` \star `\sys_if_engine_xetex:TF` $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

The underlining and italic of `TF` indicates that three functions are available:

- `\sys_if_engine_xetex:T`
- `\sys_if_engine_xetex:F`
- `\sys_if_engine_xetex:TF`

Usually, the illustration will use the `TF` variant, and so both $\langle true\ code \rangle$ and $\langle false\ code \rangle$ will be shown. The two variant forms `T` and `F` take only $\langle true\ code \rangle$ and $\langle false\ code \rangle$, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the `expl3` modules should be defined in this way.

Variables, constants and so on are described in a similar manner:

`\l_tmpa_tl` A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in $\text{\LaTeX} 2_{\epsilon}$ or plain \TeX . In these cases, the text will include an extra “ **\TeX hackers note**” section:

`\token_to_str:N` * `\token_to_str:N` (*token*)

The normal description text.

TeXhackers note: Detail for the experienced TeX or L^AT_EX 2_ε programmer. In this case, it would point out that this function is the TeX primitive `\string`.

Changes to behaviour When new functions are added to `expl3`, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of `expl3` after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behaviour.

1.3 Formal language conventions which apply generally

As this is a formal reference guide for L^AT_EX 3 programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

For tests which have a `TF` argument specification, the test if evaluated to give a logically `TRUE` or `FALSE` result. Depending on this result, either the `<true code>` or the `<false code>` will be left in the input stream. In the case where the test is expandable, and a predicate (`_p`) variant is available, the logical value determined by the test is left in the input stream: this will typically be part of a larger logical construct.

1.4 TeX concepts not supported by L^AT_EX 3

The TeX concept of an “`\outer`” macro is *not supported* at all by L^AT_EX 3. As such, the functions provided here may break when used on top of L^AT_EX 2_ε if `\outer` tokens are used in the arguments.

Part II
Bootstrapping

Chapter 2

The l3bootstrap module Bootstrap code

2.1 Using the L^AT_EX₃ modules

The modules documented in `interface3` (and this file) are designed to be used on top of L^AT_EX_{2 ϵ} and are already pre-loaded since L^AT_EX_{2 ϵ} 2020-02-02. To support older formats, the `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions are still available to load them all as one.

As the modules use a coding syntax different from standard L^AT_EX_{2 ϵ} it provides a few functions for setting it up.

`\ExplSyntaxOn` `\ExplSyntaxOn <code> \ExplSyntaxOff`

`\ExplSyntaxOff`

Updated: 2011-08-13

The `\ExplSyntaxOn` function switches to a category code regime in which spaces and new lines are ignored, and in which the colon (`:`) and underscore (`_`) are treated as “letters”, thus allowing access to the names of code functions and variables. Within this environment, `~` is used to input a space. The `\ExplSyntaxOff` reverts to the document category code regime.

T_EXhackers note: Spaces introduced by `~` behave much in the same way as normal space characters in the standard category code regime: they are ignored after a control word or at the start of a line, and multiple consecutive `~` are equivalent to a single one. However, `~` is *not* ignored at the end of a line.

`\ProvidesExplPackage` `\ProvidesExplPackage <package> <date> <version> <description>`

`\ProvidesExplClass`

`\ProvidesExplFile`

Updated: 2023-08-03

These functions act broadly in the same way as the corresponding L^AT_EX_{2 ϵ} kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as L^AT_EX_{2 ϵ} provides in turning on `\makeatletter` within package and class code.) The `<date>` should be given in the format `<year>/<month>/<day>` or in the ISO date format `<year>-<month>-<day>`. If the `<version>` is given then a leading `v` is optional: if given as a “pure” version string, a `v` will be prepended.

`\GetIdInfo` `\GetIdInfo $Id: <SVN info field> $ {(description)}`

Updated: 2012-06-04 Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\ExplFileName` for the part of the file name leading up to the period, `\ExplFileDate` for date, `\ExplFileVersion` for version and `\ExplFileDescription` for the description.

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or similar are loaded with usual L^AT_EX 2_ε category codes and the L^AT_EX 3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

```
\ProvidesExplPackage{\ExplFileName}
  {\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

Chapter 3

The `l3names` module Namespace for primitives

3.1 Setting up the `LATEX3` programming language

This module is at the core of the `LATEX3` programming language. It performs the following tasks:

- defines new names for all `TEX` primitives;
- emulate required primitives not provided by default in `LuaTEX`;
- switches to the category code régime for programming;

This module is entirely dedicated to primitives (and emulations of these), which should not be used directly within `LATEX3` code (outside of “kernel-level” code). As such, the primitives are not documented here: *The T_EXbook*, *T_EX by Topic* and the manuals for `pdfTEX`, `XYTEX`, `LuaTEX`, `pTEX` and `upTEX` should be consulted for details of the primitives. These are named `\tex_<name>:D`, typically based on the primitive’s `<name>` in `pdfTEX` and omitting a leading `pdf` when the primitive is not related to pdf output.

Part III
Programming Flow

Chapter 4

The `l3basics` module

Basic definitions

As the name suggests, this module holds some basic definitions which are needed by most or all other modules in this set.

Here we describe those functions that are used all over the place. By that, we mean functions dealing with the construction and testing of control sequences. Furthermore the basic parts of conditional processing are covered; conditional processing dealing with specific data types is described in the modules specific for the respective data types.

4.1 No operation functions

`\prg_do_nothing:` ★ `\prg_do_nothing:`

An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

`\scan_stop:` `\scan_stop:`

A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

4.2 Grouping material

`\group_begin:` `\group_begin:`

`\group_end:` `\group_end:`

These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each `\group_begin:` must be matched by a `\group_end:`, although this does not have to occur within the same function. Indeed, it is often necessary to start a group within one function and finish it within another, for example when seeking to use non-standard category codes.

`TeX`hackers note: These are the `TeX` primitives `\begingroup` and `\endgroup`.

`\group_insert_after:N` `\group_insert_after:N` $\langle token \rangle$

Adds $\langle token \rangle$ to the list of $\langle tokens \rangle$ to be inserted when the current group level ends. The list of $\langle tokens \rangle$ to be inserted is empty at the beginning of a group: multiple applications of `\group_insert_after:N` may be used to build the inserted list one $\langle token \rangle$ at a time. The current group level may be closed by a `\group_end:` function or by a token with category code 2 (close-group), namely a `}` if standard category codes apply.

TeXhackers note: This is the TeX primitive `\aftergroup`.

`\group_show_list:` `\group_show_list:`

`\group_log_list:` `\group_log_list:`

New: 2021-05-11

Display (to the terminal or log file) a list of the groups that are currently opened. This is intended for tracking down problems.

TeXhackers note: This is a wrapper around the ϵ -TeX primitive `\showgroups`.

4.3 Control sequences and functions

As TeX is a macro language, creating new functions means creating macros. At point of use, a function is replaced by the replacement text (“code”) in which each parameter in the code (`#1`, `#2`, *etc.*) is replaced the appropriate arguments absorbed by the function. In the following, $\langle code \rangle$ is therefore used as a shorthand for “replacement text”.

Functions which are not “protected” are fully expanded inside an `e`-type or `x`-type expansion. In contrast, “protected” functions are not expanded within `e` and `x` expansions.

4.3.1 Defining functions

Functions can be created with no requirement that they are declared first (in contrast to variables, which must always be declared). Declaring a function before setting up the code means that the name chosen is checked and an error raised if it is already in use. The name of a function can be checked at the point of definition using the `\cs_new...` functions: this is recommended for all functions which are defined for the first time.

There are three ways to define new functions. All classes define a function to expand to the substitution text. Within the substitution text the actual parameters are substituted for the formal parameters (`#1`, `#2`, ...).

new Create a new function with the `new` scope, such as `\cs_new:Npn`. The definition is global and results in an error if it is already defined.

set Create a new function with the `set` scope, such as `\cs_set:Npn`. The definition is restricted to the current TeX group and does not result in an error if the function is already defined.

gset Create a new function with the `gset` scope, such as `\cs_gset:Npn`. The definition is global and does not result in an error if the function is already defined.

Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

nopar Create a new function with the `nopar` restriction, such as `\cs_set_nopar:Npn`. The parameter may not contain `\par` tokens.

protected Create a new function with the `protected` restriction, such as `\cs_set_protected:Npn`. The parameter may contain `\par` tokens but the function will not expand within an `e`-type or `x`-type expansion.

Finally, the functions in Subsections 4.3.2 and 4.3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

N and n No manipulation.

T and F Functionally equivalent to `n` (you are actually encouraged to use the family of `\prg_new_conditional`: functions described in Section 9.1).

p and w These are special cases.

The `\cs_new`: functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 5.2.

4.3.2 Defining new functions using parameter text

<code>\cs_new:Npn</code>	<code>\cs_new:Npn <function> <parameters> {<code>}</code>
<code>\cs_new:cpn</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new:Npe</code>	<code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The
<code>\cs_new:cpe</code>	definition is global and an error results if the <code><function></code> is already defined.
<code>\cs_new:Npx</code>	
<code>\cs_new:cpx</code>	

<code>\cs_new_nopar:Npn</code>	<code>\cs_new_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_nopar:cpn</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new_nopar:Npe</code>	<code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. When
<code>\cs_new_nopar:cpe</code>	the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The
<code>\cs_new_nopar:Npx</code>	definition is global and an error results if the <code><function></code> is already defined.
<code>\cs_new_nopar:cpx</code>	

<code>\cs_new_protected:Npn</code>	<code>\cs_new_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_protected:cpn</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new_protected:Npe</code>	<code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The
<code>\cs_new_protected:cpe</code>	<code><function></code> will not expand within an <code>e</code> -type or <code>x</code> -type argument. The definition is
<code>\cs_new_protected:Npx</code>	global and an error results if the <code><function></code> is already defined.
<code>\cs_new_protected:cpx</code>	

```

\cs_new_protected_nopar:Npn \cs_new_protected_nopar:Npn <function> <parameters> {<code>}
\cs_new_protected_nopar:cpn
\cs_new_protected_nopar:Npe
\cs_new_protected_nopar:cpe
\cs_new_protected_nopar:Npx
\cs_new_protected_nopar:cpx

```

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain $\backslash par$ tokens. The $\langle function \rangle$ will not expand within an e-type or x-type argument. The definition is global and an error results if the $\langle function \rangle$ is already defined.

```

\cs_set:Npn \cs_set:Npn <function> <parameters> {<code>}
\cs_set:cpn
\cs_set:Npe
\cs_set:cpe
\cs_set:Npx
\cs_set:cpx

```

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level.

```

\cs_set_nopar:Npn \cs_set_nopar:Npn <function> <parameters> {<code>}
\cs_set_nopar:cpn
\cs_set_nopar:Npe
\cs_set_nopar:cpe
\cs_set_nopar:Npx
\cs_set_nopar:cpx

```

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain $\backslash par$ tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level.

```

\cs_set_protected:Npn \cs_set_protected:Npn <function> <parameters> {<code>}
\cs_set_protected:cpn
\cs_set_protected:Npe
\cs_set_protected:cpe
\cs_set_protected:Npx
\cs_set_protected:cpx

```

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level. The $\langle function \rangle$ will not expand within an e-type or x-type argument.

```

\cs_set_protected_nopar:Npn \cs_set_protected_nopar:Npn <function> <parameters> {<code>}
\cs_set_protected_nopar:cpn
\cs_set_protected_nopar:Npe
\cs_set_protected_nopar:cpe
\cs_set_protected_nopar:Npx
\cs_set_protected_nopar:cpx

```

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain $\backslash par$ tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level. The $\langle function \rangle$ will not expand within an e-type or x-type argument.

<code>\cs_gset:Npn</code>	<code>\cs_gset:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset:cpn</code>	Globally sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> ,
<code>\cs_gset:Npe</code>	the <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The
<code>\cs_gset:cpe</code>	assignment of a meaning to the <code><function></code> is <i>not</i> restricted to the current T _E X group
<code>\cs_gset:Npx</code>	level: the assignment is global.
<code>\cs_gset:cpx</code>	

<code>\cs_gset_nopar:Npn</code>	<code>\cs_gset_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_nopar:cpn</code>	Globally sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> ,
<code>\cs_gset_nopar:Npe</code>	the <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function.
<code>\cs_gset_nopar:cpe</code>	When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens.
<code>\cs_gset_nopar:Npx</code>	The assignment of a meaning to the <code><function></code> is <i>not</i> restricted to the current T _E X
<code>\cs_gset_nopar:cpx</code>	group level: the assignment is global.

<code>\cs_gset_protected:Npn</code>	<code>\cs_gset_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected:cpn</code>	Globally sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> ,
<code>\cs_gset_protected:Npe</code>	the <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The
<code>\cs_gset_protected:cpe</code>	assignment of a meaning to the <code><function></code> is <i>not</i> restricted to the current T _E X group
<code>\cs_gset_protected:Npx</code>	level: the assignment is global. The <code><function></code> will not expand within an e-type or
<code>\cs_gset_protected:cpx</code>	x-type argument.

<code>\cs_gset_protected_nopar:Npn</code>	<code>\cs_gset_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected_nopar:cpn</code>	
<code>\cs_gset_protected_nopar:Npe</code>	
<code>\cs_gset_protected_nopar:cpe</code>	
<code>\cs_gset_protected_nopar:Npx</code>	
<code>\cs_gset_protected_nopar:cpx</code>	

Globally sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The assignment of a meaning to the `<function>` is *not* restricted to the current T_EX group level: the assignment is global. The `<function>` will not expand within an e-type or x-type argument.

4.3.3 Defining new functions using the signature

<code>\cs_new:Nn</code>	<code>\cs_new:Nn <function> {<code>}</code>
<code>\cs_new:(cn Ne ce)</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
	number of <code><parameters></code> is detected automatically from the function signature. These
	<code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The
	definition is global and an error results if the <code><function></code> is already defined.

<code>\cs_new_nopar:Nn</code>	<code>\cs_new_nopar:Nn <function> {<code>}</code>
<code>\cs_new_nopar:(cn Ne ce)</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
	number of <code><parameters></code> is detected automatically from the function signature. These
	<code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. When
	the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The
	definition is global and an error results if the <code><function></code> is already defined.

<code>\cs_new_protected:Nn</code>	<code>\cs_new_protected:Nn <function> {<code>}</code>
<code>\cs_new_protected:(cn Ne ce)</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The <code><function></code> will not expand within an <code>e</code> -type or <code>x</code> -type argument. The definition is global and an error results if the <code><function></code> is already defined.

<code>\cs_new_protected_nopar:Nn</code>	<code>\cs_new_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_new_protected_nopar:(cn Ne ce)</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The <code><function></code> will not expand within an <code>e</code> -type or <code>x</code> -type argument. The definition is global and an error results if the <code><function></code> is already defined.

<code>\cs_set:Nn</code>	<code>\cs_set:Nn <function> {<code>}</code>
<code>\cs_set:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The assignment of a meaning to the <code><function></code> is restricted to the current <code>T_EX</code> group level.

<code>\cs_set_nopar:Nn</code>	<code>\cs_set_nopar:Nn <function> {<code>}</code>
<code>\cs_set_nopar:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The assignment of a meaning to the <code><function></code> is restricted to the current <code>T_EX</code> group level.

<code>\cs_set_protected:Nn</code>	<code>\cs_set_protected:Nn <function> {<code>}</code>
<code>\cs_set_protected:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The <code><function></code> will not expand within an <code>e</code> -type or <code>x</code> -type argument. The assignment of a meaning to the <code><function></code> is restricted to the current <code>T_EX</code> group level.

<code>\cs_set_protected_nopar:Nn</code>	<code>\cs_set_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_set_protected_nopar:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The assignment of a meaning to the <code><function></code> is restricted to the current <code>T_EX</code> group level.

<code>\cs_gset:Nn</code>	<code>\cs_gset:Nn <function> {<code>}</code>
<code>\cs_gset:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The assignment of a meaning to the <code><function></code> is global.

<code>\cs_gset_nopar:Nn</code>	<code>\cs_gset_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_nopar:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The assignment of a meaning to the <code><function></code> is global.

<code>\cs_gset_protected:Nn</code>	<code>\cs_gset_protected:Nn <function> {<code>}</code>
<code>\cs_gset_protected:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. The <code><function></code> will not expand within an <code>e</code> -type or <code>x</code> -type argument. The assignment of a meaning to the <code><function></code> is global.

<code>\cs_gset_protected_nopar:Nn</code>	<code>\cs_gset_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_protected_nopar:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , <i>etc.</i>) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The <code><function></code> will not expand within an <code>e</code> -type or <code>x</code> -type argument. The assignment of a meaning to the <code><function></code> is global.

<code>\cs_generate_from_arg_count:NNnn</code>	<code>\cs_generate_from_arg_count:NNnn <function> <creator></code>
<code>\cs_generate_from_arg_count:(NNno cNnn Ncnn)</code>	<code>{<number>} {<code>}</code>

Updated: 2012-01-14

Uses the `<creator>` function (which should have signature `Npn`, for example `\cs_new:Npn`) to define a `<function>` which takes `<number>` arguments and has `<code>` as replacement text. The `<number>` of arguments is an integer expression, evaluated as detailed for `\int_eval:n`.

4.3.4 Copying control sequences

Control sequences (not just functions as defined above) can be set to have the same meaning using the functions described here. Making two control sequences equivalent means that the second control sequence is a *copy* of the first (rather than a pointer to it). Thus the old and new control sequence are not tied together: changes to one are not reflected in the other.

In the following text “cs” is used as an abbreviation for “control sequence”.

<code>\cs_new_eq:NN</code>	<code>\cs_new_eq:NN</code> $\langle cs_1 \rangle$ $\langle cs_2 \rangle$
<code>\cs_new_eq:(Nc cN cc)</code>	<code>\cs_new_eq:NN</code> $\langle cs_1 \rangle$ $\langle token \rangle$

Globally creates $\langle control\ sequence_1 \rangle$ and sets it to have the same meaning as $\langle control\ sequence_2 \rangle$ or $\langle token \rangle$. The second control sequence may subsequently be altered without affecting the copy.

<code>\cs_set_eq:NN</code>	<code>\cs_set_eq:NN</code> $\langle cs_1 \rangle$ $\langle cs_2 \rangle$
<code>\cs_set_eq:(Nc cN cc)</code>	<code>\cs_set_eq:NN</code> $\langle cs_1 \rangle$ $\langle token \rangle$

Sets $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ is restricted to the current T_EX group level.

<code>\cs_gset_eq:NN</code>	<code>\cs_gset_eq:NN</code> $\langle cs_1 \rangle$ $\langle cs_2 \rangle$
<code>\cs_gset_eq:(Nc cN cc)</code>	<code>\cs_gset_eq:NN</code> $\langle cs_1 \rangle$ $\langle token \rangle$

Globally sets $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ is *not* restricted to the current T_EX group level: the assignment is global.

4.3.5 Deleting control sequences

There are occasions where control sequences need to be deleted. This is handled in a very simple manner.

<code>\cs_undefine:N</code>	<code>\cs_undefine:N</code> $\langle control\ sequence \rangle$
<code>\cs_undefine:c</code>	Sets $\langle control\ sequence \rangle$ to be globally undefined.

Updated: 2011-09-15

4.3.6 Showing control sequences

<code>\cs_meaning:N</code> *	<code>\cs_meaning:N</code> $\langle control\ sequence \rangle$
<code>\cs_meaning:c</code> *	This function expands to the <i>meaning</i> of the $\langle control\ sequence \rangle$ control sequence. For a macro, this includes the $\langle replacement\ text \rangle$.

Updated: 2011-12-22

T_EXhackers note: This is the T_EX primitive `\meaning`. For tokens that are not control sequences, it is more logical to use `\token_to_meaning:N`. The `c` variant correctly reports undefined arguments.

<code>\cs_show:N</code>	<code>\cs_show:N</code> $\langle control\ sequence \rangle$
<code>\cs_show:c</code>	Displays the definition of the $\langle control\ sequence \rangle$ on the terminal.

Updated: 2017-02-14

T_EXhackers note: This is similar to the T_EX primitive `\show`, wrapped to a fixed number of characters per line.

<code>\cs_log:N</code>	<code>\cs_log:N <control sequence></code>
<code>\cs_log:c</code>	Writes the definition of the <code><control sequence></code> in the log file. See also <code>\cs_show:N</code> which displays the result in the terminal.
New: 2014-08-22	
Updated: 2017-02-14	

4.3.7 Converting to and from control sequences

`\use:c` * `\use:c {<control sequence name>}`

Expands the `<control sequence name>` until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other `c`-type arguments the `<control sequence name>` must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

As an example of the `\use:c` function, both

```
\use:c { a b c }
```

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\use:c { \tl_use:N \l_my_tl }
```

would be equivalent to

```
\abc
```

after two expansions of `\use:c`.

<code>\cs_if_exist_use:N</code>	* <code>\cs_if_exist_use:N <control sequence></code>
<code>\cs_if_exist_use:c</code>	* <code>\cs_if_exist_use:NTF <control sequence> {<true code>} {<false code>}</code>
<code>\cs_if_exist_use:NTF</code>	* Tests whether the <code><control sequence></code> is currently defined according to the conditional
<code>\cs_if_exist_use:cTF</code>	* <code>\cs_if_exist:NTF</code> (whether as a function or another control sequence type), and if it is inserts the <code><control sequence></code> into the input stream followed by the <code><true code></code> . Otherwise the <code><false code></code> is used.
New: 2012-11-10	

`\cs:w` * `\cs:w <control sequence name> \cs_end:`

`\cs_end:` * Converts the given `<control sequence name>` into a single control sequence token. This process requires one expansion. The content for `<control sequence name>` may be literal material or from other expandable functions. The `<control sequence name>` must, when fully expanded, consist of character tokens which are not active: typically of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

TeXhackers note: These are the TeX primitives `\csname` and `\endcsname`.

As an example of the `\cs:w` and `\cs_end:` functions, both

```
\cs:w a b c \cs_end:
```

and

```

\l_my_tl
\l_my_tl { a b c }
\l_my_tl \cs_end:

```

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

```
\cs_to_str:N * \cs_to_str:N <control sequence>
```

Converts the given *<control sequence>* into a series of characters with category code 12 (other), except spaces, of category code 10. The result does *not* include the current escape token, contrarily to `\token_to_str:N`. Full expansion of this function requires exactly 2 expansion steps, and so an e-type or x-type expansion, or two o-type expansions are required to convert the *<control sequence>* to a sequence of characters in the input stream. In most cases, an f-expansion is correct as well, but this loses a space at the start of the result.

4.4 Analysing control sequences

```
\cs_split_function:N * \cs_split_function:N <function>
```

New: 2018-04-06

Splits the *<function>* into the *<name>* (*i.e.* the part before the colon) and the *<signature>* (*i.e.* after the colon). This information is then placed in the input stream in three parts: the *<name>*, the *<signature>* and a logic token indicating if a colon was found (to differentiate variables from function names). The *<name>* does not include the escape character, and both the *<name>* and *<signature>* are made up of tokens with category code 12 (other).

The next three functions decompose \TeX macros into their constituent parts: if the *<token>* passed is not a macro then no decomposition can occur. In the latter case, all three functions leave `\scan_stop:` in the input stream.

```
\cs_prefix_spec:N * \cs_prefix_spec:N <token>
```

New: 2019-02-27

If the *<token>* is a macro, this function leaves the applicable \TeX prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

```

\next:nn #1#2 { x #1~y #2 }
\cs_prefix_spec:N \next:nn

```

leaves `\long` in the input stream. If the *<token>* is not a macro then `\scan_stop:` is left in the input stream.

\TeX hackers note: The prefix can be empty, `\long`, `\protected` or `\protected\long` with backslash replaced by the current escape character.

`\cs_parameter_spec:N` * `\cs_parameter_spec:N` $\langle token \rangle$

New: 2022-06-24

If the $\langle token \rangle$ is a macro, this function leaves the primitive \TeX parameter specification in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1 y #2 }
\cs_parameter_spec:N \next:nn
```

leaves `#1#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

\TeX hackers note: If the parameter specification contains the string `->`, then the function produces incorrect results.

`\cs_replacement_spec:N` * `\cs_replacement_spec:N` $\langle token \rangle$

`\cs_replacement_spec:c` *

New: 2019-02-27

If the $\langle token \rangle$ is a macro, this function leaves the replacement text in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_replacement_spec:N \next:nn
```

leaves `x#1~y#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

\TeX hackers note: If the parameter specification contains the string `->`, then the function produces incorrect results.

4.5 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it is read more than once, the category code is determined by the situation in force when first function absorbs the token).

```

\use:n    * \use:n    {\group_1}
\use:nn   * \use:nn   {\group_1} {\group_2}
\use:nnn  * \use:nnn  {\group_1} {\group_2} {\group_3}
\use:nnnn * \use:nnnn {\group_1} {\group_2} {\group_3} {\group_4}

```

As illustrated, these functions absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also fixed by this process (if it has not already been by some other absorption). All of these functions require only a single expansion to operate, so that one expansion of

```
\use:nn { abc } { { def } }
```

results in the input stream containing

```
abc { def }
```

i.e. only the outer braces are removed.

TeXhackers note: The `\use:n` function is equivalent to L^AT_EX 2_ε's `\@firstofone`.

```

\use_i:nn      * \use_i:nn {\arg1} {\arg2}
\use_ii:nn     * \use_i:nnn {\arg1} {\arg2} {\arg3}
\use_i:nnn    * \use_i:nnnn {\arg1} {\arg2} {\arg3} {\arg4}
\use_iii:nnn  * \use_i:nnnnn {\arg1} {\arg2} {\arg3} {\arg4} {\arg5}
\use_iiii:nnn * \use_i:nnnnnn {\arg1} {\arg2} {\arg3} {\arg4} {\arg5} {\arg6}
\use_i:nnnn   * \use_i:nnnnnnn {\arg1} {\arg2} {\arg3} {\arg4} {\arg5} {\arg6} {\arg7}
\use_ii:nnnn  * \use_i:nnnnnnnn {\arg1} {\arg2} {\arg3} {\arg4} {\arg5} {\arg6} {\arg7}
\use_iii:nnnn * {\arg8}
\use_iv:nnnn  * \use_i:nnnnnnnnn {\arg1} {\arg2} {\arg3} {\arg4} {\arg5} {\arg6} {\arg7}
\use_i:nnnnn  * {\arg8} {\arg9}
\use_ii:nnnnn *
\use_iii:nnnnn *
\use_iv:nnnnn *
\use_v:nnnnn  *
\use_i:nnnnnn *
\use_ii:nnnnnn *
\use_iii:nnnnnn *
\use_iv:nnnnnn *
\use_v:nnnnnn *
\use_vi:nnnnnn *
\use_vii:nnnnnn *
\use_viii:nnnnnn *
\use_ix:nnnnnn *

```

* These functions absorb a number (n) arguments from the input stream. They then discard all arguments other than that indicated by the roman numeral, which is left in the input stream. For example, `\use_i:nn` discards the second argument, and leaves the content of the first argument in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

`\use_ii:nnn` * `\use_ii:nnn {⟨arg1⟩} {⟨arg2⟩} {⟨arg3⟩}`

This function absorbs three arguments and leaves the content of the first and second in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect. An example:

```
\use_ii:nnn { abc } { { def } } { ghi }
```

results in the input stream containing

```
abc { def }
```

i.e. the outer braces are removed and the third group is removed.

`\use_ii_i:nn` * `\use_ii_i:nn {⟨arg1⟩} {⟨arg2⟩}`

New: 2019-06-02

This function absorbs two arguments and leaves the content of the second and first in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect.

`\use_none:n` * `\use_none:n {⟨group1⟩}`

`\use_none:nn` *

`\use_none:nnn` *

`\use_none:nnnn` *

`\use_none:nnnnn` *

`\use_none:nnnnnn` *

`\use_none:nnnnnnn` *

`\use_none:nnnnnnnn` *

`\use_none:nnnnnnnnn` *

TeXhackers note: These are equivalent to L^AT_EX 2_ε's `\@gobble`, `\@gobbletwo`, *etc.*

`\use:e` * `\use:e {⟨expandable tokens⟩}`

New: 2018-06-18

Updated: 2023-07-05

Fully expands the `⟨token list⟩` in an `e`-type manner, in which parameter character (usually `#`) need not be doubled, *and* the function remains fully expandable.

TeXhackers note: `\use:e` is a wrapper around the primitive `\expanded`. It requires two expansions to complete its action.

4.5.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.

`\use_none_delimit_by_q_nil:w` * `\use_none_delimit_by_q_nil:w {balanced text} \q_nil`
`\use_none_delimit_by_q_stop:w` * `\use_none_delimit_by_q_stop:w {balanced text} \q_stop`
`\use_none_delimit_by_q_recursion_stop:w` * `\use_none_delimit_by_q_recursion_stop:w {balanced text}`
`\q_recursion_stop`

Absorb the `⟨balanced text⟩` from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

```

\use_i_delimit_by_q_nil:nw      * \use_i_delimit_by_q_nil:nw {<inserted tokens>} <balanced text>
\use_i_delimit_by_q_stop:nw    * \q_nil
\use_i_delimit_by_q_recursion_stop:nw * \use_i_delimit_by_q_recursion_stop:nw {<inserted tokens>} <balanced
text> \q_stop
\use_i_delimit_by_q_recursion_stop:nw {<inserted tokens>}
<balanced text> \q_recursion_stop

```

Absorb the *<balanced text>* from the input stream delimited by the marker given in the function name, leaving *<inserted tokens>* in the input stream for further processing.

4.6 Predicates and conditionals

L^AT_EX3 has three concepts for conditional flow processing:

Branching conditionals Functions that carry out a test and then execute, depending on its result, either the code supplied as the *<true code>* or the *<false code>*. These arguments are denoted with T and F, respectively. An example would be

```
\cs_if_free:cTF {abc} {<true code>} {<false code>}
```

a function that turns the first argument into a control sequence (since it's marked as c) then checks whether this control sequence is still free and then depending on the result carries out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as “conditionals”; whenever a TF function is defined it is usually accompanied by T and F functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions always provide all three versions.

Important to note is that these branching conditionals with *<true code>* and/or *<false code>* are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they are accompanied by a “predicate” for the same test as described below.

Predicates “Predicates” are functions that return a special type of boolean value which can be tested by the boolean expression parser. All functions of this type are expandable and have names that end with *_p* in the description part. For example,

```
\cs_if_free_p:N
```

would be a predicate function for the same type of test as the conditional described above. It would return “true” if its argument (a single token denoted by N) is still free for definition. It would be used in constructions like

```

\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {<true code>} {<false code>}

```

For each predicate defined, a “branching conditional” also exists that behaves like a conditional described above.

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain T_EX and L^AT_EX 2_ε. Their use is discouraged in expl3 (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

4.6.1 Tests on control sequences

```
\cs_if_eq_p:NN * \cs_if_eq_p:NN <cs1> <cs2>
\cs_if_eq:NNTF * \cs_if_eq:NNTF <cs1> <cs2> {\true code} {\false code}
```

Compares the definition of two *<control sequences>* and is logically true if they are the same, *i.e.* if they have exactly the same definition when examined with `\cs_show:N`.

```
\cs_if_exist_p:N * \cs_if_exist_p:N <control sequence>
\cs_if_exist_p:c * \cs_if_exist:NNTF <control sequence> {\true code} {\false code}
\cs_if_exist:NTF * Tests whether the <control sequence> is currently defined (whether as a function or another control sequence type). Any definition of <control sequence> other than \relax evaluates as true.
\cs_if_exist:cTF * 
```

```
\cs_if_free_p:N * \cs_if_free_p:N <control sequence>
\cs_if_free_p:c * \cs_if_free:NNTF <control sequence> {\true code} {\false code}
\cs_if_free:NTF * Tests whether the <control sequence> is currently free to be defined. This test is false
\cs_if_free:cTF * if the <control sequence> currently exists (as defined by \cs_if_exist:NNTF).
```

4.6.2 Primitive conditionals

The ε-T_EX engine itself provides many different conditionals. Some expand whatever comes after them and others don't. Hence the names for these underlying functions often contains a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_int_compare:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We prefix primitive conditionals with `\if_`, except for `\if:w`.

```
\if_true: * \if_true: <true code> \else: <false code> \fi:
\if_false: * \if_false: <true code> \else: <false code> \fi:
\else: * \reverse_if:N <primitive conditional>
\fi: * \if_true: always executes <true code>, while \if_false: always executes <false code>. \reverse_if:N reverses any two-way primitive conditional. \else: and \fi: delimit the branches of the conditional. The function \or: is documented in l3int and used in case switches.
```

T_EXhackers note: `\if_true:` and `\if_false:` are equivalent to their corresponding T_EX primitive conditionals `\iftrue` and `\iffalse`; `\else:` and `\fi:` are the T_EX primitives `\else` and `\fi`; `\reverse_if:N` is the ε-T_EX primitive `\unless`.

`\if_meaning:w` * `\if_meaning:w <arg1> <arg2> <true code> \else: <false code> \fi:`

`\if_meaning:w` executes `<true code>` when `<arg1>` and `<arg2>` are the same, otherwise it executes `<false code>`. `<arg1>` and `<arg2>` could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

TeXhackers note: This is the TeX primitive `\ifx`.

`\if:w` * `\if:w <token(s)> <true code> \else: <false code> \fi:`

`\if_charcode:w` * `\if_catcode:w <token(s)> <true code> \else: <false code> \fi:`

`\if_catcode:w` * `\if_charcode:w` is an alternative name for `\if:w`. These conditionals expand `<token(s)>` until two unexpandable tokens `<token1>` and `<token2>` are found; any further tokens up to the next unbalanced `\else:` are the true branch, ending with `<true code>`. It is executed if the condition is fulfilled, otherwise `<false code>` is executed. You can omit `\else:` when just in front of `\fi:` and you can nest `\if... \else:... \fi:` constructs inside the true branch or the `<false code>`. With `\exp_not:N`, you can prevent the expansion of a token.

`\if_catcode:w` tests if `<token1>` and `<token2>` have the same category code whereas `\if:w` and `\if_charcode:w` test if they have the same character code.

TeXhackers note: `\if:w` and `\if_charcode:w` are both the TeX primitive `\if`. `\if_catcode:w` is the TeX primitive `\ifcat`.

`\if_cs_exist:N` * `\if_cs_exist:N <cs> <true code> \else: <false code> \fi:`

`\if_cs_exist:w` * `\if_cs_exist:w <tokens> \cs_end: <true code> \else: <false code> \fi:`

Check if `<cs>` appears in the hash table or if the control sequence that can be formed from `<tokens>` appears in the hash table. The latter function does not turn the control sequence in question into `\scan_stop:!` This can be useful when dealing with control sequences which cannot be entered as a single token.

TeXhackers note: These are the TeX primitives `\ifdefined` and `\ifcsname`.

`\if_mode_horizontal:` * `\if_mode_horizontal: <true code> \else: <false code> \fi:`

`\if_mode_vertical:` * Execute `<true code>` if currently in horizontal mode, otherwise execute `<false code>`.

`\if_mode_math:` * Similar for the other functions.

`\if_mode_inner:` *

TeXhackers note: These are the TeX primitives `\ifhmode`, `\ifvmode`, `\ifmmode`, and `\ifinner`.

4.7 Starting a paragraph

`\mode_leave_vertical:` `\mode_leave_vertical:`

New: 2017-07-04

Ensures that `TEX` is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width `\parindent`, followed by the `\everypar` token list.

T_EXhackers note: This results in the contents of the `\everypar` token register being inserted, after `\mode_leave_vertical:` is complete. Notice that in contrast to the L^AT_EX 2_ε `\leavevmode` approach, no box is used by the method implemented here.

4.8 Debugging support

`\debug_on:n` `\debug_on:n { <comma-separated list> }`

`\debug_off:n` `\debug_off:n { <comma-separated list> }`

New: 2017-07-16

Updated: 2023-05-23

Turn on and off within a group various debugging code, some of which is also available as `expl3` load-time options. The items that can be used in the `<list>` are

- **check-declarations** that checks all `expl3` variables used were previously declared and that local/global variables (based on their name or on their first assignment) are only locally/globally assigned;
- **check-expressions** that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- **deprecation** that makes deprecated commands produce errors;
- **log-functions** that logs function definitions and variable declarations;
- **all** that does all of the above.

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call `\debug_on:n`, and load the code that one is interested in testing.

`\debug_suspend:` `\debug_suspend: ... \debug_resume:`

`\debug_resume:`

New: 2017-11-28

Suppress (locally) errors and logging from `debug` commands, except for the **deprecation** errors. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance `l3cctab` and `l3coffins`.

Chapter 5

The `\l3expan` module Argument expansion

This module provides generic methods for expanding `TEX` arguments in a systematic manner. The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the `LATEX3` kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

5.1 Defining new variants

The definition of variant forms for base functions may be necessary when writing new functions or when applying a kernel function in a situation that we haven't thought of before.

Internally preprocessing of arguments is done with functions of the form `\exp_....`. They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying `\exp_args:NNo` expands the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:No` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  { \l_tmpa_tl }
```

In other words, the first argument to `\exp_args:NNo` is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate `\exp_` function followed by the desired base function, *e.g.*

```
\cs_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of `\seq_gpush:No`

```
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is safe as the `\cs_generate_variant:Nn` function will only create new definitions if there is not already one available. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function `\cs_generate_variant:Nn`, described next.

5.2 Methods for defining variants

We recall the set of available argument specifiers.

- `N` is used for single-token arguments while `c` constructs a control sequence from its name and passes it to a parent function as an `N`-type argument.
- Many argument types extract or expand some tokens and provide it as an `n`-type argument, namely a braced multiple-token argument: `V` extracts the value of a variable, `v` extracts the value from the name of a variable, `n` uses the argument as it is, `o` expands once, `f` expands fully the front of the token list, `e` and `x` expand fully all tokens (differences are explained later).
- A few odd argument types remain: `T` and `F` for conditional processing, otherwise identical to `n`-type arguments, `p` for the parameter text in definitions, `w` for arguments with a specific syntax, and `D` to denote primitives that should not be used directly.

`\cs_generate_variant:Nn` `\cs_generate_variant:Nn` \langle *parent control sequence* \rangle $\{$ \langle *variant argument specifiers* \rangle $\}$

`\cs_generate_variant:cn`

Updated: 2017-11-28

This function is used to define argument-specifier variants of the \langle *parent control sequence* \rangle for L^AT_EX3 code-level macros. The \langle *parent control sequence* \rangle is first separated into the \langle *base name* \rangle and \langle *original argument specifier* \rangle . The comma-separated list of \langle *variant argument specifiers* \rangle is then used to define variants of the \langle *original argument specifier* \rangle if these are not already defined; entries which correspond to existing functions are silently ignored. For each \langle *variant* \rangle given, a function is created that expands its arguments as detailed and passes them to the \langle *parent control sequence* \rangle . So for example

```
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
```

creates a new function `\foo:cn` which expands its first argument into a control sequence name and passes the result to `\foo:Nn`. Similarly

```
\cs_generate_variant:Nn \foo:Nn { NV , cV }
```

generates the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function should only be applied if the \langle *parent control sequence* \rangle is already defined. (This is only enforced if debugging support `check-declarations` is enabled.) If the \langle *parent control sequence* \rangle is protected or if the \langle *variant* \rangle involves any `x` argument, then the \langle *variant control sequence* \rangle is also protected. The \langle *variant* \rangle is created globally, as is any `\exp_args:N` \langle *variant* \rangle function needed to carry out the expansion. There is no need to re-apply `\cs_generate_variant:Nn` after changing the definition of the parent function: the variant will always use the current definition of the parent. Providing variants repeatedly is safe as `\cs_generate_variant:Nn` will only create new definitions if there is not already one available.

Only `n` and `N` arguments can be changed to other types. The only allowed changes are

- `c` variant of an `N` parent;
- `o`, `V`, `v`, `f`, `e`, or `x` variant of an `n` parent;
- `N`, `n`, `T`, `F`, or `p` argument unchanged.

This means the \langle *parent* \rangle of a \langle *variant* \rangle form is always unambiguous, even in cases where both an `n`-type parent and an `N`-type parent exist, such as for `\tl_count:n` and `\tl_count:N`.

When creating variants for conditional functions, `\prg_generate_conditional_variant:Nnn` provides a convenient way of handling the related function set.

For backward compatibility it is currently possible to make `n`, `o`, `V`, `v`, `f`, `e`, or `x`-type variants of an `N`-type argument or `N` or `c`-type variants of an `n`-type argument. Both are deprecated. The first because passing more than one token to an `N`-type argument will typically break the parent function's code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a `V`-type or `v`-type variant instead of `c`-type. In those cases, using the lower-level `\exp_args:No` or `\exp_args:Nc` functions explicitly is preferred to defining confusing variants.

`\exp_args_generate:n` `\exp_args_generate:n {⟨variant argument specifiers⟩}`

New: 2018-04-04 Defines `\exp_args:N⟨variant⟩` functions for each `⟨variant⟩` given in the comma list
Updated: 2019-02-08 `{⟨variant argument specifiers⟩}`. Each `⟨variant⟩` should consist of the letters N, c, n, V, v, o, f, e, x, p and the resulting function is protected if the letter x appears in the `⟨variant⟩`. This is only useful for cases where `\cs_generate_variant:Nn` is not applicable.

5.3 Introducing the variants

The `V` type returns the value of a register, which can be one of `tl`, `clist`, `int`, `skip`, `dim`, `muskip`, or built-in `TEX` registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a `V` specifier should be used. For those referred to by `(cs)name`, the `v` specifier is available for the same purpose. Only when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with `o` specifiers be employed.

The `e` type expands all tokens fully, starting from the first. More precisely the expansion is identical to that of `TEX`'s `\message` (in particular `#` needs not be doubled). It relies on the primitive `\expanded` hence is fast.

The `x` type expands all tokens fully, starting from the first. In contrast to `e`, all macro parameter characters `#` must be doubled, and omitting this leads to low-level errors. In addition this type of expansion is not expandable, namely functions that have `x` in their signature do not themselves expand when appearing inside `e` or `x` expansion.

The `f` type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then `x`-expansion cannot be used, and `f`-expansion provides an alternative that expands the front of the token list as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression `3 + 4` and pass the result `7` as an argument to an expandable function `\example:n`. For this, one should define a variant using `\cs_generate_variant:Nn \example:n { f }`, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that `x`-expansion would also expand `\int_eval:n` fully to its result `7`, but the variant `\example:x` cannot be expandable. Note also that `o`-expansion would not expand `\int_eval:n` fully to its result since that function requires several expansions. Besides the fact that `x`-expansion is protected rather than expandable, another difference between `f`-expansion and `x`-expansion is that `f`-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while `x`-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

results in the call

```
\example:n { 3 , \int_eval:n { 3 + 4 } }
```

while using `\example:x` or `\example:e` instead results in

```
\example:n { 3 , 7 }
```

at the cost of being protected for x-type. If you use f type expansion in conditional processing then you should stick to using TF type functions only as the expansion does not finish any `\if... \fi`: itself!

It is important to note that both f- and o-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, o-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

```
\exp_after:wN <base function> \exp_after:wN { <argument> }
```

At the same time, f-type expansion stops at the *first* non-expandable token. This means for example that both

```
\tl_set:No \l_tmpa_tl { { \g_tmpb_tl } }
```

and

```
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
```

leave `\g_tmpb_tl` unchanged: { is the first token in the argument and is non-expandable.

It is usually best to keep the following in mind when using variant forms.

- Variants with x-type arguments (that are fully expanded before being passed to the n-type base function) are never expandable even when the base function is. Such variants cannot work correctly in arguments that are themselves subject to expansion. Consider using f or e expansion.
- In contrast, e expansion (full expansion, almost like x except for the treatment of #) does not prevent variants from being expandable (if the base function is).
- Finally f expansion only expands the front of the token list, stopping at the first non-expandable token. This may fail to fully expand the argument.

When speed is essential (for functions that do very little work and whose variants are used numerous times in a document) the following considerations apply because the speed of internal functions that expand the arguments of a base function depend on what needs doing with each argument and where this happens in the list of arguments:

- for fastest processing any c-type arguments should come first followed by all other modified arguments;
- unchanged N-type args that appear before modified ones have a small performance hit;
- unchanged n-type args that appear before modified ones have a relative larger performance hit.

5.4 Manipulating the first argument

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.

<code>\exp_args:Nc</code> *	<code>\exp_args:Nc</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
<code>\exp_args:cc</code> *	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged. The <code>:cc</code> variant constructs the $\langle function \rangle$ name in the same manner as described for the $\langle tokens \rangle$.
<code>\exp_args:No</code> *	<code>\exp_args:No</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$...
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded once, and the result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.
<code>\exp_args:NV</code> *	<code>\exp_args:NV</code> $\langle function \rangle$ $\langle variable \rangle$
	This function absorbs two arguments (the names of the $\langle function \rangle$ and the $\langle variable \rangle$). The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.
<code>\exp_args:Nv</code> *	<code>\exp_args:Nv</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. This control sequence should be the name of a $\langle variable \rangle$. The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.
<code>\exp_args:Ne</code> *	<code>\exp_args:Ne</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
<small>New: 2018-05-15</small>	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$. The result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.
<code>\exp_args:Nf</code> *	<code>\exp_args:Nf</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are fully expanded until the first non-expandable token is found (if that is a space it is removed), and the result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.
<code>\exp_args:Nx</code>	<code>\exp_args:Nx</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$. The result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

5.5 Manipulating two arguments

```
\exp_args:Nnc * \exp_args:Nnc <token_1> <token_2> {\tokens}
```

```
\exp_args:Nno * These optimized functions absorb three arguments and expand the second and third as  
\exp_args:NNV * detailed by their argument specifier. The first argument of the function is then the next  
\exp_args:NNv * item on the input stream, followed by the expansion of the second and third arguments.  
\exp_args:NNe *  
\exp_args:Nnf *  
\exp_args:Ncc *  
\exp_args:Nco *  
\exp_args:NcV *  
\exp_args:Ncv *  
\exp_args:Ncf *  
\exp_args:NVV *
```

Updated: 2018-05-15

```
\exp_args:Nnc * \exp_args:Noo <token> {\tokens_1} {\tokens_2}
```

```
\exp_args:Nno * These functions absorb three arguments and expand the second and third as detailed by  
\exp_args:NnV * their argument specifier. The first argument of the function is then the next item on the  
\exp_args:Nnv * input stream, followed by the expansion of the second and third arguments.  
\exp_args:Nne *  
\exp_args:Nnf *  
\exp_args:Noc *  
\exp_args:Noo *  
\exp_args:Nof *  
\exp_args:NVo *  
\exp_args:Nfo *  
\exp_args:Nff *  
\exp_args:Nee *
```

Updated: 2018-05-15

```
\exp_args:NNx \exp_args:NNx <token_1> <token_2> {\tokens}
```

```
\exp_args:Ncx * These functions absorb three arguments and expand the second and third as detailed by  
\exp_args:NNx * their argument specifier. The first argument of the function is then the next item on  
\exp_args:Nox * the input stream, followed by the expansion of the second and third arguments. These  
\exp_args:Nxo * functions are not expandable due to their x-type argument.  
\exp_args:Nxx *
```

5.6 Manipulating three arguments

```
\exp_args:NNNo * \exp_args:NNNo <token_1> <token_2> <token_3> {\tokens}
```

```
\exp_args:NNNV * These optimized functions absorb four arguments and expand the second, third and  
\exp_args:NNNv * fourth as detailed by their argument specifier. The first argument of the function is then  
\exp_args:NNNe * the next item on the input stream, followed by the expansion of the second argument,  
\exp_args:Nccc * etc.  
\exp_args:NcNc *  
\exp_args:NcNo *  
\exp_args:Ncco *
```

`\exp_args:NNcf` * `\exp_args:NNoo` $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle token_3 \rangle\}$ $\{\langle tokens \rangle\}$
`\exp_args:NNno` * These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*
`\exp_args:NNnV` *
`\exp_args:NNoo` *
`\exp_args:NNVV` *
`\exp_args:Ncno` *
`\exp_args:NcnV` *
`\exp_args:Ncoo` *
`\exp_args:NcVV` *
`\exp_args:Nnnc` *
`\exp_args:Nnno` *
`\exp_args:Nnnf` *
`\exp_args:Nnff` *
`\exp_args:Nooo` *
`\exp_args:Noof` *
`\exp_args:Nffo` *
`\exp_args:Neee` *

`\exp_args:NNNx` `\exp_args:NNnx` $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle tokens_1 \rangle\}$ $\{\langle tokens_2 \rangle\}$
`\exp_args:NNnx` These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*
`\exp_args:NNox`
`\exp_args:Nccx`
`\exp_args:Ncnx`
`\exp_args:Nnnx`
`\exp_args:Nnox`
`\exp_args:Noox`

New: 2015-08-12

5.7 Unbraced expansion

```

\exp_last_unbraced:No * \exp_last_unbraced:Nno <token> {<tokens1>} {<tokens2>}
\exp_last_unbraced:NV *
\exp_last_unbraced:Nv * These functions absorb the number of arguments given by their specification, carry out
\exp_last_unbraced:Ne * the expansion indicated and leave the results in the input stream, with the last argument
\exp_last_unbraced:Nf * not surrounded by the usual braces. Of these, the :Nno, :Noo, :Nfo and :NnNo variants
\exp_last_unbraced:NNo * need slower processing.
\exp_last_unbraced:NNV *
\exp_last_unbraced:NNf * TeXhackers note: As an optimization, the last argument is unbraced by some of those
\exp_last_unbraced:Nco * functions before expansion. This can cause problems if the argument is empty: for instance,
\exp_last_unbraced:NcV * \exp_last_unbraced:Nf \foo_bar:w { } \q_stop leads to an infinite loop, as the quark is f-
\exp_last_unbraced:Nno * expanded.
\exp_last_unbraced:Nnf *
\exp_last_unbraced:Noo *
\exp_last_unbraced:Nfo *
\exp_last_unbraced:NNNo *
\exp_last_unbraced:NNNV *
\exp_last_unbraced:NNNf *
\exp_last_unbraced:NnNo *
\exp_last_unbraced:NNNNo *
\exp_last_unbraced:NNNNf *

```

Updated: 2018-05-15

```

\exp_last_unbraced:Nx \exp_last_unbraced:Nx <function> {<tokens>}

```

This function fully expands the `<tokens>` and leaves the result in the input stream after reinsertion of the `<function>`. This function is not expandable.

```

\exp_last_two_unbraced:Noo * \exp_last_two_unbraced:Noo <token> {<tokens1>} {<tokens2>}

```

This function absorbs three arguments and expands the second and third once. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments, which are not wrapped in braces. This function needs special (slower) processing.

```

\exp_after:wN * \exp_after:wN <token1> <token2>

```

Carries out a single expansion of `<token2>` (which may consume arguments) prior to the expansion of `<token1>`. If `<token2>` has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that `<token1>` may be *any* single token, including group-opening and -closing tokens (`{` or `}` assuming normal TeX category codes). Unless specifically required this should be avoided: expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_<variant>` function.

TeXhackers note: This is the TeX primitive `\expandafter`.

5.8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expandable' since they themselves disappear after the expansion has completed.

`\exp_not:N` * `\exp_not:N` $\langle token \rangle$

Prevents expansion of the $\langle token \rangle$ in a context where it would otherwise be expanded, for example an **e**-type or **x**-type argument or the first token in an **o**-type or **f**-type argument.

TeXhackers note: This is the TeX primitive `\noexpand`. It only prevents expansion. At the beginning of an **f**-type argument, a space $\langle token \rangle$ is removed even if it appears as `\exp_not:N \c_space_token`. In an **e**-expanding definition (`\cs_new:Npe`), a macro parameter introduces an argument even if it appears as `\exp_not:N # 1`. This differs from `\exp_not:n`.

`\exp_not:c` * `\exp_not:c` $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited using `\exp_not:N`.

`\exp_not:n` * `\exp_not:n` $\{\langle tokens \rangle\}$

Prevents expansion of the $\langle tokens \rangle$ in an **e**-type or **x**-type argument. In all other cases the $\langle tokens \rangle$ continue to be expanded, for example in the input stream or in other types of arguments such as **c**, **f**, **v**. The argument of `\exp_not:n` *must* be surrounded by braces.

TeXhackers note: This is the ϵ -TeX primitive `\unexpanded`. In an **e**-expanding definition (`\cs_new:Npe`), `\exp_not:n {#1}` is equivalent to `##1` rather than to `#1`, namely it inserts the two characters `#` and `1`, and `\exp_not:n {#}` is equivalent to `#`, namely it inserts the character `#`.

`\exp_not:o` * `\exp_not:o` $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ once, then prevents any further expansion in **e**-type or **x**-type arguments using `\exp_not:n`.

`\exp_not:V` * `\exp_not:V` $\langle variable \rangle$

Recovers the content of the $\langle variable \rangle$, then prevents expansion of this material in **e**-type or **x**-type arguments using `\exp_not:n`.

`\exp_not:v` * `\exp_not:v` $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ until only characters remains, and then converts this into a control sequence which should be a $\langle variable \rangle$ name. The content of the $\langle variable \rangle$ is recovered, and further expansion in **e**-type or **x**-type arguments is prevented using `\exp_not:n`.

`\exp_not:e` * `\exp_not:e {<tokens>}`

Expands `<tokens>` exhaustively, then protects the result of the expansion (including any tokens which were not expanded) from further expansion in `e`-type or `x`-type arguments using `\exp_not:n`. This is very rarely useful but is provided for consistency.

`\exp_not:f` * `\exp_not:f {<tokens>}`

Expands `<tokens>` fully until the first unexpandable token is found (if it is a space it is removed). Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion in `e`-type or `x`-type arguments using `\exp_not:n`.

`\exp_stop_f:` * `\foo_bar:f { <tokens> \exp_stop_f: <more tokens> }`

Updated: 2011-06-03

This function terminates an `f`-type expansion. Thus if a function `\foo_bar:f` starts an `f`-type expansion and all of `<tokens>` are expandable `\exp_stop_f:` terminates the expansion of tokens even if `<more tokens>` are also expandable. The function itself is an implicit space token. Inside an `e`-type or `x`-type expansion, it retains its form, but when typeset it produces the underlying space (`\`).

5.9 Controlled expansion

The `expl3` language makes all efforts to hide the complexity of `TEX` expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to calling the “base” functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down `TEX` is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. These commands are used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of `<expandable-tokens>` as that will break badly if unexpandable tokens are encountered in that place!

`\exp:w` \star `\exp:w` \langle *expandable tokens* \rangle `\exp_end:`
`\exp_end:` \star Expands \langle *expandable-tokens* \rangle until reaching `\exp_end:` at which point expansion stops. The full expansion of \langle *expandable tokens* \rangle has to be empty. If any token in \langle *expandable tokens* \rangle or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end:` will be misinterpreted later on.³

New: 2015-08-23

In typical use cases the `\exp_end:` is hidden somewhere in the replacement text of \langle *expandable-tokens* \rangle rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

```
\exp:w \@@_case:NnTF #1 {#2} { } { }
```

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

TeXhackers note: The current implementation uses `\romannumeral` hence ignores space tokens and explicit signs + and - in the expansion of the \langle *expandable tokens* \rangle , but this should not be relied upon.

`\exp:w` \star `\exp:w` \langle *expandable-tokens* \rangle `\exp_end_continue_f:w` \langle *further-tokens* \rangle
`\exp_end_continue_f:w` \star Expands \langle *expandable-tokens* \rangle until reaching `\exp_end_continue_f:w` at which point expansion continues as an f-type expansion expanding \langle *further-tokens* \rangle until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by `\exp_stop_f:`). As with all f-type expansions a space ending the expansion gets removed.

New: 2015-08-23

The full expansion of \langle *expandable-tokens* \rangle has to be empty. If any token in \langle *expandable-tokens* \rangle or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end_continue_f:w` will be misinterpreted later on.⁴

In typical use cases \langle *expandable-tokens* \rangle contains no tokens at all, e.g., you will see code such as

```
\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }
```

where the `\exp_after:wN` triggers an f-expansion of the tokens in #2. For technical reasons this has to happen using two tokens (if they would be hidden inside another command `\exp_after:wN` would only expand the command but not trigger any additional f-expansion).

You might wonder why there are two different approaches available, after all the effect of

```
\exp:w \langle expandable-tokens \rangle \exp_end:
```

can be alternatively achieved through an f-type expansion by using `\exp_stop_f:`, i.e.

```
\exp:w \exp_end_continue_f:w \langle expandable-tokens \rangle \exp_stop_f:
```

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

³Due to the implementation you might get the character in position 0 in the current font (typically “”) in the output without any error message!

<code>\exp:w</code>	*	<code>\exp:w</code> <i>(expandable-tokens)</i> <code>\exp_end_continue_f:nw</code> <i>(further-tokens)</i>
<code>\exp_end_continue_f:nw</code>	*	The difference to <code>\exp_end_continue_f:w</code> is that we first we pick up an argument which is then returned to the input stream. If <i>(further-tokens)</i> starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the f-type expansion.

New: 2015-08-23

5.10 Internal functions

<code>\::n</code>	<code>\cs_new:Npn</code> <code>\exp_args:Ncof</code> { <code>\::c</code> <code>\::o</code> <code>\::f</code> <code>\:::</code> }
<code>\::N</code>	Internal forms for the base expansion types. These names do <i>not</i> conform to the general
<code>\::P</code>	\LaTeX 3 approach as this makes them more readily visible in the log and so forth. They
<code>\::c</code>	should not be used outside this module.
<code>\::o</code>	
<code>\::e</code>	
<code>\::f</code>	
<code>\::x</code>	
<code>\::v</code>	
<code>\::V</code>	
<code>\:::</code>	

<code>\::o_unbraced</code>	<code>\cs_new:Npn</code> <code>\exp_last_unbraced:Nno</code> { <code>\::n</code> <code>\::o_unbraced</code> <code>\:::</code> }
<code>\::e_unbraced</code>	Internal forms for the expansion types which leave the terminal argument unbraced.
<code>\::f_unbraced</code>	These names do <i>not</i> conform to the general \LaTeX 3 approach as this makes them more
<code>\::x_unbraced</code>	readily visible in the log and so forth. They should not be used outside this module.
<code>\::v_unbraced</code>	
<code>\::V_unbraced</code>	

⁴In this particular case you may get a character into the output as well as an error message.

Chapter 6

The `l3sort` module

Sorting functions

6.1 Controlling sorting

L^AT_EX3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
{
  \int_compare:nNnTF { #1 } > { #2 }
  { \sort_return_swapped: }
  { \sort_return_same: }
}
```

results in `\l_foo_clist` holding the values `{ -2 , 01 , +1 , 3 , 5 }` sorted in non-decreasing order.

The code defining the comparison should call `\sort_return_swapped:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_return_same:` to indicate that the order of this pair of items should not be changed.

For instance, a *comparison code* consisting only of `\sort_return_same:` with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a *comparison code* consisting only of `\sort_return_swapped:` reverses the list (in a fairly inefficient way).

T_EXhackers note: The current implementation is limited to sorting approximately 20000 items (40000 in Lua_{T_EX}), depending on what other packages are loaded.

Internally, the code from `l3sort` stores items in `\toks` registers allocated locally. Thus, the *comparison code* should not call `\newtoks` or other commands that allocate new `\toks` registers. On the other hand, altering the value of a previously allocated `\toks` register is not a problem.

`\sort_return_same:` `\seq_sort:Nn` *<seq var>*
`\sort_return_swapped:` { ... `\sort_return_same:` or `\sort_return_swapped:` ... }

New: 2017-02-06 Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the `\sort_return_...` functions should be used by the code, according to the results of some tests on the items #1 and #2 to be compared.

Chapter 7

The l3tl-analysis module

Analysing token lists

This module provides functions that are particularly useful in the l3regex module for mapping through a token list one $\langle token \rangle$ at a time (including begin-group/end-group tokens). For $\backslash\text{tl_analysis_map_inline:Nn}$ or $\backslash\text{tl_analysis_map_inline:nn}$, the token list is given as an argument; the analogous function $\backslash\text{peek_analysis_map_inline:n}$ documented in l3token finds tokens in the input stream instead. In both cases the user provides $\langle inline\ code \rangle$ that receives three arguments for each $\langle token \rangle$:

- $\langle tokens \rangle$, which both o-expand and e/x-expand to the $\langle token \rangle$. The detailed form of $\langle tokens \rangle$ may change in later releases.
- $\langle char\ code \rangle$, a decimal representation of the character code of the $\langle token \rangle$, -1 if it is a control sequence.
- $\langle catcode \rangle$, a capital hexadecimal digit which denotes the category code of the $\langle token \rangle$ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing " $\langle catcode \rangle$ ".

In addition, there is a debugging function $\backslash\text{tl_analysis_show:n}$, very similar to the $\backslash\text{ShowTokens}$ macro from the ted package.

$\backslash\text{tl_analysis_show:N}$	$\backslash\text{tl_analysis_show:n}$ $\{\langle token\ list \rangle\}$
$\backslash\text{tl_analysis_show:n}$	$\backslash\text{tl_analysis_log:n}$ $\{\langle token\ list \rangle\}$
$\backslash\text{tl_analysis_log:N}$	Displays to the terminal (or log) the detailed decomposition of the $\langle token\ list \rangle$ into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.
$\backslash\text{tl_analysis_log:n}$	

New: 2021-05-11

$\backslash\text{tl_analysis_map_inline:nn}$	$\backslash\text{tl_analysis_map_inline:nn}$ $\{\langle token\ list \rangle\}$ $\{\langle inline\ function \rangle\}$
$\backslash\text{tl_analysis_map_inline:Nn}$	Applies the $\langle inline\ function \rangle$ to each individual $\langle token \rangle$ in the $\langle token\ list \rangle$. The $\langle inline\ function \rangle$ receives three arguments as explained above. As all other mappings the mapping is done at the current group level, <i>i.e.</i> any local assignments made by the $\langle inline\ function \rangle$ remain in effect after the loop.

New: 2018-04-09
Updated: 2022-03-26

Chapter 8

The `l3regex` module

Regular expressions in `TeX`

The `l3regex` module provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that `TeX` manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text “`This cat.`”, where the first occurrence of “`at`” was replaced by “`is`”. A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
```

The `\w` sequence represents any “word” character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB\{` and `\cE\}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_set:Nn`. For example,

```
\regex_new:N \l_foo_regex
\regex_set:Nn \l_foo_regex { \c{begin} \cB. (\c[~BE].*) \cE. }
```

stores in `\l_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[~BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses “capture” the result of `\c[~BE].*`, giving us access to the name of the environment when doing replacements.

8.1 Syntax of regular expressions

8.1.1 Regular expression examples

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word “Cat” capitalized in this way, but also matches the beginning of the word “Cattle”: use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among “a”, “b”, “c”; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).
- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).
- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.
- `_[^_]*_[^_]*` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `_.*?_[^_]*` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.
- `[\+\\-]?d+` matches an explicit integer with at most one sign.
- `[\+\\-_]*d+_*` matches an explicit integer with any number of `+` and `-` signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[\+\\-_]*(d+|d*\\.d+)_*` matches an explicit integer or decimal number; using `[.,]` instead of `\\.` would allow the comma as a decimal marker.
- `[\+\\-_]*(d+|d*\\.d+)_*(?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)_*` matches an explicit dimension with any unit that `TEX` knows, where `(?i)` means to treat lowercase and uppercase letters identically.
- `[\+\\-_]*(?i)nan|inf|(d+|d*\\.d+)(_*e[\\+\\-_]*d+)?_*` matches an explicit floating point number or the special values `nan` and `inf` (with signs and spaces allowed).
- `[\+\\-_]*(d+|cC)_*` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\\G.*?\\K` at the beginning of a regular expression matches and discards (due to `\\K`) everything between the end of the previous match (`\\G`) and what is matched by the rest of the regular expression; this is useful in `\\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[\+\\-_]*(d+|cC)_*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

8.1.2 Characters in regular expressions

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `*` matches a star character). Some escape sequences of the form backslash–letter also have a special meaning (for instance `\d` matches any digit). As a rule,

- every alphanumeric character (A–Z, a–z, 0–9) matches exactly itself, and should not be escaped, because `\A`, `\B`, ... have special meanings;
- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\(`, `\)`, `\?`, `\.`, `\^`);
- spaces should always be escaped (even in character classes);
- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into \TeX under normal category codes. For instance, `\abc%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{<regex>}` syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code `hh...`

`\xhh` Character with hex code `hh`.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

8.1.3 Characters classes

Character properties.

`.` A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^\^I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^\^I\^\^J\^\^L\^\^M]`.

`\v` Any vertical space character, equivalent to `[\^\^J\^\^K\^\^L\^\^M]`. Note that `\^\^K` is a vertical space, but not a space, for compatibility with Perl.

`\w` Any word character, *i.e.*, alphanumerics and underscore, equivalent to the explicit class `[A-Za-z0-9_]`.

`\D` Any token not matched by `\d`.

`\H` Any token not matched by `\h`.

`\N` Any token other than the `\n` character (hex 0A).

`\S` Any token not matched by `\s`.

`\V` Any token not matched by `\v`.

`\W` Any token not matched by `\w`.

Of those, `.`, `\D`, `\H`, `\N`, `\S`, `\V`, and `\W` match arbitrary control sequences.

Character classes match exactly one token in the subject.

`[...]` Positive character class. Matches any of the specified tokens.

`[^...]` Negative character class. Matches any token other than the specified characters.

`[x-y]` Within a character class, this denotes a range (can be used with escaped characters).

`[:<name>:]` Within a character class (one more set of brackets), this denotes the POSIX character class `<name>`, which can be `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, or `xdigit`.

`[:^<name>:]` Negative POSIX character class.

For instance, `[a-oq-z\cC.]` matches any lowercase latin letter except `p`, as well as control sequences (see below for a description of `\c`).

In character classes, only `[`, `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (`\d`, `\D`, *etc.*) is supported in character classes. If the first character is `^`, then the meaning of the character class is inverted; `^` appearing anywhere else in the range is not special. If the first character (possibly following a leading `^`) is `]` then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using `-`, for instance, `[\D 0-5]` and `[^6-9]` are equivalent.

8.1.4 Structure: alternatives, groups, repetitions

Quantifiers (repetition).

`?` 0 or 1, greedy.

`??` 0 or 1, lazy.

`*` 0 or more, greedy.

`*?` 0 or more, lazy.

`+` 1 or more, greedy.

+? 1 or more, lazy.

{*n*} Exactly *n*.

{*n*,} *n* or more, greedy.

{*n*,}? *n* or more, lazy.

{*n*,*m*} At least *n*, no more than *m*, greedy.

{*n*,*m*}? At least *n*, no more than *m*, lazy.

For greedy quantifiers the regex code will first investigate matches that involve as many repetitions as possible, while for lazy quantifiers it investigates matches with as few repetitions as possible first.

Alternation and capturing groups.

A|B|C Either one of A, B, or C, investigating A first.

(...) Capturing group.

(?:...) Non-capturing group.

(?|...) Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the “best” match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnNTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

8.1.5 Matching exact tokens

The `\c` escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- C for control sequences;
- B for begin-group tokens;
- E for end-group tokens;

- M for math shift;
- T for alignment tab tokens;
- P for macro parameter tokens;
- U for superscript tokens (up);
- D for subscript tokens (down);
- S for spaces;
- L for letters;
- O for others; and
- A for active characters.

The `\c` escape sequence is used as follows.

`\c{regex}` A control sequence whose csname matches the *regex*, anchored at the beginning and end, so that `\c{begin}` matches exactly `\begin`, and nothing else.

`\cX` Applies to the next object, which can be a character, escape character sequence such as `\x{0A}`, character class, or group, and forces this object to only match tokens with category X (any of CBEMTPUDSLOA). For instance, `\cL[A-Z\d]` matches uppercase letters and digits of category code letter, `\cC` matches any control sequence, and `\cO(abc)` matches `abc` where each character has category other.⁵

`\c[XYZ]` Applies to the next object, and forces it to only match tokens with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, `\c[LSO](..)` matches two tokens of category letter, space, or other.

`\c[^XYZ]` Applies to the next object and prevents it from matching any token with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, `\c[^O]\d` matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, `[\cO\d \c[L0][A-F]]` matches what T_EX considers as hexadecimal digits, namely digits with category other, or uppercase letters from A to F with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, `\cL(ab\cO*cd)` matches `ab*cd` where all characters are of category letter, except `*` which has category other.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters. Namely, `\u{var name}` matches the exact contents (both character codes and category codes) of the variable `\var name`, which are obtained by applying `\exp_not:v{var name}` at the time the regular expression is compiled. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\t1_to_str:v`. Quantifiers are supported.

The `\ur` escape sequence allows to insert the contents of a `regex` variable into a larger regular expression. For instance, `A\ur{1_tmpa_regex}D` matches the tokens A and

⁵This last example also captures “abc” as a regex group; to avoid this use a non-capturing group `\cO(?:abc)`.

D separated by something that matches the regular expression `\1_tmpa_regex`. This behaves as if a non-capturing group were surrounding `\1_tmpa_regex`, and any group contained in `\1_tmpa_regex` is converted to a non-capturing group. Quantifiers are supported.

For instance, if `\1_tmpa_regex` has value `B|C`, then `A\ur{\1_tmpa_regex}D` is equivalent to `A(?:B|C)D` (matching `ABD` or `ACD`) and not to `AB|CD` (matching `AB` or `CD`). To get the latter effect, it is simplest to use \TeX 's expansion machinery directly: if `\1_mymodule_BC_tl` contains `B|C` then the following two lines show the same result:

```
\regex_show:n { A \u{\1_mymodule_BC_tl} D }
\regex_show:n { A B | C D }
```

8.1.6 Miscellaneous

Anchors and simple assertions.

`\b` Word boundary: either the previous token is matched by `\w` and the next by `\W`, or the opposite. For this purpose, the ends of the token list are considered as `\W`.

`\B` Not a word boundary: between two `\w` tokens or two `\W` tokens (including the boundary).

`^` or `\A` Start of the subject token list.

`$`, `\Z` or `\z` End of the subject token list.

`\G` Start of the current match. This is only different from `^` in the case of multiple matches: for instance `\regex_count:nnN { \G a } { aaba } \1_tmpa_int` yields 2, but replacing `\G` by `^` would result in `\1_tmpa_int` holding the value 1.

The option `(?i)` makes the match case insensitive (treating `A-Z` and `a-z` as equivalent, with no support yet for Unicode case changing). This applies until the end of the group in which it appears, and can be reverted using `(?-i)`. For instance, in `(?i)(a(?-i)b|c)d`, the letters `a` and `d` are affected by the `i` option. Characters within ranges and classes are affected individually: `(?i)[\?-B]` is equivalent to `[\?@ABab]` (and differs from the much larger class `[\?-b]`), and `(?i)[^aeiou]` matches any character which is not a vowel. The `i` option has no effect on `\c{...}`, on `\u{...}`, on character properties, or on character classes, for instance it has no effect at all in `(?i)\u{\1_foo_tl}\d\d[[:lower:]]`.

8.2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- `\0` is the whole match;
- `\1` is the submatch that was matched by the first (capturing) group (...); similarly for `\2`, ..., `\9` and `\g{<number>}`;
- `_` inserts a space (spaces are ignored when not escaped);

- `\a, \e, \f, \n, \r, \t, \xhh, \x{hhh}` correspond to single characters as in regular expressions;
- `\c{<cs name>}` inserts a control sequence;
- `\c{<category>}<character>` (see below);
- `\u{<tl var name>}` inserts the contents of the `<tl var>` (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for \TeX , for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_tl { Hello,~world! }
\regex_replace_all:nnN { ([er]?l|o) . } { (\0--\1) } \l_my_tl
```

results in `\l_my_tl` holding `H(e1l--e1)(o,--o)w(or--o)(ld--l)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n -th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

By default, the category code of characters inserted by the replacement are determined by the prevailing category code regime at the time where the replacement is made, with two exceptions:

- space characters (with character code 32) inserted with `_` or `\x20` or `\x{20}` have category code 10 regardless of the prevailing category code regime;
- if the category code would be 0 (escape), 5 (newline), 9 (ignore), 14 (comment) or 15 (invalid), it is replaced by 12 (other) instead.

The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters “...” with category X , which must be one of `CBEMTPUDSLOA` as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{<text>}` Produces the control sequence with csname `<text>`. The `<text>` may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{<var name>}` allows to insert the contents of the variable with name `<var name>` directly into the replacement, giving an easier control of category codes. When nested in `\c{...}` and `\u{...}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{1_my_\0_tl} } \l_my_tl
```

results in `\l_my_tl` holding `first,\emph{second},first,first`.

Regex replacement is also a convenient way to produce token lists with arbitrary category codes. For instance

```
\tl_clear:N \l_tmpa_tl
\regex_replace_all:nnN { } { \cU\% \cA\~ } \l_tmpa_tl
```

results in `\l_tmpa_tl` containing the percent character with category code 7 (superscript) and an active tilde character.

8.3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the `l3regex` module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

`\regex_new:N` `\regex_new:N` \langle *regex var* \rangle

New: 2017-05-26 Creates a new \langle *regex var* \rangle or raises an error if the name is already taken. The declaration is global. The \langle *regex var* \rangle is initially such that it never matches.

`\regex_set:Nn` `\regex_set:Nn` \langle *regex var* \rangle $\{$ \langle *regex* \rangle $\}$

`\regex_gset:Nn` Stores a compiled version of the \langle *regular expression* \rangle in the \langle *regex var* \rangle . The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. For instance, this function can be used as

```
\regex_new:N \l_my_regex
\regex_set:Nn \l_my_regex { my\ (simple\ )? reg(ex|ular\ expression) }
```

`\regex_const:Nn` `\regex_const:Nn` \langle *regex var* \rangle $\{$ \langle *regex* \rangle $\}$

New: 2017-05-26 Creates a new constant \langle *regex var* \rangle or raises an error if the name is already taken. The value of the \langle *regex var* \rangle is set globally to the compiled version of the \langle *regular expression* \rangle .

`\regex_show:N` `\regex_show:n` $\{$ \langle *regex* \rangle $\}$

`\regex_show:n` `\regex_log:n` $\{$ \langle *regex* \rangle $\}$

`\regex_log:N` Displays in the terminal or writes in the log file (respectively) how `l3regex` interprets the \langle *regex* \rangle . For instance, `\regex_show:n {\A X|Y}` shows

New: 2021-04-26

Updated: 2021-04-29

```
+--branch
  anchor at start (\A)
  char code 88 (X)
+--branch
  char code 89 (Y)
```

indicating that the anchor `\A` only applies to the first branch: the second branch is not anchored to the beginning of the match.

8.4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a “standard” regular expression, while the later require a compiled expression as generated by `\regex_set:Nn`.

```
\regex_match:nnTF \regex_match:nnTF {<regex>} {<token list>} {<>true code>} {<>false code>}
\regex_match:nVTF
\regex_match:NnTF Tests whether the <regular expression> matches any part of the <token list>. For
\regex_match:NVTF instance,
```

```
New: 2017-05-26 \regex_match:nnTF { b [cde]* } { abecdcx } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
```

leaves TRUE then FALSE in the input stream.

```
\regex_count:nnN \regex_count:nnN {<regex>} {<token list>} <int var>
\regex_count:nVN Sets <int var> within the current TEX group level equal to the number of times <regular
\regex_count:NnN expression> appears in <token list>. The search starts by finding the left-most longest
\regex_count:NVN match, respecting greedy and lazy (non-greedy) operators. Then the search starts again
from the character following the last character of the previous match, until reaching the
end of the token list. Infinite loops are prevented in the case where the regular expression
can match an empty token list: then we count one match between each pair of characters.
For instance,
```

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcb } \l_foo_int
```

results in `\l_foo_int` taking the value 5.

```
\regex_match_case:nn \regex_match_case:nnTF
\regex_match_case:nnTF {
{<regex1>} {<code case1>}
{<regex2>} {<code case2>}
...
{<regexn>} {<code casen>}
} {<token list>}
{<>true code>} {<>false code>}
```

New: 2022-01-10

Determines which of the `<regular expressions>` matches at the earliest point in the `<token list>`, and leaves the corresponding `<codei>` followed by the `<>true code>` in the input stream. If several `<regex>` match starting at the same point, then the first one in the list is selected and the others are discarded. If none of the `<regex>` match, the `<>false code>` is left in the input stream. Each `<regex>` can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the `<token list>`, each of the `<regex>` is searched in turn. If one of them matches then the corresponding `<code>` is used and everything else is discarded, while if none of the `<regex>` match at a given position then the next starting position is attempted. If none of the `<regex>` match anywhere in the `<token list>` then nothing is left in the input stream. Note that this differs from nested `\regex_match:nnTF` statements since all `<regex>` are attempted at each position rather than attempting to match `<regex1>` at every position before moving on to `<regex2>`.

8.5 Submatch extraction

<code>\regex_extract_once:nnN</code>	<code>\regex_extract_once:nnN {<regex>} {<token list>} <seq var></code>
<code>\regex_extract_once:nVN</code>	<code>\regex_extract_once:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}</code>
<code>\regex_extract_once:nnNTF</code>	
<code>\regex_extract_once:nVNTF</code>	Finds the first match of the <i><regular expression></i> in the <i><token list></i> . If it exists, the match is stored as the first item of the <i><seq var></i> , and further items are the contents of capturing groups, in the order of their opening parenthesis. The <i><seq var></i> is assigned locally. If there is no match, the <i><seq var></i> is cleared. The testing versions insert the <i><true code></i> into the input stream if a match was found, and the <i><false code></i> otherwise.
<code>\regex_extract_once:NnN</code>	
<code>\regex_extract_once:NVN</code>	
<code>\regex_extract_once:NnNTF</code>	
<code>\regex_extract_once:NVNTF</code>	

New: 2017-05-26

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
{ true } { false }
```

Then the regular expression (anchored at the start with `\A` and at the end with `\Z`) must match the whole token list. The first capturing group, `(La)?`, matches `La`, and the second capturing group, `(!*)`, matches `!!!`. Thus, `\l_foo_seq` contains as a result the items `{LaTeX!!!}`, `{La}`, and `{!!!}`, and the `true` branch is left in the input stream. Note that the n -th item of `\l_foo_seq`, as obtained using `\seq_item:Nn`, correspond to the submatch numbered $(n - 1)$ in functions such as `\regex_replace_once:nnN`.

<code>\regex_extract_all:nnN</code>	<code>\regex_extract_all:nnN {<regex>} {<token list>} <seq var></code>
<code>\regex_extract_all:nVN</code>	<code>\regex_extract_all:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}</code>
<code>\regex_extract_all:nnNTF</code>	
<code>\regex_extract_all:nVNTF</code>	Finds all matches of the <i><regular expression></i> in the <i><token list></i> , and stores all the submatch information in a single sequence (concatenating the results of multiple <code>\regex_extract_once:nnN</code> calls). The <i><seq var></i> is assigned locally. If there is no match, the <i><seq var></i> is cleared. The testing versions insert the <i><true code></i> into the input stream if a match was found, and the <i><false code></i> otherwise. For instance, assume that you type
<code>\regex_extract_all:NnN</code>	
<code>\regex_extract_all:NVN</code>	
<code>\regex_extract_all:NnNTF</code>	
<code>\regex_extract_all:NVNTF</code>	

New: 2017-05-26

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
{ true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items `{Hello}` and `{world}`, and the `true` branch is left in the input stream.

<code>\regex_split:nnN</code>	<code>\regex_split:nnN {<regular expression>} {<token list>} <seq var></code>
<code>\regex_split:nVN</code>	<code>\regex_split:nnNTF {<regular expression>} {<token list>} <seq var> {<true code>}</code>
<code>\regex_split:nnNTF</code>	<code>{<false code>}</code>
<code>\regex_split:nVNTF</code>	Splits the <code><token list></code> into a sequence of parts, delimited by matches of the <code><regular expression></code> . If the <code><regular expression></code> has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to <code><seq var></code> is local. If no match is found the resulting <code><seq var></code> has the <code><token list></code> as its sole item. If the <code><regular expression></code> matches the empty token list, then the <code><token list></code> is split into single tokens. The testing versions insert the <code><true code></code> into the input stream if a match was found, and the <code><false code></code> otherwise. For example, after
<code>\regex_split:NnN</code>	
<code>\regex_split:NVN</code>	
<code>\regex_split:NnNTF</code>	
<code>\regex_split:NVNTF</code>	

New: 2017-05-26

```

\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
{ true } { false }

```

the sequence `\l_path_seq` contains the items `{the}`, `{path}`, `{for}`, `{this}`, and `{file.tex}`, and the `true` branch is left in the input stream.

8.6 Replacement

<code>\regex_replace_once:nnN</code>	<code>\regex_replace_once:nnN {<regular expression>} {<replacement>} <t1 var></code>
<code>\regex_replace_once:nVN</code>	<code>\regex_replace_once:nnNTF {<regular expression>} {<replacement>} <t1 var> {<true code>}</code>
<code>\regex_replace_once:nnNTF</code>	<code>{<false code>}</code>
<code>\regex_replace_once:nVNTF</code>	Searches for the <code><regular expression></code> in the contents of the <code><t1 var></code> and replaces the first match with the <code><replacement></code> . In the <code><replacement></code> , <code>\0</code> represents the full match, <code>\1</code> represent the contents of the first capturing group, <code>\2</code> of the second, <i>etc.</i> The result is assigned locally to <code><t1 var></code> .
<code>\regex_replace_once:NnN</code>	
<code>\regex_replace_once:NVN</code>	
<code>\regex_replace_once:NnNTF</code>	
<code>\regex_replace_once:NVNTF</code>	

New: 2017-05-26

<code>\regex_replace_all:nnN</code>	<code>\regex_replace_all:nnN {<regular expression>} {<replacement>} <t1 var></code>
<code>\regex_replace_all:nVN</code>	<code>\regex_replace_all:nnNTF {<regular expression>} {<replacement>} <t1 var> {<true code>}</code>
<code>\regex_replace_all:nnNTF</code>	<code>{<false code>}</code>
<code>\regex_replace_all:nVNTF</code>	Replaces all occurrences of the <code><regular expression></code> in the contents of the <code><t1 var></code> by the <code><replacement></code> , where <code>\0</code> represents the full match, <code>\1</code> represent the contents of the first capturing group, <code>\2</code> of the second, <i>etc.</i> Every match is treated independently, and matches cannot overlap. The result is assigned locally to <code><t1 var></code> .
<code>\regex_replace_all:NnN</code>	
<code>\regex_replace_all:NVN</code>	
<code>\regex_replace_all:NnNTF</code>	
<code>\regex_replace_all:NVNTF</code>	

New: 2017-05-26

```

\regex_replace_case_once:nN \regex_replace_case_once:nNTF
\regex_replace_case_once:nNTF {
    New: 2022-01-10    {\langle regex_1 \rangle} {\langle replacement_1 \rangle}
                      {\langle regex_2 \rangle} {\langle replacement_2 \rangle}
                      ...
                      {\langle regex_n \rangle} {\langle replacement_n \rangle}
} \langle tl var \rangle
{\langle true code \rangle} {\langle false code \rangle}

```

Replaces the earliest match of the regular expression $(?|\langle regex_1 \rangle|\dots|\langle regex_n \rangle)$ in the $\langle token list variable \rangle$ by the $\langle replacement \rangle$ corresponding to which $\langle regex_i \rangle$ matched, then leaves the $\langle true code \rangle$ in the input stream. If none of the $\langle regex \rangle$ match, then the $\langle tl var \rangle$ is not modified, and the $\langle false code \rangle$ is left in the input stream. Each $\langle regex \rangle$ can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the $\langle token list \rangle$, each of the $\langle regex \rangle$ is searched in turn. If one of them matches then it is replaced by the corresponding $\langle replacement \rangle$ as described for `\regex_replace_once:nN`. This is equivalent to checking with `\regex_match_case:nn` which $\langle regex \rangle$ matches, then performing the replacement with `\regex_replace_once:nnN`.

```

\regex_replace_case_all:nN \regex_replace_case_all:nNTF
\regex_replace_case_all:nNTF {
    New: 2022-01-10    {\langle regex_1 \rangle} {\langle replacement_1 \rangle}
                      {\langle regex_2 \rangle} {\langle replacement_2 \rangle}
                      ...
                      {\langle regex_n \rangle} {\langle replacement_n \rangle}
} \langle tl var \rangle
{\langle true code \rangle} {\langle false code \rangle}

```

Replaces all occurrences of all $\langle regex \rangle$ in the $\langle token list \rangle$ by the corresponding $\langle replacement \rangle$. Every match is treated independently, and matches cannot overlap. The result is assigned locally to $\langle tl var \rangle$, and the $\langle true code \rangle$ or $\langle false code \rangle$ is left in the input stream depending on whether any replacement was made or not.

In detail, for each starting position in the $\langle token list \rangle$, each of the $\langle regex \rangle$ is searched in turn. If one of them matches then it is replaced by the corresponding $\langle replacement \rangle$, and the search resumes at the position that follows this match (and replacement). For instance

```

\tl_set:Nn \l_tmpa_tl { Hello,~world! }
\regex_replace_case_all:nN
{
  { [A-Za-z]+ } { ‘\0’ }
  { \b } { --- }
  { . } { [\0] }
} \l_tmpa_tl

```

results in \l_tmpa_tl having the contents ‘Hello’---[,][_L]‘world’---[!]. Note in particular that the word-boundary assertion `\b` did not match at the start of words because the case `[A-Za-z]+` matched at these positions. To change this, one could simply swap the order of the two cases in the argument of `\regex_replace_case_all:nN`.

8.7 Scratch regular expressions

`\l_tmpa_regex` Scratch regex for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\l_tmpb_regex`

New: 2017-12-11

`\g_tmpa_regex` Scratch regex for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpb_regex`

New: 2017-12-11

8.8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?
Additional error-checking to come.
- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.
- Test for the maximum register `\c_max_register_int`.
- Find out whether the fact that `\W` and friends match the end-marker leads to bugs. Possibly update `_regex_item_reverse:n`.
- The empty `cs` should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.

Code improvements to come.

- Shift arrays so that the useful information starts at position 1.
- Only build `\c{...}` once.
- Use arrays for the left and right state stacks when compiling a regex.
- Should `_regex_action_free_group:n` only be used for greedy `{n,}` quantifier? (I think not.)
- Quantifiers for `\u` and assertions.
- When matching, keep track of an explicit stack of `curr_state` and `curr_submatches`.
- If possible, when a state is reused by the same thread, kill other subthreads.

- Use an array rather than `\g__regex_balance_t1` to build the function `__regex_replacement_balance_one_match:n`.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single `__regex_action_free:n`.
- Optimize the use of `__regex_action_success:` by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of `\int_step_...` functions.
- Groups don't capture within regexes for csnames; optimize and document.
- Better “show” for anchors, properties, and catcode tests.
- Does `\K` really need a new state for itself?
- When compiling, use a boolean `in_cs` and less magic numbers.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.
- Conditional subpatterns with look ahead/behind: “if what follows is [...], then [...]”.
- `(*..)` and `(?..)` sequences to set some options.
- UTF-8 mode for pdf \TeX .
- Newline conventions are not done. In particular, we should have an option for `.` not to match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Unicode properties: `\p{..}` and `\P{..}`; `\X` which should match any “extended” Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.

The following features of PCRE or Perl may or may not be implemented.

- Callout with `(?C...)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break:` and then of playing well with `\t1_map_break:` called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.
- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?

- Named subpatterns: \TeX programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- $\backslash\text{ddd}$, matching the character with octal code ddd : we already have $\backslash\text{x}\{\dots\}$ and the syntax is confusingly close to what we could have used for backreferences ($\backslash 1$, $\backslash 2$, ...), making it harder to produce useful error message.
- $\backslash\text{cx}$, similar to \TeX 's own $\backslash\hat{\hat{x}}$.
- Comments: \TeX already has its own system for comments.
- $\backslash\text{Q}\dots\backslash\text{E}$ escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- $\backslash\text{C}$ single byte in UTF-8 mode: $\text{Xe}\text{\TeX}$ and $\text{Lua}\text{\TeX}$ serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

Chapter 9

The `l3prg` module Control structures

Conditional processing in \LaTeX 3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The states returned are `<true>` and `<false>`.

\LaTeX 3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean `<true>` or `<false>`. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean `<true>` or `<false>` values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the `N`) and then executes either `true` or `false` depending on the result.

\TeX hackers note: The arguments are executed after exiting the underlying `\if... \fi:` structure.

9.1 Defining a set of conditional functions

<code>\prg_new_conditional:Npnn</code>	<code>\prg_new_conditional:Npnn <name>:<arg spec> <parameters> {<conditions>} {<code>}</code>
<code>\prg_set_conditional:Npnn</code>	<code>\prg_new_conditional:Nnn <name>:<arg spec> {<conditions>} {<code>}</code>

<code>\prg_gset_conditional:Npnn</code>	These functions create a family of conditionals using the same <code>{<code>}</code> to perform the test created. Those conditionals are expandable if <code><code></code> is. The <code>new</code> versions check for existing definitions and perform assignments globally (<i>cf.</i> <code>\cs_new:Npn</code>) whereas the <code>set</code> versions do no check and perform assignments locally (<i>cf.</i> <code>\cs_set:Npn</code>). The conditionals created are dependent on the comma-separated list of <code><conditions></code> , which should be one or more of <code>p</code> , <code>T</code> , <code>F</code> and <code>TF</code> .
<code>\prg_new_conditional:Nnn</code>	
<code>\prg_set_conditional:Nnn</code>	
<code>\prg_gset_conditional:Nnn</code>	

Updated: 2022-11-01

```

\prg_new_protected_conditional:Npnn \prg_new_protected_conditional:Npnn \<name>:\<arg spec>
\prg_set_protected_conditional:Npnn \<parameters> \<conditions> \<code>
\prg_gset_protected_conditional:Npnn \prg_new_protected_conditional:Nnn \<name>:\<arg spec>
\prg_new_protected_conditional:Nnn \<conditions> \<code>
\prg_set_protected_conditional:Nnn
\prg_gset_protected_conditional:Nnn

```

Updated: 2012-02-06

These functions create a family of protected conditionals using the same $\{\langle code \rangle\}$ to perform the test created. The $\langle code \rangle$ does not need to be expandable. The **new** version check for existing definitions and perform assignments globally (*cf.* $\backslash cs_new:Npn$) whereas the **set** version do not (*cf.* $\backslash cs_set:Npn$). The conditionals created are depended on the comma-separated list of $\langle conditions \rangle$, which should be one or more of T, F and TF (not p).

The conditionals are defined by $\backslash prg_new_conditional:Npnn$ and friends as:

- $\backslash \langle name \rangle_p:\langle arg\ spec \rangle$ — a predicate function which will supply either a logical **true** or logical **false**. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for **protected** conditionals.
- $\backslash \langle name \rangle:\langle arg\ spec \rangle T$ — a function with one more argument than the original $\langle arg\ spec \rangle$ demands. The $\langle true\ branch \rangle$ code in this additional argument will be left on the input stream only if the test is **true**.
- $\backslash \langle name \rangle:\langle arg\ spec \rangle F$ — a function with one more argument than the original $\langle arg\ spec \rangle$ demands. The $\langle false\ branch \rangle$ code in this additional argument will be left on the input stream only if the test is **false**.
- $\backslash \langle name \rangle:\langle arg\ spec \rangle TF$ — a function with two more argument than the original $\langle arg\ spec \rangle$ demands. The $\langle true\ branch \rangle$ code in the first additional argument will be left on the input stream if the test is **true**, while the $\langle false\ branch \rangle$ code in the second argument will be left on the input stream if the test is **false**.

The $\langle code \rangle$ of the test may use $\langle parameters \rangle$ as specified by the second argument to $\backslash prg_set_conditional:Npnn$: this should match the $\langle argument\ specification \rangle$ but this is not enforced. The **Nnn** versions infer the number of arguments from the argument specification given (*cf.* $\backslash cs_new:Nn$, *etc.*). Within the $\langle code \rangle$, the functions $\backslash prg_return_true:$ and $\backslash prg_return_false:$ are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```

\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
{
  \if_meaning:w \l_tmpa_tl #1
  \prg_return_true:
\else:
  \if_meaning:w \l_tmpa_tl #2
  \prg_return_true:
\else:
  \prg_return_false:
\fi:

```



```

    \fi:
}

```

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because F is missing from the `<conditions>` list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

The special case where the code of a conditional ends with `\prg_return_true:` `\else:` `\prg_return_false:` `\fi:` is optimized.

```

\prg_new_eq_conditional:NNn \prg_new_eq_conditional:NNn \<name_1>:<arg spec_1> \<name_2>:<arg spec_2>
\prg_set_eq_conditional:NNn  {\<conditions>}
\prg_gset_eq_conditional:NNn

```

Updated: 2023-05-26

These functions copy a family of conditionals. The `new` version checks for existing definitions (*cf.* `\cs_new_eq:NN`) whereas the `set` version does not (*cf.* `\cs_set_eq:NN`). The conditionals copied are depended on the comma-separated list of `<conditions>`, which should be one or more of `p`, `T`, `F` and `TF`.

```

\prg_return_true:  * \prg_return_true:
\prg_return_false: * \prg_return_false:

```

These “return” functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc.*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an `f`-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

```

\prg_generate_conditional_variant:NNn \prg_generate_conditional_variant:NNn \<name>:<arg spec>
\prg_generate_conditional_variant:NNn \prg_generate_conditional_variant:NNn \<name>:<arg spec>

```

New: 2017-12-12

Defines argument-specifier variants of conditionals. This is equivalent to running `\cs_generate_variant:Nn <conditional> {\<variant argument specifiers>}` on each `<conditional>` described by the `<condition specifiers>`. These base-form `<conditionals>` are obtained from the `<name>` and `<arg spec>` as described for `\prg_new_conditional:Npnn`, and they should be defined.

9.2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and

`\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, *etc.* which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions except assignments are expandable and expect the input to also be fully expandable (which generally means being constructed from predicate functions and booleans, possibly nested).

T_EXhackers note: The `bool` data type is not implemented using the `\iffalse`/`\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain T_EX, L^AT_EX 2_ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

<code>\bool_new:N</code>	<code>\bool_new:N</code> $\langle boolean \rangle$
<code>\bool_new:c</code>	Creates a new $\langle boolean \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle boolean \rangle$ is initially <code>false</code> .

<code>\bool_const:Nn</code>	<code>\bool_const:Nn</code> $\langle boolean \rangle$ $\{ \langle boolexpr \rangle \}$
<code>\bool_const:cn</code>	Creates a new constant $\langle boolean \rangle$ or raises an error if the name is already taken. The value of the $\langle boolean \rangle$ is set globally to the result of evaluating the $\langle boolexpr \rangle$.

New: 2017-11-28

<code>\bool_set_false:N</code>	<code>\bool_set_false:N</code> $\langle boolean \rangle$
<code>\bool_set_false:c</code>	Sets $\langle boolean \rangle$ logically <code>false</code> .
<code>\bool_gset_false:N</code>	
<code>\bool_gset_false:c</code>	

<code>\bool_set_true:N</code>	<code>\bool_set_true:N</code> $\langle boolean \rangle$
<code>\bool_set_true:c</code>	Sets $\langle boolean \rangle$ logically <code>true</code> .
<code>\bool_gset_true:N</code>	
<code>\bool_gset_true:c</code>	

<code>\bool_set_eq:NN</code>	<code>\bool_set_eq:NN</code> $\langle boolean_1 \rangle$ $\langle boolean_2 \rangle$
<code>\bool_set_eq:(cN Nc cc)</code>	Sets $\langle boolean_1 \rangle$ to the current value of $\langle boolean_2 \rangle$.
<code>\bool_gset_eq:NN</code>	
<code>\bool_gset_eq:(cN Nc cc)</code>	

<code>\bool_set:Nn</code>	<code>\bool_set:Nn</code> $\langle boolean \rangle$ $\{ \langle boolexpr \rangle \}$
<code>\bool_set:cn</code>	Evaluates the $\langle boolean \text{ expression} \rangle$ as described for <code>\bool_if:nTF</code> , and sets the $\langle boolean \rangle$ variable to the logical truth of this evaluation.
<code>\bool_gset:Nn</code>	
<code>\bool_gset:cn</code>	

Updated: 2017-07-15

<code>\bool_set_inverse:N</code>	<code>\bool_set_inverse:N</code> $\langle boolean \rangle$
<code>\bool_set_inverse:c</code>	Toggles the $\langle boolean \rangle$ from <code>true</code> to <code>false</code> and conversely: sets it to the inverse of its current value.
<code>\bool_gset_inverse:N</code>	
<code>\bool_gset_inverse:c</code>	

New: 2018-05-10

`\bool_if_p:N` * `\bool_if_p:N` \langle *boolean* \rangle
`\bool_if_p:c` * `\bool_if:NTF` \langle *boolean* \rangle $\{$ *true code* $\}$ $\{$ *false code* $\}$
`\bool_if:NTF` * Tests the current truth of \langle *boolean* \rangle , and continues expansion based on this result.
`\bool_if:cTF` *

Updated: 2017-07-15

`\bool_to_str:N` * `\bool_to_str:N` \langle *boolean* \rangle
`\bool_to_str:c` * `\bool_to_str:n` \langle *boolean expression* \rangle
`\bool_to_str:n` * Expands to the string `true` or `false` depending on the logical truth of the \langle *boolean* \rangle or \langle *boolean expression* \rangle .

New: 2021-11-01

Updated: 2023-11-14

`\bool_show:N` * `\bool_show:N` \langle *boolean* \rangle
`\bool_show:c` * Displays the logical truth of the \langle *boolean* \rangle on the terminal.

New: 2012-02-09

Updated: 2021-04-29

`\bool_show:n` * `\bool_show:n` $\{$ \langle *boolean expression* \rangle $\}$
New: 2012-02-09 Displays the logical truth of the \langle *boolean expression* \rangle on the terminal.
Updated: 2017-07-15

`\bool_log:N` * `\bool_log:N` \langle *boolean* \rangle
`\bool_log:c` * Writes the logical truth of the \langle *boolean* \rangle in the log file.

New: 2014-08-22

Updated: 2021-04-29

`\bool_log:n` * `\bool_log:n` $\{$ \langle *boolean expression* \rangle $\}$
New: 2014-08-22 Writes the logical truth of the \langle *boolean expression* \rangle in the log file.
Updated: 2017-07-15

`\bool_if_exist_p:N` * `\bool_if_exist_p:N` \langle *boolean* \rangle
`\bool_if_exist_p:c` * `\bool_if_exist:NTF` \langle *boolean* \rangle $\{$ *true code* $\}$ $\{$ *false code* $\}$
`\bool_if_exist:NTF` * Tests whether the \langle *boolean* \rangle is currently defined. This does not check that the \langle *boolean* \rangle really is a boolean variable.
`\bool_if_exist:cTF` *

New: 2012-03-03

9.2.1 Constant and scratch booleans

`\c_true_bool` * Constants that represent `true` and `false`, respectively. Used to implement predicates.
`\c_false_bool` *

`\l_tmpa_bool` * A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any L^AT_EX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.
`\l_tmpb_bool` *

`\g_tmpa_bool` A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any L^AT_EX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpb_bool`

9.3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean `<true>` or `<false>` values, it seems only fitting that we also provide a parser for `<boolean expressions>`.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean `<true>` or `<false>`. It supports the logical operations And, Or and Not as the well-known infix operators `&&` and `||` and prefix `!` with their usual precedences (namely, `&&` binds more tightly than `||`). In addition to this, parentheses can be used to isolate sub-expressions. For example,

```
\int_compare_p:n { 1 = 1 } &&
(
  \int_compare_p:n { 2 = 3 } ||
  \int_compare_p:n { 4 <= 4 } ||
  \str_if_eq_p:nn { abc } { def }
) &&
! \int_compare_p:n { 2 = 4 }
```

is a valid boolean expression.

Contrarily to some other programming languages, the operators `&&` and `||` evaluate both operands in all cases, even when the first operand is enough to determine the result. This “eager” evaluation should be contrasted with the “lazy” evaluation of `\bool_lazy_...` functions.

T_EXhackers note: The eager evaluation of boolean expressions is unfortunately necessary in T_EX. Indeed, a lazy parser can get confused if `&&` or `||` or parentheses appear as (unbraced) arguments of some predicates. For instance, the innocuous-looking expression below would break (in a lazy parser) if `#1` were a closing parenthesis and `\l_tmpa_bool` were `true`.

```
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
```

Minimal (lazy) evaluation can be obtained using the conditionals `\bool_lazy_all:nTF`, `\bool_lazy_and:nnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:nnTF`, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```
\bool_lazy_and_p:nn
{
  \bool_lazy_any_p:n
  {
    { \int_compare_p:n { 2 = 3 } }
    { \int_compare_p:n { 4 <= 4 } }
    { \int_compare_p:n { 1 = \error } } % skipped
  }
}
{ ! \int_compare_p:n { 2 = 4 } }
```

the line marked with `skipped` is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically `true`. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

```
\bool_if_p:n * \bool_if_p:n {<boolean expression>}
\bool_if:nTF * \bool_if:nTF {<boolean expression>} {<true code>} {<false code>}
Updated: 2017-07-15
```

Tests the current truth of `<boolean expression>`, and continues expansion based on this result. The `<boolean expression>` should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` (“And”), `||` (“Or”), `!` (“Not”) and parentheses. The logical Not applies to the next predicate or group.

```
\bool_lazy_all_p:n * \bool_lazy_all_p:n { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} }
\bool_lazy_all:nTF * \bool_lazy_all:nTF { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} } {<true code>}
{<false code>}
New: 2015-11-15
Updated: 2017-07-15
```

Implements the “And” operation on the `<boolean expressions>`, hence is `true` if all of them are `true` and `false` if any of them is `false`. Contrarily to the infix operator `&&`, only the `<boolean expressions>` which are needed to determine the result of `\bool_lazy_all:nTF` are evaluated. See also `\bool_lazy_and:nnTF` when there are only two `<boolean expressions>`.

```
\bool_lazy_and_p:nn * \bool_lazy_and_p:nn {<boolexpr1>} {<boolexpr2>}
\bool_lazy_and:nnTF * \bool_lazy_and:nnTF {<boolexpr1>} {<boolexpr2>} {<true code>} {<false code>}
New: 2015-11-15
Updated: 2017-07-15
```

Implements the “And” operation between two boolean expressions, hence is `true` if both are `true`. Contrarily to the infix operator `&&`, the `<boolexpr2>` is only evaluated if it is needed to determine the result of `\bool_lazy_and:nnTF`. See also `\bool_lazy_all:nTF` when there are more than two `<boolean expressions>`.

```
\bool_lazy_any_p:n * \bool_lazy_any_p:n { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} }
\bool_lazy_any:nTF * \bool_lazy_any:nTF { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} } {<true code>}
{<false code>}
New: 2015-11-15
Updated: 2017-07-15
```

Implements the “Or” operation on the `<boolean expressions>`, hence is `true` if any of them is `true` and `false` if all of them are `false`. Contrarily to the infix operator `||`, only the `<boolean expressions>` which are needed to determine the result of `\bool_lazy_any:nTF` are evaluated. See also `\bool_lazy_or:nnTF` when there are only two `<boolean expressions>`.

```
\bool_lazy_or_p:nn * \bool_lazy_or_p:nn {<boolexpr1>} {<boolexpr2>}
\bool_lazy_or:nnTF * \bool_lazy_or:nnTF {<boolexpr1>} {<boolexpr2>} {<true code>} {<false code>}
New: 2015-11-15
Updated: 2017-07-15
```

Implements the “Or” operation between two boolean expressions, hence is `true` if either one is `true`. Contrarily to the infix operator `||`, the `<boolexpr2>` is only evaluated if it is needed to determine the result of `\bool_lazy_or:nnTF`. See also `\bool_lazy_any:nTF` when there are more than two `<boolean expressions>`.

```
\bool_not_p:n * \bool_not_p:n {<boolean expression>}
Updated: 2017-07-15
```

Function version of `!(<boolean expression>)` within a boolean expression.

<code>\bool_xor_p:n</code>	★	<code>\bool_xor_p:nn</code>	{ <i>boolexpr</i> ₁ }	{ <i>boolexpr</i> ₂ }		
<code>\bool_xor:nnTF</code>	★	<code>\bool_xor:nnTF</code>	{ <i>boolexpr</i> ₁ }	{ <i>boolexpr</i> ₂ }	{ <i>true code</i> }	{ <i>false code</i> }

New: 2018-05-09 Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operation.

9.4 Logical loops

Loops using either boolean expressions or stored boolean values.

<code>\bool_do_until:Nn</code>	☆	<code>\bool_do_until:Nn</code>	<i>boolean</i>	{ <i>code</i> }
<code>\bool_do_until:cn</code>	☆			

Updated: 2017-07-15 Places the *code* in the input stream for T_EX to process, and then checks the logical value of the *boolean*. If it is `false` then the *code* is inserted into the input stream again and the process loops until the *boolean* is `true`.

<code>\bool_do_while:Nn</code>	☆	<code>\bool_do_while:Nn</code>	<i>boolean</i>	{ <i>code</i> }
<code>\bool_do_while:cn</code>	☆			

Updated: 2017-07-15 Places the *code* in the input stream for T_EX to process, and then checks the logical value of the *boolean*. If it is `true` then the *code* is inserted into the input stream again and the process loops until the *boolean* is `false`.

<code>\bool_until_do:Nn</code>	☆	<code>\bool_until_do:Nn</code>	<i>boolean</i>	{ <i>code</i> }
<code>\bool_until_do:cn</code>	☆			

Updated: 2017-07-15 This function first checks the logical value of the *boolean*. If it is `false` the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean* is re-evaluated. The process then loops until the *boolean* is `true`.

<code>\bool_while_do:Nn</code>	☆	<code>\bool_while_do:Nn</code>	<i>boolean</i>	{ <i>code</i> }
<code>\bool_while_do:cn</code>	☆			

Updated: 2017-07-15 This function first checks the logical value of the *boolean*. If it is `true` the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean* is re-evaluated. The process then loops until the *boolean* is `false`.

<code>\bool_do_until:nn</code>	☆	<code>\bool_do_until:nn</code>	{ <i>boolean expression</i> }	{ <i>code</i> }
--------------------------------	---	--------------------------------	-------------------------------	-----------------

Updated: 2017-07-15 Places the *code* in the input stream for T_EX to process, and then checks the logical value of the *boolean expression* as described for `\bool_if:nTF`. If it is `false` then the *code* is inserted into the input stream again and the process loops until the *boolean expression* evaluates to `true`.

<code>\bool_do_while:nn</code>	☆	<code>\bool_do_while:nn</code>	{ <i>boolean expression</i> }	{ <i>code</i> }
--------------------------------	---	--------------------------------	-------------------------------	-----------------

Updated: 2017-07-15 Places the *code* in the input stream for T_EX to process, and then checks the logical value of the *boolean expression* as described for `\bool_if:nTF`. If it is `true` then the *code* is inserted into the input stream again and the process loops until the *boolean expression* evaluates to `false`.

<code>\bool_until_do:nn</code>	☆	<code>\bool_until_do:nn</code>	{ <i>boolean expression</i> }	{ <i>code</i> }
--------------------------------	---	--------------------------------	-------------------------------	-----------------

Updated: 2017-07-15 This function first checks the logical value of the *boolean expression* (as described for `\bool_if:nTF`). If it is `false` the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean expression* is re-evaluated. The process then loops until the *boolean expression* is `true`.

`\bool_while_do:nn` ☆ `\bool_while_do:nn {<boolean expression>} {<code>}`

Updated: 2017-07-15

This function first checks the logical value of the `<boolean expression>` (as described for `\bool_if:nTF`). If it is `true` the `<code>` is placed in the input stream and expanded. After the completion of the `<code>` the truth of the `<boolean expression>` is re-evaluated. The process then loops until the `<boolean expression>` is `false`.

`\bool_case:n` ☆ `\bool_case:nTF`

`\bool_case:nTF` ☆ `{`
 `{<boolexpr case1>} {<code case1>}`
 `{<boolexpr case2>} {<code case2>}`
 `...`
 `{<boolexpr casen>} {<code casen>}`
`}`
`{<>true code>}`
`{<>false code>}`

New: 2023-05-03

Evaluates in turn each of the `<boolean expression cases>` until the first one that evaluates to `true`. The `<code>` associated to this first case is left in the input stream, followed by the `<>true code>`, and other cases are discarded. If none of the cases match then only the `<>false code>` is inserted. The function `\bool_case:n`, which does nothing if there is no match, is also available. For example

```
\bool_case:nF
{
  { \dim_compare_p:n { \l__mypkg_wd_dim <= 10pt } }
  { Fits }
  { \int_compare_p:n { \l__mypkg_total_int >= 10 } }
  { Many }
  { \l__mypkg_special_bool }
  { Special }
}
{ No idea! }
```

leaves “Fits” or “Many” or “Special” or “No idea!” in the input stream, in a way similar to some other language’s “if ... elseif ... elseif ... else ...”.

9.5 Producing multiple copies

`\prg_replicate:nn` ☆ `\prg_replicate:nn {<integer expression>} {<tokens>}`

Updated: 2011-07-04

Evaluates the `<integer expression>` (which should be zero or positive) and creates the resulting number of copies of the `<tokens>`. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

9.6 Detecting T_EX’s mode

`\mode_if_horizontal_p:` ☆ `\mode_if_horizontal_p:`

`\mode_if_horizontal:TF` ☆ `\mode_if_horizontal:TF {<>true code>} {<>false code>}`

Detects if T_EX is currently in horizontal mode.

```
\mode_if_inner_p: * \mode_if_inner_p:
\mode_if_inner:TF * \mode_if_inner:TF {\true code} {\false code}
```

Detects if T_EX is currently in inner mode.

```
\mode_if_math_p: * \mode_if_math_p:
\mode_if_math:TF * \mode_if_math:TF {\true code} {\false code}
```

Updated: 2011-09-05 Detects if T_EX is currently in maths mode.

```
\mode_if_vertical_p: * \mode_if_vertical_p:
\mode_if_vertical:TF * \mode_if_vertical:TF {\true code} {\false code}
```

Detects if T_EX is currently in vertical mode.

9.7 Primitive conditionals

```
\if_predicate:w * \if_predicate:w <predicate> <true code> \else: <false code> \fi:
```

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the *<predicate>* but to make the coding clearer this should be done through `\if_bool:N`.)

```
\if_bool:N * \if_bool:N <boolean> <true code> \else: <false code> \fi:
```

This function takes a boolean variable and branches according to the result.

9.8 Nestable recursions and mappings

There are a number of places where recursion or mapping constructs are used in `expl3`. At a low-level, these typically require insertion of tokens at the end of the content to allow “clean up”. To support such mappings in a nestable form, the following functions are provided.

```
\prg_break_point:Nn * \prg_break_point:Nn \<type>_map_break: {\code}
```

New: 2018-03-26

Used to mark the end of a recursion or mapping: the functions `\<type>_map_break:` and `\<type>_map_break:n` use this to break out of the loop (see `\prg_map_break:Nn` for how to set these up). After the loop ends, the *<code>* is inserted into the input stream. This occurs even if the break functions are *not* applied: `\prg_break_point:Nn` is functionally-equivalent in these cases to `\use_ii:nn`.

<code>\prg_map_break:Nn</code>	* <code>\prg_map_break:Nn \langle type \rangle_map_break: {\langle user code \rangle}</code>
<small>New: 2018-03-26</small>	...
	<code>\prg_break_point:Nn \langle type \rangle_map_break: {\langle ending code \rangle}</code>

Breaks a recursion in mapping contexts, inserting in the input stream the `\langle user code \rangle` after the `\langle ending code \rangle` for the loop. The function breaks loops, inserting their `\langle ending code \rangle`, until reaching a loop with the same `\langle type \rangle` as its first argument. This `\langle type \rangle_map_break:` argument must be defined; it is simply used as a recognizable marker for the `\langle type \rangle`.

For types with mappings defined in the kernel, `\langle type \rangle_map_break:` and `\langle type \rangle_map_break:n` are defined as `\prg_map_break:Nn \langle type \rangle_map_break: {}` and the same with `{}` omitted.

9.8.1 Simple mappings

In addition to the more complex mappings above, non-nestable mappings are used in a number of locations and support is provided for these.

<code>\prg_break_point:</code>	* This copy of <code>\prg_do_nothing:</code> is used to mark the end of a fast short-term recursion: the function <code>\prg_break:n</code> uses this to break out of the loop.
<small>New: 2018-03-27</small>	

<code>\prg_break:</code>	* <code>\prg_break:n {\langle code \rangle} ... \prg_break_point:</code>
<code>\prg_break:n</code>	* Breaks a recursion which has no <code>\langle ending code \rangle</code> and which is not a user-breakable mapping (see for instance implementation of <code>\int_step_function:nnnN</code>), and inserts the <code>\langle code \rangle</code> in the input stream.
<small>New: 2018-03-27</small>	

9.9 Internal programming functions

<code>\group_align_safe_begin:</code>	* <code>\group_align_safe_begin:</code>
<code>\group_align_safe_end:</code>	* ...
<small>Updated: 2011-08-11</small>	<code>\group_align_safe_end:</code>

These functions are used to enclose material in a `TeX` alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as `TeX` uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` would result in a forbidden comparison of the internal `\endtemplate` token, yielding a fatal error. Each `\group_align_safe_begin:` must be matched by a `\group_align_safe_end:`, although this does not have to occur within the same function.

Chapter 10

The l3sys module System/runtime functions

10.1 The name of the job

`\c_sys_jobname_str` Constant that gets the “job name” assigned when T_EX starts.

New: 2015-09-19
Updated: 2019-10-27

T_EXhackers note: This is the T_EX primitive `\jobname`. For technical reasons, the string here is not of the same internal form as other, but may be manipulated using normal string functions.

10.2 Date and time

`\c_sys_minute_int` The date and time at which the current job was started: these are all reported as integers.

`\c_sys_hour_int`

`\c_sys_day_int`

`\c_sys_month_int`

`\c_sys_year_int`

New: 2015-09-22

T_EXhackers note: Whilst the underlying T_EX primitives `\time`, `\day`, `\month`, and `\year` can be altered by the user, this interface to the time and date is intended to be the “real” values.

`\c_sys_timestamp_str` The timestamp for the current job: the format is as described for `\file_timestamp:n`.

New: 2023-08-27

10.3 Engine

`\sys_if_engine_luatex_p:` * `\sys_if_engine_pdftex_p:`
`\sys_if_engine_luatex:TF` * `\sys_if_engine_pdftex:TF` `{(true code)}` `{(false code)}`
`\sys_if_engine_pdftex_p:` *
`\sys_if_engine_pdftex:TF` *
`\sys_if_engine_ptex_p:` *
`\sys_if_engine_ptex:TF` *
`\sys_if_engine_uptex_p:` *
`\sys_if_engine_uptex:TF` *
`\sys_if_engine_xetex_p:` *
`\sys_if_engine_xetex:TF` *

New: 2015-09-07

`\c_sys_engine_str` The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

New: 2015-09-19

`\c_sys_engine_exec_str` The name of the standard executable for the current \TeX engine given as a lower case string: one of `luatex`, `luahtex`, `pdftex`, `eptex`, `euptex` or `xetex`.

New: 2020-08-20

`\c_sys_engine_format_str` The name of the preloaded format for the current \TeX run given as a lower case string: one of `lualatex` (or `dvilualatex`), `pdflatex` (or `latex`), `platex`, `uplatex` or `xelatex` for \LaTeX , similar names for plain \TeX (except `pdfTeX` in DVI mode yields `etex`), and `cont-en` for `ConTeXt` (i.e. the `\fmtname`).

New: 2020-08-20

`\c_sys_engine_version_str` The version string of the current engine, in the same form as given in the banner issued when running a job. For `pdfTeX` and `LuaTeX` this is of the form

New: 2018-05-02

$\langle major \rangle . \langle minor \rangle . \langle revision \rangle$

For `XYTeX`, the form is

$\langle major \rangle . \langle minor \rangle$

For `pTeX` and `upTeX`, only releases since \TeX Live 2018 make the data available, and the form is more complex, as it comprises the `pTeX` version, the `upTeX` version and the `e-pTeX` version.

$p \langle major \rangle . \langle minor \rangle . \langle revision \rangle - u \langle major \rangle . \langle minor \rangle - \langle epTeX \rangle$

where the `u` part is only present for `upTeX`.

`\sys_timer:` * `\sys_timer:`

New: 2021-05-12

Expands to the current value of the engine's timer clock, a non-negative integer. This function is only defined for engines with timer support. This command measures not just CPU time but real time (including time waiting for user input). The unit are scaled seconds (2^{-16} seconds).

`\sys_if_timer_exist_p:` * `\sys_if_timer_exist_p:`
`\sys_if_timer_exist:TF` * `\sys_if_timer_exist:TF` `{\true code}` `{\false code}`

New: 2021-05-12 Tests whether current engine has timer support.

10.4 Output format

`\sys_if_output_dvi_p:` * `\sys_if_output_dvi_p:`
`\sys_if_output_dvi:TF` * `\sys_if_output_dvi:TF` `{\true code}` `{\false code}`
`\sys_if_output_pdf_p:` * Conditionals which give the current output mode the \TeX run is operating in. This is
`\sys_if_output_pdf:TF` * always one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are
thus complementary and are both provided to allow the programmer to emphasise the
most appropriate case.

`\c_sys_output_str` The current output mode given as a lower case string: one of `dvi` or `pdf`.

New: 2015-09-19

10.5 Platform

`\sys_if_platform_unix_p:` * `\sys_if_platform_unix_p:`
`\sys_if_platform_unix:TF` * `\sys_if_platform_unix:TF` `{\true code}` `{\false code}`
`\sys_if_platform_windows_p:` *
`\sys_if_platform_windows:TF` *

New: 2018-07-27

Conditionals which allow platform-specific code to be used. The names follow the Lua `os.type()` function, *i.e.* all Unix-like systems are `unix` (including Linux and MacOS).

`\c_sys_platform_str` The current platform given as a lower case string: one of `unix`, `windows` or `unknown`.

New: 2018-07-27

10.6 Random numbers

`\sys_rand_seed:` * `\sys_rand_seed:`

New: 2017-05-27 Expands to the current value of the engine's random seed, a non-negative integer. In engines without random number support this expands to 0.

`\sys_gset_rand_seed:n` `\sys_gset_rand_seed:n` $\langle\text{int expr}\rangle$

New: 2017-05-27

Globally sets the seed for the engine's pseudo-random number generator to the $\langle\text{integer expression}\rangle$. This random seed affects all `\..._rand` functions (such as `\int_rand:nn` or `\clist_rand_item:n`) as well as other packages relying on the engine's random number generator. In engines without random number support this produces an error.

TeXhackers note: While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond 2^{28} is divided by an appropriate power of 2. We recommend using an integer in $[0, 2^{28} - 1]$.

10.7 Access to the shell

`\sys_get_shell:nnN` `\sys_get_shell:nnN` $\langle\text{shell command}\rangle$ $\langle\text{setup}\rangle$ $\langle\text{tl var}\rangle$

`\sys_get_shell:nnNTF` `\sys_get_shell:nnNTF` $\langle\text{shell command}\rangle$ $\langle\text{setup}\rangle$ $\langle\text{tl var}\rangle$ $\langle\text{true code}\rangle$ $\langle\text{false code}\rangle$

New: 2019-09-20

Defines $\langle\text{tl var}\rangle$ to the text returned by the $\langle\text{shell command}\rangle$. The $\langle\text{shell command}\rangle$ is converted to a string using `\tl_to_str:n`. Category codes may need to be set appropriately via the $\langle\text{setup}\rangle$ argument, which is run just before running the $\langle\text{shell command}\rangle$ (in a group). If shell escape is disabled, the $\langle\text{tl var}\rangle$ will be set to `\q_no_value` in the non-branching version. Note that quote characters (") *cannot* be used inside the $\langle\text{shell command}\rangle$. The `\sys_get_shell:nnNTF` conditional inserts the $\langle\text{true code}\rangle$ if the shell is available and no quote is detected, and the $\langle\text{false code}\rangle$ otherwise.

Note: It is not possible to tell from TeX if a command is allowed in restricted shell escape. If restricted escape is enabled, the **true** branch is taken: if the command is forbidden at this stage, a low-level TeX error will arise.

`\c_sys_shell_escape_int` This variable exposes the internal triple of the shell escape status. The possible values are

New: 2017-05-27

- 0 Shell escape is disabled
- 1 Unrestricted shell escape is enabled
- 2 Restricted shell escape is enabled

`\sys_if_shell_p: *` `\sys_if_shell_p:`
`\sys_if_shell:TF *` `\sys_if_shell:TF` $\langle\text{true code}\rangle$ $\langle\text{false code}\rangle$

New: 2017-05-27

Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

`\sys_if_shell_unrestricted_p: *` `\sys_if_shell_unrestricted_p:`
`\sys_if_shell_unrestricted:TF *` `\sys_if_shell_unrestricted:TF` $\langle\text{true code}\rangle$ $\langle\text{false code}\rangle$

New: 2017-05-27

Performs a check for whether *unrestricted* shell escape is enabled.

```
\sys_if_shell_restricted_p: * \sys_if_shell_restricted_p:
\sys_if_shell_restricted:TF * \sys_if_shell_restricted:TF {\true code} {\false code}
```

New: 2017-05-27

Performs a check for whether *restricted* shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use `\sys_if_shell:TF`.

```
\sys_shell_now:n \sys_shell_now:n {\tokens}
\sys_shell_now:e Execute <tokens> through shell escape immediately.
```

New: 2017-05-27

```
\sys_shell_shipout:n \sys_shell_shipout:n {\tokens}
\sys_shell_shipout:e Execute <tokens> through shell escape at shipout.
```

New: 2017-05-27

10.8 System queries

Some queries can be made about the file system, etc., without needing to use unrestricted shell escape. This is carried out using the script `l3sys-query`, which is documented separately. The wrappers here use this script, if available, to obtain system information that is not directly available within the T_EX run. Note that if restricted shell escape is disabled, no results can be obtained.

```
\sys_get_query:nN \sys_get_query:nN {\cmd} {\tl var}
\sys_get_query:nnN \sys_get_query:nnN {\cmd} {\spec} {\tl var}
\sys_get_query:nnnN \sys_get_query:nnnN {\cmd} {\options} {\spec} {\tl var}
```

New: 2024-03-08
Updated: 2024-04-08

Sets the `<tl var>` to the information returned by the `l3sys-query <cmd>`, potentially supplying the `<options>` and `<spec>` to the query call. The valid `<cmd>` names are at present

- `pwd` Returns the present working directory
- `ls` Returns a directory listing, using the `<spec>` to select files and applying the `<options>` if given

The `<spec>` is likely to contain the wildcards `*` or `?`, and will automatically be passed to the script without shell expansion. In a glob is needed within the `<options>`, this will need to be protected from shell expansion using `'` tokens.

The `<spec>` and `<options>`, if given, are expanded fully before passing to the underlying script.

Spaces in the output are stored as active tokens, allowing them to be replaced by for example a visible space easily. Other non-letter characters in the ASCII range are set to category code 12. The category codes for characters out of the ASCII range are left unchanged: typically this will mean that with an 8-bit engine, accented values can be typeset directly whilst in Unicode engines, standard category code setup will apply.

If more than one line of text is returned by the `<cmd>`, these will be separated by character 13 (`^M`) tokens of category code 12. In most cases, `\sys_split_query:nnnN` should be preferred when multi-line output is expected.

<code>\sys_split_query:nN</code>	<code>\sys_split_query:nN {<cmd>} {<seq>}</code>
<code>\sys_split_query:nnN</code>	<code>\sys_split_query:nnN {<cmd>} {<spec>} {<seq>}</code>
<code>\sys_split_query:nnnN</code>	<code>\sys_split_query:nnnN {<cmd>} {<options>} {<spec>} {<seq>}</code>

New: 2024-03-08 Works as described for `\sys_split_query:nnnN`, but sets the `<seq>` to contain one entry for each line returned by `l3sys-query`. This function should therefore be preferred where multi-line return is expected, e.g. for the `ls` command.

10.9 Loading configuration data

<code>\sys_load_backend:n</code>	<code>\sys_load_backend:n {<backend>}</code>
----------------------------------	--

New: 2019-09-12 Loads the additional configuration file needed for backend support. If the `<backend>` is empty, the standard backend for the engine in use will be loaded. This command may only be used once.

<code>\sys_ensure_backend:</code>	<code>\sys_ensure_backend:</code>
-----------------------------------	-----------------------------------

New: 2022-07-29 Ensures that a backend has been loaded by calling `\sys_load_backend:n` if required.

<code>\c_sys_backend_str</code>	Set to the name of the backend in use by <code>\sys_load_backend:n</code> when issued. Possible values are
---------------------------------	--

- `pdftex`
- `luatex`
- `xetex`
- `dvips`
- `dvipdfmx`
- `dvisvgm`

<code>\sys_load_debug:</code>	<code>\sys_load_debug:</code>
-------------------------------	-------------------------------

New: 2019-09-12 Load the additional configuration file for debugging support.

10.9.1 Final settings

<code>\sys_finalise:</code>	<code>\sys_finalise:</code>
-----------------------------	-----------------------------

New: 2019-10-06 Finalises all system-dependent functionality: required before loading a backend.

Chapter 11

The l3msg module

Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

11.1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\` may be used to force a new line and `_` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the L^AT_EX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

Some authors may find the need to include spaces as `~` characters tedious. This can be avoided by locally resetting the category code of `_`.


```

\char_set_catcode_space:n { '\ }
\msg_new:nnn { foo } { bar }
  {Some message text using '#1' and usual message shorthands \{ \ \ \}.}
\char_set_catcode_ignore:n { '\ }

```

although in general this may be confusing; simply writing the messages using ~ characters is the method favored by the team.

```

\msg_new:nnnn \msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}
\msg_new:nnee
\msg_new:nnn
\msg_new:nne
Updated: 2011-08-16

```

Creates a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (#1 to #4) can be used: these will be supplied at the time the message is used. An error is raised if the *<message>* already exists.

```

\msg_set:nnnn \msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}
\msg_set:nnn

```

Sets up the text for a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (#1 to #4) can be used: these will be supplied at the time the message is used.

```

\msg_if_exist_p:nn * \msg_if_exist_p:nn {<module>} {<message>}
\msg_if_exist:nnTF * \msg_if_exist:nnTF {<module>} {<message>} {<>true code>} {<>false code>}
New: 2012-03-03

```

Tests whether the *<message>* for the *<module>* is currently defined.

11.2 Customizable information for message modules

```

\msg_module_name:n * \msg_module_name:n {<module>}
New: 2018-10-10

```

Expands to the public name of the *<module>* as defined by `\g_msg_module_name_prop` (or otherwise leaves the *<module>* unchanged).

```

\msg_module_type:n * \msg_module_type:n {<module>}
New: 2018-10-10

```

Expands to the description which applies to the *<module>*, for example a `Package` or `Class`. The information here is defined in `\g_msg_module_type_prop`, and will default to `Package` if an entry is not present.

```

\g_msg_module_name_prop
New: 2018-10-10

```

Provides a mapping between the module name used for messages, and that for documentation.

```

\g_msg_module_type_prop
New: 2018-10-10

```

Provides a mapping between the module name used for messages, and that type of module. For example, for L^AT_EX3 core messages, an empty entry is set here meaning that they are not described using the standard `Package` text.

11.3 Contextual information for messages

`\msg_line_context: ☆ \msg_line_context:`

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is preceded by the text `on line`.

`\msg_line_number: ★ \msg_line_number:`

Prints the current line number when a message is given.

`\msg_fatal_text:n ★ \msg_fatal_text:n {⟨module⟩}`

Produces the standard text

Fatal Package ⟨*module*⟩ Error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the ⟨*module*⟩ to be included. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the `expl3` mechanism.

`\msg_critical_text:n ★ \msg_critical_text:n {⟨module⟩}`

Produces the standard text

Critical Package ⟨*module*⟩ Error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the ⟨*module*⟩ to be included. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the `expl3` mechanism.

`\msg_error_text:n ★ \msg_error_text:n {⟨module⟩}`

Produces the standard text

Package ⟨*module*⟩ Error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the ⟨*module*⟩ to be included. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the `expl3` mechanism.

`\msg_warning_text:n ★ \msg_warning_text:n {⟨module⟩}`

Produces the standard text

Package ⟨*module*⟩ Warning

This function can be redefined to alter the language in which the message is given, using #1 as the name of the ⟨*module*⟩ to be included. The ⟨*type*⟩ of ⟨*module*⟩ may be adjusted: `Package` is the standard outcome: see `\msg_module_type:n`. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the `expl3` mechanism.

`\msg_info_text:n` * `\msg_info_text:n {<module>}`

Produces the standard text:

Package `<module>` Info

This function can be redefined to alter the language in which the message is given, using #1 as the name of the `<module>` to be included. The `<type>` of `<module>` may be adjusted: `Package` is the standard outcome: see `\msg_module_type:n`. Any redefinition *must* produce output containing the `<module>` name, and will affect all messages using the `expl3` mechanism.

`\msg_see_documentation_text:n` * `\msg_see_documentation_text:n {<module>}`

Updated: 2018-09-30

Produces the standard text

See the `<module>` documentation for further information.

This function can be redefined to alter the language in which the message is given, using #1 as the name of the `<module>` to be included. The name of the `<module>` is produced using `\msg_module_name:n`.

11.4 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the `e`-type variants should be used to expand material. Note that this expansion takes place with the standard definitions in effect, which means that shorthands such as `\~` or `\` are *not* available; instead one should use `\iow_char:N \~` and `\iow_newline:`, respectively. The following message classes exist:

- `fatal`, ending the `TEX` run;
- `critical`, ending the file being input;
- `error`, interrupting the `TEX` run without ending it;
- `warning`, written to terminal and log file, for important messages that may require corrections by the user;
- `note` (less common than `info`) for important information messages written to the terminal and log file;
- `info` for normal information messages written to the log file only;
- `term` and `log` for un-decorated messages written to the terminal and log file, or to the log file only;
- `none` for suppressed messages.

```

\msg_fatal:nnnnnn      \msg_fatal:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>}
\msg_fatal:nneeee      {<arg three>} {<arg four>}
\msg_fatal:nnnnn
\msg_fatal:(nneee|nnnee)
\msg_fatal:nnnn
\msg_fatal:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_fatal:nnn
\msg_fatal:(nnV|nne)
\msg_fatal:nn

```

Updated: 2012-08-11

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a fatal error the \TeX run halts. No PDF file will be produced in this case (DVI mode runs may produce a truncated DVI file).

```

\msg_critical:nnnnnn   \msg_critical:nnnnnn {<module>} {<message>} {<arg one>} {<arg
\msg_critical:nneeee   two>} {<arg three>} {<arg four>}
\msg_critical:nnnnn
\msg_critical:(nneee|nnnee)
\msg_critical:nnnn
\msg_critical:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_critical:nnn
\msg_critical:(nnV|nne)
\msg_critical:nn

```

Updated: 2012-08-11

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a critical error, \TeX stops reading the current input file. This may halt the \TeX run (if the current file is the main file) or may abort reading a sub-file.

\TeX hackers note: The \TeX $\backslash\text{endinput}$ primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

```

\msg_error:nnnnnn     \msg_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>}
\msg_error:nneeee     {<arg three>} {<arg four>}
\msg_error:nnnnn
\msg_error:(nneee|nnnee)
\msg_error:nnnn
\msg_error:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_error:nnn
\msg_error:(nnV|nne)
\msg_error:nn

```

Updated: 2012-08-11

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

```

\msg_warning:nnnnnn          \msg_warning:nnnnnn {\module} {\message} {\arg one} {\arg
\msg_warning:nneeee          two} {\arg three} {\arg four}
\msg_warning:nnnnn
\msg_warning:(nneee|nnnee)
\msg_warning:nnnn
\msg_warning:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_warning:nnn
\msg_warning:(nnV|nne)
\msg_warning:nn

```

Updated: 2012-08-11

Issues $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The warning text is added to the log file and the terminal, but the \TeX run is not interrupted.

```

\msg_note:nnnnnn          \msg_note:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg
\msg_note:nneeee          three} {\arg four}
\msg_note:nnnnn          \msg_info:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg
\msg_note:(nneee|nnnee)    three} {\arg four}
\msg_note:nnnn
\msg_note:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_note:nnn
\msg_note:(nnV|nne)
\msg_note:nn
\msg_info:nnnnnn
\msg_info:nneeee
\msg_info:nnnnn
\msg_info:(nneee|nnnee)
\msg_info:nnnn
\msg_info:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_info:nnn
\msg_info:(nnV|nne)
\msg_info:nn

```

New: 2021-05-18

Issues $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. For the more common \msg_info:nnnnnn , the information text is added to the log file only, while \msg_note:nnnnnn adds the info text to both the log file and the terminal. The \TeX run is not interrupted.

```

\msg_term:nnnnnn          \msg_term:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg
\msg_term:nneeee          three>} {<arg four>}
\msg_term:nnnnn          \msg_log:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg
\msg_term:(nneee|nnee)    three>} {<arg four>}
\msg_term:nnnn
\msg_term:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_term:nnn
\msg_term:(nnV|nne)
\msg_term:nn
\msg_log:nnnnnn
\msg_log:nneeee
\msg_log:nnnnn
\msg_log:(nneee|nnee)
\msg_log:nnnn
\msg_log:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_log:nnn
\msg_log:(nnV|nne)
\msg_log:nn

```

Updated: 2012-08-11

Issues *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The output is briefer than `\msg_info:nnnnnn`, omitting for instance the module name. It is added to the log file by `\msg_log:nnnnnn` while `\msg_term:nnnnnn` also prints it on the terminal.

```

\msg_none:nnnnnn          \msg_none:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg
\msg_none:nneeee          three>} {<arg four>}
\msg_none:nnnnn
\msg_none:(nneee|nnee)
\msg_none:nnnn
\msg_none:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_none:nnn
\msg_none:(nnV|nne)
\msg_none:nn

```

Updated: 2012-08-11

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

11.4.1 Messages for showing material

```

\msg_show:nnnnnn      \msg_show:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg
\msg_show:nneeee      three>} {<arg four>}
\msg_show:nnnnn
\msg_show:(nneee|nnnee)
\msg_show:nnnn
\msg_show:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_show:nnn
\msg_show:(nnV|nne)
\msg_show:nn

```

New: 2017-12-04

Issues `<module>` information `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. The information text is shown on the terminal and the \TeX run is interrupted in a manner similar to `\tl_show:n`. This is used in conjunction with `\msg_show_item:n` and similar functions to print complex variable contents completely. If the formatted text does not contain `>~` at the start of a line, an additional line `>~.` will be put at the end. In addition, a final period is added if not present.

```

\msg_show_item:n      * \seq_map_function:NN <seq> \msg_show_item:n
\msg_show_item_unbraced:n * \prop_map_function:NN <prop> \msg_show_item:nn
\msg_show_item:nn      *
\msg_show_item_unbraced:nn *

```

New: 2017-12-04

Used in the text of messages for `\msg_show:nnnnnn` to show or log a list of items or key-value pairs. The output of `\msg_show_item:n` produces a newline, the prefix `>`, two spaces, then the braced string representation of its argument. The two-argument versions separates the key and value using `uu=>uu`, and the `unbraced` versions don't print the surrounding braces.

These functions are suitable for usage with iterator functions like `\seq_map_function:NN`, `\prop_map_function:NN`, etc. For example, with a sequence `\l_tmpa_seq` containing `a`, `{b}` and `\c`,

```
\seq_map_function:NN \l_tmpa_seq \msg_show_item:n
```

would expand to three lines:

```

>uu{a}
>uu{{b}}
>uu{\c}

```

11.4.2 Expandable error messages

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using `\msg_error:nnnnnn` or other non-expandable commands from the previous section. Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools

to print to the terminal or the log file are expandable. As a result, short-hands such as `\{` or `\}` do not work, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

```

\msg_expandable_error:nnnnn * \msg_expandable_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg
\msg_expandable_error:nnffff * two>} {<arg three>} {<arg four>}
\msg_expandable_error:nnnnn *
\msg_expandable_error:nnffff *
\msg_expandable_error:nnnn *
\msg_expandable_error:nnff *
\msg_expandable_error:nnn *
\msg_expandable_error:nnf *
\msg_expandable_error:nn *

```

New: 2015-08-06

Updated: 2019-02-28

Issues an “Undefined error” message from T_EX itself using the undefined control sequence `\???` then prints “! *<module>*: ”*<error message>*”, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

11.5 Redirecting messages

Each message has a “name”, which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some-text } { Some-more-text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error

immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$ in this order, then the $A \rightarrow B$ redirection is cancelled.

`\msg_redirect_class:nn` `\msg_redirect_class:nn {<class one>} {<class two>}`

Updated: 2012-04-27

Changes the behaviour of messages of `<class one>` so that they are processed using the code for those of `<class two>`. Each `<class>` can be one of `fatal`, `critical`, `error`, `warning`, `note`, `info`, `term`, `log`, `none`.

`\msg_redirect_module:nnn` `\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}`

Updated: 2012-04-27

Redirects message of `<class one>` for `<module>` to act as though they were from `<class two>`. Messages of `<class one>` from sources other than `<module>` are not affected by this redirection. This function can be used to make some messages “silent” by default. For example, all of the `warning` messages of `<module>` could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

`\msg_redirect_name:nnn` `\msg_redirect_name:nnn {<module>} {<message>} {<class>}`

Updated: 2012-04-27

Redirects a specific `<message>` from a specific `<module>` to act as a member of `<class>` of messages. No further redirection is performed. This function can be used to make a selected message “silent” without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

Chapter 12

The `l3file` module

File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files `TeX` attempts to locate them using both the operating system path and entries in the `TeX` file database (most `TeX` systems use such a database). Thus the “current path” for `TeX` is somewhat broader than that for other programs.

For functions which expect a `<file name>` argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* expanded, allowing the direct use of these in file names. Quote tokens (`"`) are not permitted in file names as they are reserved for internal use by some `TeX` primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

12.1 Input–output stream management

As `TeX` engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in `LATeX3`. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

<code>\ior_new:N</code>	<code>\ior_new:N <stream></code>
<code>\ior_new:c</code>	<code>\iow_new:N <stream></code>
<code>\iow_new:N</code>	Globally reserves the name of the <code><stream></code> , either for reading or for writing as appropriate. The <code><stream></code> is not opened until the appropriate <code>\..._open:Nn</code> function is used.
<code>\iow_new:c</code>	Attempting to use a <code><stream></code> which has not been opened is an error, and the <code><stream></code> will behave as the corresponding <code>\c_term_...</code>

New: 2011-09-26
Updated: 2011-12-27

<code>\ior_open:Nn</code>	<code>\ior_open:Nn <stream> {<file name>}</code>
<code>\ior_open:cn</code>	Opens <code><file name></code> for reading using <code><stream></code> as the control sequence for file access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><file name></code> until an <code>\ior_close:N</code> instruction is given or the T _E X run ends. If the file is not found, an error is raised.

Updated: 2012-02-10

<code>\ior_open:NnTF</code>	<code>\ior_open:NnTF <stream> {<file name>} {<true code>} {<false code>}</code>
<code>\ior_open:cnTF</code>	Opens <code><file name></code> for reading using <code><stream></code> as the control sequence for file access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><file name></code> until a <code>\ior_close:N</code> instruction is given or the T _E X run ends. The <code><true code></code> is then inserted into the input stream. If the file is not found, no error is raised and the <code><false code></code> is inserted into the input stream.

New: 2013-01-12

<code>\iow_open:Nn</code>	<code>\iow_open:Nn <stream> {<file name>}</code>
<code>\iow_open:(NV cn cV)</code>	Opens <code><file name></code> for writing using <code><stream></code> as the control sequence for file access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><file name></code> until a <code>\iow_close:N</code> instruction is given or the T _E X run ends. Opening a file for writing clears any existing content in the file (<i>i.e.</i> writing is <i>not</i> additive).

Updated: 2012-02-09

<code>\ior_shell_open:Nn</code>	<code>\ior_shell_open:Nn <stream> {<shell command>}</code>
New: 2019-05-08	Opens the <i>pseudo</i> -file created by the output of the <code><shell command></code> for reading using <code><stream></code> as the control sequence for access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><shell command></code> until a <code>\ior_close:N</code> instruction is given or the T _E X run ends. If piped system calls are disabled an error is raised. For details of handling of the <code><shell command></code> , see <code>\sys_get_shell:nNTF</code> .

<code>\iow_shell_open:Nn</code>	<code>\iow_shell_open:Nn <stream> {<shell command>}</code>
New: 2023-05-25	Opens the <i>pseudo</i> -file created by the output of the <code><shell command></code> for writing using <code><stream></code> as the control sequence for access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><shell command></code> until an <code>\iow_close:N</code> instruction is given or the T _E X run ends. If piped system calls are disabled an error is raised. For details of handling of the <code><shell command></code> , see <code>\sys_get_shell:nNTF</code> .

```
\ior_close:N \ior_close:N <stream>
\ior_close:c \iow_close:N <stream>
\iow_close:N Closes the <stream>. Streams should always be closed when they are finished with as
\iow_close:c this ensures that they remain available to other programmers.
```

Updated: 2012-07-31

```
\ior_show:N \ior_show:N <stream>
\ior_show:c \ior_log:N <stream>
\ior_log:N \iow_show:N <stream>
\ior_log:c \iow_log:N <stream>
\iow_show:N Display (to the terminal or log file) the file name associated to the (read or write)
\iow_show:c <stream>.
\iow_log:N
\iow_log:c
```

New: 2021-05-11

```
\ior_show_list: \ior_show_list:
\ior_log_list: \ior_log_list:
\iow_show_list: \iow_show_list:
\iow_log_list: \iow_log_list:
```

New: 2017-06-27 Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

12.1.1 Reading from files

Reading from files and reading from the terminal are separate processes in `expl3`. The functions `\ior_get:NN` and `\ior_str_get:NN`, and their branching equivalents, are designed to work with *files*.

<code>\ior_get:NN</code>	<code>\ior_get:NN <stream> <token list variable></code>
<code>\ior_get:NNTF</code>	<code>\ior_get:NNTF <stream> <token list variable> <true code> <false code></code>

New: 2012-06-24
Updated: 2019-03-23

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input `<stream>` and stores the result locally in the `<token list>` variable. The material read from the `<stream>` is tokenized by T_EX according to the category codes and `\endlinechar` in force when the function is used. Assuming normal settings, any lines which do not end in a comment character `%` have the line ending converted to a space, so for example input

```
a b c
```

results in a token list `a b c`. Any blank line is converted to the token `\par`. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NMF \l_tmpa_tl \l_tmpb_tl
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens. In the non-branching version, where the `<stream>` is not open the `<tl var>` is set to `\q_no_value`.

T_EXhackers note: This protected macro is a wrapper around the T_EX primitive `\read`. Regardless of settings, T_EX replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code `\endlinechar`, omitted if `\endlinechar` is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

<code>\ior_str_get:NN</code>	<code>\ior_str_get:NN <stream> <token list variable></code>
<code>\ior_str_get:NNTF</code>	<code>\ior_str_get:NNTF <stream> <token list variable> <true code> <false code></code>

New: 2016-12-04
Updated: 2019-03-23

Function that reads one line from the file input `<stream>` and stores the result locally in the `<token list>` variable. The material is read from the `<stream>` as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the `<token list variable>` being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

```
a b c
```

results in a token list `a b c` with the letters `a`, `b`, and `c` having category code 12. In the non-branching version, where the `<stream>` is not open the `<tl var>` is set to `\q_no_value`.

T_EXhackers note: This protected macro is a wrapper around the ε -T_EX primitive `\readline`. Regardless of settings, T_EX removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of `\ior_str_get:NN`.

All mappings are done at the current group level, *i.e.* any local assignments made

by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

$\backslash\text{ior_map_inline:Nn}$ $\backslash\text{ior_map_inline:Nn}$ $\langle stream \rangle$ $\{\langle inline\ function \rangle\}$

New: 2012-02-11 Applies the $\langle inline\ function \rangle$ to each set of $\langle lines \rangle$ obtained by calling $\backslash\text{ior_get:NN}$ until reaching the end of the file. T_EX ignores any trailing new-line marker from the file it reads. The $\langle inline\ function \rangle$ should consist of code which receives the $\langle line \rangle$ as #1.

$\backslash\text{ior_str_map_inline:Nn}$ $\backslash\text{ior_str_map_inline:Nn}$ $\langle stream \rangle$ $\{\langle inline\ function \rangle\}$

New: 2012-02-11 Applies the $\langle inline\ function \rangle$ to every $\langle line \rangle$ in the $\langle stream \rangle$. The material is read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The $\langle inline\ function \rangle$ should consist of code which receives the $\langle line \rangle$ as #1. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads.

$\backslash\text{ior_map_variable:NNn}$ $\backslash\text{ior_map_variable:NNn}$ $\langle stream \rangle$ $\langle tl\ var \rangle$ $\{\langle code \rangle\}$

New: 2019-01-13 For each set of $\langle lines \rangle$ obtained by calling $\backslash\text{ior_get:NN}$ until reaching the end of the file, stores the $\langle lines \rangle$ in the $\langle tl\ var \rangle$ then applies the $\langle code \rangle$. The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last set of $\langle lines \rangle$, or its original value if the $\langle stream \rangle$ is empty. T_EX ignores any trailing new-line marker from the file it reads. This function is typically faster than $\backslash\text{ior_map_inline:Nn}$.

$\backslash\text{ior_str_map_variable:NNn}$ $\backslash\text{ior_str_map_variable:NNn}$ $\langle stream \rangle$ $\langle variable \rangle$ $\{\langle code \rangle\}$

New: 2019-01-13 For each $\langle line \rangle$ in the $\langle stream \rangle$, stores the $\langle line \rangle$ in the $\langle variable \rangle$ then applies the $\langle code \rangle$. The material is read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last $\langle line \rangle$, or its original value if the $\langle stream \rangle$ is empty. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads. This function is typically faster than $\backslash\text{ior_str_map_inline:Nn}$.

`\ior_map_break:` `\ior_map_break:`

New: 2012-06-29

Used to terminate a `\ior_map...` function before all lines from the $\langle stream \rangle$ have been processed. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map...` scenario leads to low level \TeX errors.

\TeX hackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\ior_map_break:n` `\ior_map_break:n {<code>}`

New: 2012-06-29

Used to terminate a `\ior_map...` function before all lines in the $\langle stream \rangle$ have been processed, inserting the $\langle code \rangle$ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map...` scenario leads to low level \TeX errors.

\TeX hackers note: When the mapping is broken, additional tokens may be inserted before the $\langle code \rangle$ is inserted into the input stream. This depends on the design of the mapping function.

`\ior_if_eof_p:N` \star `\ior_if_eof_p:N` $\langle stream \rangle$

`\ior_if_eof:NTF` \star `\ior_if_eof:NTF` $\langle stream \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$

Updated: 2012-02-10

Tests if the end of a file $\langle stream \rangle$ has been reached during a reading operation. The test also returns a `true` value if the $\langle stream \rangle$ is not open.

12.1.2 Reading from the terminal

<code>\ior_get_term:nN</code>	<code>\ior_get_term:nN {⟨prompt⟩} ⟨token list variable⟩</code>
<code>\ior_str_get_term:nN</code>	Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the <code>⟨token list⟩</code> variable. Tokenization occurs as described for <code>\ior_get:NN</code> or <code>\ior_str_get:NN</code> , respectively. When the <code>⟨prompt⟩</code> is empty, <code>TeX</code> will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. <code>\iow_term:n</code> . Where the <code>⟨prompt⟩</code> is given, it will appear in the terminal followed by an =, e.g.
<small>New: 2019-03-23</small>	

prompt=

12.1.3 Writing to files

<code>\iow_now:Nn</code>	<code>\iow_now:Nn ⟨stream⟩ {⟨tokens⟩}</code>
<code>\iow_now:(NV Ne cn cV ce)</code>	This function writes <code>⟨tokens⟩</code> to the specified <code>⟨stream⟩</code> immediately (<i>i.e.</i> the write operation is called on expansion of <code>\iow_now:Nn</code>).
<small>Updated: 2012-06-05</small>	

<code>\iow_log:n</code>	<code>\iow_log:n {⟨tokens⟩}</code>
<code>\iow_log:e</code>	This function writes the given <code>⟨tokens⟩</code> to the log (transcript) file immediately: it is a dedicated version of <code>\iow_now:Nn</code> .

<code>\iow_term:n</code>	<code>\iow_term:n {⟨tokens⟩}</code>
<code>\iow_term:e</code>	This function writes the given <code>⟨tokens⟩</code> to the terminal file immediately: it is a dedicated version of <code>\iow_now:Nn</code> .

<code>\iow_shipout:Nn</code>	<code>\iow_shipout:Nn ⟨stream⟩ {⟨tokens⟩}</code>
<code>\iow_shipout:(Ne cn ce)</code>	This function writes <code>⟨tokens⟩</code> to the specified <code>⟨stream⟩</code> when the current page is finalised (<i>i.e.</i> at shipout). The e-type variants expand the <code>⟨tokens⟩</code> at the point where the function is used but <i>not</i> when the resulting tokens are written to the <code>⟨stream⟩</code> (<i>cf.</i> <code>\iow_shipout_e:Nn</code>).

TeXhackers note: When using `expl3` with a format other than `LATeX`, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

<code>\iow_shipout_e:Nn</code> <code>\iow_shipout_e:(Ne cn ce)</code>	<code>\iow_shipout_e:Nn <stream> {<tokens>}</code>
<small>Updated: 2023-09-17</small>	<p>This function writes <code><tokens></code> to the specified <code><stream></code> when the current page is finalised (<i>i.e.</i> at shipout). The <code><tokens></code> are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).</p>

T_EXhackers note: This is a wrapper around the T_EX primitive `\write`. When using `expl3` with a format other than L^AT_EX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

<code>\iow_char:N *</code>	<code>\iow_char:N \<char></code>
	<p>Inserts <code><char></code> into the output stream. Useful when trying to write difficult characters such as <code>%</code>, <code>{</code>, <code>}</code>, <i>etc.</i> in messages, for example:</p>

```
\iow_now:Ne \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

<code>\iow_newline: *</code>	<code>\iow_newline:</code>
	<p>Function to add a new line within the <code><tokens></code> written to a file. The function has no effect if writing is taking place without expansion (<i>e.g.</i> in the second argument of <code>\iow_now:Nn</code>).</p>

T_EXhackers note: When using `expl3` with a format other than L^AT_EX, the character inserted by `\iow_newline:` is not recognized by T_EX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_e:Nn` and direct uses of primitive operations.

12.1.4 Wrapping lines in output

`\iow_wrap:nnnN` `\iow_wrap:nnnN` `{<text>}` `{<run-on text>}` `{<set up>}` `<function>`

`\iow_wrap:nenN`

New: 2012-06-28
Updated: 2017-12-04

This function wraps the `<text>` to a fixed number of characters per line. At the start of each line which is wrapped, the `<run-on text>` is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the `<run-on text>` for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The `<text>` and `<run-on text>` are exhaustively expanded by the function, with the following substitutions:

- `\` or `\iow_newline`: may be used to force a new line,
- `_` may be used to represent a forced space (for example after a control sequence),
- `\#`, `\%`, `\{`, `\}`, `\~` may be used to represent the corresponding character,
- `\iow_wrap_allow_break`: may be used to allow a line-break without inserting a space,
- `\iow_indent:n` may be used to indent a part of the `<text>` (not the `<run-on text>`).

Additional functions may be added to the wrapping by using the `<set up>`, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the `<text>` which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, *etc.*

The result of the wrapping operation is passed as a braced argument to the `<function>`, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the `<function>`) consists of characters of category “other” (category code 12), with the exception of spaces which have category “space” (category code 10). This means that the output does *not* expand further when written to a file.

T_EXhackers note: Internally, `\iow_wrap:nnnN` carries out an e-type expansion on the `<text>` to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the `<text>`.

`\iow_wrap_allow_break`: `\iow_wrap_allow_break`:

New: 2023-04-25

In the first argument of `\iow_wrap:nnnN` (for instance in messages), inserts a break-point that allows a line break. If no break occurs, this function adds nothing to the output.

`\iow_indent:n` `\iow_indent:n` `{<text>}`

New: 2011-09-21

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents `<text>` by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the `<text>`. In case the indented `<text>` should appear on separate lines from the surrounding text, use `\` to force line breaks.

<code>\l_iow_line_count_int</code> <small>New: 2012-06-24</small>	The maximum number of characters in a line to be written by the <code>\iow_wrap:nnnN</code> function. This value depends on the T _E X system in use: the standard value is 78, which is typically correct for unmodified T _E X Live and MiK _T E _X systems.
--	--

12.1.5 Constant input–output streams, and variables

<code>\g_tmpa_iow</code> <code>\g_tmpb_iow</code> <small>New: 2017-12-11</small>	Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	--

<code>\c_log_iow</code> <code>\c_term_iow</code>	Constant output streams for writing to the log and to the terminal (plus the log), respectively.
---	--

<code>\g_tmpa_iow</code> <code>\g_tmpb_iow</code> <small>New: 2017-12-11</small>	Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	---

12.1.6 Primitive conditionals

<code>\if_eof:w</code> ★	<pre> \if_eof:w <stream> <true code> \else: <false code> \fi: </pre> <p>Tests if the <code><stream></code> returns “end of file”, which is true for non-existent files. The <code>\else:</code> branch is optional.</p>
--------------------------	---

T_EXhackers note: This is the T_EX primitive `\ifeof`.

12.2 File operations

12.2.1 Basic file operations

<code>\g_file_curr_dir_str</code> <code>\g_file_curr_name_str</code> <code>\g_file_curr_ext_str</code> <small>New: 2017-06-21</small>	Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (<i>i.e.</i> if it is in the T _E X search path), and does <i>not</i> end in / other than the case that it is exactly equal to the root directory. The <code><name></code> and <code><ext></code> parts together make up the file name, thus the <code><name></code> part may be thought of as the “job name” for the current file.
--	--

Note that T_EX does not provide information on the `<dir>` and `<ext>` part for the main (top level) file and that this file always has empty `<dir>` and `<ext>` components. Also, the `<name>` here will be equal to `\c_sys_jobname_str`, which may be different from the real file name (if set using `--jobname`, for example).

`\l_file_search_path_seq` Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and need not include the trailing slash. Spaces need not be quoted.

New: 2017-06-18
Updated: 2023-06-15

TeXhackers note: When working as a package in L^AT_EX 2_ε, `expl3` will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

`\file_if_exist_p:n` ☆ `\file_if_exist_p:n {<file name>}`
`\file_if_exist_p:V` ☆ `\file_if_exist:nTF {<file name>} {<>true code>} {<>false code>}`
`\file_if_exist:nTF` ☆ Tests if `<file name>` is found in the path as detailed for `\file_if_exist:nTF`.
`\file_if_exist:VTF` ☆

Updated: 2023-09-18

12.2.2 Information about files and file contents

Functions in this section return information about files as `expl3` `str` data, *except* that the non-expandable functions set their return *token list* to `\q_no_value` if the file requested is not found. As such, comparison of file names, hashes, sizes, etc., should use `\str_if_eq:nnTF` rather than `\tl_if_eq:nnTF` and so on.

`\file_hex_dump:n` ☆ `\file_hex_dump:n {<file name>}`
`\file_hex_dump:V` ☆ `\file_hex_dump:nnn {<file name>} {<start index>} {<end index>}`
`\file_hex_dump:nnn` ☆ Searches for `<file name>` using the current T_EX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most T_EX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The `{<start index>}` and `{<end index>}` values work as described for `\str_range:nnn`.
`\file_hex_dump:Vnn` ☆

New: 2019-11-19

`\file_get_hex_dump:nN` `\file_get_hex_dump:nN {<file name>} <tl var>`
`\file_get_hex_dump:VN` `\file_get_hex_dump:nnnN {<file name>} {<start index>} {<end index>} <tl var>`
`\file_get_hex_dump:nNTF` Sets the `<tl var>` to the result of applying `\file_hex_dump:n/\file_hex_dump:nnn` to
`\file_get_hex_dump:VNTF` the `<file>`. If the file is not found, the `<tl var>` will be set to `\q_no_value`.
`\file_get_hex_dump:nnnN`
`\file_get_hex_dump:VnnN`
`\file_get_hex_dump:nnnNTF`
`\file_get_hex_dump:VnnNTF`

New: 2019-11-19

`\file_md5hash:n` ☆ `\file_md5hash:n {<file name>}`
`\file_md5hash:V` ☆
 New: 2019-09-03

Searches for `<file name>` using the current \TeX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most \TeX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.

`\file_get_md5hash:nN` `\file_get_md5hash:nN {<file name>} <t1 var>`
`\file_get_md5hash:VN`
`\file_get_md5hash:nNTF` Sets the `<t1 var>` to the result of applying `\file_md5hash:n` to the `<file>`. If the
`\file_get_md5hash:VNTF` file is not found, the `<t1 var>` will be set to `\q_no_value`.
 New: 2017-07-11
 Updated: 2019-02-16

`\file_size:n` ☆ `\file_size:n {<file name>}`
`\file_size:V` ☆
 New: 2019-09-03

Searches for `<file name>` using the current \TeX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the size of the file in bytes in the input stream. When the file is not found, the result of expansion is empty.

`\file_get_size:nN` `\file_get_size:nN {<file name>} <t1 var>`
`\file_get_size:VN`
`\file_get_size:nNTF` Sets the `<t1 var>` to the result of applying `\file_size:n` to the `<file>`. If the file is not
`\file_get_size:VNTF` found, the `<t1 var>` will be set to `\q_no_value`. This is not available in older versions of \XeTeX .
 New: 2017-07-09
 Updated: 2019-02-16

`\file_timestamp:n` ☆ `\file_timestamp:n {<file name>}`
`\file_timestamp:V` ☆
 New: 2019-09-03

Searches for `<file name>` using the current \TeX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form `D:<year><month><day><hour><minute><second><offset>`, where the latter may be `Z` (UTC) or `<plus-minus><hours>'<minutes>'`. When the file is not found, the result of expansion is empty. This is not available in older versions of \XeTeX .

`\file_get_timestamp:nN` `\file_get_timestamp:nN {<file name>} <t1 var>`
`\file_get_timestamp:VN`
`\file_get_timestamp:nNTF` Sets the `<t1 var>` to the result of applying `\file_timestamp:n` to the `<file>`. If the
`\file_get_timestamp:VNTF` file is not found, the `<t1 var>` will be set to `\q_no_value`. This is not available in older versions of \XeTeX .
 New: 2017-07-09
 Updated: 2019-02-16

```

\file_compare_timestamp_p:nNn      * \file_compare_timestamp_p:nNn {<file-1>} <comparator>
\file_compare_timestamp_p:(nNV|VNn|VNV) * {<file-2>}
\file_compare_timestamp:nNnTF      * \file_compare_timestamp:nNnTF {<file-1>} <comparator>
\file_compare_timestamp:(nNV|VNn|VNV)TF * {<file-2>} {<>true code>} {<>false code>}

```

New: 2019-05-13

Updated: 2019-09-20

Compares the file stamps on the two $\langle files \rangle$ as indicated by the $\langle comparator \rangle$, and inserts either the $\langle true code \rangle$ or $\langle false case \rangle$ as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```

\file_compare_timestamp:nNnT { source-file } > { derived-file }
{
  % Code to regenerate derived file
}

```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different. This is not available in older versions of Xe_{La}TeX.

```

\file_get_full_name:nN   \file_get_full_name:nN {<file name>} <tl>
\file_get_full_name:VN   \file_get_full_name:nNTF {<file name>} <tl> {<>true code>} {<>false code>}

```

```

\file_get_full_name:nNTF
\file_get_full_name:VNTF

```

Searches for $\langle file name \rangle$ in the path as detailed for $\backslash file_if_exist:nTF$, and if found sets the $\langle tl var \rangle$ the fully-qualified name of the file, *i.e.* the path and file name. This includes an extension `.tex` when the given $\langle file name \rangle$ has no extension but the file found has that extension. In the non-branching version, the $\langle tl var \rangle$ will be set to $\backslash q_no_value$ in the case that the file does not exist.

Updated: 2019-02-16

```

\file_full_name:n ☆ \file_full_name:n {<file name>}
\file_full_name:V ☆

```

New: 2019-09-23

Searches for $\langle file name \rangle$ in the path as detailed for $\backslash file_if_exist:nTF$, and if found leaves the fully-qualified name of the file, *i.e.* the path and file name, in the input stream. This includes an extension `.tex` when the given $\langle file name \rangle$ has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.

```

\file_parse_full_name:nNNN \file_parse_full_name:nNNN {<full name>} <dir> <name> <ext>
\file_parse_full_name:VNNN

```

New: 2017-06-23

Updated: 2020-06-24

Parses the $\langle full name \rangle$ and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The $\langle dir \rangle$: everything up to the last / (path separator) in the $\langle file path \rangle$. As with system PATH variables and related functions, the $\langle dir \rangle$ does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), $\langle dir \rangle$ is empty.
- The $\langle name \rangle$: everything after the last / up to the last ., where both of those characters are optional. The $\langle name \rangle$ may contain multiple . characters. It is empty if $\langle full name \rangle$ consists only of a directory name.
- The $\langle ext \rangle$: everything after the last . (including the dot). The $\langle ext \rangle$ is empty if there is no . after the last /.

Before parsing, the $\langle full name \rangle$ is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (") are invalid in file names and are discarded from the input.

<code>\file_parse_full_name:n</code> *	<code>\file_parse_full_name:n</code> { <i>⟨full name⟩</i> }
<code>\file_parse_full_name:V</code> *	Parses the <i>⟨full name⟩</i> as described for <code>\file_parse_full_name:nNNN</code> , and leaves
New: 2020-06-24	<i>⟨dir⟩</i> , <i>⟨name⟩</i> , and <i>⟨ext⟩</i> in the input stream, each inside a pair of braces.

<code>\file_parse_full_name_apply:nN</code> *	<code>\file_parse_full_name_apply:nN</code> { <i>⟨full name⟩</i> } <i>⟨function⟩</i>
<code>\file_parse_full_name_apply:VN</code> *	
New: 2020-06-24	

Parses the *⟨full name⟩* as described for `\file_parse_full_name:nNNN`, and passes *⟨dir⟩*, *⟨name⟩*, and *⟨ext⟩* as arguments to *⟨function⟩*, as an n-type argument each, in this order.

12.2.3 Accessing file contents

<code>\file_get:nnN</code>	<code>\file_get:nnN</code> { <i>⟨file name⟩</i> } { <i>⟨setup⟩</i> } <i>⟨t1⟩</i>
<code>\file_get:VnN</code>	<code>\file_get:nnNTF</code> { <i>⟨file name⟩</i> } { <i>⟨setup⟩</i> } <i>⟨t1⟩</i> { <i>⟨true code⟩</i> } { <i>⟨false code⟩</i> }
<code>\file_get:nnNTF</code>	Defines <i>⟨t1⟩</i> to the contents of <i>⟨file name⟩</i> . Category codes may need to be set appropriately via the <i>⟨setup⟩</i> argument. The non-branching version sets the <i>⟨t1⟩</i> to <code>\q_no_value</code> if the file is not found. The branching version runs the <i>⟨true code⟩</i> after the assignment to <i>⟨t1⟩</i> if the file is found, and <i>⟨false code⟩</i> otherwise. The file content will be tokenized using the current category code régime,
<code>\file_get:VnNTF</code>	
New: 2019-01-16	
Updated: 2019-02-16	

<code>\file_input:n</code>	<code>\file_input:n</code> { <i>⟨file name⟩</i> }
<code>\file_input:V</code>	Searches for <i>⟨file name⟩</i> in the path as detailed for <code>\file_if_exist:nTF</code> , and if found reads in the file as additional L ^A T _E X source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.
Updated: 2017-06-26	

<code>\file_input_raw:n</code> *	<code>\file_input_raw:n</code> { <i>⟨file name⟩</i> }
<code>\file_input_raw:V</code> *	Searches for <i>⟨file name⟩</i> in the path as detailed for <code>\file_if_exist:nTF</code> , and if found reads in the file as additional T _E X source. No data concerning the file is tracked. If the file is not found, no action is taken.
New: 2023-05-18	

T_EXhackers note: This function is intended only for contexts where files must be read purely by expansion, for example at the start of a table cell in an `\halign`.

<code>\file_if_exist_input:n</code>	<code>\file_if_exist_input:n</code> { <i>⟨file name⟩</i> }
<code>\file_if_exist_input:V</code>	<code>\file_if_exist_input:nF</code> { <i>⟨file name⟩</i> } { <i>⟨false code⟩</i> }
<code>\file_if_exist_input:nF</code>	Searches for <i>⟨file name⟩</i> using the current T _E X search path and the additional paths included in <code>\l_file_search_path_seq</code> . If found then reads in the file as additional L ^A T _E X source as described for <code>\file_input:n</code> , otherwise inserts the <i>⟨false code⟩</i> . Note that these functions do not raise an error if the file is not found, in contrast to <code>\file_input:n</code> .
<code>\file_if_exist_input:VF</code>	
New: 2014-07-02	

`\file_input_stop:` `\file_input_stop:`

New: 2017-07-07

Ends the reading of a file started by `\file_input:n` or similar before the end of the file is reached. Where the file reading is being terminated due to an error, `\msg_critical:nn(nn)` should be preferred.

TeXhackers note: This function must be used on a line on its own: TeX reads files line-by-line and so any additional tokens in the “current” line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn’t help as it is the line where it is used that counts!

`\file_show_list:` `\file_show_list:`

`\file_log_list:` `\file_log_list:`

These functions list all files loaded by L^AT_EX 2_ε commands that populate `\@filelist` or by `\file_input:n`. While `\file_show_list:` displays the list in the terminal, `\file_log_list:` outputs it to the log file only.

Chapter 13

The `l3luatex` module Lua_{TeX}-specific functions

The Lua_{TeX} engine provides access to the Lua programming language, and with it access to the “internals” of _{TeX}. In order to use this within the framework provided here, a family of functions is available. When used with pdf_{TeX}, p_{TeX}, up_{TeX} or X_{TeX} these raise an error: use `\sys_if_engine luatex:T` to avoid this. Details on using Lua with the Lua_{TeX} engine are given in the Lua_{TeX} manual.

13.1 Breaking out to Lua

`\lua_now:n` ★ `\lua_now:n` {*token list*}

`\lua_now:e` ★

New: 2018-06-18

The *token list* is first tokenized by _{TeX}, which includes converting line ends to spaces in the usual _{TeX} manner and which respects currently-applicable _{TeX} category codes. The resulting *Lua input* is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the *Lua input* immediately, and in an expandable manner.

_{TeX}hackers note: `\lua_now:e` is a macro wrapper around `\directlua`: when Lua_{TeX} is in use two expansions are required to yield the result of the Lua code.

`\lua_shipout_e:n` `\lua_shipout:n` {*token list*}

`\lua_shipout:n`

New: 2018-06-18

The *token list* is first tokenized by _{TeX}, which includes converting line ends to spaces in the usual _{TeX} manner and which respects currently-applicable _{TeX} category codes. The resulting *Lua input* is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the *Lua input* during the page-building routine: no _{TeX} expansion of the *Lua input* will occur at this stage.

In the case of the `\lua_shipout_e:n` version the input is fully expanded by _{TeX} in an e-type manner during the shipout operation.

_{TeX}hackers note: At a _{TeX} level, the *Lua input* is stored as a “whatsit”.

`\lua_escape:n` * `\lua_escape:n` {*token list*}

`\lua_escape:e` *

New: 2015-06-29

Converts the *token list* such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to `\n` and `\r`, respectively.

TeXhackers note: `\lua_escape:e` is a macro wrapper around `\luaescapestring:` when LuaTeX is in use two expansions are required to yield the result of the Lua code.

`\lua_load_module:n` `\lua_load_module:n` {*Lua module name*}

New: 2022-05-14

Loads a Lua module into the Lua interpreter.

`\lua_now:n` passes its {*token list*} argument to the Lua interpreter as a single line, with characters interpreted under the current catcode regime. These two facts mean that `\lua_now:n` rarely behaves as expected for larger pieces of code. Therefore, package authors should **not** write significant amounts of Lua code in the arguments to `\lua_now:n`. Instead, it is strongly recommended that they write the majority of their Lua code in a separate file, and then load it using `\lua_load_module:n`.

TeXhackers note: This is a wrapper around the Lua call `require` '*module*'.

13.2 Lua interfaces

As well as interfaces for TeX, there are a small number of Lua functions provided here.

`ltx.utils`

Most public interfaces provided by the module are stored within the `ltx.utils` table.

`ltx.utils.filedump` `<dump> = ltx.utils.filedump(<file>, <offset>, <length>)`

Returns the uppercase hexadecimal representation of the content of the *file* read as bytes. If the *length* is given, only this part of the file is returned; similarly, one may specify the *offset* from the start of the file. If the *length* is not given, the entire file is read starting at the *offset*.

`ltx.utils.filemd5sum` `<hash> = ltx.utils.filemd5sum(<file>)`

Returns the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal TeX behaviour. If the *file* is not found, nothing is returned with *no error raised*.

`ltx.utils.filemoddate` `<date> = ltx.utils.filemoddate(<file>)`

Returns the date/time of last modification of the *file* in the format

`D:<year><month><day><hour><minute><second><offset>`

where the latter may be Z (UTC) or `<plus-minus><hours>'<minutes>'`. If the *file* is not found, nothing is returned with *no error raised*.

`ltx.utils.filesize` `size = ltx.utils.filesize(<file>)`

Returns the size of the `<file>` in bytes. If the `<file>` is not found, nothing is returned with *no error raised*.

Chapter 14

The **l3**legacy module

Interfaces to legacy concepts

There are a small number of T_EX or L^AT_EX 2_ε concepts which are not used in expl3 code but which need to be manipulated when working as a L^AT_EX 2_ε package. To allow these to be integrated cleanly into expl3 code, a set of legacy interfaces are provided here.

```
\legacy_if_p:n * \legacy_if_p:n {<name>}
\legacy_if:nTF * \legacy_if:nTF {<name>} {<true code>} {<false code>}
```

Tests if the L^AT_EX 2_ε/plain T_EX conditional (generated by `\newif`) is true or false and branches accordingly. The `<name>` of the conditional should *omit* the leading `if`.

```
\legacy_if_set_true:n \legacy_if_set_true:n {<name>}
\legacy_if_set_false:n \legacy_if_set_false:n {<name>}
```

```
\legacy_if_gset_true:n Sets the LATEX 2ε/plain TEX conditional \if<name> (generated by \newif) to be true or
\legacy_if_gset_false:n false.
```

New: 2021-05-10

```
\legacy_if_set:nn \legacy_if_set:nn {<name>} {<boolexpr>}
\legacy_if_gset:nn
```

New: 2021-05-10

```
Sets the LATEX 2ε/plain TEX conditional \if<name> (generated by \newif) to the result of evaluating the <boolean expression>.
```

Part IV
Data types

Chapter 15

The `l3tl` module

Token lists

`TEX` works with tokens, and `LATEX3` therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called “token list variable”, which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, functions which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two “views” of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of “items”, or a list of “tokens”. An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or `␣`, `{`, or `}` (assuming normal `TEX` category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (`Hello`, `w`, `o`, `r`, `l` and `d`), but thirteen tokens (`{`, `H`, `e`, `l`, `l`, `o`, `}`, `␣`, `w`, `o`, `r`, `l` and `d`). Functions which act on items are often faster than their analogue acting directly on tokens.

15.1 Creating and initialising token list variables

```
\tl_new:N \tl_new:N <tl var>
```

```
\tl_new:c
```

Creates a new `<tl var>` or raises an error if the name is already taken. The declaration is global. The `<tl var>` is initially empty.

<code>\tl_const:Nn</code>	<code>\tl_const:Nn <tl var> {<tokens>}</code>
<code>\tl_const:(Ne cn ce)</code>	Creates a new constant <code><tl var></code> or raises an error if the name is already taken. The value of the <code><tl var></code> is set globally to the <code><tokens></code> .

<code>\tl_clear:N</code>	<code>\tl_clear:N <tl var></code>
<code>\tl_clear:c</code>	Clears all entries from the <code><tl var></code> .
<code>\tl_gclear:N</code>	
<code>\tl_gclear:c</code>	

<code>\tl_clear_new:N</code>	<code>\tl_clear_new:N <tl var></code>
<code>\tl_clear_new:c</code>	Ensures that the <code><tl var></code> exists globally by applying <code>\tl_new:N</code> if necessary, then applies <code>\tl_(g)clear:N</code> to leave the <code><tl var></code> empty.
<code>\tl_gclear_new:N</code>	
<code>\tl_gclear_new:c</code>	

<code>\tl_set_eq:NN</code>	<code>\tl_set_eq:NN <tl var1> <tl var2></code>
<code>\tl_set_eq:(cN Nc cc)</code>	Sets the content of <code><tl var1></code> equal to that of <code><tl var2></code> .
<code>\tl_gset_eq:NN</code>	
<code>\tl_gset_eq:(cN Nc cc)</code>	

<code>\tl_concat:NNN</code>	<code>\tl_concat:NNN <tl var1> <tl var2> <tl var3></code>
<code>\tl_concat:ccc</code>	Concatenates the content of <code><tl var2></code> and <code><tl var3></code> together and saves the result in <code><tl var1></code> . The <code><tl var2></code> is placed at the left side of the new token list.
<code>\tl_gconcat:NNN</code>	
<code>\tl_gconcat:ccc</code>	

New: 2012-05-18

<code>\tl_if_exist_p:N *</code>	<code>\tl_if_exist_p:N <tl var></code>
<code>\tl_if_exist_p:c *</code>	<code>\tl_if_exist:NTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_exist:NTF *</code>	Tests whether the <code><tl var></code> is currently defined. This does not check that the <code><tl var></code> really is a token list variable.
<code>\tl_if_exist:cTF *</code>	

New: 2012-03-03

15.2 Adding data to token list variables

<code>\tl_set:Nn</code>	<code>\tl_set:Nn <tl var> {<tokens>}</code>
<code>\tl_set:(NV Nv No Ne Nf cn cV cv co ce cf)</code>	Sets <code><tl var></code> to contain <code><tokens></code> , removing any previous content from the variable.
<code>\tl_gset:Nn</code>	
<code>\tl_gset:(NV Nv No Ne Nf cn cV cv co ce cf)</code>	

<code>\tl_put_left:Nn</code>	<code>\tl_put_left:Nn <tl var> {<tokens>}</code>
<code>\tl_put_left:(NV Nv Ne No cn cV cv ce co)</code>	Appends <code><tokens></code> to the left side of the current content of <code><tl var></code> .
<code>\tl_gput_left:Nn</code>	
<code>\tl_gput_left:(NV Nv Ne No cn cV cv ce co)</code>	

```

\__tl_put_right:Nn \tl_put_right:Nn <tl var> {<tokens>}
\__tl_put_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\__tl_gput_right:Nn
\__tl_gput_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

```

Appends `<tokens>` to the right side of the current content of `<tl var>`.

15.3 Token list conditionals

```

\__tl_if_blank_p:n * \tl_if_blank_p:n {<token list>}
\__tl_if_blank_p:(e|V|o) * \tl_if_blank:nTF {<token list>} {<true code>} {<false code>}
\__tl_if_blank:nTF *
\__tl_if_blank:(e|V|o)TF * Tests if the <token list> consists only of blank spaces (i.e. contains no item). The test

```

Updated: 2019-09-04

is `true` if `<token list>` is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is `false` otherwise.

```

\__tl_if_empty_p:N * \tl_if_empty_p:N <tl var>
\__tl_if_empty_p:c * \tl_if_empty:NNTF <tl var> {<true code>} {<false code>}
\__tl_if_empty:NNTF *
\__tl_if_empty:cTF * Tests if the <tl var> is entirely empty (i.e. contains no tokens at all).

```

```

\__tl_if_empty_p:n * \tl_if_empty_p:n {<token list>}
\__tl_if_empty_p:(V|o|e) * \tl_if_empty:nTF {<token list>} {<true code>} {<false code>}
\__tl_if_empty:nTF *
\__tl_if_empty:(V|o|e)TF * Tests if the <token list> is entirely empty (i.e. contains no tokens at all).

```

New: 2012-05-24
Updated: 2012-06-05

```

\__tl_if_eq_p:NN * \tl_if_eq_p:NN <tl var1> <tl var2>
\__tl_if_eq_p:(Nc|cN|cc) * \tl_if_eq:NNTF <tl var1> <tl var2> {<true code>} {<false code>}
\__tl_if_eq:NNTF *
\__tl_if_eq:(Nc|cN|cc)TF * Compares the content of <tl var1> and <tl var2> and is logically true if the two contain

```

the same list of tokens (*i.e.* identical in both the list of characters they contain and the category codes of those characters). Thus for example

```

\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Ne \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }

```

yields `false`. See also `\str_if_eq:nnTF` for a comparison that ignores category codes.

```

\__tl_if_eq:NnTF \tl_if_eq:NnTF <tl var1> {<token list2>} {<true code>} {<false code>}
\__tl_if_eq:cnTF

```

New: 2020-07-14

Tests if the `<tl var1>` and the `<token list2>` contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see `\tl_if_eq:NNTF` for an expandable version when both token lists are stored in variables, or `\str_if_eq:nnTF` if category codes are not important.

`\tl_if_eq:nnTF` `\tl_if_eq:nnTF {<token list1>} {<token list2>} {<>true code>} {<>false code>}`
`\tl_if_eq:(nV|ne|Vn|en|ee)TF` Tests if `<token list1>` and `<token list2>` contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see `\tl_if_eq:NNTF` for an expandable version when token lists are stored in variables, or `\str_if_eq:nnTF` if category codes are not important.

`\tl_if_in:NnTF` `\tl_if_in:NnTF <tl var> {<token list>} {<>true code>} {<>false code>}`
`\tl_if_in:(NV|No|cn|cV|co)TF` Tests if the `<token list>` is found in the content of the `<tl var>`. The `<token list>` cannot contain the tokens `{, }` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

`\tl_if_in:nnTF` `\tl_if_in:nnTF {<token list1>} {<token list2>} {<>true code>} {<>false code>}`
`\tl_if_in:(Vn|VW|on|oo|nV|no)TF` Tests if `<token list2>` is found inside `<token list1>`. The `<token list2>` cannot contain the tokens `{, }` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). The search does *not* enter brace (category code 1/2) groups.

`\tl_if_novalue_p:n *` `\tl_if_novalue_p:n {<token list>}`
`\tl_if_novalue:nTF *` `\tl_if_novalue:nTF {<token list>} {<>true code>} {<>false code>}`
New: 2017-11-14 Tests if the `<token list>` and the special `\c_novalue_tl` marker contain the same list of tokens, both in respect of character codes and category codes. This means that `\exp_args:No \tl_if_novalue:nTF { \c_novalue_tl }` is logically **true** but `\tl_if_novalue:nTF { \c_novalue_tl }` is logically **false**. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

`\tl_if_single_p:N *` `\tl_if_single_p:N <tl var>`
`\tl_if_single_p:c *` `\tl_if_single:NnTF <tl var> {<>true code>} {<>false code>}`
`\tl_if_single:NnTF *` Tests if the content of the `<tl var>` consists of a single `<item>`, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:N`.
`\tl_if_single:cTF *`

`\tl_if_single_p:n *` `\tl_if_single_p:n {<token list>}`
`\tl_if_single:nTF *` `\tl_if_single:nTF {<token list>} {<>true code>} {<>false code>}`
Updated: 2011-08-13 Tests if the `<token list>` has exactly one `<item>`, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:n`.

`\tl_if_single_token_p:n *` `\tl_if_single_token_p:n {<token list>}`
`\tl_if_single_token:nTF *` `\tl_if_single_token:nTF {<token list>} {<>true code>} {<>false code>}`
Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single normal token. Token groups (`{...}`) are not single tokens.

15.3.1 Testing the first token

```
\tl_if_head_eq_catcode_p:nN      * \tl_if_head_eq_catcode_p:nN {\token list} \test token}
\tl_if_head_eq_catcode_p:(VN|eN|oN) * \tl_if_head_eq_catcode:nNTF {\token list} \test token}
\tl_if_head_eq_catcode:nNTF      *   {\true code} {\false code}
\tl_if_head_eq_catcode:(VN|eN|oN)TF *
```

Updated: 2012-07-09

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same category code as the $\langle test token \rangle$. In the case where the $\langle token list \rangle$ is empty, the test is always **false**.

```
\tl_if_head_eq_charcode_p:nN      * \tl_if_head_eq_charcode_p:nN {\token list} \test token}
\tl_if_head_eq_charcode_p:(VN|eN|fN) * \tl_if_head_eq_charcode:nNTF {\token list} \test token}
\tl_if_head_eq_charcode:nNTF      *   {\true code} {\false code}
\tl_if_head_eq_charcode:(VN|eN|fN)TF *
```

Updated: 2012-07-09

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same character code as the $\langle test token \rangle$. In the case where the $\langle token list \rangle$ is empty, the test is always **false**.

```
\tl_if_head_eq_meaning_p:nN      * \tl_if_head_eq_meaning_p:nN {\token list} \test token}
\tl_if_head_eq_meaning_p:(VN|eN) * \tl_if_head_eq_meaning:nNTF {\token list} \test token}
\tl_if_head_eq_meaning:nNTF      *   {\true code} {\false code}
\tl_if_head_eq_meaning:(VN|eN)TF *
```

Updated: 2012-07-09

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same meaning as the $\langle test token \rangle$. In the case where $\langle token list \rangle$ is empty, the test is always **false**.

```
\tl_if_head_is_group_p:n * \tl_if_head_is_group_p:n {\token list}
\tl_if_head_is_group:nTF * \tl_if_head_is_group:nTF {\token list} {\true code} {\false code}
```

New: 2012-07-08

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ is an explicit begin-group character (with category code 1 and any character code), in other words, if the $\langle token list \rangle$ starts with a brace group. In particular, the test is **false** if the $\langle token list \rangle$ starts with an implicit token such as $\backslash c_group_begin_token$, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

```
\tl_if_head_is_N_type_p:n * \tl_if_head_is_N_type_p:n {\token list}
\tl_if_head_is_N_type:nTF * \tl_if_head_is_N_type:nTF {\token list} {\true code} {\false code}
```

New: 2012-07-08

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields **false**, as it does not have a normal first token. This function is useful to implement actions on token lists on a token by token basis.

```

\tl_if_head_is_space_p:n * \tl_if_head_is_space_p:n {<token list>}
\tl_if_head_is_space:nTF * \tl_if_head_is_space:nTF {<token list>} {<true code>} {<false code>}

```

Updated: 2012-07-08 Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ is an explicit space character (explicit token with character code 32 and category code 10). In particular, the test is **false** if the $\langle token list \rangle$ starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

15.4 Working with token lists as a whole

15.4.1 Using token lists

```

\tl_to_str:n * \tl_to_str:n {<token list>}
\tl_to_str:(o|V|v|e) *

```

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, leaving the resulting character tokens in the input stream. A $\langle string \rangle$ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). The base function requires only a single expansion. Its argument *must* be braced.

T_EXhackers note: This is the ε -T_EX primitive `\detokenize`. Converting a $\langle token list \rangle$ to a $\langle string \rangle$ yields a concatenation of the string representations of every token in the $\langle token list \rangle$. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not “letter”.

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally `#`). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character “backslash”, “lower-case a”, “space”, but it may also produce a single “lower-case a” if the escape character is negative and `a` is currently not a letter.

```

\tl_to_str:N * \tl_to_str:N <tl var>

```

```

\tl_to_str:c *

```

Converts the content of the $\langle tl var \rangle$ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This $\langle string \rangle$ is then left in the input stream. For low-level details, see the notes given for `\tl_to_str:n`.

```

\tl_use:N * \tl_use:N <tl var>

```

```

\tl_use:c *

```

Recovers the content of a $\langle tl var \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a $\langle tl var \rangle$ directly without an accessor function.

15.4.2 Counting and reversing token lists

<code>\tl_count:n</code>	* <code>\tl_count:n {⟨token list⟩}</code>
<code>\tl_count:(V v e o)</code>	* Counts the number of <code>⟨items⟩</code> in the <code>⟨token list⟩</code> and leaves this information in the input stream. Unbraced tokens count as one element as do each token group <code>{...}</code> . This process ignores any unprotected spaces within the <code>⟨token list⟩</code> . See also <code>\tl_count:N</code> . This function requires three expansions, giving an <code>⟨integer denotation⟩</code> .
New: 2012-05-13	
<code>\tl_count:N</code>	* <code>\tl_count:N ⟨tl var⟩</code>
<code>\tl_count:c</code>	* Counts the number of <code>⟨items⟩</code> in the <code>⟨tl var⟩</code> and leaves this information in the input stream. Unbraced tokens count as one element as do each token group <code>{...}</code> . This process ignores any unprotected spaces within the <code>⟨tl var⟩</code> . See also <code>\tl_count:n</code> . This function requires three expansions, giving an <code>⟨integer denotation⟩</code> .
New: 2012-05-13	
<code>\tl_count_tokens:n</code>	* <code>\tl_count_tokens:n {⟨token list⟩}</code>
New: 2019-02-25	Counts the number of \TeX tokens in the <code>⟨token list⟩</code> and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of <code>a~{bc}</code> is 6.
<code>\tl_reverse:n</code>	* <code>\tl_reverse:n {⟨token list⟩}</code>
<code>\tl_reverse:(V o f e)</code>	* Reverses the order of the <code>⟨items⟩</code> in the <code>⟨token list⟩</code> , so that <code>⟨item₁⟩⟨item₂⟩⟨item₃⟩...⟨item_n⟩</code> becomes <code>⟨item_n⟩...⟨item₃⟩⟨item₂⟩⟨item₁⟩</code> . This process preserves unprotected space within the <code>⟨token list⟩</code> . Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider <code>\tl_reverse_items:n</code> . See also <code>\tl_reverse:N</code> .
Updated: 2012-01-08	
\TeXhackers note: The result is returned within <code>\unexpanded</code> , which means that the token list does not expand further when appearing in an <code>e</code> -type or <code>x</code> -type argument expansion.	
<code>\tl_reverse:N</code>	* <code>\tl_reverse:N ⟨tl var⟩</code>
<code>\tl_reverse:c</code>	* Sets the <code>⟨tl var⟩</code> to contain the result of reversing the order of its <code>⟨items⟩</code> , so that <code>⟨item₁⟩⟨item₂⟩⟨item₃⟩...⟨item_n⟩</code> becomes <code>⟨item_n⟩...⟨item₃⟩⟨item₂⟩⟨item₁⟩</code> . This process preserves unprotected spaces within the <code>⟨tl var⟩</code> . Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. This is equivalent to a combination of an assignment and <code>\tl_reverse:V</code> . See also <code>\tl_reverse_items:n</code> for improved performance.
<code>\tl_greverse:N</code>	* <code>\tl_greverse:N ⟨tl var⟩</code>
<code>\tl_greverse:c</code>	* <code>\tl_greverse:c</code>
Updated: 2012-01-08	
<code>\tl_reverse_items:n</code>	* <code>\tl_reverse_items:n {⟨token list⟩}</code>
New: 2012-01-08	Reverses the order of the <code>⟨items⟩</code> in the <code>⟨token list⟩</code> , so that <code>⟨item₁⟩⟨item₂⟩⟨item₃⟩...⟨item_n⟩</code> becomes <code>{⟨item_n⟩} ... {⟨item₃⟩}{⟨item₂⟩}{⟨item₁⟩}</code> . This process removes any unprotected space within the <code>⟨token list⟩</code> . Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function <code>\tl_reverse:n</code> .

\TeX hackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `e`-type or `x`-type argument expansion.

<code>\tl_trim_spaces:n</code>	*	<code>\tl_trim_spaces:n {⟨token list⟩}</code>
<code>\tl_trim_spaces:(V v e o)</code>	*	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the <code>⟨token list⟩</code> and leaves the result in the input stream.
<small>New: 2011-07-09 Updated: 2012-06-25</small>		

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `e`-type or `x`-type argument expansion.

<code>\tl_trim_spaces_apply:nN</code>	*	<code>\tl_trim_spaces_apply:nN {⟨token list⟩} ⟨function⟩</code>
<code>\tl_trim_spaces_apply:oN</code>	*	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the <code>⟨token list⟩</code> and passes the result to the <code>⟨function⟩</code> as an <code>n</code> -type argument.
<small>New: 2018-04-12</small>		

<code>\tl_trim_spaces:N</code>		<code>\tl_trim_spaces:N ⟨tl var⟩</code>
<code>\tl_trim_spaces:c</code>		Sets the <code>⟨tl var⟩</code> to contain the result of removing any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.
<code>\tl_gtrim_spaces:N</code>		
<code>\tl_gtrim_spaces:c</code>		
<small>New: 2011-07-09</small>		

15.4.3 Viewing token lists

<code>\tl_show:N</code>		<code>\tl_show:N ⟨tl var⟩</code>
<code>\tl_show:c</code>		Displays the content of the <code>⟨tl var⟩</code> on the terminal.
<small>Updated: 2021-04-29</small>		

TeXhackers note: This is similar to the TeX primitive `\show`, wrapped to a fixed number of characters per line.

<code>\tl_show:n</code>		<code>\tl_show:n {⟨token list⟩}</code>
<code>\tl_show:e</code>		Displays the <code>⟨token list⟩</code> on the terminal.
<small>Updated: 2015-08-07</small>		

TeXhackers note: This is similar to the ϵ -TeX primitive `\showtokens`, wrapped to a fixed number of characters per line.

<code>\tl_log:N</code>		<code>\tl_log:N ⟨tl var⟩</code>
<code>\tl_log:c</code>		Writes the content of the <code>⟨tl var⟩</code> in the log file. See also <code>\tl_show:N</code> which displays the result in the terminal.
<small>New: 2014-08-22 Updated: 2021-04-29</small>		

<code>\tl_log:n</code>		<code>\tl_log:n {⟨token list⟩}</code>
<code>\tl_log:(e x)</code>		Writes the <code>⟨token list⟩</code> in the log file. See also <code>\tl_show:n</code> which displays the result in the terminal.
<small>New: 2014-08-22 Updated: 2015-08-07</small>		

15.5 Manipulating items in token lists

15.5.1 Mapping over token lists

All mappings are done at the current group level, *i.e.* any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

<code>\tl_map_function:NN</code> ☆	<code>\tl_map_function:NN <tl var> <function></code>
<code>\tl_map_function:cN</code> ☆	Applies <code><function></code> to every <code><item></code> in the <code><tl var></code> . The <code><function></code> receives one argument for each iteration. This may be a number of tokens if the <code><item></code> was stored within braces. Hence the <code><function></code> should anticipate receiving n-type arguments. See also <code>\tl_map_function:nN</code> .
Updated: 2012-06-29	

<code>\tl_map_function:nN</code> ☆	<code>\tl_map_function:nN {(token list)} <function></code>
Updated: 2012-06-29	Applies <code><function></code> to every <code><item></code> in the <code><token list></code> , The <code><function></code> receives one argument for each iteration. This may be a number of tokens if the <code><item></code> was stored within braces. Hence the <code><function></code> should anticipate receiving n-type arguments. See also <code>\tl_map_function:NN</code> .

<code>\tl_map_inline:Nn</code>	<code>\tl_map_inline:Nn <tl var> {<inline function>}</code>
<code>\tl_map_inline:cN</code>	Applies the <code><inline function></code> to every <code><item></code> stored within the <code><tl var></code> . The <code><inline function></code> should consist of code which receives the <code><item></code> as #1. See also <code>\tl_map_function:NN</code> .
Updated: 2012-06-29	

<code>\tl_map_inline:nn</code>	<code>\tl_map_inline:nn {(token list)} {<inline function>}</code>
Updated: 2012-06-29	Applies the <code><inline function></code> to every <code><item></code> stored within the <code><token list></code> . The <code><inline function></code> should consist of code which receives the <code><item></code> as #1. See also <code>\tl_map_function:nN</code> .

<code>\tl_map_tokens:Nn</code> ☆	<code>\tl_map_tokens:Nn <tl var> {<code>}</code>
<code>\tl_map_tokens:cN</code> ☆	<code>\tl_map_tokens:nn {(token list)} {<code>}</code>
<code>\tl_map_tokens:nn</code> ☆	Analogue of <code>\tl_map_function:NN</code> which maps several tokens instead of a single function. The <code><code></code> receives each <code><item></code> in the <code><tl var></code> or in the <code><token list></code> as a trailing brace group. For instance,
New: 2019-09-02	

```
\tl_map_tokens:Nn \l_my_tl { \prg_replicate:nn { 2 } }
```

expands to twice each `<item>` in the `<tl var>`: for each `<item>` in `\l_my_tl` the function `\prg_replicate:nn` receives 2 and `<item>` as its two arguments. The function `\tl_map_inline:Nn` is typically faster but is not expandable.

<code>\tl_map_variable:NNn</code>	<code>\tl_map_variable:NNn <tl var> <variable> {<code>}</code>
<code>\tl_map_variable:cNn</code>	Stores each <code><item></code> of the <code><tl var></code> in turn in the (token list) <code><variable></code> and applies the <code><code></code> . The <code><code></code> will usually make use of the <code><variable></code> , but this is not enforced. The assignments to the <code><variable></code> are local. Its value after the loop is the last <code><item></code> in the <code><tl var></code> , or its original value if the <code><tl var></code> is blank. See also <code>\tl_map_inline:Nn</code> .
Updated: 2012-06-29	

`\tl_map_variable:nNn` `\tl_map_variable:nNn {<token list>} <variable> {<code>}`
Updated: 2012-06-29 Stores each *<item>* of the *<token list>* in turn in the (token list) *<variable>* and applies the *<code>*. The *<code>* will usually make use of the *<variable>*, but this is not enforced. The assignments to the *<variable>* are local. Its value after the loop is the last *<item>* in the *<tl var>*, or its original value if the *<tl var>* is blank. See also `\tl_map_inline:nn`.

`\tl_map_break: ☆` `\tl_map_break:`
Updated: 2012-06-29 Used to terminate a `\tl_map...` function before all entries in the *<token list>* have been processed. This normally takes place within a conditional statement, for example

```

\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: }
  % Do something useful
}

```

See also `\tl_map_break:n`. Use outside of a `\tl_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\tl_map_break:n ☆` `\tl_map_break:n {<code>}`
Updated: 2012-06-29 Used to terminate a `\tl_map...` function before all entries in the *<token list>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```

\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo }
  { \tl_map_break:n { <code> } }
  % Do something useful
}

```

Use outside of a `\tl_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

15.5.2 Head and tail of token lists

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

<code>\tl_head:N</code>	<code>*</code>	<code>\tl_head:n {⟨token list⟩}</code>
<code>\tl_head:n</code>	<code>*</code>	Leaves in the input stream the first <i>⟨item⟩</i> in the <i>⟨token list⟩</i> , discarding the rest of the <i>⟨token list⟩</i> . All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example
<code>\tl_head:(V v f)</code>	<code>*</code>	

Updated: 2012-09-09

`\tl_head:n { abc }`

and

`\tl_head:n { ~ abc }`

both leave `a` in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so

`\tl_head:n { ~ { ~ ab } c }`

yields `▯ab`. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) results in `\tl_head:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

<code>\tl_head:w</code>	<code>*</code>	<code>\tl_head:w ⟨token list⟩ { } \q_stop</code>
-------------------------	----------------	--

Leaves in the input stream the first *⟨item⟩* in the *⟨token list⟩*, discarding the rest of the *⟨token list⟩*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank *⟨token list⟩* (which consists only of space characters) results in a low-level TeX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a “blank” argument. This function requires only a single expansion, and thus is suitable for use within an o-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

<code>\tl_tail:N</code>	<code>*</code>	<code>\tl_tail:n {⟨token list⟩}</code>
<code>\tl_tail:n</code>	<code>*</code>	Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first <i>⟨item⟩</i> in the <i>⟨token list⟩</i> , and leaves the remaining tokens in the input stream. Thus for example
<code>\tl_tail:(V v f)</code>	<code>*</code>	

Updated: 2012-09-01

`\tl_tail:n { a ~ {bc} d }`

and

`\tl_tail:n { ~ a ~ {bc} d }`

both leave `▯{bc}d` in the input stream. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) results in `\tl_tail:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

If you wish to handle token lists where the first token may be a space, and this

needs to be treated as the head/tail, this can be accomplished using `\tl_if_head_is_space:nTF`, for example

```

\exp_last_unbraced:NNo
  \cs_new:Npn \__mypkg_gobble_space:w \c_space_tl { }
\cs_new:Npn \mypkg_tl_head_keep_space:n #1
  {
    \tl_if_head_is_space:nTF {#1}
      { ~ }
      { \tl_head:n {#1} }
  }
\cs_new:Npn \mypkg_tl_tail_keep_space:n #1
  {
    \tl_if_head_is_space:nTF {#1}
      { \exp_not:o { \__mypkg_gobble_space:w #1 } }
      { \tl_tail:n {#1} }
  }

```

15.5.3 Items and ranges in token lists

```
\tl_item:nn * \tl_item:nn {<token list>} {<integer expression>}
```

```
\tl_item:Nn * Indexing items in the <token list> from 1 on the left, this function evaluates the
\tl_item:cn * <integer expression> and leaves the appropriate item from the <token list> in the
New: 2014-07-17 input stream. If the <integer expression> is negative, indexing occurs from the right
of the token list, starting at -1 for the right-most item. If the index is out of bounds,
then the function expands to nothing.
```

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `e`-type or `x`-type argument expansion.

```
\tl_rand_item:N * \tl_rand_item:N <tl var>
```

```
\tl_rand_item:c * \tl_rand_item:n {<token list>}
```

```
\tl_rand_item:n * Selects a pseudo-random item of the <token list>. If the <token list> is blank, the
New: 2016-12-06 result is empty. This is not available in older versions of XƎTEX.
```

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `e`-type or `x`-type argument expansion.

```

\Tl_range:Nnn * \Tl_range:Nnn <tl var> {<start index>} {<end index>}
\Tl_range:nnn * \Tl_range:nnn {<token list>} {<start index>} {<end index>}

```

New: 2017-02-17 Leaves in the input stream the items from the $\langle \textit{start index} \rangle$ to the $\langle \textit{end index} \rangle$ inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here $\langle \textit{start index} \rangle$ and $\langle \textit{end index} \rangle$ should be $\langle \textit{integer expressions} \rangle$. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', and a negative index means 'from the right end'. Let l be the count of the token list.

Updated: 2017-07-15

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$.

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to `\Tl_trim_spaces:n` to begin with).

Thus, with $l = 7$ as in the examples below, all of the following are equivalent and result in the whole token list

```

\Tl_range:nnn { abcd~{e}}fg } { 1 } { 7 }
\Tl_range:nnn { abcd~{e}}fg } { 1 } { 12 }
\Tl_range:nnn { abcd~{e}}fg } { -7 } { 7 }
\Tl_range:nnn { abcd~{e}}fg } { -12 } { 7 }

```

Here are some more interesting examples. The calls

```

\iow_term:e { \Tl_range:nnn { abcd~{e}}fg } { 2 } { 5 } }
\iow_term:e { \Tl_range:nnn { abcd~{e}}fg } { 2 } { -3 } }
\iow_term:e { \Tl_range:nnn { abcd~{e}}fg } { -6 } { 5 } }
\iow_term:e { \Tl_range:nnn { abcd~{e}}fg } { -6 } { -3 } }

```

are all equivalent and will print `bcd{e}` on the terminal; similarly

```

\iow_term:e { \Tl_range:nnn { abcd~{e}}fg } { 2 } { 5 } }
\iow_term:e { \Tl_range:nnn { abcd~{e}}fg } { 2 } { -3 } }
\iow_term:e { \Tl_range:nnn { abcd~{e}}fg } { -6 } { 5 } }
\iow_term:e { \Tl_range:nnn { abcd~{e}}fg } { -6 } { -3 } }

```

are all equivalent and will print `bcd {e}` on the terminal (note the space in the middle). To the contrary,

```

\Tl_range:nnn { abcd~{e}}f } { 2 } { 4 }

```

will discard the space after 'd'.

If we want to get the items from, say, the third to the last in a token list $\langle \textit{tl} \rangle$, the call is `\Tl_range:nnn { <tl> } { 3 } { -1 }`. Similarly, for discarding the last item, we can do `\Tl_range:nnn { <tl> } { 1 } { -2 }`.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle \textit{item} \rangle$ does not expand further when appearing in an e-type or x-type argument expansion.

15.5.4 Sorting token lists

`\tl_sort:Nn` `\tl_sort:Nn` $\langle tl\ var \rangle$ $\{ \langle comparison\ code \rangle \}$
`\tl_sort:cn`
`\tl_gsort:Nn` `\tl_gsort:cn` Sorts the items in the $\langle tl\ var \rangle$ according to the $\langle comparison\ code \rangle$, and assigns the result to $\langle tl\ var \rangle$. The details of sorting comparison are described in Section 6.1.

New: 2017-02-06

`\tl_sort:nN` \star `\tl_sort:nN` $\{ \langle token\ list \rangle \}$ $\langle conditional \rangle$

New: 2017-02-06 Sorts the items in the $\langle token\ list \rangle$, using the $\langle conditional \rangle$ to compare items, and leaves the result in the input stream. The $\langle conditional \rangle$ should have signature `:nnTF`, and return `true` if the two items being compared should be left in the same order, and `false` if the items should be swapped. The details of sorting comparison are described in Section 6.1.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

15.6 Manipulating tokens in token lists

15.6.1 Replacing tokens

Within token lists, replacement takes place at the top level: there is no recursion into brace groups (more precisely, within a group defined by a category code 1/2 pair).

`\tl_replace_once:Nnn` `\tl_replace_once:Nnn` $\langle tl\ var \rangle$ $\{ \langle old\ tokens \rangle \}$ $\{ \langle new\ tokens \rangle \}$
`\tl_replace_once:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)`
`\tl_greplace_once:Nnn`
`\tl_greplace_once:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)`

Updated: 2011-08-11

Replaces the first (leftmost) occurrence of $\langle old\ tokens \rangle$ in the $\langle tl\ var \rangle$ with $\langle new\ tokens \rangle$. $\langle Old\ tokens \rangle$ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```

\l_tl_replace_all:Nnn          \l_tl_replace_all:Nnn <tl var> {<old tokens>} {<new
\l_tl_replace_all:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|
                             cne|cee)
\l_greplace_all:Nnn
\l_tl_greplace_all:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|
                    cne|cee)

```

Updated: 2011-08-11

Replaces all occurrences of `<old tokens>` in the `<tl var>` with `<new tokens>`. `<Old tokens>` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern `<old tokens>` may remain after the replacement (see `\l_tl_remove_all:Nn` for an example).

```

\l_tl_remove_once:Nn          \l_tl_remove_once:Nn <tl var> {<tokens>}
\l_tl_remove_once:(NV|Ne|cn|cV|ce)
\l_gremove_once:Nn
\l_tl_gremove_once:(NV|Ne|cn|cV|ce)

```

Updated: 2011-08-11

Removes the first (leftmost) occurrence of `<tokens>` from the `<tl var>`. The `<tokens>` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```

\l_tl_remove_all:Nn          \l_tl_remove_all:Nn <tl var> {<tokens>}
\l_tl_remove_all:(NV|Ne|cn|cV|ce)
\l_gremove_all:Nn
\l_tl_gremove_all:(NV|Ne|cn|cV|ce)

```

Updated: 2011-08-11

Removes all occurrences of `<tokens>` from the `<tl var>`. The `<tokens>` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern `<tokens>` may remain after the removal, for instance,

```
\l_tl_set:Nn \l_tmpa_tl {abbccd} \l_tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in `\l_tmpa_tl` containing `abcd`.

15.6.2 Reassigning category codes

These functions allow the rescanning of tokens: re-apply \TeX 's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using `\char_generate:nn`).

```

\tl_set_rescan:Nnn \tl_set_rescan:Nnn <tl var> {<setup>} {<tokens>}
\tl_set_rescan:(NnV|Nne|Nno|cnn|cnV|cne|cno)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(NnV|Nne|Nno|cnn|cnV|cne|cno)

```

Updated: 2015-08-11

Sets $\langle tl\ var\rangle$ to contain $\langle tokens\rangle$, applying the category code régime specified in the $\langle setup\rangle$ before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the $\langle setup\rangle$ are those in force at the point of use of $\backslash tl_set_rescan:Nnn$.) This allows the $\langle tl\ var\rangle$ to contain material with category codes other than those that apply when $\langle tokens\rangle$ are absorbed. The $\langle setup\rangle$ is run within a group and may contain any valid input, although only changes in category codes, such as uses of $\backslash cctab_select:N$, are relevant. See also $\backslash tl_rescan:nn$.

T_EXhackers note: The $\langle tokens\rangle$ are first turned into a string (using $\backslash tl_to_str:n$). If the string contains one or more characters with character code $\backslash newlinechar$ (set equal to $\backslash endlinechar$ unless that is equal to 32, before the user $\langle setup\rangle$), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

```

\tl_rescan:nn \tl_rescan:nn {<setup>} {<tokens>}

```

```

\tl_rescan:nV

```

Updated: 2015-08-11

Rescans $\langle tokens\rangle$ applying the category code régime specified in the $\langle setup\rangle$, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the $\langle setup\rangle$ are those in force at the point of use of $\backslash tl_rescan:nn$.) The $\langle setup\rangle$ is run within a group and may contain any valid input, although only changes in category codes, such as uses of $\backslash cctab_select:N$, are relevant. See also $\backslash tl_set_rescan:Nnn$, which is more robust than using $\backslash tl_set:Nn$ in the $\langle tokens\rangle$ argument of $\backslash tl_rescan:nn$.

T_EXhackers note: The $\langle tokens\rangle$ are first turned into a string (using $\backslash tl_to_str:n$). If the string contains one or more characters with character code $\backslash newlinechar$ (set equal to $\backslash endlinechar$ unless that is equal to 32, before the user $\langle setup\rangle$), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

Contrarily to the $\backslash scantokens$ ϵ -T_EX primitive, $\backslash tl_rescan:nn$ tokenizes the whole string in the same category code régime rather than one token at a time, so that directives such as $\backslash verb$ that rely on changing category codes will not function properly.

15.7 Constant token lists

```

\c_empty_tl

```

Constant that is always empty.

`\c_novalue_tl` A marker for the absence of an argument. This constant `tl` can safely be typeset (*cf.* `\q_nil`), with the result being `-NoValue-`. It is important to note that `\c_novalue_tl` is constructed such that it will *not* match the simple text input `-NoValue-`, *i.e.* that

New: 2017-11-14

`\tl_if_eq:NnTF \c_novalue_tl { -NoValue- }`

is logically `false`. The `\c_novalue_tl` marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty `tl`.

`\c_space_tl` An explicit space character contained in a token list (compare this with `\c_space_token`). For use where an explicit space is required.

15.8 Scratch token lists

`\l_tmpa_tl` Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\l_tmpb_tl`

`\g_tmpa_tl` Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpb_tl`

Chapter 16

The l3tl-build module

Piecewise t1 constructions

16.1 Constructing $\langle t1 var \rangle$ by accumulation

When creating a $\langle t1 var \rangle$ by accumulation of many tokens, the performance available using a combination of `\tl_set:Nn` and `\tl_put_right:Nn` or similar begins to become an issue. To address this, a set of functions are available to “build” a $\langle t1 var \rangle$. The performance of this approach is much more efficient than the standard `\tl_put_right:Nn`, but the constructed token list cannot be accessed during construction other than by methods provided in this section.

Whilst the exact performance difference is dependent on the size of each added block of tokens and the total number of blocks, in general, the `\tl_build_(g)put...` functions will out-perform the basic `\tl_(g)put...` equivalent if more than 100 non-empty addition operations occur. See <https://github.com/latex3/latex3/issues/1393#issuecomment-1880164756> for a more detailed analysis.

```
\tl_build_begin:N \tl_build_begin:N <t1 var>
```

```
\tl_build_gbegin:N
```

New: 2018-04-01

Clears the $\langle t1 var \rangle$ and sets it up to support other `\tl_build_...` functions. Until `\tl_build_end:N <t1 var>` or `\tl_build_gend:N <t1 var>` is called, applying any function from l3tl other than `\tl_build_...` will lead to incorrect results. The `begin` and `gbegin` functions must be used for local and global $\langle t1 var \rangle$ respectively.

```
\tl_build_put_left:Nn \tl_build_put_left:Nn <t1 var> {<tokens>}
```

```
\tl_build_put_left:Ne \tl_build_put_right:Nn <t1 var> {<tokens>}
```

```
\tl_build_gput_left:Nn
```

```
\tl_build_gput_left:Ne
```

```
\tl_build_put_right:Nn
```

```
\tl_build_put_right:Ne
```

```
\tl_build_gput_right:Nn
```

```
\tl_build_gput_right:Ne
```

New: 2018-04-01

Adds $\langle tokens \rangle$ to the left or right side of the current contents of $\langle t1 var \rangle$. The $\langle t1 var \rangle$ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `put` and `gput` functions must be used for local and global $\langle t1 var \rangle$ respectively. The `right` functions are about twice faster than the `left` functions.

`\tl_build_end:N` `\tl_build_end:N` $\langle t1\ var\rangle$

`\tl_build_gend:N`

New: 2018-04-01

Gets the contents of $\langle t1\ var\rangle$ and stores that into the $\langle t1\ var\rangle$ using `\tl_set:Nn` or `\tl_gset:Nn`. The $\langle t1\ var\rangle$ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `end` and `gend` functions must be used for local and global $\langle t1\ var\rangle$ respectively. These functions completely remove the setup code that enabled $\langle t1\ var\rangle$ to be used for other `\tl_build_...` functions. After the action of `end/gend`, the $\langle t1\ var\rangle$ may be manipulated using standard `tl` functions.

`\tl_build_get_intermediate:NN` `\tl_build_get_intermediate:NN` $\langle t1\ var_1\rangle$ $\langle t1\ var_2\rangle$

New: 2023-12-14

Stores the contents of the $\langle t1\ var_1\rangle$ in the $\langle t1\ var_2\rangle$. The $\langle t1\ var_1\rangle$ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The $\langle t1\ var_2\rangle$ is a “normal” token list variable, assigned locally using `\tl_set:Nn`.

Chapter 17

The `l3str` module Strings

`TeX` associates each character with a category code: as such, there is no concept of a “string” as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense “ignoring” category codes: this is done by treating token lists as strings in a `TeX` sense.

A `TeX` string (and thus an `expl3` string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a `TeX` string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn’t primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in `l3basics`, `l3tl` and `l3token`, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

17.1 Creating and initialising string variables

<code>\str_new:N</code>	<code>\str_new:N <str var></code>
<code>\str_new:c</code>	Creates a new <code><str var></code> or raises an error if the name is already taken. The declaration is global. The <code><str var></code> is initially empty.

New: 2015-09-18

<code>\str_const:Nn</code>	<code>\str_const:Nn <str var> {(token list)}</code>
<code>\str_const:(NV Ne cn cV ce)</code>	Creates a new constant <code><str var></code> or raises an error if the name is already taken. The value of the <code><str var></code> is set globally to the <code><token list></code> , converted to a string.

New: 2015-09-18
Updated: 2018-07-28

<code>\str_clear:N</code>	<code>\str_clear:N <str var></code>
<code>\str_clear:c</code>	Clears the content of the <code><str var></code> .
<code>\str_gclear:N</code>	
<code>\str_gclear:c</code>	

New: 2015-09-18

<code>\str_clear_new:N</code>	<code>\str_clear_new:N <str var></code>
<code>\str_clear_new:c</code>	Ensures that the <code><str var></code> exists globally by applying <code>\str_new:N</code> if necessary, then applies <code>\str_(g)clear:N</code> to leave the <code><str var></code> empty.
<code>\str_gclear_new:N</code>	
<code>\str_gclear_new:c</code>	

New: 2015-09-18

<code>\str_set_eq:NN</code>	<code>\str_set_eq:NN <str var₁₂</code>
<code>\str_set_eq:(cN Nc cc)</code>	Sets the content of <code><str var_{1 equal to that of <code><str var_{2.}</code>}</code>
<code>\str_gset_eq:NN</code>	
<code>\str_gset_eq:(cN Nc cc)</code>	

New: 2015-09-18

<code>\str_concat:NNN</code>	<code>\str_concat:NNN <str var₁₂₃</code>
<code>\str_concat:ccc</code>	Concatenates the content of <code><str var_{2 and <code><str var_{3 together and saves the result in <code><str var_{1. The <code><str var_{2 is placed at the left side of the new string variable. The <code><str var_{2 and <code><str var_{3 must indeed be strings, as this function does not convert their contents to a string.}</code>}</code>}</code>}</code>}</code>}</code>
<code>\str_gconcat:NNN</code>	
<code>\str_gconcat:ccc</code>	

New: 2017-10-08

<code>\str_if_exist_p:N *</code>	<code>\str_if_exist_p:N <str var></code>
<code>\str_if_exist_p:c *</code>	<code>\str_if_exist:NTF <str var> {(true code)} {(false code)}</code>
<code>\str_if_exist:NTF *</code>	Tests whether the <code><str var></code> is currently defined. This does not check that the <code><str var></code> really is a string.
<code>\str_if_exist:cTF *</code>	

New: 2015-09-18

17.2 Adding data to string variables

<code>\str_set:Nn</code>	<code>\str_set:Nn <str var> {<token list>}</code>
<code>\str_set:(NV Ne cn cV ce)</code>	Converts the <code><token list></code> to a <code><string></code> , and stores the result in <code><str var></code> .
<code>\str_gset:Nn</code>	
<code>\str_gset:(NV Ne cn cV ce)</code>	

New: 2015-09-18
Updated: 2018-07-28

<code>\str_put_left:Nn</code>	<code>\str_put_left:Nn <str var> {<token list>}</code>
<code>\str_put_left:(NV Ne cn cV ce)</code>	
<code>\str_gput_left:Nn</code>	
<code>\str_gput_left:(NV Ne cn cV ce)</code>	

New: 2015-09-18
Updated: 2018-07-28

Converts the `<token list>` to a `<string>`, and prepends the result to `<str var>`. The current contents of the `<str var>` are not automatically converted to a string.

<code>\str_put_right:Nn</code>	<code>\str_put_right:Nn <str var> {<token list>}</code>
<code>\str_put_right:(NV Ne cn cV Ne)</code>	
<code>\str_gput_right:Nn</code>	
<code>\str_gput_right:(NV Ne cn cV ce)</code>	

New: 2015-09-18
Updated: 2018-07-28

Converts the `<token list>` to a `<string>`, and appends the result to `<str var>`. The current contents of the `<str var>` are not automatically converted to a string.

17.3 String conditionals

<code>\str_if_empty_p:N *</code>	<code>\str_if_empty_p:N <str var></code>
<code>\str_if_empty_p:c *</code>	<code>\str_if_empty:NTF <str var> {<true code>} {<false code>}</code>
<code>\str_if_empty:NTF *</code>	Tests if the <code><string variable></code> is entirely empty (<i>i.e.</i> contains no characters at all).
<code>\str_if_empty:cTF *</code>	
<code>\str_if_empty_p:n *</code>	
<code>\str_if_empty:nTF *</code>	

New: 2015-09-18
Updated: 2022-03-21

<code>\str_if_eq_p:NN</code>	<code>\str_if_eq_p:NN <str var₁₂</code>
<code>\str_if_eq_p:(Nc cN cc) *</code>	<code>\str_if_eq:NNTF <str var₁₂</code>
<code>\str_if_eq:NNTF *</code>	Compares the content of two <code><str variables></code> and is logically true if the two contain the same characters in the same order. See <code>\tl_if_eq:NNTF</code> to compare tokens (including their category codes) rather than characters.
<code>\str_if_eq:(Nc cN cc)TF *</code>	

New: 2015-09-18

```

\str_if_eq_p:nn          * \str_if_eq_p:nn {<tl1>} {<tl2>}
\str_if_eq_p:(Vn|on|no|nV|VV|vn|nv|ee) * \str_if_eq:nnTF {<tl1>} {<tl2>} {<true code>} {<false code>}
\str_if_eq:nnTF          *
\str_if_eq:(Vn|on|no|nV|VV|vn|nv|ee)TF *

```

Updated: 2018-06-18

Compares the two *<token lists>* on a character by character basis (namely after converting them to strings), and is *true* if the two *<strings>* contain the same characters in the same order. Thus for example

```
\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }
```

is logically true. See `\tl_if_eq:nnTF` to compare tokens (including their category codes) rather than characters.

```

\str_if_in:NnTF \str_if_in:NnTF <str var> {<token list>} {<true code>} {<false code>}
\str_if_in:cnTF

```

New: 2017-10-08 Converts the *<token list>* to a *<string>* and tests if that *<string>* is found in the content of the *<str var>*.

```

\str_if_in:nnTF \str_if_in:nnTF {<tl1>} {<tl2>} {<true code>} {<false code>}

```

New: 2017-10-08 Converts both *<token lists>* to *<strings>* and tests whether *<string2>* is found inside *<string1>*.

```

\str_case:nn          * \str_case:nnTF {<test string>}
\str_case:(Vn|on|en|nV|nv) * {
\str_case:nnTF          *   {<string case1>} {<code case1>}
\str_case:(Vn|on|en|nV|nv)TF *   {<string case2>} {<code case2>}
\str_case:Nn          *   ...
\str_case:NnTF          *   {<string case_n>} {<code case_n>}

```

New: 2013-07-24
Updated: 2022-03-21

```

}
{<true code>}
{<false code>}

```

Compares the *<test string>* in turn with each of the *<string case>*s (all token lists are converted to strings). If the two are equal (as described for `\str_if_eq:nnTF`) then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

This set of functions performs no expansion on each *<string case>* argument, so any variable in there will be compared as a string. If expansion is needed in the *<string case>*s, then `\str_case_e:nn(TF)` should be used instead.

```

\str_case_e:nn  * \str_case_e:nnTF {\langle test string \rangle}
\str_case_e:en  * {
\str_case_e:nnTF *   {\langle string case1 \rangle} {\langle code case1 \rangle}
\str_case_e:enTF *   {\langle string case2 \rangle} {\langle code case2 \rangle}
New: 2018-06-19     *   ...
                    *   {\langle string casen \rangle} {\langle code casen \rangle}
                    * }
                    * {\langle true code \rangle}
                    * {\langle false code \rangle}

```

Compares the full expansion of the $\langle test\ string \rangle$ in turn with the full expansion of the $\langle string\ case \rangle$ s (all token lists are converted to strings). If the two full expansions are equal (as described for $\backslash\text{str_if_eq:eeTF}$) then the associated $\langle code \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function $\backslash\text{str_case_e:nn}$, which does nothing if there is no match, is also available. In $\backslash\text{str_case_e:nn(TF)}$, the $\langle test\ string \rangle$ is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

```

\str_compare_p:nNn * \str_compare_p:nNn {\langle t1 \rangle} \langle relation \rangle {\langle t2 \rangle}
\str_compare_p:eNe * \str_compare:nNnTF {\langle t1 \rangle} \langle relation \rangle {\langle t2 \rangle} {\langle true code \rangle} {\langle false code \rangle}
\str_compare:nNnTF *
\str_compare:eNeTF * Compares the two \langle token lists \rangle on a character by character basis (namely after con-
New: 2021-05-17     * verting them to strings) in a lexicographic order according to the character codes of the
                    * characters. The \langle relation \rangle can be <, =, or > and the test is true under the following
                    * conditions:

```

- for <, if the first string is earlier than the second in lexicographic order;
- for =, if the two strings have exactly the same characters;
- for >, if the first string is later than the second in lexicographic order.

Thus for example the following is logically **true**:

```
\str_compare_p:nNn { ab } < { abc }
```

T_EXhackers note: This is a wrapper around the T_EX primitive $\backslash(\text{pdf})\text{strcmp}$. It is meant for programming and not for sorting textual contents, as it simply considers character codes and not more elaborate considerations of grapheme clusters, locale, etc.

17.4 Mapping over strings

All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

```

\str_map_function:nN ☆ \str_map_function:nN {\langle token list \rangle} \langle function \rangle
\str_map_function:NN ☆ \str_map_function:NN \langle str var \rangle \langle function \rangle
\str_map_function:cN ☆ Converts the \langle token list \rangle to a \langle string \rangle then applies \langle function \rangle to every \langle character \rangle
New: 2017-11-14     * in the \langle string \rangle including spaces.

```

<code>\str_map_inline:nn</code>	<code>\str_map_inline:nn {<token list>} {<inline function>}</code>
<code>\str_map_inline:Nn</code>	<code>\str_map_inline:Nn <str var> {<inline function>}</code>
<code>\str_map_inline:cn</code>	Converts the <token list> to a <string> then applies the <inline function> to every <character> in the <str var> including spaces. The <inline function> should consist of code which receives the <character> as #1.

<code>\str_map_tokens:nn</code>	☆ <code>\str_map_tokens:nn {<token list>} {<code>}</code>
<code>\str_map_tokens:Nn</code>	☆ <code>\str_map_tokens:Nn <str var> {<code>}</code>
<code>\str_map_tokens:cn</code>	☆ Converts the <token list> to a <string> then applies <code> to every <character> in the <string> including spaces. The <code> receives each character as a trailing brace group. This is equivalent to <code>\str_map_function:nN</code> if the <code> consists of a single function.

<code>\str_map_variable:nNn</code>	<code>\str_map_variable:nNn {<token list>} <variable> {<code>}</code>
<code>\str_map_variable:NNn</code>	<code>\str_map_variable:NNn <str var> <variable> {<code>}</code>
<code>\str_map_variable:cNn</code>	Converts the <token list> to a <string> then stores each <character> in the <string> (including spaces) in turn in the (string or token list) <variable> and applies the <code>. The <code> will usually make use of the <variable>, but this is not enforced. The assignments to the <variable> are local. Its value after the loop is the last <character> in the <string>, or its original value if the <string> is empty. See also <code>\str_map_inline:Nn</code> .

<code>\str_map_break:</code>	☆ <code>\str_map_break:</code>
------------------------------	--------------------------------

New: 2017-10-08

Used to terminate a `\str_map...` function before all characters in the <string> have been processed. This normally takes place within a conditional statement, for example

```

\str_map_inline:Nn \l_my_str
{
  \str_if_eq:nnT { #1 } { bingo } { \str_map_break: }
  % Do something useful
}

```

See also `\str_map_break:n`. Use outside of a `\str_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.

`\str_map_break:n` ☆ `\str_map_break:n {<code>}`

New: 2017-10-08

Used to terminate a `\str_map_...` function before all characters in the *<string>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
{
  \str_if_eq:nnT { #1 } { bingo }
  { \str_map_break:n { <code> } }
  % Do something useful
}
```

Use outside of a `\str_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

17.5 Working with the content of strings

`\str_use:N` ☆ `\str_use:N <str var>`

`\str_use:c` ☆

New: 2015-09-18

Recovers the content of a *<str var>* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a *<str>* directly without an accessor function.

`\str_count:N` ☆ `\str_count:n {<token list>}`
`\str_count:c` ☆
`\str_count:n` ☆
`\str_count_ignore_spaces:n` ☆

New: 2015-09-18

Leaves in the input stream the number of characters in the string representation of *<token list>*, as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_count:N` and `\str_count:n`, all characters including spaces are counted. The `\str_count_ignore_spaces:n` function leaves the number of non-space characters in the input stream.

`\str_count_spaces:N` ☆ `\str_count_spaces:n {<token list>}`

`\str_count_spaces:c` ☆

`\str_count_spaces:n` ☆

New: 2015-09-18

Leaves in the input stream the number of space characters in the string representation of *<token list>*, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

```

\str_head:N          * \str_head:n {\token list}
\str_head:c          *
\str_head:n          *
\str_head_ignore_spaces:n *

```

New: 2015-09-18

Converts the $\langle token list \rangle$ into a $\langle string \rangle$. The first character in the $\langle string \rangle$ is then left in the input stream, with category code “other”. The functions differ if the first character is a space: $\backslashstr_head:N$ and $\backslashstr_head:n$ return a space token with category code 10 (blank space), while the $\backslashstr_head_ignore_spaces:n$ function ignores this space character and leaves the first non-space character in the input stream. If the $\langle string \rangle$ is empty (or only contains spaces in the case of the $_ignore_spaces$ function), then nothing is left on the input stream.

```

\str_tail:N          * \str_tail:n {\token list}
\str_tail:c          *
\str_tail:n          *
\str_tail_ignore_spaces:n *

```

New: 2015-09-18

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: $\backslashstr_tail:N$ and $\backslashstr_tail:n$ only trim that space, while $\backslashstr_tail_ignore_spaces:n$ removes the first non-space character and any space before it. If the $\langle token list \rangle$ is empty (or blank in the case of the $_ignore_spaces$ variant), then nothing is left on the input stream.

```

\str_item:Nn        * \str_item:nn {\token list} {\integer expression}
\str_item:nn        *
\str_item_ignore_spaces:nn *

```

New: 2015-09-18

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the character in position $\langle integer expression \rangle$ of the $\langle string \rangle$, starting at 1 for the first (left-most) character. In the case of $\backslashstr_item:Nn$ and $\backslashstr_item:nn$, all characters including spaces are taken into account. The $\backslashstr_item_ignore_spaces:nn$ function skips spaces when counting characters. If the $\langle integer expression \rangle$ is negative, characters are counted from the end of the $\langle string \rangle$. Hence, -1 is the right-most character, *etc.*

```

\str_range:Nnn      * \str_range:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}
\str_range:cnn      *
\str_range:nnn      *
\str_range_ignore_spaces:nnn *

```

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the characters from the $\langle start\ index \rangle$ to the $\langle end\ index \rangle$ inclusive. Spaces are preserved and counted as items (contrast this with `\tl_range:nnn` where spaces are not counted as items and are possibly discarded from the output).

Here $\langle start\ index \rangle$ and $\langle end\ index \rangle$ should be integer denotations. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', a negative index means 'start counting from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$. For instance,

```

\iow_term:e { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:e { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:e { \str_range:nnn { abcdef } { 0 } { -1 } }

```

prints `bcde`, `cdef`, `ef`, and an empty line to the terminal. The $\langle start\ index \rangle$ must always be smaller than or equal to the $\langle end\ index \rangle$: if this is not the case then no output is generated. Thus

```

\iow_term:e { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:e { \str_range:nnn { abcdef } { -1 } { -4 } }

```

both yield empty strings.

The behavior of `\str_range_ignore_spaces:nnn` is similar, but spaces are removed before starting the job. The input

```

\iow_term:e { \str_range:nnn { abcdefg } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abcdefg } { 2 } { -3 } }
\iow_term:e { \str_range:nnn { abcdefg } { -6 } { 5 } }
\iow_term:e { \str_range:nnn { abcdefg } { -6 } { -3 } }

```

```

\iow_term:e { \str_range:nnn { abc~efg } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abc~efg } { 2 } { -3 } }
\iow_term:e { \str_range:nnn { abc~efg } { -6 } { 5 } }
\iow_term:e { \str_range:nnn { abc~efg } { -6 } { -3 } }

```

```

\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { -3 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { -3 } }

```

```

\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { -3 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { -3 } }

```

will print four instances of bcde, four instances of bc e and eight instances of bcde.

17.6 Modifying string variables

```

\str_replace_once:Nnn \str_replace_once:Nnn <str var> {<old>} {<new>}
\str_replace_once:cnn
\str_greplace_once:Nnn
\str_greplace_once:cnn

```

Converts the *<old>* and *<new>* token lists to strings, then replaces the first (leftmost) occurrence of *<old string>* in the *<str var>* with *<new string>*.

New: 2017-10-08

```

\str_replace_all:Nnn \str_replace_all:Nnn <str var> {<old>} {<new>}
\str_replace_all:cnn
\str_greplace_all:Nnn
\str_greplace_all:cnn

```

Converts the *<old>* and *<new>* token lists to strings, then replaces all occurrences of *<old string>* in the *<str var>* with *<new string>*. As this function operates from left to right, the pattern *<old string>* may remain after the replacement (see `\str_remove_all:Nn` for an example).

New: 2017-10-08

```

\str_remove_once:Nn \str_remove_once:Nn <str var> {<token list>}
\str_remove_once:cn
\str_gremove_once:Nn
\str_gremove_once:cn

```

Converts the *<token list>* to a *<string>* then removes the first (leftmost) occurrence of *<string>* from the *<str var>*.

New: 2017-10-08

```

\str_remove_all:Nn \str_remove_all:Nn <str var> {<token list>}
\str_remove_all:cn
\str_gremove_all:Nn
\str_gremove_all:cn

```

Converts the *<token list>* to a *<string>* then removes all occurrences of *<string>* from the *<str var>*. As this function operates from left to right, the pattern *<string>* may remain after the removal, for instance,

New: 2017-10-08

```

\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str
{bc}

```

results in `\l_tmpa_str` containing abcd.

17.7 String manipulation

```
\str_lowercase:n * \str_lowercase:n {(tokens)}
\str_lowercase:f * \str_uppercase:n {(tokens)}
\str_uppercase:n *
\str_uppercase:f * 
```

Converts the input $\langle tokens \rangle$ to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

New: 2019-11-26

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
{
  \cs_set_protected:cpn
  {
    user
    \str_uppercase:f { \tl_head:n {#1} }
    \str_lowercase:f { \tl_tail:n {#1} }
  }
  { #2 }
}
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_casefold:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\text_lowercase:n(n)`, `\text_uppercase:n(n)` and `\text_titlecase_(all|once):n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

<code>\str_casefold:n</code> *	<code>\str_casefold:n</code> $\{\langle tokens \rangle\}$
<code>\str_casefold:V</code> *	Converts the input $\langle tokens \rangle$ to their string representation, as described for <code>\tl_to_str:n</code> , and then folds the case of the resulting $\langle string \rangle$ to remove case information. The result of this process is left in the input stream.

New: 2022-10-16

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by `\str_casefold:n` follows the mappings provided by the [Unicode Consortium](#), who [state](#):

Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by `\str_casefold:n` follows the “full” scheme defined by the Unicode Consortium (*e.g.* SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* I always folds to i and not to ı).

<code>\str_md5hash:n</code> *	<code>\str_md5hash:n</code> $\{\langle tl \rangle\}$
<code>\str_md5hash:e</code> *	Expands to the MD5 sum generated from the $\langle tl \rangle$, which is converted to a $\langle string \rangle$ as described for <code>\tl_to_str:n</code> .

New: 2023-05-19

17.8 Viewing strings

<code>\str_show:N</code>	<code>\str_show:N</code> $\langle str var \rangle$
<code>\str_show:c</code>	Displays the content of the $\langle str var \rangle$ on the terminal.
<code>\str_show:n</code>	

New: 2015-09-18
Updated: 2021-04-29

<code>\str_log:N</code>	<code>\str_log:N</code> $\langle str var \rangle$
<code>\str_log:c</code>	Writes the content of the $\langle str var \rangle$ in the log file.
<code>\str_log:n</code>	

New: 2019-02-15
Updated: 2021-04-29

17.9 Constant strings

`\c_ampersand_str` Constant strings, containing a single character token, with category code 12.
`\c_atsign_str`
`\c_backslash_str`
`\c_left_brace_str`
`\c_right_brace_str`
`\c_circumflex_str`
`\c_colon_str`
`\c_dollar_str`
`\c_hash_str`
`\c_percent_str`
`\c_tilde_str`
`\c_underscore_str`
`\c_zero_str`

New: 2015-09-19

Updated: 2020-12-22

`\c_empty_str` Constant that is always empty.

New: 2023-12-07

17.10 Scratch strings

`\l_tmpa_str` Scratch strings for local assignment. These are never used by the kernel code, and so
`\l_tmpb_str` are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by
other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_str` Scratch strings for global assignment. These are never used by the kernel code, and so
`\g_tmpb_str` are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by
other non-kernel code and so should only be used for short-term storage.

Chapter 18

The `l3str-convert` module

String encoding conversions

18.1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be UTF-16, ISO 8859-1, *etc.* See Table 1 for a list of supported encodings.⁶
- Bytes are translated to `TeX` tokens through a given “escaping”. Those are defined for the most part by the `pdf` file format. See Table 2 for a list of escaping methods supported.⁶

⁶Encodings and escapings will be added as they are requested.

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

<i>⟨Encoding⟩</i>	description
<code>utf8</code>	UTF-8
<code>utf16</code>	UTF-16, with byte-order mark
<code>utf16be</code>	UTF-16, big-endian
<code>utf16le</code>	UTF-16, little-endian
<code>utf32</code>	UTF-32, with byte-order mark
<code>utf32be</code>	UTF-32, big-endian
<code>utf32le</code>	UTF-32, little-endian
<code>iso88591, latin1</code>	ISO 8859-1
<code>iso88592, latin2</code>	ISO 8859-2
<code>iso88593, latin3</code>	ISO 8859-3
<code>iso88594, latin4</code>	ISO 8859-4
<code>iso88595</code>	ISO 8859-5
<code>iso88596</code>	ISO 8859-6
<code>iso88597</code>	ISO 8859-7
<code>iso88598</code>	ISO 8859-8
<code>iso88599, latin5</code>	ISO 8859-9
<code>iso885910, latin6</code>	ISO 8859-10
<code>iso885911</code>	ISO 8859-11
<code>iso885913, latin7</code>	ISO 8859-13
<code>iso885914, latin8</code>	ISO 8859-14
<code>iso885915, latin9</code>	ISO 8859-15
<code>iso885916, latin10</code>	ISO 8859-16
<code>clist</code>	comma-list of integers
<i>⟨empty⟩</i>	native (Unicode) string
<code>default</code>	like <code>utf8</code> with 8-bit engines, and like <code>native</code> with <code>unicode-engines</code>

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

<i>⟨Escaping⟩</i>	description
<code>bytes</code> , or <code>empty</code>	arbitrary bytes
<code>hex</code> , <code>hexadecimal</code>	byte = two hexadecimal digits
<code>name</code>	see <code>\pdfescapename</code>
<code>string</code>	see <code>\pdfescapestring</code>
<code>url</code>	encoding used in URLs

18.2 Conversion functions

`\str_set_convert:Nnnn` `\str_set_convert:Nnnn` $\langle str\ var \rangle$ $\{\langle string \rangle\}$ $\{\langle name\ 1 \rangle\}$ $\{\langle name\ 2 \rangle\}$
`\str_gset_convert:Nnnn`

This function converts the $\langle string \rangle$ from the encoding given by $\langle name\ 1 \rangle$ to the encoding given by $\langle name\ 2 \rangle$, and stores the result in the $\langle str\ var \rangle$. Each $\langle name \rangle$ can have the form $\langle encoding \rangle$ or $\langle encoding \rangle / \langle escaping \rangle$, where the possible values of $\langle encoding \rangle$ and $\langle escaping \rangle$ are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty $\langle name \rangle$ indicates the use of “native” strings, 8-bit for pdfTeX, and Unicode strings for the other two engines.

For example,

```
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
```

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark `"FEFF`, which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the $\langle string \rangle$ is not valid according to the $\langle escaping\ 1 \rangle$ and $\langle encoding\ 1 \rangle$, or if it cannot be reencoded in the $\langle encoding\ 2 \rangle$ and $\langle escaping\ 2 \rangle$ (for instance, if a character does not exist in the $\langle encoding\ 2 \rangle$). Erroneous input is replaced by the Unicode replacement character `"FFFD`, and characters which cannot be reencoded are replaced by either the replacement character `"FFFD` if it exists in the $\langle encoding\ 2 \rangle$, or an encoding-specific replacement character, or the question mark character.

`\str_set_convert:NnnnTF` `\str_set_convert:NnnnTF` $\langle str\ var \rangle$ $\{\langle string \rangle\}$ $\{\langle name\ 1 \rangle\}$ $\{\langle name\ 2 \rangle\}$ $\{\langle true\ code \rangle\}$
`\str_gset_convert:NnnnTF` $\{\langle false\ code \rangle\}$

As `\str_set_convert:Nnnn`, converts the $\langle string \rangle$ from the encoding given by $\langle name\ 1 \rangle$ to the encoding given by $\langle name\ 2 \rangle$, and assigns the result to $\langle str\ var \rangle$. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the $\langle string \rangle$ is not valid according to the $\langle name\ 1 \rangle$ encoding, or cannot be expressed in the $\langle name\ 2 \rangle$ encoding. Instead, the $\langle false\ code \rangle$ is performed.

18.3 Conversion by expansion (for PDF contexts)

A small number of expandable functions are provided for use in PDF string/name contexts. These *assume UTF-8* and *no escaping* in the input.

`\str_convert_pdfname:n` \star `\str_convert_pdfname:n` $\langle string \rangle$

As `\str_set_convert:Nnnn`, converts the $\langle string \rangle$ on a byte-by-byte basis with non-ASCII codepoints escaped using hashes.

18.4 Possibilities, and things to do

Encoding/escaping-related tasks.

- In Xe_{La}TeX/LuaTeX, would it be better to use the `^^^...` approach to build a string from a given list of character codes? Namely, within a group, assign `0-9a-f` and all characters we want to category “other”, then assign `^` the category superscript, and use `\scantokens`.
- Change `\str_set_convert:Nnnn` to expand its last two arguments.
- Describe the internal format in the code comments. Refuse code points in `["D800,"DFFF]` in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps the characters `! '()*-. /0123456789_` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.
- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko’s `stringenc`). CESU?
- More escapings: ASCII85, shell escapes, lua escapes, *etc.*?

Chapter 19

The I3quark module

Quarks and scan marks

Two special types of constants in L^AT_EX3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

19.1 Quarks

Quarks are control sequences (and in fact, token lists) that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the ‘stop token’ (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
  { <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster.

19.2 Defining quarks

`\quark_new:N` `\quark_new:N <quark>`
 Creates a new `<quark>` which expands only to `<quark>`. The `<quark>` is defined globally, and an error message is raised if the name was already taken.

`\q_stop` Used as a marker for delimited arguments, such as

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

`\q_mark` Used as a marker for delimited arguments when `\q_stop` is already in use.

`\q_nil` Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to `\q_stop`, which is only ever used as a delimiter).

`\q_no_value` A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as `\prop_get:NnN` if there is no data to return.

19.3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

`\quark_if_nil_p:N` * `\quark_if_nil_p:N <token>`
`\quark_if_nil:NTF` * `\quark_if_nil:NTF <token> {<true code>} {<false code>}`

Tests if the `<token>` is equal to `\q_nil`.

`\quark_if_nil_p:n` * `\quark_if_nil_p:n {<token list>}`
`\quark_if_nil_p:(o|V)` * `\quark_if_nil:nTF {<token list>} {<true code>} {<false code>}`
`\quark_if_nil:nTF` * Tests if the `<token list>` contains only `\q_nil` (distinct from `<token list>` being empty or containing `\q_nil` plus one or more other tokens).
`\quark_if_nil:(o|V)TF` *

`\quark_if_no_value_p:N` * `\quark_if_no_value_p:N <token>`
`\quark_if_no_value_p:c` * `\quark_if_no_value:NTF <token> {<true code>} {<false code>}`
`\quark_if_no_value:NTF` * Tests if the `<token>` is equal to `\q_no_value`.
`\quark_if_no_value:cTF` *

`\quark_if_no_value_p:n` * `\quark_if_no_value_p:n {<token list>}`
`\quark_if_no_value:nTF` * `\quark_if_no_value:nTF {<token list>} {<true code>} {<false code>}`
 Tests if the `<token list>` contains only `\q_no_value` (distinct from `<token list>` being empty or containing `\q_no_value` plus one or more other tokens).

19.4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 19.4.1.

`\q_recursion_tail` This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

`\q_recursion_stop` This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

`\quark_if_recursion_tail_stop:N * \quark_if_recursion_tail_stop:N <token>`

Tests if `<token>` contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop:n * \quark_if_recursion_tail_stop:n {{token list}}`
`\quark_if_recursion_tail_stop:o *`

Updated: 2011-09-06

Tests if the `<token list>` contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop_do:Nn * \quark_if_recursion_tail_stop_do:Nn <token> {{insertion}}`

Tests if `<token>` contains only the marker `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The `<insertion>` code is then added to the input stream after the recursion has ended.

`\quark_if_recursion_tail_stop_do:nn * \quark_if_recursion_tail_stop_do:nn {{token list}} {{insertion}}`
`\quark_if_recursion_tail_stop_do:on *`

Updated: 2011-09-06

Tests if the `<token list>` contains only `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The `<insertion>` code is then added to the input stream after the recursion has ended.

```
\quark_if_recursion_tail_break:NN * \quark_if_recursion_tail_break:nN {<token list>}
\quark_if_recursion_tail_break:nN * \<type>_map_break:
```

New: 2018-04-10

Tests if `<token list>` contains only `\q_recursion_tail`, and if so terminates the recursion using `\<type>_map_break:.` The recursion end should be marked by `\prg_break_point:Nn \<type>_map_break:.`

19.4.1 An example of recursion with quarks

Quarks are mainly used internally in the `expl3` code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce “`[-a-b-] [-c-d-]`”. Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here’s the definition of `\my_map_dbl:nn`. First of all, define the function that does the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map_dbl:nn #1#2
{
  \cs_set:Npn \_my_map_dbl_fn:nn ##1 ##2 {#2}
  \_my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail
  \q_recursion_stop
}
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \_my_map_dbl:nn
{
  \quark_if_recursion_tail_stop:n {#1}
  \quark_if_recursion_tail_stop:n {#2}
  \_my_map_dbl_fn:nn {#1} {#2}
}
```

Finally, recurse:

```
\_my_map_dbl:nn
}
```

Note that contrarily to `LATEX3` built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `_my_map_dbl_fn:nn`.

19.5 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by \TeX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see `!3regex`).

`\scan_new:N` `\scan_new:N` $\langle scan\ mark \rangle$

New: 2018-04-01 Creates a new $\langle scan\ mark \rangle$ which is set equal to `\scan_stop:`. The $\langle scan\ mark \rangle$ is defined globally, and an error message is raised if the name was already taken by another scan mark.

`\s_stop` Used at the end of a set of instructions, as a marker that can be jumped to using `\use_`
`none_delimit_by_s_stop:w`.

New: 2018-04-01

`\use_none_delimit_by_s_stop:w` \star `\use_none_delimit_by_s_stop:w` $\langle tokens \rangle$ `\s_stop`

New: 2018-04-01

Removes the $\langle tokens \rangle$ and `\s_stop` from the input stream. This leads to a low-level \TeX error if `\s_stop` is absent.

Chapter 20

The l3seq module

Sequences and stacks

L^AT_EX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any *⟨balanced text⟩*. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in L^AT_EX3. This is achieved using a number of dedicated stack functions.

20.1 Creating and initialising sequences

<code>\seq_new:N</code>	<code>\seq_new:N</code>	<code>⟨seq var⟩</code>
<code>\seq_new:c</code>		

Creates a new `⟨seq var⟩` or raises an error if the name is already taken. The declaration is global. The `⟨seq var⟩` initially contains no items.

<code>\seq_clear:N</code>	<code>\seq_clear:N</code>	<code>⟨seq var⟩</code>
<code>\seq_clear:c</code>		
<code>\seq_gclear:N</code>		
<code>\seq_gclear:c</code>		

Clears all items from the `⟨seq var⟩`.

<code>\seq_clear_new:N</code>	<code>\seq_clear_new:N</code>	<code>⟨seq var⟩</code>
<code>\seq_clear_new:c</code>		
<code>\seq_gclear_new:N</code>		
<code>\seq_gclear_new:c</code>		

Ensures that the `⟨seq var⟩` exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the `⟨seq var⟩` empty.

<code>\seq_set_eq:NN</code>	<code>\seq_set_eq:NN</code>	<code>⟨seq var₁⟩</code>	<code>⟨seq var₂⟩</code>
<code>\seq_set_eq:(cN Nc cc)</code>			
<code>\seq_gset_eq:NN</code>			
<code>\seq_gset_eq:(cN Nc cc)</code>			

Sets the content of `⟨seq var1⟩` equal to that of `⟨seq var2⟩`.

```

\seq_set_from_clist:NN \seq_set_from_clist:NN <seq var> <comma-list>
\seq_set_from_clist:(cN|Nc|cc)
\seq_set_from_clist:Nn
\seq_set_from_clist:cn
\seq_gset_from_clist:NN
\seq_gset_from_clist:(cN|Nc|cc)
\seq_gset_from_clist:Nn
\seq_gset_from_clist:cn

```

New: 2014-07-17

Converts the data in the `<comma list>` into a `<seq var>`: the original `<comma list>` is unchanged.

```

\seq_const_from_clist:Nn \seq_const_from_clist:Nn <seq var> {(comma-list)}
\seq_const_from_clist:cn

```

New: 2017-11-28

Creates a new constant `<seq var>` or raises an error if the name is already taken. The `<seq var>` is set globally to contain the items in the `<comma list>`.

```

\seq_set_split:Nnn \seq_set_split:Nnn <seq var> {(delimiter)} {(token list)}
\seq_set_split:(NVn|NnV|NVV|Nne|Nee)
\seq_gset_split:Nnn
\seq_gset_split:(NVn|NnV|NVV|Nne|Nee)

```

New: 2011-08-15

Updated: 2012-07-02

Splits the `<token list>` into `<items>` separated by `<delimiter>`, and assigns the result to the `<seq var>`. Spaces on both sides of each `<item>` are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of `\l3clist` functions. Empty `<items>` are preserved by `\seq_set_split:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <seq var> {}`. The `<delimiter>` may not contain `{, }` or `#` (assuming `TEX`'s normal category code régime). If the `<delimiter>` is empty, the `<token list>` is split into `<items>` as a `<token list>`. See also `\seq_set_split_keep_spaces:Nnn`, which omits space stripping.

```

\seq_set_split_keep_spaces:Nnn \seq_set_split_keep_spaces:Nnn <seq var> {(delimiter)} {(token list)}
\seq_set_split_keep_spaces:NnV
\seq_gset_split_keep_spaces:Nnn
\seq_gset_split_keep_spaces:NnV

```

New: 2021-03-24

Splits the `<token list>` into `<items>` separated by `<delimiter>`, and assigns the result to the `<seq var>`. One set of outer braces is removed (if any) but any surrounding spaces are retained: any braces *inside* one or more spaces are therefore kept. Empty `<items>` are preserved by `\seq_set_split_keep_spaces:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <seq var> {}`. The `<delimiter>` may not contain `{, }` or `#` (assuming `TEX`'s normal category code régime). If the `<delimiter>` is empty, the `<token list>` is split into `<items>` as a `<token list>`. See also `\seq_set_split:Nnn`, which removes spaces around the delimiters.

<code>\seq_set_filter:NNn</code>	<code>\seq_set_filter:NNn <seq var₁> <seq var₂> {<inline boolexpr>}</code>
<code>\seq_gset_filter:NNn</code>	Evaluates the <code><inline boolexpr></code> for every <code><item></code> stored within the <code><seq var₂></code> . The <code><inline boolexpr></code> receives the <code><item></code> as #1. The sequence of all <code><items></code> for which the <code><inline boolexpr></code> evaluated to <code>true</code> is assigned to <code><seq var₁></code> .

New: 2012-06-15

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

<code>\seq_concat:NNN</code>	<code>\seq_concat:NNN <seq var₁> <seq var₂> <seq var₃></code>
<code>\seq_concat:ccc</code>	Concatenates the content of <code><seq var₂></code> and <code><seq var₃></code> together and saves the result in <code><seq var₁></code> . The items in <code><seq var₂></code> are placed at the left side of the new sequence.
<code>\seq_gconcat:NNN</code>	
<code>\seq_gconcat:ccc</code>	

<code>\seq_if_exist_p:N *</code>	<code>\seq_if_exist_p:N <seq var></code>
<code>\seq_if_exist_p:c *</code>	<code>\seq_if_exist:NTF <seq var> {<true code>} {<false code>}</code>
<code>\seq_if_exist:NTF *</code>	Tests whether the <code><seq var></code> is currently defined. This does not check that the <code><seq var></code> really is a sequence variable.
<code>\seq_if_exist:cTF *</code>	

New: 2012-03-03

20.2 Appending data to sequences

<code>\seq_put_left:Nn</code>	<code>\seq_put_left:Nn <seq var> {<item>}</code>
<code>\seq_put_left:(NV Nv Ne No cn cV cv ce co)</code>	
<code>\seq_gput_left:Nn</code>	
<code>\seq_gput_left:(NV Nv Ne No cn cV cv ce co)</code>	

Appends the `<item>` to the left of the `<seq var>`.

<code>\seq_put_right:Nn</code>	<code>\seq_put_right:Nn <seq var> {<item>}</code>
<code>\seq_put_right:(NV Nv Ne No cn cV cv ce co)</code>	
<code>\seq_gput_right:Nn</code>	
<code>\seq_gput_right:(NV Nv Ne No cn cV cv ce co)</code>	

Appends the `<item>` to the right of the `<seq var>`.

20.3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, *i.e.* setting the `<token list variable>` used with `\tl_set:Nn` and `never \tl_gset:Nn`.

<code>\seq_get_left:NN</code>	<code>\seq_get_left:NN <seq var> <token list variable></code>
<code>\seq_get_left:cN</code>	Stores the left-most item from a <code><seq var></code> in the <code><token list variable></code> without removing it from the <code><seq var></code> . The <code><token list variable></code> is assigned locally. If <code><seq var></code> is empty the <code><token list variable></code> is set to the special marker <code>\q_no_value</code> .
<code>\seq_get_left:cn</code>	

Updated: 2012-05-14

`\seq_get_right:NN` `\seq_get_right:NN` $\langle seq\ var \rangle$ $\langle token\ list\ variable \rangle$
`\seq_get_right:cN`
Updated: 2012-05-19

Stores the right-most item from a $\langle seq\ var \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle seq\ var \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_pop_left:NN` `\seq_pop_left:NN` $\langle seq\ var \rangle$ $\langle token\ list\ variable \rangle$
`\seq_pop_left:cN`
Updated: 2012-05-14

Pops the left-most item from a $\langle seq\ var \rangle$ into the $\langle token\ list\ variable \rangle$, *i.e.* removes the item from the sequence and stores it in the $\langle token\ list\ variable \rangle$. Both of the variables are assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_gpop_left:NN` `\seq_gpop_left:NN` $\langle seq\ var \rangle$ $\langle token\ list\ variable \rangle$
`\seq_gpop_left:cN`
Updated: 2012-05-14

Pops the left-most item from a $\langle seq\ var \rangle$ into the $\langle token\ list\ variable \rangle$, *i.e.* removes the item from the sequence and stores it in the $\langle token\ list\ variable \rangle$. The $\langle seq\ var \rangle$ is modified globally, while the assignment of the $\langle token\ list\ variable \rangle$ is local. If $\langle seq\ var \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_pop_right:NN` `\seq_pop_right:NN` $\langle seq\ var \rangle$ $\langle token\ list\ variable \rangle$
`\seq_pop_right:cN`
Updated: 2012-05-19

Pops the right-most item from a $\langle seq\ var \rangle$ into the $\langle token\ list\ variable \rangle$, *i.e.* removes the item from the sequence and stores it in the $\langle token\ list\ variable \rangle$. Both of the variables are assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_gpop_right:NN` `\seq_gpop_right:NN` $\langle seq\ var \rangle$ $\langle token\ list\ variable \rangle$
`\seq_gpop_right:cN`
Updated: 2012-05-19

Pops the right-most item from a $\langle seq\ var \rangle$ into the $\langle token\ list\ variable \rangle$, *i.e.* removes the item from the sequence and stores it in the $\langle token\ list\ variable \rangle$. The $\langle seq\ var \rangle$ is modified globally, while the assignment of the $\langle token\ list\ variable \rangle$ is local. If $\langle seq\ var \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_item:Nn` * `\seq_item:Nn` $\langle seq\ var \rangle$ $\{ \langle integer\ expression \rangle \}$
`\seq_item:(NV|Ne|cn|cV|ce)` *
New: 2014-07-17

Indexing items in the $\langle seq\ var \rangle$ from 1 at the top (left), this function evaluates the $\langle integer\ expression \rangle$ and leaves the appropriate item from the sequence in the input stream. If the $\langle integer\ expression \rangle$ is negative, indexing occurs from the bottom (right) of the sequence. If the $\langle integer\ expression \rangle$ is larger than the number of items in the $\langle seq\ var \rangle$ (as calculated by `\seq_count:N`) then the function expands to nothing.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an e-type or x-type argument expansion.

<code>\seq_rand_item:N</code> *	<code>\seq_rand_item:N</code> $\langle seq\ var \rangle$
<code>\seq_rand_item:c</code> *	Selects a pseudo-random item of the $\langle seq\ var \rangle$. If the $\langle seq\ var \rangle$ is empty the result is empty. This is not available in older versions of X _ƒ TEX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an e-type or x-type argument expansion.

20.4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

<code>\seq_get_left:NNTF</code>	<code>\seq_get_left:NNTF</code> $\langle seq\ var \rangle$ $\langle token\ list\ variable \rangle$ $\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
<code>\seq_get_left:cNTF</code>	If the $\langle seq\ var \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle seq\ var \rangle$ is non-empty, stores the left-most item from the $\langle seq\ var \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle seq\ var \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle token\ list\ variable \rangle$ is assigned locally.

New: 2012-05-14
Updated: 2012-05-19

<code>\seq_get_right:NNTF</code>	<code>\seq_get_right:NNTF</code> $\langle seq\ var \rangle$ $\langle token\ list\ variable \rangle$ $\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
<code>\seq_get_right:cNTF</code>	If the $\langle seq\ var \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle seq\ var \rangle$ is non-empty, stores the right-most item from the $\langle seq\ var \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle seq\ var \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle token\ list\ variable \rangle$ is assigned locally.

New: 2012-05-19

<code>\seq_pop_left:NNTF</code>	<code>\seq_pop_left:NNTF</code> $\langle seq\ var \rangle$ $\langle token\ list\ variable \rangle$ $\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
<code>\seq_pop_left:cNTF</code>	If the $\langle seq\ var \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle seq\ var \rangle$ is non-empty, pops the left-most item from the $\langle seq\ var \rangle$ in the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the $\langle seq\ var \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. Both the $\langle seq\ var \rangle$ and the $\langle token\ list\ variable \rangle$ are assigned locally.

New: 2012-05-14
Updated: 2012-05-19

<code>\seq_gpop_left:NNTF</code>	<code>\seq_gpop_left:NNTF</code> $\langle seq\ var \rangle$ $\langle token\ list\ variable \rangle$ $\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
<code>\seq_gpop_left:cNTF</code>	If the $\langle seq\ var \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle seq\ var \rangle$ is non-empty, pops the left-most item from the $\langle seq\ var \rangle$ in the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the $\langle seq\ var \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle seq\ var \rangle$ is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally.

New: 2012-05-14
Updated: 2012-05-19

$\backslash\text{seq_pop_right:NNTF}$ $\backslash\text{seq_pop_right:cNNTF}$	$\backslash\text{seq_pop_right:NNTF}$ $\langle\text{seq var}\rangle$ $\langle\text{token list variable}\rangle$ $\{\langle\text{true code}\rangle\}$ $\{\langle\text{false code}\rangle\}$
<small>New: 2012-05-19</small>	<p>If the $\langle\text{seq var}\rangle$ is empty, leaves the $\langle\text{false code}\rangle$ in the input stream. The value of the $\langle\text{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\text{seq var}\rangle$ is non-empty, pops the right-most item from the $\langle\text{seq var}\rangle$ in the $\langle\text{token list variable}\rangle$, <i>i.e.</i> removes the item from the $\langle\text{seq var}\rangle$, then leaves the $\langle\text{true code}\rangle$ in the input stream. Both the $\langle\text{seq var}\rangle$ and the $\langle\text{token list variable}\rangle$ are assigned locally.</p>

$\backslash\text{seq_gpop_right:NNTF}$ $\backslash\text{seq_gpop_right:cNNTF}$	$\backslash\text{seq_gpop_right:NNTF}$ $\langle\text{seq var}\rangle$ $\langle\text{token list variable}\rangle$ $\{\langle\text{true code}\rangle\}$ $\{\langle\text{false code}\rangle\}$
<small>New: 2012-05-19</small>	<p>If the $\langle\text{seq var}\rangle$ is empty, leaves the $\langle\text{false code}\rangle$ in the input stream. The value of the $\langle\text{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\text{seq var}\rangle$ is non-empty, pops the right-most item from the $\langle\text{seq var}\rangle$ in the $\langle\text{token list variable}\rangle$, <i>i.e.</i> removes the item from the $\langle\text{seq var}\rangle$, then leaves the $\langle\text{true code}\rangle$ in the input stream. The $\langle\text{seq var}\rangle$ is modified globally, while the $\langle\text{token list variable}\rangle$ is assigned locally.</p>

20.5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

$\backslash\text{seq_remove_duplicates:N}$ $\backslash\text{seq_remove_duplicates:c}$ $\backslash\text{seq_gremove_duplicates:N}$ $\backslash\text{seq_gremove_duplicates:c}$	$\backslash\text{seq_remove_duplicates:N}$ $\langle\text{seq var}\rangle$
	<p>Removes duplicate items from the $\langle\text{seq var}\rangle$, leaving the left most copy of each item in the $\langle\text{seq var}\rangle$. The $\langle\text{item}\rangle$ comparison takes place on a token basis, as for $\backslash\text{tl_if_eq:nnTF}$.</p>

T_EXhackers note: This function iterates through every item in the $\langle\text{seq var}\rangle$ and does a comparison with the $\langle\text{items}\rangle$ already checked. It is therefore relatively slow with large sequences.

$\backslash\text{seq_remove_all:Nn}$ $\backslash\text{seq_remove_all:(NV Ne cn cV ce)}$ $\backslash\text{seq_gremove_all:Nn}$ $\backslash\text{seq_gremove_all:(NV Ne cn cV ce)}$	$\backslash\text{seq_remove_all:Nn}$ $\langle\text{seq var}\rangle$ $\{\langle\text{item}\rangle\}$
--	---

Removes every occurrence of $\langle\text{item}\rangle$ from the $\langle\text{seq var}\rangle$. The $\langle\text{item}\rangle$ comparison takes place on a token basis, as for $\backslash\text{tl_if_eq:nnTF}$.

$\backslash\text{seq_set_item:Nnn}$ $\backslash\text{seq_set_item:cnn}$ $\backslash\text{seq_set_item:NnnTF}$ $\backslash\text{seq_set_item:cnnTF}$ $\backslash\text{seq_gset_item:Nnn}$ $\backslash\text{seq_gset_item:cnn}$ $\backslash\text{seq_gset_item:NnnTF}$ $\backslash\text{seq_gset_item:cnnTF}$	$\backslash\text{seq_set_item:Nnn}$ $\langle\text{seq var}\rangle$ $\{\langle\text{int expr}\rangle\}$ $\{\langle\text{item}\rangle\}$ $\backslash\text{seq_set_item:NnnTF}$ $\langle\text{seq var}\rangle$ $\{\langle\text{int expr}\rangle\}$ $\{\langle\text{item}\rangle\}$ $\{\langle\text{true code}\rangle\}$ $\{\langle\text{false code}\rangle\}$
	<p>Removes the item of $\langle\text{seq var}\rangle$ at the position given by evaluating the $\langle\text{int expr}\rangle$ and replaces it by $\langle\text{item}\rangle$. Items are indexed from 1 on the left/top of the $\langle\text{seq var}\rangle$, or from -1 on the right/bottom. If the $\langle\text{int expr}\rangle$ is zero or is larger (in absolute value) than the number of items in the sequence, the $\langle\text{seq var}\rangle$ is not modified. In these cases, $\backslash\text{seq_set_item:Nnn}$ raises an error while $\backslash\text{seq_set_item:NnnTF}$ runs the $\langle\text{false code}\rangle$. In cases where the assignment was successful, $\langle\text{true code}\rangle$ is run afterwards.</p>

New: 2021-04-29

```

\seq_reverse:N \seq_reverse:N <seq var>
\seq_reverse:c
\seq_greverse:N Reverses the order of the items stored in the <seq var>.
\seq_greverse:c

```

New: 2014-07-18

```

\seq_sort:Nn \seq_sort:Nn <seq var> <comparison code>
\seq_sort:cn
\seq_gsort:Nn Sorts the items in the <seq var> according to the <comparison code>, and assigns the
\seq_gsort:cn result to <seq var>. The details of sorting comparison are described in Section 6.1.

```

New: 2017-02-06

```

\seq_shuffle:N \seq_shuffle:N <seq var>
\seq_shuffle:c
\seq_gshuffle:N Sets the <seq var> to the result of placing the items of the <seq var> in a random order.
\seq_gshuffle:c Each item is (roughly) as likely to end up in any given position.

```

New: 2018-04-29

TeXhackers note: For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed `\sys_rand_seed`: only has 28-bits. The use of `\toks` internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

20.6 Sequence conditionals

```

\seq_if_empty_p:N * \seq_if_empty_p:N <seq var>
\seq_if_empty_p:c * \seq_if_empty:NNTF <seq var> <true code> <false code>
\seq_if_empty:NNTF * Tests if the <seq var> is empty (containing no items).
\seq_if_empty:cTF *

```

```

\seq_if_in:NnTF \seq_if_in:NnTF <seq var> <item> <true code> <false code>
\seq_if_in:(NV|Nv|Ne|No|cn|cV|cv|ce|co)TF

```

Tests if the `<item>` is present in the `<seq var>`.

20.7 Mapping over sequences

All mappings are done at the current group level, *i.e.* any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

```

\seq_map_function:NN ☆ \seq_map_function:NN <seq var> <function>
\seq_map_function:cN ☆

```

Updated: 2012-06-29

Applies `<function>` to every `<item>` stored in the `<seq var>`. The `<function>` will receive one argument for each iteration. The `<items>` are returned from left to right. To pass further arguments to the `<function>`, see `\seq_map_tokens:Nn`. The function `\seq_map_inline:Nn` is faster than `\seq_map_function:NN` for sequences with more than about 10 items.

<code>\seq_map_inline:Nn</code>	<code>\seq_map_inline:Nn <seq var> {<inline function>}</code>
<code>\seq_map_inline:cn</code>	Applies <i><inline function></i> to every <i><item></i> stored within the <i><seq var></i> . The <i><inline function></i> should consist of code which will receive the <i><item></i> as #1. The <i><items></i> are returned from left to right.
Updated: 2012-06-29	

<code>\seq_map_tokens:Nn</code> ☆	<code>\seq_map_tokens:Nn <seq var> {<code>}</code>
<code>\seq_map_tokens:cn</code> ☆	Analogue of <code>\seq_map_function:NN</code> which maps several tokens instead of a single function. The <i><code></i> receives each item in the <i><seq var></i> as a trailing brace group. For instance,
New: 2019-08-30	

`\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }`

expands to twice each item in the *<seq var>*: for each item in `\l_my_seq` the function `\prg_replicate:nn` receives 2 and *<item>* as its two arguments. The function `\seq_map_inline:Nn` is typically faster but it is not expandable.

<code>\seq_map_variable:NNn</code>	<code>\seq_map_variable:NNn <seq var> <variable> {<code>}</code>
<code>\seq_map_variable:(Ncn cNn ccn)</code>	
Updated: 2012-06-29	

Stores each *<item>* of the *<seq var>* in turn in the (token list) *<variable>* and applies the *<code>*. The *<code>* will usually make use of the *<variable>*, but this is not enforced. The assignments to the *<variable>* are local. Its value after the loop is the last *<item>* in the *<seq var>*, or its original value if the *<seq var>* is empty. The *<items>* are returned from left to right.

<code>\seq_map_indexed_function:NN</code> ☆	<code>\seq_map_indexed_function:NN <seq var> <function></code>
New: 2018-05-03	

Applies *<function>* to every entry in the *<seq var>*. The *<function>* should have signature `:nn`. It receives two arguments for each iteration: the *<index>* (namely 1 for the first entry, then 2 and so on) and the *<item>*.

<code>\seq_map_indexed_inline:Nn</code>	<code>\seq_map_indexed_inline:Nn <seq var> {<inline function>}</code>
---	---

New: 2018-05-03 Applies *<inline function>* to every entry in the *<seq var>*. The *<inline function>* should consist of code which receives the *<index>* (namely 1 for the first entry, then 2 and so on) as #1 and the *<item>* as #2.

<code>\seq_map_pairwise_function:NNN</code> ☆	<code>\seq_map_pairwise_function:NNN <seq₁₂</code>
<code>\seq_map_pairwise_function:(NcN cNN ccN)</code> ☆	
New: 2023-05-10	

Applies *<function>* to every pair of items *<seq₁-item>*–*<seq₂-item>* from the two sequences, returning items from both sequences from left to right. The *<function>* receives two n-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

`\seq_map_break: ☆ \seq_map_break:`

Updated: 2012-06-29

Used to terminate a `\seq_map_...` function before all entries in the `<seq var>` have been processed. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \seq_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\seq_map_break:n ☆ \seq_map_break:n {<code>}`

Updated: 2012-06-29

Used to terminate a `\seq_map_...` function before all entries in the `<seq var>` have been processed, inserting the `<code>` after the mapping has ended. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \seq_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the `<code>` is inserted into the input stream. This depends on the design of the mapping function.

`\seq_set_map:NNn \seq_set_map:NNn <seq var1> <seq var2> {<inline function>}`

`\seq_gset_map:NNn`

New: 2011-12-22

Updated: 2020-07-16

Applies `<inline function>` to every `<item>` stored within the `<seq var2>`. The `<inline function>` should consist of code which will receive the `<item>` as `#1`. The sequence resulting applying `<inline function>` to each `<item>` is assigned to `<seq var1>`.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level T_EX errors.

<code>\seq_set_map_e:NNn</code> <code>\seq_gset_map_e:NNn</code>	<code>\seq_set_map_e:NNn</code> <code><seq var₁></code> <code><seq var₂></code> <code>{<inline function>}</code> <code>\seq_gset_map_e:NNn</code> Applies <code><inline function></code> to every <code><item></code> stored within the <code><seq var₂></code> . The <code><inline function></code> should consist of code which will receive the <code><item></code> as #1. The sequence resulting from e-expanding <code><inline function></code> applied to each <code><item></code> is assigned to <code><seq var₁></code> . As such, the code in <code><inline function></code> should be expandable.
---	--

New: 2020-07-16
Updated: 2023-10-26

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

<code>\seq_count:N</code> * <code>\seq_count:c</code> *	<code>\seq_count:N</code> <code><seq var></code> Leaves the number of items in the <code><seq var></code> in the input stream as an <code><integer denotation></code> . The total number of items in a <code><seq var></code> includes those which are empty and duplicates, <i>i.e.</i> every item in a <code><seq var></code> is unique.
--	---

New: 2012-07-13

20.8 Using the content of sequences directly

<code>\seq_use:Nnnn</code> * <code>\seq_use:cnnn</code> *	<code>\seq_use:Nnnn</code> <code><seq var></code> <code>{<separator between two>}</code> <code>\seq_use:cnnn</code> <code>{<separator between more than two>}</code> <code>{<separator between final two>}</code> Places the contents of the <code><seq var></code> in the input stream, with the appropriate <code><separator></code> between the items. Namely, if the sequence has more than two items, the <code><separator between more than two></code> is placed between each pair of items except the last, for which the <code><separator between final two></code> is used. If the sequence has exactly two items, then they are placed in the input stream separated by the <code><separator between two></code> . If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.
--	---

New: 2013-05-26

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<items>` do not expand further when appearing in an e-type or x-type argument expansion.

<code>\seq_use:Nn</code> *	<code>\seq_use:Nn <seq var> {<separator>}</code>
<code>\seq_use:cn</code> *	
<small>New: 2013-05-26</small>	

Places the contents of the `<seq var>` in the input stream, with the `<separator>` between the items. If the sequence has a single item, it is placed in the input stream with no `<separator>`, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<items>` do not expand further when appearing in an e-type or x-type argument expansion.

20.9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

<code>\seq_get:NN</code>	<code>\seq_get:NN <seq var> <token list variable></code>
<code>\seq_get:cN</code>	
<small>Updated: 2012-05-14</small>	

Reads the top item from a `<seq var>` into the `<token list variable>` without removing it from the `<seq var>`. The `<token list variable>` is assigned locally. If `<seq var>` is empty the `<token list variable>` is set to the special marker `\q_no_value`.

<code>\seq_pop:NN</code>	<code>\seq_pop:NN <seq var> <token list variable></code>
<code>\seq_pop:cN</code>	
<small>Updated: 2012-05-14</small>	

Pops the top item from a `<seq var>` into the `<token list variable>`. Both of the variables are assigned locally. If `<seq var>` is empty the `<token list variable>` is set to the special marker `\q_no_value`.

<code>\seq_gpop:NN</code>	<code>\seq_gpop:NN <seq var> <token list variable></code>
<code>\seq_gpop:cN</code>	
<small>Updated: 2012-05-14</small>	

Pops the top item from a `<seq var>` into the `<token list variable>`. The `<seq var>` is modified globally, while the `<token list variable>` is assigned locally. If `<seq var>` is empty the `<token list variable>` is set to the special marker `\q_no_value`.

<code>\seq_get:NNTF</code>	<code>\seq_get:NNTF <seq var> <token list variable> {<>true code>} {<>false code>}</code>
<code>\seq_get:cNTF</code>	
<small>New: 2012-05-14</small>	
<small>Updated: 2012-05-19</small>	

If the `<seq var>` is empty, leaves the `<>false code>` in the input stream. The value of the `<token list variable>` is not defined in this case and should not be relied upon. If the `<seq var>` is non-empty, stores the top item from a `<seq var>` in the `<token list variable>` without removing it from the `<seq var>`. The `<token list variable>` is assigned locally.

<code>\seq_pop:NNTF</code>	<code>\seq_pop:NNTF <seq var> <token list variable> {<true code>} {<false code>}</code>
<code>\seq_pop:cNTF</code>	If the <code><seq var></code> is empty, leaves the <code><false code></code> in the input stream. The value of the <code><token list variable></code> is not defined in this case and should not be relied upon.
New: 2012-05-14	
Updated: 2012-05-19	If the <code><seq var></code> is non-empty, pops the top item from the <code><seq var></code> in the <code><token list variable></code> , <i>i.e.</i> removes the item from the <code><seq var></code> . Both the <code><seq var></code> and the <code><token list variable></code> are assigned locally.

<code>\seq_gpop:NNTF</code>	<code>\seq_gpop:NNTF <seq var> <token list variable> {<true code>} {<false code>}</code>
<code>\seq_gpop:cNTF</code>	If the <code><seq var></code> is empty, leaves the <code><false code></code> in the input stream. The value of the <code><token list variable></code> is not defined in this case and should not be relied upon.
New: 2012-05-14	
Updated: 2012-05-19	If the <code><seq var></code> is non-empty, pops the top item from the <code><seq var></code> in the <code><token list variable></code> , <i>i.e.</i> removes the item from the <code><seq var></code> . The <code><seq var></code> is modified globally, while the <code><token list variable></code> is assigned locally.

<code>\seq_push:Nn</code>	<code>\seq_push:Nn <seq var> {<item>}</code>
<code>\seq_push:(NV Nv Ne No cn cV cv ce co)</code>	
<code>\seq_gpush:Nn</code>	
<code>\seq_gpush:(NV Nv Ne No cn cV cv ce co)</code>	

Adds the `{<item>}` to the top of the `<seq var>`.

20.10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a `<seq var>` only has distinct items, use `\seq_remove_duplicates:N <seq var>`. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set `<seq var>` are straightforward. For instance, `\seq_count:N <seq var>` expands to the number of items, while `\seq_if_in:NnTF <seq var> {<item>}` tests if the `<item>` is in the set.

Adding an `<item>` to a set `<seq var>` can be done by appending it to the `<seq var>` if it is not already in the `<seq var>`:

```
\seq_if_in:NnF <seq var> {<item>}
{ \seq_put_right:Nn <seq var> {<item>} }
```

Removing an `<item>` from a set `<seq var>` can be done using `\seq_remove_all:Nn`,

```
\seq_remove_all:Nn <seq var> {<item>}
```

The intersection of two sets `<seq var1>` and `<seq var2>` can be stored into `<seq var3>` by collecting items of `<seq var1>` which are in `<seq var2>`.

```

\seq_clear:N <seq var_3>
\seq_map_inline:Nn <seq var_1>
{
  \seq_if_in:NnT <seq var_2> {#1}
  { \seq_put_right:Nn <seq var_3> {#1} }
}

```

The code as written here only works if $\langle seq\ var_3 \rangle$ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence $\backslash l_ \langle pkg \rangle_ internal_ seq$, then $\langle seq\ var_3 \rangle$ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ through

```

\seq_concat:NNN <seq var_3> <seq var_1> <seq var_2>
\seq_remove_duplicates:N <seq var_3>

```

or by adding items to (a copy of) $\langle seq\ var_1 \rangle$ one by one

```

\seq_set_eq:NN <seq var_3> <seq var_1>
\seq_map_inline:Nn <seq var_2>
{
  \seq_if_in:NnF <seq var_3> {#1}
  { \seq_put_right:Nn <seq var_3> {#1} }
}

```

The second approach is faster than the first when the $\langle seq\ var_2 \rangle$ is short compared to $\langle seq\ var_1 \rangle$.

The difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by removing items of the $\langle seq\ var_2 \rangle$ from (a copy of) the $\langle seq\ var_1 \rangle$ one by one.

```

\seq_set_eq:NN <seq var_3> <seq var_1>
\seq_map_inline:Nn <seq var_2>
{ \seq_remove_all:Nn <seq var_3> {#1} }

```

The symmetric difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by computing the difference between $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ and storing the result as $\backslash l_ \langle pkg \rangle_ internal_ seq$, then the difference between $\langle seq\ var_2 \rangle$ and $\langle seq\ var_1 \rangle$, and finally concatenating the two differences to get the symmetric differences.

```

\seq_set_eq:NN \l\_ \langle pkg \rangle\_ internal\_ seq <seq var_1>
\seq_map_inline:Nn <seq var_2>
{ \seq_remove_all:Nn \l\_ \langle pkg \rangle\_ internal\_ seq {#1} }
\seq_set_eq:NN <seq var_3> <seq var_2>
\seq_map_inline:Nn <seq var_1>
{ \seq_remove_all:Nn <seq var_3> {#1} }
\seq_concat:NNN <seq var_3> <seq var_3> \l\_ \langle pkg \rangle\_ internal\_ seq

```

20.11 Constant and scratch sequences

$\backslash c_ empty_ seq$ Constant that is always empty.

New: 2012-07-02

`\l_tmpa_seq` Scratch sequences for local assignment. These are never used by the kernel code, and so
`\l_tmpb_seq` are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by
New: 2012-04-26 other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_seq` Scratch sequences for global assignment. These are never used by the kernel code, and
`\g_tmpb_seq` so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten
New: 2012-04-26 by other non-kernel code and so should only be used for short-term storage.

20.12 Viewing sequences

`\seq_show:N` `\seq_show:N` $\langle seq\ var \rangle$
`\seq_show:c` Displays the entries in the $\langle seq\ var \rangle$ in the terminal.
Updated: 2021-04-29

`\seq_log:N` `\seq_log:N` $\langle seq\ var \rangle$
`\seq_log:c` Writes the entries in the $\langle seq\ var \rangle$ in the log file.
New: 2014-08-12
Updated: 2021-04-29

Chapter 21

The `\l3int` module Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* (“`<int expr>`”).

21.1 Integer expressions

Throughout this module, (almost) all n-type argument allow for an `<intexpr>` argument with the following syntax. The `<integer expression>` should consist, after expansion, of `+`, `-`, `*`, `/`, `(`, `)` and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- `/` denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling operations $a*b/c$, for which $a*b$ may be arbitrarily large (but the operands a , b , c are still constrained to an absolute value at most $2^{31} - 1$);
- parentheses may not appear after unary `+` or `-`, namely placing `+(` or `-(` at the start of an expression or after `+`, `-`, `*`, `/` or `(` leads to an error.

Each integer operand can be either an integer variable (with no need for `\int_use:N`) or an integer denotation. For example both

```
\int_show:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
\tl_new:N \l_my_tl  
\tl_set:Nn \l_my_tl { 5 }  
\int_new:N \l_my_int  
\int_set:Nn \l_my_int { 4 }  
\int_show:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

show the same result -6 because `\l_my_tl` expands to the integer denotation 5 while the integer variable `\l_my_int` takes the value 4 . As the *integer expression* is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and `\exp_not:n` and its variants have no effect while `\exp_not:N` may incorrectly interrupt the expression.

TeXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an *internal integer*, and therefore should be terminated by a space if used in `\int_value:w` or in a TeX-style integer assignment.

As all TeX integers, integer operands can also be: `\value{LaTeX 2ε counter}`; dimension or skip variables, converted to integers in `sp`; the character code of some character given as `'char'` or `\'char'`; octal numbers given as `'` followed by digits from 0 to 7 ; or hexadecimal numbers given as `"` followed by digits and upper case letters from `A` to `F`.

`\int_eval:n` * `\int_eval:n` {*int expr*}

Evaluates the *int expr* and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with 0, for negative results – followed by such a sequence, and 0 for zero. The *int expr* should consist, after expansion, of +, –, *, /, (,) and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- / denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling operations $a*b/c$, for which $a*b$ may be arbitrarily large;
- parentheses may not appear after unary + or –, namely placing +(or -(at the start of an expression or after +, –, *, / or (leads to an error.

Each integer operand can be either an integer variable (with no need for `\int_use:N`) or an integer denotation. For example both

```
\int_eval:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
\tl_new:N \l_my_tl  
\tl_set:Nn \l_my_tl { 5 }  
\int_new:N \l_my_int  
\int_set:Nn \l_my_int { 4 }  
\int_eval:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

evaluate to -6 because `\l_my_tl` expands to the integer denotation 5. As the *int expr* is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and `\exp_not:n` and its variants have no effect while `\exp_not:N` may incorrectly interrupt the expression.

TeXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an *internal integer*, and therefore requires suitable termination if used in a TeX-style integer assignment.

As all TeX integers, integer operands can also be dimension or skip variables, converted to integers in `sp`, or octal numbers given as `'` followed by digits other than 8 and 9, or hexadecimal numbers given as `"` followed by digits or upper case letters from A to F, or the character code of some character or one-character control sequence, given as `'<char>`.

`\int_eval:w` * `\int_eval:w` {*int expr*}

New: 2018-03-30

Evaluates the *int expr* as described for `\int_eval:n`. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is `\scan_stop`: it is removed, otherwise not. Spaces do *not* terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, `\int_eval:w 1_+_1_9` (with explicit space tokens inserted using `~` in a code setting) expands to 29 since the digit 9 is not part of the expression. Expansion details, etc., are as given for `\int_eval:n`.

`\int_sign:n` * `\int_sign:n` {*int expr*}

New: 2018-11-03 Evaluates the *int expr* then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

`\int_abs:n` * `\int_abs:n` {*int expr*}

Updated: 2012-09-26 Evaluates the *int expr* as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an *integer denotation* after two expansions.

`\int_div_round:nn` * `\int_div_round:nn` {*int expr*₁} {*int expr*₂}

Updated: 2012-09-26 Evaluates the two *int expr*s as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using `/` directly in an *int expr*. The result is left in the input stream as an *integer denotation* after two expansions.

`\int_div_truncate:nn` * `\int_div_truncate:nn` {*int expr*₁} {*int expr*₂}

Updated: 2012-02-09 Evaluates the two *int expr*s as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using `/` rounds to the closest integer instead. The result is left in the input stream as an *integer denotation* after two expansions.

`\int_max:nn` * `\int_max:nn` {*int expr*₁} {*int expr*₂}

`\int_min:nn` * `\int_min:nn` {*int expr*₁} {*int expr*₂}

Updated: 2012-09-26 Evaluates the *int expr*s as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream as an *integer denotation* after two expansions.

`\int_mod:nn` * `\int_mod:nn` {*int expr*₁} {*int expr*₂}

Updated: 2012-09-26 Evaluates the two *int expr*s as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting `\int_div_truncate:nn` {*int expr*₁} {*int expr*₂} times *int expr*₂ from *int expr*₁. Thus, the result has the same sign as *int expr*₁ and its absolute value is strictly less than that of *int expr*₂. The result is left in the input stream as an *integer denotation* after two expansions.

21.2 Creating and initialising integers

`\int_new:N` `\int_new:N` *integer*

`\int_new:c` Creates a new *integer* or raises an error if the name is already taken. The declaration is global. The *integer* is initially equal to 0.

`\int_const:Nn` `\int_const:Nn` *integer* {*int expr*}

`\int_const:cn` Creates a new constant *integer* or raises an error if the name is already taken. The value of the *integer* is set globally to the *int expr*.

Updated: 2011-10-22

`\int_zero:N` `\int_zero:N` $\langle integer \rangle$
`\int_zero:c`
`\int_gzero:N` Sets $\langle integer \rangle$ to 0.
`\int_gzero:c`

`\int_zero_new:N` `\int_zero_new:N` $\langle integer \rangle$
`\int_zero_new:c`
`\int_gzero_new:N` Ensures that the $\langle integer \rangle$ exists globally by applying `\int_new:N` if necessary, then
`\int_gzero_new:c` applies `\int_(g)zero:N` to leave the $\langle integer \rangle$ set to zero.

New: 2011-12-13

`\int_set_eq:NN` `\int_set_eq:NN` $\langle integer_1 \rangle$ $\langle integer_2 \rangle$
`\int_set_eq:(cN|Nc|cc)`
`\int_gset_eq:NN` Sets the content of $\langle integer_1 \rangle$ equal to that of $\langle integer_2 \rangle$.
`\int_gset_eq:(cN|Nc|cc)`

`\int_if_exist_p:N` * `\int_if_exist_p:N` $\langle int \rangle$
`\int_if_exist_p:c` * `\int_if_exist:NTF` $\langle int \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
`\int_if_exist:NTF` * Tests whether the $\langle int \rangle$ is currently defined. This does not check that the $\langle int \rangle$ really
`\int_if_exist:cTF` * is an integer variable.

New: 2012-03-03

21.3 Setting and incrementing integers

`\int_add:Nn` `\int_add:Nn` $\langle integer \rangle$ $\{\langle int expr \rangle\}$
`\int_add:cn`
`\int_gadd:Nn` Adds the result of the $\langle int expr \rangle$ to the current content of the $\langle integer \rangle$.
`\int_gadd:cn`

Updated: 2011-10-22

`\int_decr:N` `\int_decr:N` $\langle integer \rangle$
`\int_decr:c`
`\int_gdecr:N` Decreases the value stored in $\langle integer \rangle$ by 1.
`\int_gdecr:c`

`\int_incr:N` `\int_incr:N` $\langle integer \rangle$
`\int_incr:c`
`\int_gincr:N` Increases the value stored in $\langle integer \rangle$ by 1.
`\int_gincr:c`

`\int_set:Nn` `\int_set:Nn` $\langle integer \rangle$ $\{\langle int expr \rangle\}$
`\int_set:cn`
`\int_gset:Nn` Sets $\langle integer \rangle$ to the value of $\langle int expr \rangle$, which must evaluate to an integer (as de-
`\int_gset:cn` scribed for `\int_eval:n`).

Updated: 2011-10-22

<code>\int_sub:Nn</code>	<code>\int_sub:Nn <integer> {<int expr>}</code>
<code>\int_sub:cn</code>	
<code>\int_gsub:Nn</code>	Subtracts the result of the <code><int expr></code> from the current content of the <code><integer></code> .
<code>\int_gsub:cn</code>	

Updated: 2011-10-22

21.4 Using integers

<code>\int_use:N</code>	<code>\int_use:N <integer></code>
<code>\int_use:c</code>	

Updated: 2011-10-22

Recovers the content of an `<integer>` and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an `<integer>` is required (such as in the first and third arguments of `\int_compare:nNnTF`).

TeXhackers note: `\int_use:N` is the TeX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

21.5 Integer expression conditionals

<code>\int_compare_p:nNn</code>	<code>\int_compare_p:nNn {<int expr₁>} <relation> {<int expr₂>}</code>
<code>\int_compare:nNnTF</code>	<code>\int_compare:nNnTF</code>

`{<int expr1>} <relation> {<int expr2>}`
`{<true code>} {<false code>}`

This function first evaluates each of the `<int expr>`s as described for `\int_eval:n`. The two results are then compared using the `<relation>`:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\int_compare:nTF` but around 5 times faster.

```

\int_compare_p:n * \int_compare_p:n
\int_compare:nTF * {
  Updated: 2013-01-13
  <int expr1> <relation1>
  ...
  <int exprN> <relationN>
  <int exprN+1>
}
\int_compare:nTF
{
  <int expr1> <relation1>
  ...
  <int exprN> <relationN>
  <int exprN+1>
}
{(true code)} {(false code)}

```

This function evaluates the `<int expr>`s as described for `\int_eval:n` and compares consecutive result using the corresponding `<relation>`, namely it compares `<int expr1>` and `<int expr2>` using the `<relation1>`, then `<int expr2>` and `<int expr3>` using the `<relation2>`, until finally comparing `<int exprN>` and `<int exprN+1>` using the `<relationN>`. The test yields `true` if all comparisons are `true`. Each `<int expr>` is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other `<integer expression>` is evaluated and no other comparison is performed. The `<relations>` can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

This function is more flexible than `\int_compare:nNnTF` but around 5 times slower.

```

\int_case:nn * \int_case:nnTF {<test int expr>}
\int_case:nnTF * {
  {<int expr case1>} {<code case1>}
  {<int expr case2>} {<code case2>}
  ...
  {<int expr casen>} {<code casen>}
}
{<true code>}
{<false code>}

```

New: 2013-07-24

This function evaluates the $\langle test\ int\ expr \rangle$ and compares this in turn to each of the $\langle int\ expr\ cases \rangle$. If the two are equal then the associated $\langle code \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function $\backslash int_case:nn$, which does nothing if there is no match, is also available. For example

```

\int_case:nnF
{ 2 * 5 }
{
  { 5 }      { Small }
  { 4 + 6 }  { Medium }
  { -2 * 10 } { Negative }
}
{ No idea! }

```

leaves “Medium” in the input stream.

```

\int_if_even_p:n * \int_if_odd_p:n {<int expr>}
\int_if_even:nTF * \int_if_odd:nTF {<int expr>}
\int_if_odd_p:n * {<true code>} {<false code>}
\int_if_odd:nTF *

```

This function first evaluates the $\langle int\ expr \rangle$ as described for $\backslash int_eval:n$. It then evaluates if this is odd or even, as appropriate.

```

\int_if_zero_p:n * \int_if_zero_p:n {<int expr>}
\int_if_zero:nTF * \int_if_zero:nTF {<int expr>}
                    {<true code>} {<false code>}

```

New: 2023-05-17

This function first evaluates the $\langle int\ expr \rangle$ as described for $\backslash int_eval:n$. It then evaluates if this is zero or not.

21.6 Integer expression loops

```

\int_do_until:nNnn ☆ \int_do_until:nNnn {<int expr1>} <relation> {<int expr2>} {<code>}

```

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle int\ expr \rangle$ s as described for $\backslash int_compare:nNnTF$. If the test is *false* then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is *true*.

<code>\int_do_while:nNnn</code> ☆	<code>\int_do_while:nNnn {<int expr1>} <relation> {<int expr2>} {<code>}</code>
	Places the <code><code></code> in the input stream for \TeX to process, and then evaluates the relationship between the two <code><int expr></code> s as described for <code>\int_compare:nNnTF</code> . If the test is <code>true</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is <code>false</code> .
<code>\int_until_do:nNnn</code> ☆	<code>\int_until_do:nNnn {<int expr1>} <relation> {<int expr2>} {<code>}</code>
	Evaluates the relationship between the two <code><int expr></code> s as described for <code>\int_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>false</code> . After the <code><code></code> has been processed by \TeX the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\int_while_do:nNnn</code> ☆	<code>\int_while_do:nNnn {<int expr1>} <relation> {<int expr2>} {<code>}</code>
	Evaluates the relationship between the two <code><int expr></code> s as described for <code>\int_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>true</code> . After the <code><code></code> has been processed by \TeX the test is repeated, and a loop occurs until the test is <code>false</code> .
<code>\int_do_until:nn</code> ☆	<code>\int_do_until:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <code><code></code> in the input stream for \TeX to process, and then evaluates the <code><integer relation></code> as described for <code>\int_compare:nTF</code> . If the test is <code>false</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is <code>true</code> .
<code>\int_do_while:nn</code> ☆	<code>\int_do_while:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <code><code></code> in the input stream for \TeX to process, and then evaluates the <code><integer relation></code> as described for <code>\int_compare:nTF</code> . If the test is <code>true</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is <code>false</code> .
<code>\int_until_do:nn</code> ☆	<code>\int_until_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <code><integer relation></code> as described for <code>\int_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>false</code> . After the <code><code></code> has been processed by \TeX the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\int_while_do:nn</code> ☆	<code>\int_while_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <code><integer relation></code> as described for <code>\int_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>true</code> . After the <code><code></code> has been processed by \TeX the test is repeated, and a loop occurs until the test is <code>false</code> .

21.7 Integer step functions

<code>\int_step_function:nN</code>	☆	<code>\int_step_function:nN</code>	{ <i>final value</i> }	{ <i>function</i> }		
<code>\int_step_function:nnN</code>	☆	<code>\int_step_function:nnN</code>	{ <i>initial value</i> }	{ <i>final value</i> }	{ <i>function</i> }	
<code>\int_step_function:nnnN</code>	☆	<code>\int_step_function:nnnN</code>	{ <i>initial value</i> }	{ <i>step</i> }	{ <i>final value</i> }	{ <i>function</i> }

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the *initial value*, *step* and *final value*, all of which should be integer expressions. The *function* is then placed in front of each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*). The *step* must be non-zero. If the *step* is positive, the loop stops when the *value* becomes larger than the *final value*. If the *step* is negative, the loop stops when the *value* becomes smaller than the *final value*. The *function* should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

```
[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]
```

The functions `\int_step_function:nN` and `\int_step_function:nnN` both use a fixed *step* of 1, and in the case of `\int_step_function:nN` the *initial value* is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_inline:nn</code>		<code>\int_step_inline:nn</code>	{ <i>final value</i> }	{ <i>code</i> }		
<code>\int_step_inline:nnn</code>		<code>\int_step_inline:nnn</code>	{ <i>initial value</i> }	{ <i>final value</i> }	{ <i>code</i> }	
<code>\int_step_inline:nnnn</code>		<code>\int_step_inline:nnnn</code>	{ <i>initial value</i> }	{ <i>step</i> }	{ <i>final value</i> }	{ <i>code</i> }

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the *initial value*, *step* and *final value*, all of which should be integer expressions. Then for each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*), the *code* is inserted into the input stream with `#1` replaced by the current *value*. Thus the *code* should define a function of one argument (`#1`).

The functions `\int_step_inline:nn` and `\int_step_inline:nnn` both use a fixed *step* of 1, and in the case of `\int_step_inline:nn` the *initial value* is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_variable:nNn</code>		<code>\int_step_variable:nNn</code>	{ <i>final value</i> }	{ <i>t1 var</i> }	{ <i>code</i> }		
<code>\int_step_variable:nnNn</code>		<code>\int_step_variable:nnNn</code>	{ <i>initial value</i> }	{ <i>final value</i> }	{ <i>t1 var</i> }	{ <i>code</i> }	
<code>\int_step_variable:nnnNn</code>		<code>\int_step_variable:nnnNn</code>	{ <i>initial value</i> }	{ <i>step</i> }	{ <i>final value</i> }	{ <i>t1 var</i> }	{ <i>code</i> }

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the *initial value*, *step* and *final value*, all of which should be integer expressions. Then for each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*), the *code* is inserted into the input stream, with the *t1 var* defined as the current *value*. Thus the *code* should make use of the *t1 var*.

The functions `\int_step_variable:nNn` and `\int_step_variable:nnNn` both use a fixed *step* of 1, and in the case of `\int_step_variable:nNn` the *initial value* is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

21.8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

`\int_to_arabic:n` * `\int_to_arabic:n {⟨int expr⟩}`
`\int_to_arabic:v` * Places the value of the `⟨int expr⟩` in the input stream as digits, with category code 12 (other).
Updated: 2011-10-22

`\int_to_alph:n` * `\int_to_alph:n {⟨int expr⟩}`
`\int_to_Alph:n` * Evaluates the `⟨int expr⟩` and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus
Updated: 2011-09-17

`\int_to_alph:n { 1 }`

places a in the input stream,

`\int_to_alph:n { 26 }`

is represented as z and

`\int_to_alph:n { 27 }`

is converted to aa. For conversions using other alphabets, use `\int_to_symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alph:n` and `\int_to_Alph:n` functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_symbols:nnn` * `\int_to_symbols:nnn`
`{⟨int expr⟩ {⟨total symbols⟩}`
`{⟨value to symbol mapping⟩}`
Updated: 2011-09-17

This is the low-level function for conversion of an `⟨int expr⟩` into a symbolic form (often letters). The `⟨total symbols⟩` available should be given as an integer expression. Values are actually converted to symbols according to the `⟨value to symbol mapping⟩`. This should be given as `⟨total symbols⟩` pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alph:n` function is defined as

```
\cs_new:Npn \int_to_alph:n #1
{
  \int_to_symbols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    ...
    { 26 } { z }
  }
}
```

`\int_to_bin:n` \star `\int_to_bin:n` $\{ \langle int\ expr \rangle \}$
New: 2014-02-11 Calculates the value of the $\langle int\ expr \rangle$ and places the binary representation of the result in the input stream.

`\int_to_hex:n` \star `\int_to_hex:n` $\{ \langle int\ expr \rangle \}$
`\int_to_Hex:n` \star Calculates the value of the $\langle int\ expr \rangle$ and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for `\int_to_hex:n` and upper case ones for `\int_to_Hex:n`. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
New: 2014-02-11

`\int_to_oct:n` \star `\int_to_oct:n` $\{ \langle int\ expr \rangle \}$
New: 2014-02-11 Calculates the value of the $\langle int\ expr \rangle$ and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_base:nn` \star `\int_to_base:nn` $\{ \langle int\ expr \rangle \} \{ \langle base \rangle \}$
`\int_to_Base:nn` \star Calculates the value of the $\langle int\ expr \rangle$ and converts it into the appropriate representation in the $\langle base \rangle$; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for `\int_to_base:n` and upper case ones for `\int_to_Base:n`. The maximum $\langle base \rangle$ value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
Updated: 2014-02-11

TeXhackers note: This is a generic version of `\int_to_bin:n`, *etc.*

`\int_to_roman:n` \star `\int_to_roman:n` $\{ \langle int\ expr \rangle \}$
`\int_to_Roman:n` \star Places the value of the $\langle int\ expr \rangle$ in the input stream as Roman numerals, either lower case (`\int_to_roman:n`) or upper case (`\int_to_Roman:n`). If the value is negative or zero, the output is empty. The Roman numerals are letters with category code 11 (letter). The letters used are `mdclxvi`, repeated as needed: the notation with bars (such as \bar{v} for 5000) is *not* used. For instance `\int_to_roman:n { 8249 }` expands to `mmmmmmmmccclix`.
Updated: 2011-10-22

21.9 Converting from other formats to integers

`\int_from_alph:n` \star `\int_from_alph:n` $\{ \langle letters \rangle \}$
Updated: 2014-08-25 Converts the $\langle letters \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle letters \rangle$ are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with “a” equal to 1 through to “z” equal to 26. The function also accepts a leading sign, made of + and -. This is the inverse function of `\int_to_alph:n` and `\int_to_Alph:n`.

`\int_from_bin:n` \star `\int_from_bin:n` $\{ \langle binary\ number \rangle \}$
New: 2014-02-11
Updated: 2014-08-25 Converts the $\langle binary\ number \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle binary\ number \rangle$ is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by binary digits. This is the inverse function of `\int_to_bin:n`.

`\int_from_hex:n` ★ `\int_from_hex:n {⟨hexadecimal number⟩}`

New: 2014-02-11
Updated: 2014-08-25
Converts the `⟨hexadecimal number⟩` into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the `⟨hexadecimal number⟩` by upper or lower case letters. The `⟨hexadecimal number⟩` is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and -. This is the inverse function of `\int_to_hex:n` and `\int_to_Hex:n`.

`\int_from_oct:n` ★ `\int_from_oct:n {⟨octal number⟩}`

New: 2014-02-11
Updated: 2014-08-25
Converts the `⟨octal number⟩` into the integer (base 10) representation and leaves this in the input stream. The `⟨octal number⟩` is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by octal digits. This is the inverse function of `\int_to_oct:n`.

`\int_from_roman:n` ★ `\int_from_roman:n {⟨roman numeral⟩}`

Updated: 2014-08-25
Converts the `⟨roman numeral⟩` into the integer (base 10) representation and leaves this in the input stream. The `⟨roman numeral⟩` is first converted to a string, with no expansion. The `⟨roman numeral⟩` may be in upper or lower case; if the numeral contains characters besides `mdclxvi` or `MDCLXVI` then the resulting value is -1. This is the inverse function of `\int_to_roman:n` and `\int_to_Roman:n`.

`\int_from_base:nn` ★ `\int_from_base:nn {⟨number⟩} {⟨base⟩}`

Updated: 2014-08-25
Converts the `⟨number⟩` expressed in `⟨base⟩` into the appropriate value in base 10. The `⟨number⟩` is first converted to a string, with no expansion. The `⟨number⟩` should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum `⟨base⟩` value is 36. This is the inverse function of `\int_to_base:nn` and `\int_to_Base:nn`.

21.10 Random integers

`\int_rand:nn` ★ `\int_rand:nn {⟨int expr1⟩} {⟨int expr2⟩}`

New: 2016-12-06
Updated: 2018-04-27
Evaluates the two `⟨int expr⟩`s and produces a pseudo-random number between the two (with bounds included). This is not available in older versions of X_ƒTEX.

`\int_rand:n` ★ `\int_rand:n {⟨int expr⟩}`

New: 2018-05-05
Evaluates the `⟨int expr⟩` then produces a pseudo-random number between 1 and the `⟨int expr⟩` (included). This is not available in older versions of X_ƒTEX.

21.11 Viewing integers

`\int_show:N` `\int_show:N ⟨integer⟩`

`\int_show:c` Displays the value of the `⟨integer⟩` on the terminal.

`\int_show:n` `\int_show:n {⟨int expr⟩}`
New: 2011-11-22 Displays the result of evaluating the `⟨int expr⟩` on the terminal.
Updated: 2015-08-07

`\int_log:N` `\int_log:N ⟨integer⟩`
`\int_log:c` Writes the value of the `⟨integer⟩` in the log file.
New: 2014-08-22
Updated: 2015-08-03

`\int_log:n` `\int_log:n {⟨int expr⟩}`
New: 2014-08-22 Writes the result of evaluating the `⟨int expr⟩` in the log file.
Updated: 2015-08-07

21.12 Constant integers

`\c_zero_int` Integer values used with primitive tests and assignments: their self-terminating nature
`\c_one_int` makes these more convenient and faster than literal numbers.
New: 2018-05-07

`\c_max_int` The maximum value that can be stored as an integer.

`\c_max_register_int` Maximum number of registers.

`\c_max_char_int` Maximum character code completely supported by the engine.

21.13 Scratch integers

`\l_tmpa_int` Scratch integer for local assignment. These are never used by the kernel code, and so
`\l_tmpb_int` are safe for use with any `LATEX3`-defined function. However, they may be overwritten by
other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_int` Scratch integer for global assignment. These are never used by the kernel code, and so
`\g_tmpb_int` are safe for use with any `LATEX3`-defined function. However, they may be overwritten by
other non-kernel code and so should only be used for short-term storage.

21.14 Direct number expansion

```
\int_value:w * \int_value:w <integer>  
New: 2018-03-27 \int_value:w <integer denotation> <optional space>
```

Expands the following tokens until an `<integer>` is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit 0 except for zero) in the input stream as category code 12 (other) characters. The `<integer>` can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any \TeX register except `\toks`) or
- explicit digits (or by `'<octal digits>` or `"<hexadecimal digits>` or `'<character>`).

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in `f`-expansion, and so `\exp_stop_f:` may be employed as an end marker. Note that protected functions *are* expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required “directly”. In general, `\int_eval:n` is the preferred approach to generating numbers.

\TeX hackers note: This is the \TeX primitive `\number`.

21.15 Primitive conditionals

```
\if_int_compare:w * \if_int_compare:w <integer12<true code>  
\else:  
<false code>  
\fi:
```

Compare two integers using `<relation>`, which must be one of `=`, `<` or `>` with category code 12. The `\else:` branch is optional.

\TeX hackers note: This is the \TeX primitive `\ifnum`.

```
\if_case:w * \if_case:w <integer> <case0>  
\or: * \or: <case1>  
\or: ...  
\else: <default>  
\fi:
```

Selects a case to execute based on the value of the `<integer>`. The first case (`<case0>`) is executed if `<integer>` is 0, the second (`<case1>`) if the `<integer>` is 1, *etc.* The `<integer>` may be a literal, a constant or an integer expression (*e.g.* using `\int_eval:n`).

\TeX hackers note: These are the \TeX primitives `\ifcase` and `\or`.

```
\if_int_odd:w * \if_int_odd:w <tokens> <optional space>
  <true code>
\else:
  <true code>
\fi:
```

Expands `<tokens>` until a non-numeric token or a space is found, and tests whether the resulting `<integer>` is odd. If so, `<true code>` is executed. The `\else:` branch is optional.

TeXhackers note: This is the TeX primitive `\ifodd`.

Chapter 22

The l3flag module

Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any (small) non-negative value, which we call its *height*. In expansion-only contexts, a flag can only be “raised”: this increases the *height* by 1. The *height* can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by `l3str-convert`, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height and that the memory cannot be reclaimed even if the flag is cleared. Flags should not be used unless it is unavoidable.

In earlier versions, flags were referenced by an n-type *flag name* such as `fp_overflow`, used as part of `\use:c` constructions. All of the commands described below have n-type analogues that can still appear in old code, but the N-type commands are to be preferred moving forward. The n-type *flag name* is simply mapped to `\l_<flag name>_flag`, which makes it easier for packages using public flags (such as `l3fp`) to retain backwards compatibility.

22.1 Setting up flags

`\flag_new:N` `\flag_new:N <flag var>`

`\flag_new:c`

Creates a new *flag var*, or raises an error if the name is already taken. The declaration is global, but flags are always local variables. The *flag var* initially has zero height.

New: 2024-01-12

`\flag_clear:N` `\flag_clear:N` $\langle flag\ var \rangle$
`\flag_clear:c` Sets the height of the $\langle flag\ var \rangle$ to zero. The assignment is local.
New: 2024-01-12

`\flag_clear_new:N` `\flag_clear_new:N` $\langle flag\ var \rangle$
`\flag_clear_new:c` Ensures that the $\langle flag\ var \rangle$ exists globally by applying `\flag_new:N` if necessary, then applies `\flag_clear:N`, setting the height to zero locally.
New: 2024-01-12

`\flag_show:N` `\flag_show:N` $\langle flag\ var \rangle$
`\flag_show:c` Displays the height of the $\langle flag\ var \rangle$ in the terminal.
New: 2024-01-12

`\flag_log:N` `\flag_log:N` $\langle flag\ var \rangle$
`\flag_log:c` Writes the height of the $\langle flag\ var \rangle$ in the log file.
New: 2024-01-12

22.2 Expandable flag commands

`\flag_if_exist_p:N` \star `\flag_if_exist_p:N` $\langle flag\ var \rangle$
`\flag_if_exist_p:c` \star `\flag_if_exist:NTF` $\langle flag\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
`\flag_if_exist:NTF` \star This function returns `true` if the $\langle flag\ var \rangle$ is currently defined, and `false` otherwise.
`\flag_if_exist:cTF` \star This does not check that the $\langle flag\ var \rangle$ really is a flag variable.
New: 2024-01-12

`\flag_if_raised_p:N` \star `\flag_if_raised_p:N` $\langle flag\ var \rangle$
`\flag_if_raised_p:c` \star `\flag_if_raised:NTF` $\langle flag\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
`\flag_if_raised:NTF` \star This function returns `true` if the $\langle flag\ var \rangle$ has non-zero height, and `false` if the
`\flag_if_raised:cTF` \star $\langle flag\ var \rangle$ has zero height.
New: 2024-01-12

`\flag_height:N` \star `\flag_height:N` $\langle flag\ var \rangle$
`\flag_height:c` \star Expands to the height of the $\langle flag\ var \rangle$ as an integer denotation.
New: 2024-01-12

`\flag_raise:N` \star `\flag_raise:N` $\langle flag\ var \rangle$
`\flag_raise:c` \star The height of $\langle flag\ var \rangle$ is increased by 1 locally.
New: 2024-01-12

`\flag_ensure_raised:N` \star `\flag_ensure_raised:N` $\langle flag\ var \rangle$
`\flag_ensure_raised:c` \star Ensures the $\langle flag\ var \rangle$ is raised by making its height at least 1, locally.
New: 2024-01-12

`\l_tmpa_flag` Scratch flag for local assignment. These are never used by the kernel code, and so are safe
`\l_tmpb_flag` for use with any \LaTeX 3-defined function. However, they may be overwritten by other
New: 2024-01-12 non-kernel code and so should only be used for short-term storage.

Chapter 23

The `l3clist` module

Comma separated lists

Comma lists (in short, `clist`) contain ordered data where items can be added to the left or right end of the list. This data type allows basic list manipulations such as adding/removing items, applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on. Sequences (defined in `l3seq`) are safer, faster, and provide more features, so they should often be preferred to comma lists. Comma lists are mostly useful when interfacing with $\LaTeX 2_{\epsilon}$ or other code that expects or provides items separated by commas.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an `\ExplSyntaxOn ... \ExplSyntaxOff` block, spaces are removed from both sides of each comma-delimited argument upon input. Blank arguments are ignored, to allow for trailing commas or repeated commas (which may otherwise arise when concatenating comma lists “by hand”). In addition, a set of braces is removed if the result of space-trimming is braced: this allows the storage of any item in a comma list. For instance,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~a~ , ~{b}~ , c~\d }
\clist_put_right:Nn \l_my_clist { ~{e~} , , {f} } , }
```

results in `\l_my_clist` containing `a,b,c~\d,{e~},{f}` namely the five items `a`, `b`, `c~\d`, `e~` and `{f}`. Comma lists normally do not contain empty or blank items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_set:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

and it leaves `true` in the input stream. To include an “unsafe” item (empty, or one that contains a comma, or starts or ends with a space, or is a single brace group), surround it with braces.

Any `n`-type token list is a valid comma list input for `l3clist` functions, which will split the token list at every comma and process the items as described above. On the other hand, `N`-type functions expect comma list variables, which are particular token list variables in which this processing of items (and removal of blank items) has already

occurred. Because comma list variables are token list variables, expanding them once yields their items separated by commas, and `\tl_show:N` can be applied to them. (These functions often have `\clist` analogues, which should be preferred.)

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using `\seq_set_from_clist:Nn` (see `\l3seq`) may be advisable if speed is important. The exception is that `\clist_if_in:NnTF` and `\clist_remove_duplicates:N` may be faster than their sequence analogues for large lists. However, these functions work slowly for “unsafe” items that must be braced, and may produce errors when their argument contains `{`, `}` or `#` (assuming the usual `TeX` category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

23.1 Creating and initialising comma lists

`\clist_new:N` `\clist_new:N` \langle *clist var* \rangle

`\clist_new:c` Creates a new \langle *clist var* \rangle or raises an error if the name is already taken. The declaration is global. The \langle *clist var* \rangle initially contains no items.

`\clist_const:Nn` `\clist_const:Nn` \langle *clist var* \rangle $\{$ *comma list* $\}$

`\clist_const:(Ne|cn|ce)` Creates a new constant \langle *clist var* \rangle or raises an error if the name is already taken. The value of the \langle *clist var* \rangle is set globally to the *comma list*.

New: 2014-07-05

`\clist_clear:N` `\clist_clear:N` \langle *clist var* \rangle

`\clist_clear:c` Clears all items from the \langle *clist var* \rangle .

`\clist_gclear:N`

`\clist_gclear:c`

`\clist_clear_new:N` `\clist_clear_new:N` \langle *clist var* \rangle

`\clist_clear_new:c` Ensures that the \langle *clist var* \rangle exists globally by applying `\clist_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

`\clist_gclear_new:N`

`\clist_gclear_new:c`

`\clist_set_eq:NN` `\clist_set_eq:NN` \langle *clist var*₁ \rangle \langle *clist var*₂ \rangle

`\clist_set_eq:(cN|Nc|cc)` Sets the content of \langle *clist var*₁ \rangle equal to that of \langle *clist var*₂ \rangle . To set a token list variable equal to a comma list variable, use `\tl_set_eq:NN`. Conversely, setting a comma list variable to a token list is unadvisable unless one checks space-trimming and related issues.

`\clist_gset_eq:NN`

`\clist_gset_eq:(cN|Nc|cc)`

`\clist_set_from_seq:NN` `\clist_set_from_seq:NN` \langle *clist var* \rangle \langle *seq var* \rangle

`\clist_set_from_seq:(cN|Nc|cc)`

`\clist_gset_from_seq:NN`

`\clist_gset_from_seq:(cN|Nc|cc)`

New: 2014-07-17

Converts the data in the \langle *seq var* \rangle into a \langle *clist var* \rangle : the original \langle *seq var* \rangle is unchanged. Items which contain either spaces or commas are surrounded by braces.

<code>\clist_concat:NNN</code>	<code>\clist_concat:NNN</code>	<code><clist var₁></code>	<code><clist var₂></code>	<code><clist var₃></code>
<code>\clist_concat:ccc</code>	Concatenates the content of <code><clist var₂></code> and <code><clist var₃></code> together and saves the result in <code><clist var₁></code> . The items in <code><clist var₂></code> are placed at the left side of the new comma list.			
<code>\clist_gconcat:NNN</code>				
<code>\clist_gconcat:ccc</code>				

<code>\clist_if_exist_p:N</code>	*	<code>\clist_if_exist_p:N</code>	<code><clist var></code>
<code>\clist_if_exist_p:c</code>	*	<code>\clist_if_exist:NTF</code>	<code><clist var></code> <code>{<true code>}</code> <code>{<false code>}</code>
<code>\clist_if_exist:NTF</code>	*	Tests whether the <code><clist var></code> is currently defined. This does not check that the	
<code>\clist_if_exist:cTF</code>	*	<code><clist var></code> really is a comma list.	

New: 2012-03-03

23.2 Adding data to comma lists

<code>\clist_set:Nn</code>	<code>\clist_set:Nn</code>	<code><clist var></code>	<code>{<item₁>, ..., <item_n>}</code>
<code>\clist_set:(NV Ne No cn cV ce co)</code>			
<code>\clist_gset:Nn</code>			
<code>\clist_gset:(NV Ne No cn cV ce co)</code>			

New: 2011-09-06

Sets `<clist var>` to contain the `<items>`, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some `<tokens>` as a single `<item>` even if the `<tokens>` contain commas or spaces, add a set of braces: `\clist_set:Nn <clist var> { {<tokens> } }`.

<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn</code>	<code><clist var></code>	<code>{<item₁>, ..., <item_n>}</code>
<code>\clist_put_left:(NV Nv Ne No cn cV cv ce co)</code>			
<code>\clist_gput_left:Nn</code>			
<code>\clist_gput_left:(NV Nv Ne No cn cV cv ce co)</code>			

Updated: 2011-09-05

Appends the `<items>` to the left of the `<clist var>`. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some `<tokens>` as a single `<item>` even if the `<tokens>` contain commas or spaces, add a set of braces: `\clist_put_left:Nn <clist var> { {<tokens> } }`.

<code>\clist_put_right:Nn</code>	<code>\clist_put_right:Nn</code>	<code><clist var></code>	<code>{<item₁>, ..., <item_n>}</code>
<code>\clist_put_right:(NV Nv Ne No cn cV cv ce co)</code>			
<code>\clist_gput_right:Nn</code>			
<code>\clist_gput_right:(NV Nv Ne No cn cV cv ce co)</code>			

Updated: 2011-09-05

Appends the `<items>` to the right of the `<clist var>`. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some `<tokens>` as a single `<item>` even if the `<tokens>` contain commas or spaces, add a set of braces: `\clist_put_right:Nn <clist var> { {<tokens> } }`.

23.3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

```
\clist_remove_duplicates:N \clist_remove_duplicates:N <clist var>
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c
```

Removes duplicate items from the $\langle\textit{clist var}\rangle$, leaving the left most copy of each item in the $\langle\textit{clist var}\rangle$. The $\langle\textit{item}\rangle$ comparison takes place on a token basis, as for $\backslash\textit{tl_if_eq:nnTF}$.

TeXhackers note: This function iterates through every item in the $\langle\textit{clist var}\rangle$ and does a comparison with the $\langle\textit{items}\rangle$ already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the $\langle\textit{clist var}\rangle$ contains $\{, \}$, or $\#$ (assuming the usual TeX category codes apply).

```
\clist_remove_all:Nn \clist_remove_all:Nn <clist var> {\item}
\clist_remove_all:(cn|NV|cV)
\clist_gremove_all:Nn
\clist_gremove_all:(cn|NV|cV)
```

Updated: 2011-09-06

Removes every occurrence of $\langle\textit{item}\rangle$ from the $\langle\textit{clist var}\rangle$. The $\langle\textit{item}\rangle$ comparison takes place on a token basis, as for $\backslash\textit{tl_if_eq:nnTF}$.

TeXhackers note: The function may fail if the $\langle\textit{item}\rangle$ contains $\{, \}$, or $\#$ (assuming the usual TeX category codes apply).

```
\clist_reverse:N \clist_reverse:N <clist var>
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c
```

New: 2014-07-18

```
\clist_reverse:n \clist_reverse:n {\comma list}
```

New: 2014-07-18

Leaves the items in the $\langle\textit{comma list}\rangle$ in the input stream in reverse order. Contrarily to other what is done for other n-type $\langle\textit{comma list}\rangle$ arguments, braces and spaces are preserved by this process.

TeXhackers note: The result is returned within $\backslash\textit{unexpanded}$, which means that the comma list does not expand further when appearing in an e-type or x-type argument expansion.

<code>\clist_sort:Nn</code>	<code>\clist_sort:Nn <clist var> {<comparison code>}</code>
<code>\clist_sort:cn</code>	
<code>\clist_gsort:Nn</code>	Sorts the items in the <code><clist var></code> according to the <code><comparison code></code> , and assigns the
<code>\clist_gsort:cn</code>	result to <code><clist var></code> . The details of sorting comparison are described in Section 6.1.

New: 2017-02-06

23.4 Comma list conditionals

<code>\clist_if_empty_p:N</code>	<code>\clist_if_empty_p:N <clist var></code>
<code>\clist_if_empty_p:c</code>	<code>\clist_if_empty:NTF <clist var> {<>true code>} {<>false code>}</code>
<code>\clist_if_empty:NTF</code>	Tests if the <code><clist var></code> is empty (containing no items).
<code>\clist_if_empty:cTF</code>	

<code>\clist_if_empty_p:n</code>	<code>\clist_if_empty_p:n {<comma list>}</code>
<code>\clist_if_empty:nTF</code>	<code>\clist_if_empty:nTF {<comma list>} {<>true code>} {<>false code>}</code>

New: 2014-07-05

Tests if the `<clist var>` is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list `{~,~,~,~}` (without outer braces) is empty, while `{~, { }, }` (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

<code>\clist_if_in:NnTF</code>	<code>\clist_if_in:NnTF <clist var> {<item>} {<>true code>} {<>false code>}</code>
<code>\clist_if_in:(NV No cn cV co)TF</code>	
<code>\clist_if_in:nnTF</code>	
<code>\clist_if_in:(nV no)TF</code>	

Updated: 2011-09-06

Tests if the `<item>` is present in the `<clist var>`. In the case of an n-type `<comma list>`, the usual rules of space trimming and brace stripping apply. Hence,

```
\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
```

yields true.

TeXhackers note: The function may fail if the `<item>` contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

23.5 Mapping over comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, *i.e.* any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

When the comma list is given explicitly, as an n-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is `{a, {b}, {c}, }` then the arguments passed to the mapped function are ‘a’, ‘{b}’, an empty argument, and ‘c’.

When the comma list is given as an N-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n-type comma lists.

<code>\clist_map_function:NN</code>	☆	<code>\clist_map_function:NN</code>	<code><clist var></code> <code><function></code>
<code>\clist_map_function:cN</code>	☆	Applies <code><function></code> to every <code><item></code> stored in the <code><clist var></code> . The <code><function></code> receives	
<code>\clist_map_function:nN</code>	☆	one argument for each iteration. The <code><items></code> are returned from left to right. The function	
<code>\clist_map_function:eN</code>	☆	<code>\clist_map_inline:Nn</code> is in general more efficient than <code>\clist_map_function:NN</code> .	

Updated: 2012-06-29

<code>\clist_map_inline:Nn</code>		<code>\clist_map_inline:Nn</code>	<code><clist var></code> <code>{<inline function>}</code>
<code>\clist_map_inline:cn</code>		Applies <code><inline function></code> to every <code><item></code> stored within the <code><clist var></code> . The	
<code>\clist_map_inline:nn</code>		<code><inline function></code> should consist of code which receives the <code><item></code> as #1. The <code><items></code>	

are returned from left to right.

Updated: 2012-06-29

<code>\clist_map_variable:NNn</code>		<code>\clist_map_variable:NNn</code>	<code><clist var></code> <code><variable></code> <code>{<code>}</code>
<code>\clist_map_variable:cNn</code>		Stores each <code><item></code> of the <code><clist var></code> in turn in the (token list) <code><variable></code> and applies	
<code>\clist_map_variable:nNn</code>		the <code><code></code> . The <code><code></code> will usually make use of the <code><variable></code> , but this is not enforced.	

The assignments to the `<variable>` are local. Its value after the loop is the last `<item>` in the `<clist var>`, or its original value if there were no `<item>`. The `<items>` are returned from left to right.

Updated: 2012-06-29

<code>\clist_map_tokens:Nn</code>	☆	<code>\clist_map_tokens:Nn</code>	<code><clist var></code> <code>{<code>}</code>
<code>\clist_map_tokens:cn</code>	☆	<code>\clist_map_tokens:nn</code>	<code>{<comma list>}</code> <code>{<code>}</code>
<code>\clist_map_tokens:nn</code>	☆	Calls <code><code></code> <code>{<item>}</code> for every <code><item></code> stored in the <code><clist var></code> . The <code><code></code> receives	

each `<item>` as a trailing brace group. If the `<code>` consists of a single function this is equivalent to `\clist_map_function:nN`.

New: 2021-05-05

<code>\clist_map_break:</code>	☆	<code>\clist_map_break:</code>	
--------------------------------	---	--------------------------------	--

Used to terminate a `\clist_map_...` function before all entries in the `<comma list>` have been processed. This normally takes place within a conditional statement, for example

Updated: 2012-06-29

```

\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break: }
  {
    % Do something useful
  }
}

```

Use outside of a `\clist_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\clist_map_break:n` ☆ `\clist_map_break:n {<code>}`

Updated: 2012-06-29

Used to terminate a `\clist_map_...` function before all entries in the *<comma list>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

`\clist_count:N` ☆ `\clist_count:N <clist var>`

`\clist_count:c` ☆

`\clist_count:n` ☆

`\clist_count:e` ☆

Leaves the number of items in the *<clist var>* in the input stream as an *<integer denotation>*. The total number of items in a *<clist var>* includes those which are duplicates, *i.e.* every item in a *<clist var>* is counted.

New: 2012-07-13

23.6 Using the content of comma lists directly

```
\clist_use:Nnnn * \clist_use:Nnnn <clist var> {<separator between two>}
\clist_use:cnnn * {<separator between more than two>} {<separator between final two>}
```

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the appropriate $\langle\textit{separator}\rangle$ between the items. Namely, if the comma list has more than two items, the $\langle\textit{separator between more than two}\rangle$ is placed between each pair of items except the last, for which the $\langle\textit{separator between final two}\rangle$ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the $\langle\textit{separator between two}\rangle$. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

T_EXhackers note: The result is returned within the $\backslash\textit{unexpanded}$ primitive ($\backslash\textit{exp_not:n}$), which means that the $\langle\textit{items}\rangle$ do not expand further when appearing in an e-type or x-type argument expansion.

```
\clist_use:Nn * \clist_use:Nn <clist var> {<separator>}
```

```
\clist_use:cn * *
```

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the $\langle\textit{separator}\rangle$ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

T_EXhackers note: The result is returned within the $\backslash\textit{unexpanded}$ primitive ($\backslash\textit{exp_not:n}$), which means that the $\langle\textit{items}\rangle$ do not expand further when appearing in an e-type or x-type argument expansion.

<code>\clist_use:nnnn</code>	<code>*</code>	<code>\clist_use:nnnn</code>	<code><comma list> {<separator between two>}</code>
<code>\clist_use:nn</code>	<code>*</code>		<code>{<separator between more than two>} {<separator between final two>}</code>
		<code>\clist_use:nn</code>	<code><comma list> {<separator>}</code>

New: 2021-05-10

Places the contents of the `<comma list>` in the input stream, with the appropriate `<separator>` between the items. As for `\clist_set:Nn`, blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. The `<separators>` are then inserted in the same way as for `\clist_use:Nnnn` and `\clist_use:Nn`, respectively.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<items>` do not expand further when appearing in an e-type or x-type argument expansion.

23.7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

<code>\clist_get:NN</code>	<code>\clist_get:NN</code>	<code><clist var> <token list variable></code>
----------------------------	----------------------------	--

`\clist_get:cN`

`\clist_get:NNTF`

`\clist_get:cNTF`

New: 2012-05-14

Updated: 2019-02-16

Stores the left-most item from a `<clist var>` in the `<token list variable>` without removing it from the `<clist var>`. The `<token list variable>` is assigned locally. In the non-branching version, if the `<clist var>` is empty the `<token list variable>` is set to the marker value `\q_no_value`.

<code>\clist_pop:NN</code>	<code>\clist_pop:NN</code>	<code><clist var> <token list variable></code>
----------------------------	----------------------------	--

`\clist_pop:cN`

Updated: 2011-09-06

Pops the left-most item from a `<clist var>` into the `<token list variable>`, *i.e.* removes the item from the comma list and stores it in the `<token list variable>`. Both of the variables are assigned locally.

<code>\clist_gpop:NN</code>	<code>\clist_gpop:NN</code>	<code><clist var> <token list variable></code>
-----------------------------	-----------------------------	--

`\clist_gpop:cN`

Pops the left-most item from a `<clist var>` into the `<token list variable>`, *i.e.* removes the item from the comma list and stores it in the `<token list variable>`. The `<clist var>` is modified globally, while the assignment of the `<token list variable>` is local.

<code>\clist_pop:NNTF</code>	<code>\clist_pop:NNTF</code>	<code><clist var> <token list variable> {<true code>} {<false code>}</code>
------------------------------	------------------------------	---

`\clist_pop:cNTF`

New: 2012-05-14

If the `<clist var>` is empty, leaves the `<false code>` in the input stream. The value of the `<token list variable>` is not defined in this case and should not be relied upon. If the `<clist var>` is non-empty, pops the top item from the `<clist var>` in the `<token list variable>`, *i.e.* removes the item from the `<clist var>`. Both the `<clist var>` and the `<token list variable>` are assigned locally.

`\clist_show:n` `\clist_show:n {tokens}`

Updated: 2013-08-03 Displays the entries in the comma list in the terminal.

`\clist_log:N` `\clist_log:N <clist var>`

`\clist_log:c`

Writes the entries in the `<clist var>` in the log file. See also `\clist_show:N` which displays the result in the terminal.

New: 2014-08-22

Updated: 2021-04-29

`\clist_log:n` `\clist_log:n {tokens}`

New: 2014-08-22 Writes the entries in the comma list in the log file. See also `\clist_show:n` which displays the result in the terminal.

23.10 Constant and scratch comma lists

`\c_empty_clist` Constant that is always empty.

New: 2012-07-02

`\l_tmpa_clist` Scratch comma lists for local assignment. These are never used by the kernel code, and
`\l_tmpb_clist` so are safe for use with any \LaTeX 3-defined function. However, they may be overwritten
New: 2011-09-06 by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_clist` Scratch comma lists for global assignment. These are never used by the kernel code, and
`\g_tmpb_clist` so are safe for use with any \LaTeX 3-defined function. However, they may be overwritten
New: 2011-09-06 by other non-kernel code and so should only be used for short-term storage.

Chapter 24

The `l3token` module Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in `TEX`, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its “shape” (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its “meaning”, which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of `TEX`, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, `TEX` distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }  
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section [24.7](#).

24.1 Creating character tokens

<code>\char_set_active_eq:NN</code> <code>\char_set_active_eq:Nc</code> <code>\char_gset_active_eq:NN</code> <code>\char_gset_active_eq:Nc</code>	<code>\char_set_active_eq:NN</code> $\langle char \rangle$ $\langle function \rangle$ Sets the behaviour of the $\langle char \rangle$ in situations where it is active (category code 13) to be equivalent to that of the definition of the $\langle function \rangle$ at the time <code>\char_set_active_eq:NN</code> is used. The category code of the $\langle char \rangle$ is <i>unchanged</i> by this process. The $\langle function \rangle$ may itself be an active character.
--	--

Updated: 2015-11-12

<code>\char_set_active_eq:nN</code> <code>\char_set_active_eq:nc</code> <code>\char_gset_active_eq:nN</code> <code>\char_gset_active_eq:nc</code>	<code>\char_set_active_eq:nN</code> $\{\langle integer\ expression \rangle\}$ $\langle function \rangle$ Sets the behaviour of the $\langle char \rangle$ which has character code as given by the $\langle integer\ expression \rangle$ in situations where it is active (category code 13) to be equivalent to that of the $\langle function \rangle$ at the time <code>\char_set_active_eq:nN</code> is used. The category code of the $\langle char \rangle$ is <i>unchanged</i> by this process. The $\langle function \rangle$ may itself be an active character.
--	--

New: 2015-11-12

<code>\char_generate:nn</code> \star	<code>\char_generate:nn</code> $\{\langle charcode \rangle\}$ $\{\langle catcode \rangle\}$
--	---

New: 2015-09-09
Updated: 2019-01-16

Generates a character token of the given $\langle charcode \rangle$ and $\langle catcode \rangle$ (both of which may be integer expressions). The $\langle catcode \rangle$ may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

and other values raise an error. The $\langle charcode \rangle$ may be any one valid for the engine in use, except that for $\langle catcode \rangle$ 10, $\langle charcode \rangle$ 0 is not allowed. Active characters cannot be generated in older versions of X_YTeX. Another way to build token lists with unusual category codes is `\regex_replace:nnN` $\{.*\}$ $\{\langle replacement \rangle\}$ $\langle t1\ var \rangle$.

TeXhackers note: Exactly two expansions are needed to produce the character.

<code>\c_catcode_active_space_tl</code>	Token list containing one character with category code 13, (“active”), and character code 32 (space).
---	---

New: 2017-08-07

<code>\c_catcode_other_space_tl</code>	Token list containing one character with category code 12, (“other”), and character code 32 (space).
<small>New: 2011-09-05</small>	

24.2 Manipulating and interrogating character tokens

```

\char_set_catcode_escape:N          \char_set_catcode_letter:N <character>
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N

```

Updated: 2015-11-11

Sets the category code of the `<character>` to that indicated in the function name. Depending on the current category code of the `<token>` the escape token may also be needed:

```
\char_set_catcode_other:N \%
```

The assignment is local.

```

\char_set_catcode_escape:n          \char_set_catcode_letter:n {⟨integer expression⟩}
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n

```

Updated: 2015-11-11

Sets the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

```

\char_set_catcode:nn \char_set_catcode:nn {⟨int expr1⟩} {⟨int expr2⟩}

```

Updated: 2015-11-11

These functions set the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. The first $\langle integer expression \rangle$ is the character code and the second is the category code to apply. The setting applies within the current \TeX group. In general, the symbolic functions $\backslash\text{char_set_catcode_}\langle type \rangle$ should be preferred, but there are cases where these lower-level functions may be useful.

```

\char_value_catcode:n * \char_value_catcode:n {⟨integer expression⟩}

```

Expands to the current category code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

```

\char_show_value_catcode:n \char_show_value_catcode:n {⟨integer expression⟩}

```

Displays the current category code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.

```

\char_set_lccode:nn \char_set_lccode:nn {⟨int expr1⟩} {⟨int expr2⟩}

```

Updated: 2015-08-06

Sets up the behaviour of the $\langle character \rangle$ when found inside $\backslash\text{text_lowercase:n}$, such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the \TeX ‘ $\langle character \rangle$ ’ method for converting a single character into its character code:

```

\char_set_lccode:nn { ‘\A } { ‘\a } % Standard behaviour
\char_set_lccode:nn { ‘\A } { ‘\A + 32 }
\char_set_lccode:nn { 50 } { 60 }

```

The setting applies within the current \TeX group.

<code>\char_value_lccode:n</code> *	<code>\char_value_lccode:n {<integer expression>}</code>	Expands to the current lower case code of the <code><character></code> with character code given by the <code><integer expression></code> .
<code>\char_show_value_lccode:n</code>	<code>\char_show_value_lccode:n {<integer expression>}</code>	Displays the current lower case code of the <code><character></code> with character code given by the <code><integer expression></code> on the terminal.
<code>\char_set_uccode:nn</code>	<code>\char_set_uccode:nn {<int expr1>} {<int expr2>}</code>	Sets up the behaviour of the <code><character></code> when found inside <code>\text_uppercase:n</code> , such that <code><character1></code> will be converted into <code><character2></code> . The two <code><characters></code> may be specified using an <code><integer expression></code> for the character code concerned. This may include the T _E X ‘ <code><character></code> ’ method for converting a single character into its character code:
<small>Updated: 2015-08-06</small>		<pre style="margin: 0;">\char_set_uccode:nn { '\a } { '\A } % Standard behaviour \char_set_uccode:nn { '\A } { '\A - 32 } \char_set_uccode:nn { 60 } { 50 }</pre>
		The setting applies within the current T _E X group.
<code>\char_value_uccode:n</code> *	<code>\char_value_uccode:n {<integer expression>}</code>	Expands to the current upper case code of the <code><character></code> with character code given by the <code><integer expression></code> .
<code>\char_show_value_uccode:n</code>	<code>\char_show_value_uccode:n {<integer expression>}</code>	Displays the current upper case code of the <code><character></code> with character code given by the <code><integer expression></code> on the terminal.
<code>\char_set_mathcode:nn</code>	<code>\char_set_mathcode:nn {<int expr1>} {<int expr2>}</code>	This function sets up the math code of <code><character></code> . The <code><character></code> is specified as an <code><integer expression></code> which will be used as the character code of the relevant character. The setting applies within the current T _E X group.
<small>Updated: 2015-08-06</small>		
<code>\char_value_mathcode:n</code> *	<code>\char_value_mathcode:n {<integer expression>}</code>	Expands to the current math code of the <code><character></code> with character code given by the <code><integer expression></code> .
<code>\char_show_value_mathcode:n</code>	<code>\char_show_value_mathcode:n {<integer expression>}</code>	Displays the current math code of the <code><character></code> with character code given by the <code><integer expression></code> on the terminal.
<code>\char_set_sfcode:nn</code>	<code>\char_set_sfcode:nn {<int expr1>} {<int expr2>}</code>	This function sets up the space factor for the <code><character></code> . The <code><character></code> is specified as an <code><integer expression></code> which will be used as the character code of the relevant character. The setting applies within the current T _E X group.
<small>Updated: 2015-08-06</small>		

`\char_value_sfcode:n` \star `\char_value_sfcode:n {⟨integer expression⟩}`
 Expands to the current space factor for the `⟨character⟩` with character code given by the `⟨integer expression⟩`.

`\char_show_value_sfcode:n` `\char_show_value_sfcode:n {⟨integer expression⟩}`
 Displays the current space factor for the `⟨character⟩` with character code given by the `⟨integer expression⟩` on the terminal.

`\l_char_active_seq` Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category `⟨active⟩` (catcode 13). Each entry in the sequence consists of a single escaped token, for example `\~`. Active tokens should be added to the sequence when they are defined for general document use.
New: 2012-01-23
 Updated: 2015-11-11

`\l_char_special_seq` Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories `⟨letter⟩` (catcode 11) or `⟨other⟩` (catcode 12). Each entry in the sequence consists of a single escaped token, for example `\` for the backslash or `{` for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.
New: 2012-01-23
 Updated: 2015-11-11

24.3 Generic tokens

`\c_group_begin_token`
`\c_group_end_token`
`\c_math_toggle_token`
`\c_alignment_token`
`\c_parameter_token`
`\c_math_superscript_token`
`\c_math_subscript_token`
`\c_space_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

TeXhackers note: The tokens `\c_group_begin_token`, `\c_group_end_token`, and `\c_space_token` are expl3 counterparts of L^AT_EX 2_ε's `\bgroup`, `\egroup`, and `\@sptoken`.

`\c_catcode_letter_token`
`\c_catcode_other_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

`\c_catcode_active_tl` A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.

24.4 Converting tokens

`\token_to_meaning:N` * `\token_to_meaning:N` $\langle token \rangle$

`\token_to_meaning:c` *

Inserts the current meaning of the $\langle token \rangle$ into the input stream as a series of characters of category code 12 (other). This is the primitive T_EX description of the $\langle token \rangle$, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as macros.

T_EXhackers note: This is the T_EX primitive `\meaning`. The $\langle token \rangle$ can thus be an explicit space token or an explicit begin-group or end-group character token (`{` or `}` when normal T_EX category codes apply) even though these are not valid N-type arguments.

`\token_to_str:N` * `\token_to_str:N` $\langle token \rangle$

`\token_to_str:c` *

Converts the given $\langle token \rangle$ into a series of characters with category code 12 (other). If the $\langle token \rangle$ is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the $\langle token \rangle$). This function requires only a single expansion.

T_EXhackers note: `\token_to_str:N` is the T_EX primitive `\string`. The $\langle token \rangle$ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal T_EX category codes apply) even though these are not valid N-type arguments.

`\token_to_catcode:N` * `\token_to_catcode:N` $\langle token \rangle$

New: 2023-10-15

Converts the given $\langle token \rangle$ into a number describing its category code. If $\langle token \rangle$ is a control sequence this expands to 16. This can't detect the categories 0 (escape character), 5 (end of line), 9 (ignored character), 14 (comment character), or 15 (invalid character). Control sequences or active characters let to a token of one of the detectable category codes will yield that category.

24.5 Token conditionals

`\token_if_group_begin_p:N` * `\token_if_group_begin_p:N` $\langle token \rangle$

`\token_if_group_begin:NTF` * `\token_if_group_begin:NTF` $\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a begin group token (`{` when normal T_EX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

`\token_if_group_end_p:N` * `\token_if_group_end_p:N` $\langle token \rangle$

`\token_if_group_end:NTF` * `\token_if_group_end:NTF` $\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an end group token (`}` when normal T_EX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

```
\token_if_math_toggle_p:N * \token_if_math_toggle_p:N <token>
\token_if_math_toggle:NTF * \token_if_math_toggle:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a math shift token ($\$$ when normal \TeX category codes are in force).

```
\token_if_alignment_p:N * \token_if_alignment_p:N <token>
\token_if_alignment:NTF * \token_if_alignment:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of an alignment token ($\&$ when normal \TeX category codes are in force).

```
\token_if_parameter_p:N * \token_if_parameter_p:N <token>
\token_if_parameter:NTF * \token_if_parameter:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a macro parameter token ($\#$ when normal \TeX category codes are in force).

```
\token_if_math_superscript_p:N * \token_if_math_superscript_p:N <token>
\token_if_math_superscript:NTF * \token_if_math_superscript:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a superscript token ($\^$ when normal \TeX category codes are in force).

```
\token_if_math_subscript_p:N * \token_if_math_subscript_p:N <token>
\token_if_math_subscript:NTF * \token_if_math_subscript:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a subscript token ($_$ when normal \TeX category codes are in force).

```
\token_if_space_p:N * \token_if_space_p:N <token>
\token_if_space:NTF * \token_if_space:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

```
\token_if_letter_p:N * \token_if_letter_p:N <token>
\token_if_letter:NTF * \token_if_letter:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a letter token.

```
\token_if_other_p:N * \token_if_other_p:N <token>
\token_if_other:NTF * \token_if_other:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of an “other” token.

```
\token_if_active_p:N * \token_if_active_p:N <token>
\token_if_active:NTF * \token_if_active:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of an active character.

```
\token_if_eq_catcode_p:NN * \token_if_eq_catcode_p:NN <token1> <token2>
\token_if_eq_catcode:NNTF * \token_if_eq_catcode:NNTF <token1> <token2> {\true code} {\false code}
```

Tests if the two $\langle tokens \rangle$ have the same category code.

```
\token_if_eq_charcode_p:NN * \token_if_eq_charcode_p:NN <token1> <token2>
\token_if_eq_charcode:NNTF * \token_if_eq_charcode:NNTF <token1> <token2> {\true code} {\false code}
```

Tests if the two *<tokens>* have the same character code.

```
\token_if_eq_meaning_p:NN * \token_if_eq_meaning_p:NN <token1> <token2>
\token_if_eq_meaning:NNTF * \token_if_eq_meaning:NNTF <token1> <token2> {\true code} {\false code}
```

Tests if the two *<tokens>* have the same meaning when expanded.

```
\token_if_macro_p:N * \token_if_macro_p:N <token>
\token_if_macro:NNTF * \token_if_macro:NNTF <token> {\true code} {\false code}
```

Updated: 2011-05-23 Tests if the *<token>* is a \TeX macro.

```
\token_if_cs_p:N * \token_if_cs_p:N <token>
\token_if_cs:NNTF * \token_if_cs:NNTF <token> {\true code} {\false code}
```

Tests if the *<token>* is a control sequence.

```
\token_if_expandable_p:N * \token_if_expandable_p:N <token>
\token_if_expandable:NNTF * \token_if_expandable:NNTF <token> {\true code} {\false code}
```

Tests if the *<token>* is expandable. This test returns *<false>* for an undefined token.

```
\token_if_long_macro_p:N * \token_if_long_macro_p:N <token>
\token_if_long_macro:NNTF * \token_if_long_macro:NNTF <token> {\true code} {\false code}
```

Updated: 2012-01-20 Tests if the *<token>* is a long macro.

```
\token_if_protected_macro_p:N * \token_if_protected_macro_p:N <token>
\token_if_protected_macro:NNTF * \token_if_protected_macro:NNTF <token> {\true code} {\false code}
```

Updated: 2012-01-20

Tests if the *<token>* is a protected macro: for a macro which is both protected and long this returns *false*.

```
\token_if_protected_long_macro_p:N * \token_if_protected_long_macro_p:N <token>
\token_if_protected_long_macro:NNTF * \token_if_protected_long_macro:NNTF <token> {\true code} {\false
code}}
```

Updated: 2012-01-20

Tests if the *<token>* is a protected long macro.

```
\token_if_chardef_p:N * \token_if_chardef_p:N <token>
\token_if_chardef:NNTF * \token_if_chardef:NNTF <token> {\true code} {\false code}
```

Updated: 2012-01-20 Tests if the *<token>* is defined to be a chardef.

\TeX hackers note: Booleans, boxes and small integer constants are implemented as *\chardefs*.

```
\token_if_mathchardef_p:N * \token_if_mathchardef_p:N <token>
\token_if_mathchardef:NTF * \token_if_mathchardef:NTF <token> {(true code)} {(false code)}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a mathchardef.

```
\token_if_font_selection_p:N * \token_if_font_selection_p:N <token>
\token_if_font_selection:NTF * \token_if_font_selection:NTF <token> {(true code)} {(false code)}
```

New: 2020-10-27

Tests if the $\langle token \rangle$ is defined to be a font selection command.

```
\token_if_dim_register_p:N * \token_if_dim_register_p:N <token>
\token_if_dim_register:NTF * \token_if_dim_register:NTF <token> {(true code)} {(false code)}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a dimension register.

```
\token_if_int_register_p:N * \token_if_int_register_p:N <token>
\token_if_int_register:NTF * \token_if_int_register:NTF <token> {(true code)} {(false code)}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a integer register.

TeXhackers note: Constant integers may be implemented as integer registers, $\backslash chardefs$, or $\backslash mathchardefs$ depending on their value.

```
\token_if_muskip_register_p:N * \token_if_muskip_register_p:N <token>
\token_if_muskip_register:NTF * \token_if_muskip_register:NTF <token> {(true code)} {(false code)}
```

New: 2012-02-15

Tests if the $\langle token \rangle$ is defined to be a muskip register.

```
\token_if_skip_register_p:N * \token_if_skip_register_p:N <token>
\token_if_skip_register:NTF * \token_if_skip_register:NTF <token> {(true code)} {(false code)}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a skip register.

```
\token_if_toks_register_p:N * \token_if_toks_register_p:N <token>
\token_if_toks_register:NTF * \token_if_toks_register:NTF <token> {(true code)} {(false code)}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a toks register (not used by L^AT_EX3).

```
\token_if_primitive_p:N * \token_if_primitive_p:N <token>
\token_if_primitive:NTF * \token_if_primitive:NTF <token> {(true code)} {(false code)}
```

Updated: 2020-09-11

Tests if the $\langle token \rangle$ is an engine primitive. In Lua_TE_X this includes primitive-like commands defined using `token.set_lua`.

<code>\token_case_catcode:Nn</code>	*	<code>\token_case_meaning:NnTF</code>	<code><test token></code>
<code>\token_case_catcode:NnTF</code>	*	{	
<code>\token_case_charcode:Nn</code>	*	<code><token case₁></code>	{ <code><code case₁></code> }
<code>\token_case_charcode:NnTF</code>	*	<code><token case₂></code>	{ <code><code case₂></code> }
<code>\token_case_meaning:Nn</code>	*	...	
<code>\token_case_meaning:NnTF</code>	*	<code><token case_n></code>	{ <code><code case_n></code> }

		}	
		{ <code><>true code></code> }	
		{ <code><>false code></code> }	

New: 2020-12-03

This function compares the `<test token>` in turn with each of the `<token cases>`. If the two are equal (as described for `\token_if_eq_catcode:NNTF`, `\token_if_eq_charcode:NNTF` and `\token_if_eq_meaning:NNTF`, respectively) then the associated `<code>` is left in the input stream and other cases are discarded. If any of the cases are matched, the `<>true code>` is also inserted into the input stream (after the code for the appropriate case), while if none match then the `<>false code>` is inserted. The functions `\token_case_catcode:Nn`, `\token_case_charcode:Nn`, and `\token_case_meaning:Nn`, which do nothing if there is no match, are also available.

24.6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. Peeking ahead does *not* skip spaces: rather, `\peek_remove_spaces:n` should be used. In addition, using `\peek_analysis_map_inline:n`, one can map through the following tokens in the input stream and repeatedly perform some tests.

`\peek_after:Nw` `\peek_after:Nw` `<function>` `<token>`

Locally sets the test variable `\l_peek_token` equal to `<token>` (as an implicit token, *not* as a token list), and then expands the `<function>`. The `<token>` remains in the input stream as the next item after the `<function>`. The `<token>` here may be `␣`, `{` or `}` (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

`\peek_gafter:Nw` `\peek_gafter:Nw` `<function>` `<token>`

Globally sets the test variable `\g_peek_token` equal to `<token>` (as an implicit token, *not* as a token list), and then expands the `<function>`. The `<token>` remains in the input stream as the next item after the `<function>`. The `<token>` here may be `␣`, `{` or `}` (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

`\l_peek_token` Token set by `\peek_after:Nw` and available for testing as described above.

`\g_peek_token` Token set by `\peek_gafter:Nw` and available for testing as described above.

<u>\peek_catcode:NTF</u>	\peek_catcode:NTF $\langle test\ token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
Updated: 2012-12-20	Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test\ token \rangle$ (as defined by the test <code>\token_if_eq_catcode:NNTF</code>). Spaces are respected by the test and the $\langle token \rangle$ is left in the input stream after the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ (as appropriate to the result of the test).
<u>\peek_catcode_remove:NTF</u>	\peek_catcode_remove:NTF $\langle test\ token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
Updated: 2012-12-20	Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test\ token \rangle$ (as defined by the test <code>\token_if_eq_catcode:NNTF</code>). Spaces are respected by the test and the $\langle token \rangle$ is removed from the input stream if the test is true. The function then places either the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ in the input stream (as appropriate to the result of the test).
<u>\peek_charcode:NTF</u>	\peek_charcode:NTF $\langle test\ token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
Updated: 2012-12-20	Tests if the next $\langle token \rangle$ in the input stream has the same character code as the $\langle test\ token \rangle$ (as defined by the test <code>\token_if_eq_charcode:NNTF</code>). Spaces are respected by the test and the $\langle token \rangle$ is left in the input stream after the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ (as appropriate to the result of the test).
<u>\peek_charcode_remove:NTF</u>	\peek_charcode_remove:NTF $\langle test\ token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
Updated: 2012-12-20	Tests if the next $\langle token \rangle$ in the input stream has the same character code as the $\langle test\ token \rangle$ (as defined by the test <code>\token_if_eq_charcode:NNTF</code>). Spaces are respected by the test and the $\langle token \rangle$ is removed from the input stream if the test is true. The function then places either the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ in the input stream (as appropriate to the result of the test).
<u>\peek_meaning:NTF</u>	\peek_meaning:NTF $\langle test\ token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
Updated: 2011-07-02	Tests if the next $\langle token \rangle$ in the input stream has the same meaning as the $\langle test\ token \rangle$ (as defined by the test <code>\token_if_eq_meaning:NNTF</code>). Spaces are respected by the test and the $\langle token \rangle$ is left in the input stream after the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ (as appropriate to the result of the test).
<u>\peek_meaning_remove:NTF</u>	\peek_meaning_remove:NTF $\langle test\ token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
Updated: 2011-07-02	Tests if the next $\langle token \rangle$ in the input stream has the same meaning as the $\langle test\ token \rangle$ (as defined by the test <code>\token_if_eq_meaning:NNTF</code>). Spaces are respected by the test and the $\langle token \rangle$ is removed from the input stream if the test is true. The function then places either the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ in the input stream (as appropriate to the result of the test).
<u>\peek_remove_spaces:n</u>	\peek_remove_spaces:n $\{\langle code \rangle\}$
New: 2018-10-01	Peeks ahead and detect if the following token is a space (category code 10 and character code 32). If so, removes the token and checks the next token. Once a non-space token is found, the $\langle code \rangle$ will be inserted into the input stream. Typically this will contain a peek operation, but this is not required.

`\peek_remove_filler:n` `\peek_remove_filler:n {<code>}`

New: 2022-01-10

Peeks ahead and detect if the following token is a space (category code 10) or has meaning equal to `\scan_stop:`. If so, removes the token and checks the next token. If neither of these cases apply, expands the next token using f-type expansion, then checks the resulting leading token in the same way. If after expansion the next token is neither of the two test cases, the `<code>` will be inserted into the input stream. Typically this will contain a `peek` operation, but this is not required.

TeXhackers note: This is essentially a macro-based implementation of how TeX handles the search for a left brace after for example `\everypar`, except that any non-expandable token cleanly ends the `<filler>` (i.e. it does not lead to a TeX error).

In contrast to TeX's filler removal, a construct `\exp_not:N \foo` will be treated in the same way as `\foo`.

`\peek_N_type:TF` `\peek_N_type:TF {<true code>} {<false code>}`

Updated: 2012-12-20

Tests if the next `<token>` in the input stream can be safely grabbed as an N-type argument. The test is `<false>` if the next `<token>` is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in L^AT_EX3) and `<true>` in all other cases. Note that a `<true>` result ensures that the next `<token>` is a valid N-type argument. However, if the next `<token>` is for instance `\c_space_token`, the test takes the `<false>` branch, even though the next `<token>` is in fact a valid N-type argument. The `<token>` is left in the input stream after the `<true code>` or `<false code>` (as appropriate to the result of the test).

`\peek_analysis_map_inline:n` `\peek_analysis_map_inline:n` {*inline function*}

New: 2020-12-03

Updated: 2024-02-07

Repeatedly removes one *token* from the input stream and applies the *inline function* to it, until `\peek_analysis_map_break:` is called. The *inline function* receives three arguments for each *token* in the input stream:

- *tokens*, which both *o*-expand and *e/x*-expand to the *token*. The detailed form of *tokens* may change in later releases.
- *char code*, a decimal representation of the character code of the *token*, -1 if it is a control sequence.
- *catcode*, a capital hexadecimal digit which denotes the category code of the *token* (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing "*catcode*".

These arguments are the same as for `\tl_analysis_map_inline:nn` defined in `l3tl-analysis`. The *char code* and *catcode* do not take the meaning of a control sequence or active character into account: for instance, upon encountering the token `\c_group_begin_token` in the input stream, `\peek_analysis_map_inline:n` calls the *inline function* with #1 being `\exp_not:n { \c_group_begin_token }` (with the current implementation), #2 being -1 , and #3 being 0, as for any other control sequence. In contrast, upon encountering an explicit begin-group token `{`, the *inline function* is called with arguments `\exp_after:wN { \if_false: } \fi:`, 123 and 1.

The mapping is done at the current group level, *i.e.* any local assignments made by the *inline function* remain in effect after the loop. Within the code, `\l_peek_token` is set equal (as a token, not a token list) to the token under consideration.

Peek functions cannot be used within this mapping function (nor other mapping functions) since the input stream contains trailing material necessary for the functioning of the loop.

T_EXhackers note: In case the input stream has not yet been tokenized (converted from characters to tokens), characters are tokenized one by one as needed by `\peek_analysis_map_inline:n` using the current category code regime.

`\peek_analysis_map_break:` `\peek_analysis_map_inline:n`
`\peek_analysis_map_break:n` { ... `\peek_analysis_map_break:n` {*code*} }

New: 2020-12-03

Stops the `\peek_analysis_map_inline:n` loop from seeking more tokens, and inserts *code* in the input stream (empty for `\peek_analysis_map_break:`).

`\peek_regex:nTF` `\peek_regex:nTF {<regex>} {<true code>} {<false code>}`

`\peek_regex:NTF`

New: 2020-12-03

Tests if the *<tokens>* that follow in the input stream match the *<regular expression>*. Any *<tokens>* that have been read are left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test). See `l3regex` for documentation of the syntax of regular expressions. The *<regular expression>* is implicitly anchored at the start, so for instance `\peek_regex:nTF { a }` is essentially equivalent to `\peek_charcode:NTF a`.

TeXhackers note: Implicit character tokens are correctly considered by `\peek_regex:nTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

The `\peek_regex:nTF` function only inspects as few tokens as necessary to determine whether the regular expression matches. For instance `\peek_regex:nTF { abc | [a-z] } { } { }` `abc` will only inspect the first token `a` even though the first branch `abc` of the alternative is preferred in functions such as `\peek_regex_remove_once:nTF`. This may have an effect on tokenization if the input stream has not yet been tokenized and category codes are changed.

`\peek_regex_remove_once:nTF` `\peek_regex_remove_once:nTF {<regex>} {<true code>} {<false code>}`

`\peek_regex_remove_once:NTF`

New: 2020-12-03

Tests if the *<tokens>* that follow in the input stream match the *<regex>*. If the test is true, the *<tokens>* are removed from the input stream and the *<true code>* is inserted, while if the test is false, the *<false code>* is inserted followed by the *<tokens>* that were originally in the input stream. See `l3regex` for documentation of the syntax of regular expressions. The *<regular expression>* is implicitly anchored at the start, so for instance `\peek_regex_remove_once:nTF { a }` is essentially equivalent to `\peek_charcode_remove:NTF a`.

TeXhackers note: Implicit character tokens are correctly considered by `\peek_regex_remove_once:nTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

```

\peek_regex_replace_once:nn  \peek_regex_replace_once:nnTF {<regex>} {<replacement>} {<true code>}
\peek_regex_replace_once:nnTF {<false code>}
\peek_regex_replace_once:Nn
\peek_regex_replace_once:NnTF

```

New: 2020-12-03

If the $\langle tokens \rangle$ that follow in the input stream match the $\langle regex \rangle$, replaces them according to the $\langle replacement \rangle$ as for `\regex_replace_once:nnN`, and leaves the result in the input stream, after the $\langle true code \rangle$. Otherwise, leaves $\langle false code \rangle$ followed by the $\langle tokens \rangle$ that were originally in the input stream, with no modifications. See `l3regex` for documentation of the syntax of regular expressions and of the $\langle replacement \rangle$: for instance `\0` in the $\langle replacement \rangle$ is replaced by the tokens that were matched in the input stream. The $\langle regular expression \rangle$ is implicitly anchored at the start. In contrast to `\regex_replace_once:nnN`, no error arises if the $\langle replacement \rangle$ leads to an unbalanced token list: the tokens are inserted into the input stream without issue.

TeXhackers note: Implicit character tokens are correctly considered by `\peek_regex_replace_once:nnTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

24.7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on TeX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.
- An active character token, characterized by its character code (between 0 and 1114111 for LuaTeX and XeTeX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A “frozen” `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand <token>` (when the $\langle token \rangle$ is expandable) results in an internal token, displayed (temporarily) as `\notexpanded: <token>`, whose shape coincides with the $\langle token \rangle$ and whose meaning differs from `\relax`.

- An `\outer endtemplate`: can be encountered when peeking ahead at the next token; this expands to another internal token, `end of alignment template`.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.
- In LuaTeX, there is also the strange case of “bytes” `^^1100xy` where x, y are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from `"110000 = 1114112` to `"1100ff = 1114367`. These are used to output individual bytes to files, rather than UTF-8. For the purposes of token comparisons they behave like non-expandable primitive control sequences (*not characters*) whose `\meaning` is `the_character_` followed by the given byte. If this byte is in the range `80–ff` this gives an “invalid utf-8 sequence” error: applying `\token_to_str:N` or `\token_to_meaning:N` to these tokens is unsafe. Unfortunately, they don’t seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TeX primitive `\meaning`, together with their L^AT_EX3 names and most common example:

- 1 begin-group character (`group_begin`, often `{`),
- 2 end-group character (`group_end`, often `}`),
- 3 math shift character (`math_toggle`, often `$`),
- 4 alignment tab character (`alignment`, often `&`),
- 6 macro parameter character (`parameter`, often `#`),
- 7 superscript character (`math_superscript`, often `^`),
- 8 subscript character (`math_subscript`, often `_`),
- 10 blank space (`space`, often character code 32),
- 11 the letter (`letter`, such as `A`),
- 12 the character (`other`, such as `0`).

Category code 13 (*active*) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (`escape`), 5 (`end_line`), 9 (`ignore`), 14 (`comment`), and 15 (`invalid`).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in L^AT_EX3 for most functions and some variables (`tl`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in L^AT_EX3 for some functions,
- a register such as `\count123`, used in L^AT_EX3 for the implementation of some variables (`int`, `dim`, ...),

- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros can be `\protected` or not, `\long` or not (the opposite of what L^AT_EX3 calls `nopar`), and `\outer` or not (unused in L^AT_EX3). Their `\meaning` takes the form

`⟨prefix⟩ macro:⟨argument⟩->⟨replacement⟩`

where `⟨prefix⟩` is among `\protected\long\outer`, `⟨argument⟩` describes parameters that the macro expects, such as `#1#2#3`, and `⟨replacement⟩` describes how the parameters are manipulated, such as `\int_eval:n{#2+#1*#3}`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then T_EX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature `:N`.

When a macro takes a delimited argument T_EX scans ahead until finding the delimiter (outside any pairs of begin-group/end-group explicit characters), and the resulting list of tokens (with outer braces removed) becomes the argument. Note that explicit space characters at the start of the argument are *not* ignored in this case (and they prevent brace-stripping).

Chapter 25

The l3prop module

Property lists

expl3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a $\langle key \rangle$ (string) and an associated $\langle value \rangle$ (token list). The $\langle key \rangle$ and $\langle value \rangle$ may both be given as any balanced text, and the $\langle key \rangle$ is processed using `\tl_to_str:n`, meaning that category codes are ignored. Entries can be manipulated individually, as well as collectively by applying a function to every key–value pair within the list.

Each entry in a property list must have a unique $\langle key \rangle$: if an entry is added to a property list which already contains the $\langle key \rangle$ then the new entry overwrites the existing one. The $\langle keys \rangle$ are compared on a string basis, using the same method as `\str_if_eq:nnTF`.

Property lists are intended for storing key-based information for use within code. They can be converted from and to key–value lists, which are a form of *input* parsed by the `l3keys` module. If a key–value list contains a $\langle key \rangle$ multiple times, only the last $\langle value \rangle$ associated to it will be kept in the conversion to a property list.

Internally, property lists can use two distinct implementations with different data storage, which are decided when declaring the property list variable using `\prop_new:N` (“flat” storage) or `\prop_new_linked:N` (“linked” storage). After a property list is declared with `\prop_new:N` or `\prop_new_linked:N`, the type of internal data storage can be changed by `\prop_make_flat:N` or `\prop_make_linked:N`, but only at the outermost group level. All other `l3prop` functions transparently manipulate either storage method and convert as needed.

- The (default) “flat” storage method is suited for a relatively small number of entries, or when the property list is likely to be manipulated (copied, mapped) as a whole rather than entry-wise. It is significantly faster for `\prop_set_eq:NN`, and only slightly faster for `\prop_clear:N`, `\prop_concat:NNN`, and mapping functions `\prop_map_...`
- The “linked” storage method is meant for property lists with a large numbers of entries. It takes up more of TeX’s memory during a run, but is significantly faster (for long lists) when accessing or modifying individual entries using functions such as `\prop_if_in:Nn`, `\prop_item:Nn`, `\prop_put:Nnn`, `\prop_get:NnN`, `\prop_pop:NnN`, `\prop_remove:Nn`, as it takes a constant time for these operations (rather

than the number of items for a “flat” property list). A technical drawback is that memory is permanently used⁷ by `<keys>` stored in a “linked” property list, even after they are removed and the property list is deleted.

25.1 Creating and initialising property lists

`\prop_new:N` `\prop_new:N <property list>`
`\prop_new:c`

Creates a new “flat” `<property list>` or raises an error if the name is already taken. The declaration is global. The `<property list>` initially contains no entries. See also `\prop_new_linked:N`.

`\prop_new_linked:N` `\prop_new_linked:N <property list>`
`\prop_new_linked:c`
New: 2024-02-12

Creates a new “linked” `<property list>` or raises an error if the name is already taken. The declaration is global. The `<property list>` initially contains no entries. The internal data storage differs from that produced by `\prop_new:N` and it is optimized for property lists with a large number of entries.

`\prop_clear:N` `\prop_clear:N <property list>`
`\prop_clear:c`
`\prop_gclear:N` `\prop_gclear:N <property list>`.
`\prop_gclear:c`

`\prop_clear_new:N` `\prop_clear_new:N <property list>`
`\prop_clear_new:c`
`\prop_gclear_new:N`
`\prop_gclear_new:c`

Ensures that the `<property list>` exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

TeXhackers note: If the property list exists and is of “linked” type, it is cleared but not made into a flat property list.

`\prop_clear_new_linked:N` `\prop_clear_new_linked:N <property list>`
`\prop_clear_new_linked:c`
`\prop_gclear_new_linked:N`
`\prop_gclear_new_linked:c`

Ensures that the `<property list>` exists globally by applying `\prop_new_linked:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

New: 2024-02-12

TeXhackers note: If the property list exists and is of “flat” type, it is cleared but not made into a linked property list.

`\prop_set_eq:NN` `\prop_set_eq:NN <property list12
\prop_set_eq:(cN|Nc|cc)
\prop_gset_eq:NN
\prop_gset_eq:(cN|Nc|cc)

Sets the content of <property list1 equal to that of <property list2. This converts as needed between the two storage types.`

⁷Until the end of the run, that is.

```

\prop_set_from_keyval:Nn \prop_set_from_keyval:Nn <property list>
\prop_set_from_keyval:cn {
\prop_gset_from_keyval:Nn <key1> = <value1> ,
\prop_gset_from_keyval:cn <key2> = <value2> , ...
}

```

New: 2017-11-28
Updated: 2021-11-07

Sets *<property list>* to contain key–value pairs given in the second argument. If duplicate keys appear only the last of the values is kept. In contrast to most keyval lists (e.g. those in `l3keys`), each key here *must* be followed with an = sign even to specify an empty *<value>*.

Spaces are trimmed around every *<key>* and every *<value>*, and if the result of trimming spaces consists of a single brace group then a set of outer braces is removed. This enables both the *<key>* and the *<value>* to contain spaces, commas or equal signs. The *<key>* is then processed by `\tl_to_str:n`. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

```

\prop_const_from_keyval:Nn \prop_const_from_keyval:Nn <property list>
\prop_const_from_keyval:cn {
\prop_gconst_from_keyval:Nn <key1> = <value1> ,
\prop_gconst_from_keyval:cn <key2> = <value2> , ...
}

```

New: 2017-11-28
Updated: 2021-11-07

Creates a new constant “flat” *<property list>* or raises an error if the name is already taken. The *<property list>* is set globally to contain key–value pairs given in the second argument, processed in the way described for `\prop_set_from_keyval:Nn`. If duplicate keys appear only the last of the values is kept. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

```

\prop_const_linked_from_keyval:Nn \prop_const_linked_from_keyval:Nn <prop var>
\prop_const_linked_from_keyval:cn {
\prop_gconst_linked_from_keyval:Nn <key1> = <value1> ,
\prop_gconst_linked_from_keyval:cn <key2> = <value2> , ...
}

```

New: 2024-02-12

Creates a new constant “linked” *<prop var>* or raises an error if the name is already taken. The *<prop var>* is set globally to contain key–value pairs given in the second argument, processed in the way described for `\prop_set_from_keyval:Nn`. If duplicate keys appear only the last of the values is kept. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

```

\prop_make_flat:N \prop_make_flat:N <property list>
\prop_make_flat:c

```

New: 2024-02-12

Changes the internal storage type of the *<property list>* to be the same “flat” storage as `\prop_new:N`. The key–value pairs of the *<property list>* are preserved by the change. If the property list was already flat then nothing is done. This function can only be used at the outermost group level.

```

\prop_make_linked:N \prop_make_linked:N <property list>
\prop_make_linked:c

```

New: 2024-02-12

Changes the internal storage type of the *<property list>* to be the same “linked” storage as `\prop_new_linked:N`. The key–value pairs of the *<property list>* are preserved by the change. If the property list was already linked then nothing is done. This function can only be used at the outermost group level.

25.2 Adding and updating property list entries

<code>\prop_put:Nnn</code>	<code>\prop_put:Nnn <property list> {<key>} {<value>}</code>
<code>\prop_put:(NnV Nnv Nne NVn NVV NVv Nve Nvn NvV Nvv Nve Nen NeV Nev Nee Nno Non Noo cnn cnV cnv cne cVn cVV cVv cVe cvn cvV cvv cve cen ceV cev cee cno con coo)</code>	
<code>\prop_gput:Nnn</code>	
<code>\prop_gput:(NnV Nnv Nne NVn NVV NVv Nve Nvn NvV Nvv Nve Nen NeV Nev Nee Nno Non Noo cnn cnV cnv cne cVn cVV cVv cVe cvn cvV cvv cve cen ceV cev cee cno con coo)</code>	

Updated: 2012-07-09

Adds an entry to the `<property list>` which may be accessed using the `<key>` and which has `<value>`. If the `<key>` is already present in the `<property list>`, the existing entry is overwritten by the new `<value>`. Both the `<key>` and `<value>` may contain any `<balanced text>`. The `<key>` is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored.

<code>\prop_put_if_not_in:Nnn</code>	<code>\prop_put_if_not_in:Nnn <property list> {<key>} {<value>}</code>
<code>\prop_put_if_not_in:(NnV Nnv Nne NVn NVV NVv Nve Nvn NvV Nvv Nve Nen NeV Nev Nee cnn cnV cnv cne cVn cVV cVv cVe cvn cvV cvv cve cen ceV cev cee)</code>	
<code>\prop_gput_if_not_in:Nnn</code>	
<code>\prop_gput_if_not_in:(NnV Nnv Nne NVn NVV NVv Nve Nvn NvV Nvv Nve Nen NeV Nev Nee cnn cnV cnv cne cVn cVV cVv cVe cvn cvV cvv cve cen ceV cev cee)</code>	

New: 2024-03-30

Updated: 2024-05-07

If the `<key>` is present in the `<property list>` then no action is taken. Otherwise, a new entry is added as described for `\prop_put:Nnn`.

<code>\prop_concat:NNN</code>	<code>\prop_concat:NNN <property list₁> <property list₂> <property list₃></code>
<code>\prop_concat:ccc</code>	
<code>\prop_gconcat:NNN</code>	Combines the key–value pairs of <code><property list₂></code> and <code><property list₃></code> , and saves the result in <code><property list₁></code> . If a key appears in both <code><property list₂></code> and <code><property list₃></code> then the last value, namely the value in <code><property list₃></code> is kept. This converts
<code>\prop_gconcat:ccc</code>	as needed between the two storage types.

New: 2021-05-16

```

\prop_put_from_keyval:Nn \prop_put_from_keyval:Nn <property list>
\prop_put_from_keyval:cn {
\prop_gput_from_keyval:Nn <key1> = <value1> ,
\prop_gput_from_keyval:cn <key2> = <value2> , ...
}

```

New: 2021-05-16
Updated: 2021-11-07

Updates the *<property list>* by adding entries for each key–value pair given in the second argument. The addition is done through `\prop_put:Nnn`, hence if the *<property list>* already contains some of the keys, the corresponding values are discarded and replaced by those given in the key–value list. If duplicate keys appear in the key–value list then only the last of the values is kept.

The function is equivalent to storing the key–value pairs in a temporary property list using `\prop_set_from_keyval:Nn`, then combining *<property list>* with the temporary variable using `\prop_concat:MNn`. In particular, the *<keys>* and *<values>* are space-trimmed and unbraced as described in `\prop_set_from_keyval:Nn`. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

25.3 Recovering values from property lists

```

\prop_get:NnN \prop_get:NnN <property list> {<key>} <tl var>
\prop_get:(NVN|NvN|NeN|NoN|cnN|cVN|cvN|ceN|coN|cnc)

```

Updated: 2011-08-28

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<token list variable>* is set within the current T_EX group. See also `\prop_get:NnNTF`.

```

\prop_pop:NnN \prop_pop:NnN <property list> {<key>} <tl var>
\prop_pop:(NVN|NoN|cnN|cVN|coN)

```

Updated: 2011-08-18

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. Both assignments are local. See also `\prop_pop:NnNTF`.

```

\prop_gpop:NnN \prop_gpop:NnN <property list> {<key>} <tl var>
\prop_gpop:(NVN|NoN|cnN|cVN|coN)

```

Updated: 2011-08-18

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. The *<property list>* is modified globally, while the assignment of the *<token list variable>* is local. See also `\prop_gpop:NnNTF`.

```

\prop_item:Nn * \prop_item:Nn <property list> {<key>}
\prop_item:(NV|Ne|No|cn|cV|ce|co) *

```

New: 2014-07-17

Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.

TeXhackers note: For “flat” property lists, this expandable function iterates through every key–value pair and is therefore slower than a non-expandable approach based on `\prop_get:NnN`. (For “linked” property lists both functions are fast.)

The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle value \rangle$ does not expand further when appearing in an e-type or x-type argument expansion.

```

\prop_count:N * \prop_count:N <property list>
\prop_count:c *

```

Leaves the number of key–value pairs in the $\langle property list \rangle$ in the input stream as an $\langle integer denotation \rangle$.

```

\prop_to_keyval:N * \prop_to_keyval:N <property list>

```

Expands to the $\langle property list \rangle$ in a key–value notation. Keep in mind that a $\langle property list \rangle$ is *unordered*, while key–value interfaces are not necessarily, so this cannot be used for arbitrary interfaces.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the key–value list does not expand further when appearing in an e-type or x-type argument expansion. It also needs exactly two steps of expansion.

25.4 Modifying property lists

```

\prop_remove:Nn * \prop_remove:Nn <property list> {<key>}
\prop_remove:(NV|Ne|cn|cV|ce)
\prop_gremove:Nn
\prop_gremove:(NV|Ne|cn|cV|ce)

```

New: 2012-05-12

Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e.* there is no need to test for the existence of a key before deleting it.

25.5 Property list conditionals

```

\prop_if_exist_p:N * \prop_if_exist_p:N <property list>
\prop_if_exist_p:c * \prop_if_exist:NTF <property list> {<true code>} {<false code>}
\prop_if_exist:NTF * Tests whether the <property list> is currently defined. This does not check that the
\prop_if_exist:cTF * <property list> really is a property list variable.

```

New: 2012-03-03

```

\prop_if_empty_p:N * \prop_if_empty_p:N <property list>
\prop_if_empty_p:c * \prop_if_empty:NTF <property list> {<true code>} {<>false code>}
\prop_if_empty:NTF * Tests if the <property list> is empty (containing no entries).
\prop_if_empty:cTF *

```

```

\prop_if_in_p:Nn * \prop_if_in_p:Nn <property list> {<key>}
\prop_if_in_p:(NV|Ne|No|cn|cV|ce|co) * \prop_if_in:NnTF <property list> {<key>} {<true code>} {<>false
\prop_if_in:NnTF * code>}
\prop_if_in:(NV|Ne|No|cn|cV|ce|co)TF *

```

Updated: 2011-09-15

Tests if the $\langle key \rangle$ is present in the $\langle property list \rangle$, making the comparison using the method described by `\str_if_eq:nnTF`.

T_EXhackers note: For “flat” property lists, this expandable function iterates through every key–value pair and is therefore slower than a non-expandable approach based on `\prop_get:NnNTF`. (For “linked” property lists both functions are fast.)

25.6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated value. This makes them useful for cases where different code follows depending on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

```

\prop_get:NnNTF * \prop_get:NnNTF <property list> {<key>} <token list
\prop_get:(NVN|NvN|NeN|NoN|cnN|cVN|cvN|ceN|coN| variable>
cnc)TF * {<true code>} {<>false code>}

```

Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, stores the corresponding $\langle value \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle property list \rangle$, then leaves the $\langle true code \rangle$ in the input stream. The $\langle token list variable \rangle$ is assigned locally.

```

\prop_pop:NnNTF * \prop_pop:NnNTF <property list> {<key>} <token list variable>
\prop_pop:(NVN|NoN|cnN|cVN|coN)TF * {<true code>} {<>false code>}

```

New: 2011-08-18

Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, pops the corresponding $\langle value \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from the $\langle property list \rangle$. Both the $\langle property list \rangle$ and the $\langle token list variable \rangle$ are assigned locally.

<code>\prop_gpop:NnNTF</code> <code>\prop_gpop:(NVN NoN cnN cVN coN)TF</code>	<code>\prop_gpop:NnNTF <property list> {<key>} <token list variable></code> <code>{<true code>} {<false code>}</code>
--	--

New: 2011-08-18
Updated: 2012-05-19

If the `<key>` is not present in the `<property list>`, leaves the `<false code>` in the input stream. The value of the `<token list variable>` is not defined in this case and should not be relied upon. If the `<key>` is present in the `<property list>`, pops the corresponding `<value>` in the `<token list variable>`, *i.e.* removes the item from the `<property list>`. The `<property list>` is modified globally, while the `<token list variable>` is assigned locally.

25.7 Mapping over property lists

All mappings are done at the current group level, *i.e.* any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

<code>\prop_map_function:Nn ☆</code> <code>\prop_map_function:cN ☆</code> <small>Updated: 2013-01-08</small>	<code>\prop_map_function:Nn <property list> <function></code> Applies <code><function></code> to every <code><entry></code> stored in the <code><property list></code> . The <code><function></code> receives two arguments for each iteration: the <code><key></code> and associated <code><value></code> . The order in which <code><entries></code> are returned is not defined and should not be relied upon. To pass further arguments to the <code><function></code> , see <code>\prop_map_inline:Nn</code> (non-expandable) or <code>\prop_map_tokens:Nn</code> .
--	---

<code>\prop_map_inline:Nn</code> <code>\prop_map_inline:cn</code> <small>Updated: 2013-01-08</small>	<code>\prop_map_inline:Nn <property list> {<inline function>}</code> Applies <code><inline function></code> to every <code><entry></code> stored within the <code><property list></code> . The <code><inline function></code> should consist of code which receives the <code><key></code> as #1 and the <code><value></code> as #2. The order in which <code><entries></code> are returned is not defined and should not be relied upon.
--	--

<code>\prop_map_tokens:Nn ☆</code> <code>\prop_map_tokens:cn ☆</code>	<code>\prop_map_tokens:Nn <property list> {<code>}</code> Analogue of <code>\prop_map_function:Nn</code> which maps several tokens instead of a single function. The <code><code></code> receives each key-value pair in the <code><property list></code> as two trailing brace groups. For instance,
--	--

`\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }`

expands to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the `<key>` and the `<value>` as its three arguments. For that specific task, `\prop_item:Nn` is faster.

`\prop_map_break:` ☆ `\prop_map_break:`

Updated: 2012-06-29

Used to terminate a `\prop_map...` function before all entries in the *⟨property list⟩* have been processed. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\prop_map_break:n` ☆ `\prop_map_break:n {⟨code⟩}`

Updated: 2012-06-29

Used to terminate a `\prop_map...` function before all entries in the *⟨property list⟩* have been processed, inserting the *⟨code⟩* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *⟨code⟩* is inserted into the input stream. This depends on the design of the mapping function.

25.8 Viewing property lists

`\prop_show:N` `\prop_show:N ⟨property list⟩`

`\prop_show:c`

Updated: 2021-04-29

Displays the entries in the *⟨property list⟩* in the terminal, and specifies its storage type.

`\prop_log:N` `\prop_log:N` *<property list>*
`\prop_log:c` Writes the entries in the *<property list>* in the log file, and specifies its storage type.
New: 2014-08-12
Updated: 2021-04-29

25.9 Scratch property lists

There is no need to include both flat and linked property lists as scratch variables. We arbitrarily pick the older implementation.

`\l_tmpa_prop` Scratch “flat” property lists for local assignment. These are never used by the kernel
`\l_tmpb_prop` code, and so are safe for use with any L^AT_EX3-defined function. However, they may be
New: 2012-06-23 overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_prop` Scratch “flat” property lists for global assignment. These are never used by the kernel
`\g_tmpb_prop` code, and so are safe for use with any L^AT_EX3-defined function. However, they may be
New: 2012-06-23 overwritten by other non-kernel code and so should only be used for short-term storage.

25.10 Constants

`\c_empty_prop` A permanently-empty property list used for internal comparisons.

Chapter 26

The `\l3skip` module

Dimensions and skips

L^AT_EX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in μ). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

Many functions take *dimension expressions* (“`\langle dim expr \rangle`”) or *skip expressions* (“`\langle skip expr \rangle`”) as arguments.

26.1 Creating and initialising `dim` variables

<hr/> <code>\dim_new:N</code>	<code>\dim_new:N \langle dimension \rangle</code>	
<code>\dim_new:c</code>		Creates a new <code>\langle dimension \rangle</code> or raises an error if the name is already taken. The declaration is global. The <code>\langle dimension \rangle</code> is initially equal to 0pt.
<hr/> <code>\dim_const:Nn</code>	<code>\dim_const:Nn \langle dimension \rangle { \langle dim expr \rangle }</code>	
<code>\dim_const:cn</code>		Creates a new constant <code>\langle dimension \rangle</code> or raises an error if the name is already taken. The value of the <code>\langle dimension \rangle</code> is set globally to the <code>\langle dim expr \rangle</code> .
<hr/> <code>\dim_zero:N</code>	<code>\dim_zero:N \langle dimension \rangle</code>	
<code>\dim_zero:c</code>		Sets <code>\langle dimension \rangle</code> to 0pt.
<code>\dim_gzero:N</code>		
<code>\dim_gzero:c</code>		
<hr/> <code>\dim_zero_new:N</code>	<code>\dim_zero_new:N \langle dimension \rangle</code>	
<code>\dim_zero_new:c</code>		Ensures that the <code>\langle dimension \rangle</code> exists globally by applying <code>\dim_new:N</code> if necessary, then applies <code>\dim_(g)zero:N</code> to leave the <code>\langle dimension \rangle</code> set to zero.
<code>\dim_gzero_new:N</code>		
<code>\dim_gzero_new:c</code>		
<hr/> <code>New: 2012-03-05</code>		
<hr/> <code>New: 2012-01-07</code>		

<code>\dim_if_exist_p:N</code>	<code>\dim_if_exist_p:N</code>	$\langle dimension \rangle$
<code>\dim_if_exist_p:c</code>	<code>\dim_if_exist:NTF</code>	$\langle dimension \rangle$ <code>{\true code}</code> <code>{\false code}</code>
<code>\dim_if_exist:NTF</code>	*	Tests whether the $\langle dimension \rangle$ is currently defined. This does not check that the
<code>\dim_if_exist:cTF</code>	*	$\langle dimension \rangle$ really is a dimension variable.

New: 2012-03-03

26.2 Setting dim variables

<code>\dim_add:Nn</code>	<code>\dim_add:Nn</code>	$\langle dimension \rangle$ <code>{\dim expr}</code>
<code>\dim_add:cn</code>		Adds the result of the <code>\dim expr</code> to the current content of the $\langle dimension \rangle$.
<code>\dim_gadd:Nn</code>		
<code>\dim_gadd:cn</code>		

Updated: 2011-10-22

<code>\dim_set:Nn</code>	<code>\dim_set:Nn</code>	$\langle dimension \rangle$ <code>{\dim expr}</code>
<code>\dim_set:cn</code>		Sets $\langle dimension \rangle$ to the value of <code>\dim expr</code> , which must evaluate to a length with units.
<code>\dim_gset:Nn</code>		
<code>\dim_gset:cn</code>		

Updated: 2011-10-22

<code>\dim_set_eq:NN</code>	<code>\dim_set_eq:NN</code>	$\langle dimension_1 \rangle$ $\langle dimension_2 \rangle$
<code>\dim_set_eq:(cN Nc cc)</code>		Sets the content of $\langle dimension_1 \rangle$ equal to that of $\langle dimension_2 \rangle$.
<code>\dim_gset_eq:NN</code>		
<code>\dim_gset_eq:(cN Nc cc)</code>		

<code>\dim_sub:Nn</code>	<code>\dim_sub:Nn</code>	$\langle dimension \rangle$ <code>{\dim expr}</code>
<code>\dim_sub:cn</code>		Subtracts the result of the <code>\dim expr</code> from the current content of the $\langle dimension \rangle$.
<code>\dim_gsub:Nn</code>		
<code>\dim_gsub:cn</code>		

Updated: 2011-10-22

26.3 Utilities for dimension calculations

<code>\dim_abs:n</code>	*	<code>\dim_abs:n</code>	<code>{\dim expr}</code>
<code>\dim_abs:n</code>		Updated: 2012-09-26	Converts the <code>\dim expr</code> to its absolute value, leaving the result in the input stream as a $\langle dimension denotation \rangle$.

<code>\dim_max:nn</code>	*	<code>\dim_max:nn</code>	<code>{\dim expr_1}</code> <code>{\dim expr_2}</code>
<code>\dim_min:nn</code>	*	<code>\dim_min:nn</code>	<code>{\dim expr_1}</code> <code>{\dim expr_2}</code>
<code>\dim_max:nn</code>		New: 2012-09-09	Evaluates the two <code>\dim exprs</code> and leaves either the maximum or minimum value in the
<code>\dim_min:nn</code>		Updated: 2012-09-26	input stream as appropriate, as a $\langle dimension denotation \rangle$.

`\dim_ratio:nn` ☆ `\dim_ratio:nn {<dim expr1>} {<dim expr2>}`

Updated: 2011-10-22 Parses the two `<dim exprs>` and converts the ratio of the two to a form suitable for use inside a `<dim expr>`. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
  { 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Ne \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays 327680/655360 on the terminal.

26.4 Dimension expression conditionals

`\dim_compare_p:nNn` ☆ `\dim_compare_p:nNn {<dim expr1>} <relation> {<dim expr2>}`

`\dim_compare:nNnTF` ☆ `\dim_compare:nNnTF`
`{<dim expr1>} <relation> {<dim expr2>}`
`{<>true code>} {<>false code>}`

This function first evaluates each of the `<dim exprs>` as described for `\dim_eval:n`. The two results are then compared using the `<relation>`:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\dim_compare:nTF` but around 5 times faster.

```

\dim_compare_p:n * \dim_compare_p:n
\dim_compare:nTF * {
  <dim expr1> <relation1>
  ...
  <dim exprN> <relationN>
  <dim exprN+1>
}
\dim_compare:nTF
{
  <dim expr1> <relation1>
  ...
  <dim exprN> <relationN>
  <dim exprN+1>
}
{(true code)} {(false code)}

```

Updated: 2013-01-13

This function evaluates the $\langle dim\ exprs \rangle$ as described for $\backslash dim_eval:n$ and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle dim\ expr_1 \rangle$ and $\langle dim\ expr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle dim\ expr_2 \rangle$ and $\langle dim\ expr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle dim\ expr_N \rangle$ and $\langle dim\ expr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields `true` if all comparisons are `true`. Each $\langle dim\ expr \rangle$ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other $\langle dim\ expr \rangle$ is evaluated and no other comparison is performed. The $\langle relations \rangle$ can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

This function is more flexible than $\backslash dim_compare:nNnTF$ but around 5 times slower.

```

\dim_case:nn  ☆ \dim_case:nnTF {⟨test dim expr⟩}
\dim_case:nnTF ☆ {
  {⟨dim expr case1⟩} {⟨code case1⟩}
  {⟨dim expr case2⟩} {⟨code case2⟩}
  ...
  {⟨dim expr casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}

```

New: 2013-07-24

This function evaluates the $\langle test\ dim\ expr \rangle$ and compares this in turn to each of the $\langle dim\ expr\ cases \rangle$. If the two are equal then the associated $\langle code \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function $\backslash dim_case:nn$, which does nothing if there is no match, is also available. For example

```

\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
  { 2 \l_tmpa_dim }
  {
    { 5 pt }      { Small }
    { 4 pt + 6 pt } { Medium }
    { - 10 pt }   { Negative }
  }
  { No idea! }

```

leaves “Medium” in the input stream.

26.5 Dimension expression loops

```

\dim_do_until:nNnn ☆ \dim_do_until:nNnn {⟨dim expr1⟩} ⟨relation⟩ {⟨dim expr2⟩} {⟨code⟩}

```

Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle dim\ exprs \rangle$ as described for $\backslash dim_compare:nNnTF$. If the test is **false** then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is **true**.

```

\dim_do_while:nNnn ☆ \dim_do_while:nNnn {⟨dim expr1⟩} ⟨relation⟩ {⟨dim expr2⟩} {⟨code⟩}

```

Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle dim\ exprs \rangle$ as described for $\backslash dim_compare:nNnTF$. If the test is **true** then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is **false**.

```

\dim_until_do:nNnn ☆ \dim_until_do:nNnn {⟨dim expr1⟩} ⟨relation⟩ {⟨dim expr2⟩} {⟨code⟩}

```

Evaluates the relationship between the two $\langle dim\ exprs \rangle$ as described for $\backslash dim_compare:nNnTF$, and then places the $\langle code \rangle$ in the input stream if the $\langle relation \rangle$ is **false**. After the $\langle code \rangle$ has been processed by \TeX the test is repeated, and a loop occurs until the test is **true**.

`\dim_while_do:nNnn` ☆ `\dim_while_do:nNnn {⟨dim expr1⟩} ⟨relation⟩ {⟨dim expr2⟩} {⟨code⟩}`

Evaluates the relationship between the two `⟨dim exprs⟩` as described for `\dim_compare:nNnTF`, and then places the `⟨code⟩` in the input stream if the `⟨relation⟩` is true. After the `⟨code⟩` has been processed by T_EX the test is repeated, and a loop occurs until the test is false.

`\dim_do_until:nn` ☆ `\dim_do_until:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13 Places the `⟨code⟩` in the input stream for T_EX to process, and then evaluates the `⟨dimension relation⟩` as described for `\dim_compare:nTF`. If the test is false then the `⟨code⟩` is inserted into the input stream again and a loop occurs until the `⟨relation⟩` is true.

`\dim_do_while:nn` ☆ `\dim_do_while:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13 Places the `⟨code⟩` in the input stream for T_EX to process, and then evaluates the `⟨dimension relation⟩` as described for `\dim_compare:nTF`. If the test is true then the `⟨code⟩` is inserted into the input stream again and a loop occurs until the `⟨relation⟩` is false.

`\dim_until_do:nn` ☆ `\dim_until_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13 Evaluates the `⟨dimension relation⟩` as described for `\dim_compare:nTF`, and then places the `⟨code⟩` in the input stream if the `⟨relation⟩` is false. After the `⟨code⟩` has been processed by T_EX the test is repeated, and a loop occurs until the test is true.

`\dim_while_do:nn` ☆ `\dim_while_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13 Evaluates the `⟨dimension relation⟩` as described for `\dim_compare:nTF`, and then places the `⟨code⟩` in the input stream if the `⟨relation⟩` is true. After the `⟨code⟩` has been processed by T_EX the test is repeated, and a loop occurs until the test is false.

26.6 Dimension step functions

`\dim_step_function:nnnN` ☆ `\dim_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩`

New: 2018-02-18 This function first evaluates the `⟨initial value⟩`, `⟨step⟩` and `⟨final value⟩`, all of which should be dimension expressions. The `⟨function⟩` is then placed in front of each `⟨value⟩` from the `⟨initial value⟩` to the `⟨final value⟩` in turn (using `⟨step⟩` between each `⟨value⟩`). The `⟨step⟩` must be non-zero. If the `⟨step⟩` is positive, the loop stops when the `⟨value⟩` becomes larger than the `⟨final value⟩`. If the `⟨step⟩` is negative, the loop stops when the `⟨value⟩` becomes smaller than the `⟨final value⟩`. The `⟨function⟩` should absorb one argument.

`\dim_step_inline:nnnn` ☆ `\dim_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}`

New: 2018-02-18 This function first evaluates the `⟨initial value⟩`, `⟨step⟩` and `⟨final value⟩`, all of which should be dimension expressions. Then for each `⟨value⟩` from the `⟨initial value⟩` to the `⟨final value⟩` in turn (using `⟨step⟩` between each `⟨value⟩`), the `⟨code⟩` is inserted into the input stream with `#1` replaced by the current `⟨value⟩`. Thus the `⟨code⟩` should define a function of one argument (`#1`).

`\dim_step_variable:nnnNn`

New: 2018-02-18

`\dim_step_variable:nnnNn``{⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}`

This function first evaluates the `⟨initial value⟩`, `⟨step⟩` and `⟨final value⟩`, all of which should be dimension expressions. Then for each `⟨value⟩` from the `⟨initial value⟩` to the `⟨final value⟩` in turn (using `⟨step⟩` between each `⟨value⟩`), the `⟨code⟩` is inserted into the input stream, with the `⟨tl var⟩` defined as the current `⟨value⟩`. Thus the `⟨code⟩` should make use of the `⟨tl var⟩`.

26.7 Using dim expressions and variables

`\dim_eval:n` ★

Updated: 2011-10-22

`\dim_eval:n` `{⟨dim expr⟩}`

Evaluates the `⟨dim expr⟩`, expanding any dimensions and token list variables within the `⟨expression⟩` to their content (without requiring `\dim_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a `⟨dimension denotation⟩` after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a `TeX`-style assignment as it is *not* an `⟨internal dimension⟩`.

`\dim_sign:n` ★

New: 2018-11-03

`\dim_sign:n` `{⟨dim expr⟩}`

Evaluates the `⟨dim expr⟩` then leaves 1 or 0 or `-1` in the input stream according to the sign of the result.

`\dim_use:N` ★

`\dim_use:c` ★**`\dim_use:N`** `⟨dimension⟩`

Recovers the content of a `⟨dimension⟩` and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a `⟨dimension⟩` is required (such as in the argument of `\dim_eval:n`).

TeXhackers note: `\dim_use:N` is the `TeX` primitive `\the`: this is one of several `LATeX3` names for this primitive.

`\dim_to_decimal:n` ★

New: 2014-07-15

`\dim_to_decimal:n` `{⟨dim expr⟩}`

Evaluates the `⟨dim expr⟩`, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by `TeX` to at most five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal:n { 1bp }
```

leaves 1.00374 in the input stream, *i.e.* the magnitude of one “big point” when converted to (`TeX`) points.

`\dim_to_decimal_in_bp:n` * `\dim_to_decimal_in_bp:n {<dim expr>}`

New: 2014-07-15
Updated: 2023-05-20

Evaluates the `<dim expr>`, and leaves the result, expressed in big points (`bp`) in the input stream, with *no units*. The result is rounded by `TeX` to at most five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves 0.99628 in the input stream, *i.e.* the magnitude of one (`TeX`) point when converted to big points.

TeXhackers note: The implementation of this function is re-entrant: the result of

```
\dim_compare:nNnTF  
{ <x>bp } =  
  { \dim_to_decimal_in_bp:n { <x>bp } bp }
```

will be logically `true`. The decimal representations may differ provided they produce the same `TeX` dimension.

`\dim_to_decimal_in_cc:n` * `\dim_to_decimal_in_cm:n` {<dim expr>}

`\dim_to_decimal_in_cm:n` *
`\dim_to_decimal_in_dd:n` *
`\dim_to_decimal_in_in:n` *
`\dim_to_decimal_in_mm:n` *
`\dim_to_decimal_in_pc:n` *

Evaluates the `<dim expr>`, and leaves the result, expressed with the appropriate scaling in the input stream, with *no units*. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precisions of the result is limited to a maximum of five decimal places with trailing zeros omitted.

New: 2023-05-20

The maximum `TeX` allowable dimension value (available as `\maxdimen` in plain `TeX` and `\LaTeX` and `\c_max_dim` in `expl3`) can only be expressed exactly in the units `pt`, `bp` and `sp`. The maximum allowable input values to five decimal places are

```
1276.00215 cc  
575.83174 cm  
15312.02584 dd  
226.70540 in  
5758.31742 mm  
1365.33333 pc
```

(Note that these are not all equal, but rather any larger value will overflow due to the way `TeX` converts to `sp`.) Values given to five decimal places larger than these will result in `TeX` errors; the behavior if additional decimal places are given depends on the `TeX` internals and thus larger values are *not* supported by `expl3`.

TeXhackers note: The implementation of these functions is re-entrant: the result of

```
\dim_compare:nNnTF  
{ <x><unit> } =  
  { \dim_to_decimal_in_<unit>:n { <x><unit> } <unit> }
```

will be logically `true`. The decimal representations may differ provided they produce the same `TeX` dimension.

`\dim_to_decimal_in_sp:n` * `\dim_to_decimal_in_sp:n` {*dim expr*}

New: 2015-05-18 Evaluates the *dim expr*, and leaves the result, expressed in scaled points (*sp*) in the input stream, with *no units*. The result is necessarily an integer.

`\dim_to_decimal_in_unit:nn` * `\dim_to_decimal_in_unit:nn` {*dim expr*₁} {*dim expr*₂}

New: 2014-07-15
Updated: 2023-05-20

Evaluates the *dim exprs*, and leaves the value of *dim expr*₁, expressed in a unit given by *dim expr*₂, in the input stream. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precisions of the result is limited to a maximum of five decimal places with trailing zeros omitted.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves 0.35278 in the input stream, *i.e.* the magnitude of one big point when expressed in millimetres. The conversions do *not* guarantee that \TeX would yield identical results for the direct input in an equality test, thus for instance

```
\dim_compare:nNnTF  
  { 1bp } =  
    { \dim_to_decimal_in_unit:nn { 1bp } { 1mm } mm }
```

will take the *false* branch.

`\dim_to_fp:n` * `\dim_to_fp:n` {*dim expr*}

New: 2012-05-08 Expands to an internal floating point number equal to the value of the *dim expr* in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, `\dim_to_fp:n` can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

26.8 Viewing dim variables

`\dim_show:N` `\dim_show:N` *dimension*

`\dim_show:c` Displays the value of the *dimension* on the terminal.

`\dim_show:n` `\dim_show:n` {*dim expr*}

New: 2011-11-22
Updated: 2015-08-07 Displays the result of evaluating the *dim expr* on the terminal.

`\dim_log:N` `\dim_log:N` *dimension*

`\dim_log:c` Writes the value of the *dimension* in the log file.

New: 2014-08-22
Updated: 2015-08-03

`\dim_log:n` `\dim_log:n {⟨dim expr⟩}`
New: 2014-08-22 Writes the result of evaluating the `⟨dim expr⟩` in the log file.
Updated: 2015-08-07

26.9 Constant dimensions

`\c_max_dim` The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

`\c_zero_dim` A zero length as a dimension. This can also be used as a component of a skip.

26.10 Scratch dimensions

`\l_tmpa_dim` Scratch dimension for local assignment. These are never used by the kernel code, and so
`\l_tmpb_dim` are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_dim` Scratch dimension for global assignment. These are never used by the kernel code, and
`\g_tmpb_dim` so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

26.11 Creating and initialising skip variables

`\skip_new:N` `\skip_new:N ⟨skip⟩`
`\skip_new:c` Creates a new `⟨skip⟩` or raises an error if the name is already taken. The declaration is global. The `⟨skip⟩` is initially equal to 0 pt.

`\skip_const:Nn` `\skip_const:Nn ⟨skip⟩ {⟨skip expr⟩}`
`\skip_const:cn` Creates a new constant `⟨skip⟩` or raises an error if the name is already taken. The value of the `⟨skip⟩` is set globally to the `⟨skip expr⟩`.
New: 2012-03-05

`\skip_zero:N` `\skip_zero:N ⟨skip⟩`
`\skip_zero:c` Sets `⟨skip⟩` to 0 pt.
`\skip_gzero:N`
`\skip_gzero:c`

<code>\skip_zero_new:N</code>	<code>\skip_zero_new:N</code> $\langle skip \rangle$
<code>\skip_zero_new:c</code>	
<code>\skip_gzero_new:N</code>	Ensures that the $\langle skip \rangle$ exists globally by applying <code>\skip_new:N</code> if necessary, then applies <code>\skip_(g)zero:N</code> to leave the $\langle skip \rangle$ set to zero.
<code>\skip_gzero_new:c</code>	

New: 2012-01-07

<code>\skip_if_exist_p:N</code> *	<code>\skip_if_exist_p:N</code> $\langle skip \rangle$
<code>\skip_if_exist_p:c</code> *	<code>\skip_if_exist:NTF</code> $\langle skip \rangle$ $\{(true\ code)\}$ $\{(false\ code)\}$
<code>\skip_if_exist:NTF</code> *	Tests whether the $\langle skip \rangle$ is currently defined. This does not check that the $\langle skip \rangle$ really is a skip variable.
<code>\skip_if_exist:cTF</code> *	

New: 2012-03-03

26.12 Setting skip variables

<code>\skip_add:Nn</code>	<code>\skip_add:Nn</code> $\langle skip \rangle$ $\{(skip\ expr)\}$
<code>\skip_add:cn</code>	
<code>\skip_gadd:Nn</code>	Adds the result of the $\langle skip\ expr \rangle$ to the current content of the $\langle skip \rangle$.
<code>\skip_gadd:cn</code>	

Updated: 2011-10-22

<code>\skip_set:Nn</code>	<code>\skip_set:Nn</code> $\langle skip \rangle$ $\{(skip\ expr)\}$
<code>\skip_set:cn</code>	
<code>\skip_gset:Nn</code>	Sets $\langle skip \rangle$ to the value of $\langle skip\ expr \rangle$, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).
<code>\skip_gset:cn</code>	

Updated: 2011-10-22

<code>\skip_set_eq:NN</code>	<code>\skip_set_eq:NN</code> $\langle skip_1 \rangle$ $\langle skip_2 \rangle$
<code>\skip_set_eq:(cN Nc cc)</code>	
<code>\skip_gset_eq:NN</code>	Sets the content of $\langle skip_1 \rangle$ equal to that of $\langle skip_2 \rangle$.
<code>\skip_gset_eq:(cN Nc cc)</code>	

<code>\skip_sub:Nn</code>	<code>\skip_sub:Nn</code> $\langle skip \rangle$ $\{(skip\ expr)\}$
<code>\skip_sub:cn</code>	
<code>\skip_gsub:Nn</code>	Subtracts the result of the $\langle skip\ expr \rangle$ from the current content of the $\langle skip \rangle$.
<code>\skip_gsub:cn</code>	

Updated: 2011-10-22

26.13 Skip expression conditionals

`\skip_if_eq_p:nn` * `\skip_if_eq_p:nn` $\langle\textit{skip expr}_1\rangle$ $\langle\textit{skip expr}_2\rangle$
`\skip_if_eq:nnTF` * `\skip_if_eq:nnTF`
 $\langle\textit{skip expr}_1\rangle$ $\langle\textit{skip expr}_2\rangle$
 $\langle\textit{true code}\rangle$ $\langle\textit{false code}\rangle$

This function first evaluates each of the $\langle\textit{skip exprs}\rangle$ as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

`\skip_if_finite_p:n` * `\skip_if_finite_p:n` $\langle\textit{skip expr}\rangle$
`\skip_if_finite:nTF` * `\skip_if_finite:nTF` $\langle\textit{skip expr}\rangle$ $\langle\textit{true code}\rangle$ $\langle\textit{false code}\rangle$

New: 2012-03-05 Evaluates the $\langle\textit{skip expr}\rangle$ as described for `\skip_eval:n`, and then tests if all of its components are finite.

26.14 Using skip expressions and variables

`\skip_eval:n` * `\skip_eval:n` $\langle\textit{skip expr}\rangle$

Updated: 2011-10-22 Evaluates the $\langle\textit{skip expr}\rangle$, expanding any skips and token list variables within the $\langle\textit{expression}\rangle$ to their content (without requiring `\skip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle\textit{glue denotation}\rangle$ after two expansions. This is expressed in points (pt), and requires suitable termination if used in a T_EX-style assignment as it is *not* an $\langle\textit{internal glue}\rangle$.

`\skip_use:N` * `\skip_use:N` $\langle\textit{skip}\rangle$

`\skip_use:c` * Recovers the content of a $\langle\textit{skip}\rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle\textit{dimension}\rangle$ or $\langle\textit{skip}\rangle$ is required (such as in the argument of `\skip_eval:n`).

T_EXhackers note: `\skip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

26.15 Viewing skip variables

`\skip_show:N` * `\skip_show:N` $\langle\textit{skip}\rangle$

`\skip_show:c` Displays the value of the $\langle\textit{skip}\rangle$ on the terminal.

Updated: 2015-08-03

`\skip_show:n` * `\skip_show:n` $\langle\textit{skip expr}\rangle$

New: 2011-11-22 Displays the result of evaluating the $\langle\textit{skip expr}\rangle$ on the terminal.

Updated: 2015-08-07

`\skip_log:N` `\skip_log:N` $\langle skip \rangle$
`\skip_log:c` Writes the value of the $\langle skip \rangle$ in the log file.
New: 2014-08-22
Updated: 2015-08-03

`\skip_log:n` `\skip_log:n` $\{\langle skip expr \rangle\}$
New: 2014-08-22 Writes the result of evaluating the $\langle skip expr \rangle$ in the log file.
Updated: 2015-08-07

26.16 Constant skips

`\c_max_skip` The maximum value that can be stored as a skip (equal to `\c_max_dim` in length), with no stretch nor shrink component.
Updated: 2012-11-02

`\c_zero_skip` A zero length as a skip, with no stretch nor shrink component.
Updated: 2012-11-01

26.17 Scratch skips

`\l_tmpa_skip` Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
`\l_tmpb_skip`

`\g_tmpa_skip` Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
`\g_tmpb_skip`

26.18 Inserting skips into the output

`\skip_horizontal:N` `\skip_horizontal:N` $\langle skip \rangle$
`\skip_horizontal:c` `\skip_horizontal:n` $\{\langle skip expr \rangle\}$
`\skip_horizontal:n` Inserts a horizontal $\langle skip \rangle$ into the current list. The argument can also be a $\langle dim \rangle$.
Updated: 2011-10-22

T_EXhackers note: `\skip_horizontal:N` is the T_EX primitive `\hskip`.

<code>\skip_vertical:N</code>	<code>\skip_vertical:N</code> $\langle skip \rangle$
<code>\skip_vertical:c</code>	<code>\skip_vertical:n</code> $\{ \langle skip \textit> expr \rangle \}$
<code>\skip_vertical:n</code>	Inserts a vertical $\langle skip \rangle$ into the current list. The argument can also be a $\langle dim \rangle$.

Updated: 2011-10-22

T_EXhackers note: `\skip_vertical:N` is the T_EX primitive `\vskip`.

26.19 Creating and initialising muskip variables

<code>\muskip_new:N</code>	<code>\muskip_new:N</code> $\langle muskip \rangle$
<code>\muskip_new:c</code>	Creates a new $\langle muskip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle muskip \rangle$ is initially equal to 0 mu.

<code>\muskip_const:Nn</code>	<code>\muskip_const:Nn</code> $\langle muskip \rangle$ $\{ \langle muskip \textit> expr \rangle \}$
<code>\muskip_const:cn</code>	Creates a new constant $\langle muskip \rangle$ or raises an error if the name is already taken. The value of the $\langle muskip \rangle$ is set globally to the $\langle muskip \textit> expr \rangle$.

New: 2012-03-05

<code>\muskip_zero:N</code>	<code>\skip_zero:N</code> $\langle muskip \rangle$
<code>\muskip_zero:c</code>	Sets $\langle muskip \rangle$ to 0 mu.
<code>\muskip_gzero:N</code>	
<code>\muskip_gzero:c</code>	

<code>\muskip_zero_new:N</code>	<code>\muskip_zero_new:N</code> $\langle muskip \rangle$
<code>\muskip_zero_new:c</code>	Ensures that the $\langle muskip \rangle$ exists globally by applying <code>\muskip_new:N</code> if necessary, then applies <code>\muskip_(g)zero:N</code> to leave the $\langle muskip \rangle$ set to zero.
<code>\muskip_gzero_new:N</code>	
<code>\muskip_gzero_new:c</code>	

New: 2012-01-07

<code>\muskip_if_exist_p:N</code>	<code>\muskip_if_exist_p:N</code> $\langle muskip \rangle$
<code>\muskip_if_exist_p:c</code>	<code>\muskip_if_exist:NTF</code> $\langle muskip \rangle$ $\{ \langle true \textit> code \rangle \}$ $\{ \langle false \textit> code \rangle \}$
<code>\muskip_if_exist:NTF</code>	*
<code>\muskip_if_exist:cTF</code>	*
	Tests whether the $\langle muskip \rangle$ is currently defined. This does not check that the $\langle muskip \rangle$ really is a muskip variable.

New: 2012-03-03

26.20 Setting muskip variables

<code>\muskip_add:Nn</code>	<code>\muskip_add:Nn</code> $\langle muskip \rangle$ $\{ \langle muskip \textit> expr \rangle \}$
<code>\muskip_add:cn</code>	Adds the result of the $\langle muskip \textit> expr \rangle$ to the current content of the $\langle muskip \rangle$.
<code>\muskip_gadd:Nn</code>	
<code>\muskip_gadd:cn</code>	

Updated: 2011-10-22

<code>\muskip_set:Nn</code>	<code>\muskip_set:Nn <muskip> {<muskip expr>}</code>
<code>\muskip_set:cn</code>	Sets $\langle muskip \rangle$ to the value of $\langle muskip expr \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).
<code>\muskip_gset:Nn</code>	
<code>\muskip_gset:cn</code>	

Updated: 2011-10-22

<code>\muskip_set_eq:NN</code>	<code>\muskip_set_eq:NN <muskip₁> <muskip₂></code>
<code>\muskip_set_eq:(cN Nc cc)</code>	Sets the content of $\langle muskip_1 \rangle$ equal to that of $\langle muskip_2 \rangle$.
<code>\muskip_gset_eq:NN</code>	
<code>\muskip_gset_eq:(cN Nc cc)</code>	

<code>\muskip_sub:Nn</code>	<code>\muskip_sub:Nn <muskip> {<muskip expr>}</code>
<code>\muskip_sub:cn</code>	Subtracts the result of the $\langle muskip expr \rangle$ from the current content of the $\langle muskip \rangle$.
<code>\muskip_gsub:Nn</code>	
<code>\muskip_gsub:cn</code>	

Updated: 2011-10-22

26.21 Using muskip expressions and variables

<code>\muskip_eval:n</code> *	<code>\muskip_eval:n {<muskip expr>}</code>
-------------------------------	---

Updated: 2011-10-22

Evaluates the $\langle muskip expr \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring `\muskip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle muglue denotation \rangle$ after two expansions. This is expressed in mu, and requires suitable termination if used in a T_EX-style assignment as it is *not* an $\langle internal muglue \rangle$.

<code>\muskip_use:N</code> *	<code>\muskip_use:N <muskip></code>
<code>\muskip_use:c</code> *	Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of <code>\muskip_eval:n</code>).

T_EXhackers note: `\muskip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

26.22 Viewing muskip variables

<code>\muskip_show:N</code>	<code>\muskip_show:N <muskip></code>
<code>\muskip_show:c</code>	Displays the value of the $\langle muskip \rangle$ on the terminal.

Updated: 2015-08-03

`\muskip_show:n` `\muskip_show:n {⟨muskip expr⟩}`
New: 2011-11-22 Displays the result of evaluating the `⟨muskip expr⟩` on the terminal.
Updated: 2015-08-07

`\muskip_log:N` `\muskip_log:N ⟨muskip⟩`
`\muskip_log:c` Writes the value of the `⟨muskip⟩` in the log file.
New: 2014-08-22
Updated: 2015-08-03

`\muskip_log:n` `\muskip_log:n {⟨muskip expr⟩}`
New: 2014-08-22 Writes the result of evaluating the `⟨muskip expr⟩` in the log file.
Updated: 2015-08-07

26.23 Constant muskips

`\c_max_muskip` The maximum value that can be stored as a muskip, with no stretch nor shrink component.

`\c_zero_muskip` A zero length as a muskip, with no stretch nor shrink component.

26.24 Scratch muskips

`\l_tmpa_muskip` Scratch muskip for local assignment. These are never used by the kernel code, and so
`\l_tmpb_muskip` are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_muskip` Scratch muskip for global assignment. These are never used by the kernel code, and so
`\g_tmpb_muskip` are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

26.25 Primitive conditional

`\if_dim:w` ★ `\if_dim:w ⟨dimen1⟩ ⟨relation⟩ ⟨dimen2⟩`
 `⟨true code⟩`
 `\else:`
 `⟨false⟩`
 `\fi:`
Compare two dimensions. The `⟨relation⟩` is one of `<`, `=` or `>` with category code 12.

T_EXhackers note: This is the T_EX primitive `\ifdim`.

Chapter 27

The I3keys module

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
{
  key-one .code:n = code including parameter #1,
  key-two .tl_set:N = \l_mymodule_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
{
  key-one = value one,
  key-two = value two
}
```

As illustrated, keys are created inside a `<module>`: a set of related keys, typically those for a single module/L^AT_ΕX 2_ε package. See Section 27.2 for suggestions on how to divide large numbers of keys for a single module.

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
  { \keys_set:nn { mymodule } { #1 } }
\DeclareDocumentCommand \MyModuleMacro { o m }
  {
    \group_begin:
      \keys_set:nn { mymodule } { #1 }
      % Main code for \MyModuleMacro
    \group_end:
  }
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As discussed in section 27.2, it is suggested that the character `/` is reserved for sub-division of keys into different subsets. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
  {
    \l_mymodule_tmp_tl .code:n = code
  }
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

27.1 Creating keys

```
\keys_define:nn <module> {<keyval list>}
```

`\keys_define:ne` Parses the `<keyval list>` and defines the keys listed there for `<module>`. The `<module>` name is treated as a string. In practice the `<module>` should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

Updated: 2017-11-14

The `<keyval list>` should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
  {
    keyname .code:n = Some-code-using~#1,
    keyname .value_required:n = true
  }
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary `<key>`, which when used may be supplied with a `<value>`. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define “actions”, such as `.code:n`, `.tl_set:N`, *etc.*, override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
{
  keyname .code:n          = Some~code~using~#1,
  keyname .value_required:n = true
}
\keys_define:nn { mymodule }
{
  keyname .value_required:n = true,
  keyname .code:n          = Some~code~using~#1
}
```

Note that all key properties define the key within the current T_EX group, with an exception that the special `.undefine:` property *undefines* the key within the current T_EX group.

<code>.bool_set:N</code>	<code><key> .bool_set:N = <boolean variable></code>
<code>.bool_set:c</code>	Defines <code><key></code> to set <code><boolean variable></code> to <code><value></code> (which must be either “true” or “false”). If the variable does not exist, it will be created globally at the point that the key is set up.
<code>.bool_gset:N</code>	
<code>.bool_gset:c</code>	

Updated: 2013-07-08

<code>.bool_set_inverse:N</code>	<code><key> .bool_set_inverse:N = <boolean variable></code>
<code>.bool_set_inverse:c</code>	Defines <code><key></code> to set <code><boolean variable></code> to the logical inverse of <code><value></code> (which must be either “true” or “false”). If the <code><boolean variable></code> does not exist, it will be created globally at the point that the key is set up.
<code>.bool_gset_inverse:N</code>	
<code>.bool_gset_inverse:c</code>	

New: 2011-08-28
Updated: 2013-07-08

<code>.choice:</code>	<code><key> .choice:</code>
-----------------------	-----------------------------------

Sets `<key>` to act as a choice key. Each valid choice for `<key>` must then be created, as discussed in section 27.3.

<code>.choices:nn</code>	<code><key> .choices:nn = {<choices>} {<code>}</code>
<code>.choices:(Vn en on)</code>	Sets <code><key></code> to act as a choice key, and defines a series <code><choices></code> which are implemented using the <code><code></code> . Inside <code><code></code> , <code>\l_keys_choice_tl</code> will be the name of the choice made, and <code>\l_keys_choice_int</code> will be the position of the choice in the list of <code><choices></code> (indexed from 1). Choices are discussed in detail in section 27.3.
New: 2011-08-21	
Updated: 2013-07-10	

<code>.clist_set:N</code>	<code><key> .clist_set:N = <comma list variable></code>
<code>.clist_set:c</code>	Defines <code><key></code> to set <code><comma list variable></code> to <code><value></code> . Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.
<code>.clist_gset:N</code>	
<code>.clist_gset:c</code>	

New: 2011-09-11

<code>.code:n</code>	<code><key> .code:n = {<code>}</code>
----------------------	---

Updated: 2013-07-10

Stores the `<code>` for execution when `<key>` is used. The `<code>` can include one parameter (`#1`), which will be the `<value>` given for the `<key>`.

<code>.cs_set:Np</code>	<code><key> .cs_set:Np = <control sequence> <arg. spec.></code>
<code>.cs_set:cp</code>	Defines <code><key></code> to set <code><control sequence></code> to have <code><arg. spec.></code> and replacement text <code><value></code> .
<code>.cs_set_protected:Np</code>	
<code>.cs_set_protected:cp</code>	
<code>.cs_gset:Np</code>	
<code>.cs_gset:cp</code>	
<code>.cs_gset_protected:Np</code>	
<code>.cs_gset_protected:cp</code>	

New: 2020-01-11

<code>.default:n</code>	<code><key> .default:n = {<default>}</code>
-------------------------	---

Updated: 2013-07-09

Creates a `<default>` value for `<key>`, which is used if no value is given. This will be used if only the key name is given, but not if a blank `<value>` is given:

```

\keys_define:nn { mymodule }
{
  key .code:n      = Hello~#1,
  key .default:n = World
}
\keys_set:nn { mymodule }
{
  key = Fred, % Prints 'Hello Fred'
  key,      % Prints 'Hello World'
  key = ,   % Prints 'Hello '
}

```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

When no value is given for a key as part of `\keys_set:nn`, the `.default:n` value provides the value before key properties are considered. The only exception is when the `.value_required:n` property is active: a required value cannot be supplied by the default, and must be explicitly given as part of `\keys_set:nn`.

<code>.dim_set:N</code>	<code><key> .dim_set:N = <dimension></code>
<code>.dim_set:c</code>	Defines <code><key></code> to set <code><dimension></code> to <code><value></code> (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
<code>.dim_gset:N</code>	
<code>.dim_gset:c</code>	

Updated: 2020-01-17

<code>.fp_set:N</code>	<code><key> .fp_set:N = <floating point></code>
<code>.fp_set:c</code>	Defines <code><key></code> to set <code><floating point></code> to <code><value></code> (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
<code>.fp_gset:N</code>	
<code>.fp_gset:c</code>	

Updated: 2020-01-17

`.groups:n` $\langle key \rangle$ `.groups:n = { $\langle groups \rangle$ }`

New: 2013-07-14 Defines $\langle key \rangle$ as belonging to the $\langle groups \rangle$ (a comma-separated list). Groups provide a “secondary axis” for selectively setting keys, and are described in Section 27.7.

T_EXhackers note: The $\langle groups \rangle$ argument is turned into a string then interpreted as a comma-separated list, so group names cannot contain commas nor start or end with a space character.

`.inherit:n` $\langle key \rangle$ `.inherit:n = { $\langle parents \rangle$ }`

New: 2016-11-22 Specifies that the $\langle key \rangle$ path should inherit the keys listed as any of the $\langle parents \rangle$ (a comma list), which can be a module or a sub-division thereof. For example, after setting

```
\keys_define:nn { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:nn { } { bar .inherit:n = foo }
```

setting

```
\keys_set:nn { bar } { test = a }
```

will be equivalent to

```
\keys_set:nn { foo } { test = a }
```

Inheritance applies at point of use, not at definition, thus keys may be added to the $\langle parent \rangle$ after the use of `.inherit:n` and will be active. If more than one $\langle parent \rangle$ is specified, the presence of the $\langle key \rangle$ will be tested for each in turn, with the first successful hit taking priority.

`.initial:n` $\langle key \rangle$ `.initial:n = { $\langle value \rangle$ }`

`.initial:(V|e|o)` Initialises the $\langle key \rangle$ with the $\langle value \rangle$, equivalent to

Updated: 2013-07-09

```
\keys_set:nn { $\langle module \rangle$ } {  $\langle key \rangle$  =  $\langle value \rangle$  }
```

`.int_set:N` $\langle key \rangle$ `.int_set:N = $\langle integer \rangle$`

`.int_set:c`

`.int_gset:N`

`.int_gset:c`

Updated: 2020-01-17

Defines $\langle key \rangle$ to set $\langle integer \rangle$ to $\langle value \rangle$ (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

`.legacy_if_set:n` $\langle key \rangle$ `.legacy_if_set:n = $\langle switch \rangle$`

`.legacy_if_gset:n`

`.legacy_if_set_inverse:n`

`.legacy_if_gset_inverse:n`

Updated: 2022-01-15

Defines $\langle key \rangle$ to set legacy `\if $\langle switch \rangle$` to $\langle value \rangle$ (which must be either “true” or “false”). The $\langle switch \rangle$ is the name of the switch *without the leading if*.

The inverse versions will set the $\langle switch \rangle$ to the logical opposite of the $\langle value \rangle$.

`.meta:n` $\langle key \rangle$ `.meta:n = { $\langle keyval list \rangle$ }`

Updated: 2013-07-10 Makes $\langle key \rangle$ a meta-key, which will set $\langle keyval list \rangle$ in one go. The $\langle keyval list \rangle$ can refer as `#1` to the value given at the time the $\langle key \rangle$ is used (or, if no value is given, the $\langle key \rangle$'s default value).

`.meta:nn` $\langle key \rangle$ `.meta:nn = { $\langle path \rangle$ } { $\langle keyval list \rangle$ }`

New: 2013-07-10 Makes $\langle key \rangle$ a meta-key, which will set $\langle keyval list \rangle$ in one go using the $\langle path \rangle$ in place of the current one. The $\langle keyval list \rangle$ can refer as #1 to the value given at the time the $\langle key \rangle$ is used (or, if no value is given, the $\langle key \rangle$'s default value).

`.multichoice:` $\langle key \rangle$ `.multichoice:`

New: 2011-08-21 Sets $\langle key \rangle$ to act as a multiple choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 27.3.

`.multichoices:nn` $\langle key \rangle$ `.multichoices:nn { $\langle choices \rangle$ } { $\langle code \rangle$ }`

`.multichoices:(Vn|en|on)`

New: 2011-08-21
Updated: 2013-07-10

Sets $\langle key \rangle$ to act as a multiple choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section 27.3.

`.muskip_set:N` $\langle key \rangle$ `.muskip_set:N = $\langle muskip \rangle$`

`.muskip_set:c`

`.muskip_gset:N`

`.muskip_gset:c`

New: 2019-05-05
Updated: 2020-01-17

Defines $\langle key \rangle$ to set $\langle muskip \rangle$ to $\langle value \rangle$ (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

`.prop_put:N` $\langle key \rangle$ `.prop_put:N = $\langle property list \rangle$`

`.prop_put:c`

`.prop_gput:N`

`.prop_gput:c`

New: 2019-01-31

Defines $\langle key \rangle$ to put the $\langle value \rangle$ onto the $\langle property list \rangle$ stored under the $\langle key \rangle$. If the variable does not exist, it is created globally at the point that the key is set up.

`.skip_set:N` $\langle key \rangle$ `.skip_set:N = $\langle skip \rangle$`

`.skip_set:c`

`.skip_gset:N`

`.skip_gset:c`

Updated: 2020-01-17

Defines $\langle key \rangle$ to set $\langle skip \rangle$ to $\langle value \rangle$ (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

`.str_set:N` $\langle key \rangle$ `.str_set:N = $\langle string variable \rangle$`

`.str_set:c`

`.str_gset:N`

`.str_gset:c`

New: 2021-10-30

Defines $\langle key \rangle$ to set $\langle string variable \rangle$ to $\langle value \rangle$. If the variable does not exist, it is created globally at the point that the key is set up.

`.str_set_e:N` $\langle key \rangle$ `.str_set_e:N = $\langle string variable \rangle$`

`.str_set_e:c`

`.str_gset_e:N`

`.str_gset_e:c`

New: 2023-09-18

Defines $\langle key \rangle$ to set $\langle string variable \rangle$ to $\langle value \rangle$, which will be subjected to an e-type expansion (*i.e.* using `\str_set:Ne`). If the variable does not exist, it is created globally at the point that the key is set up.

```
.tl_set:N <key> .tl_set:N = <token list variable>
```

```
.tl_set:c
```

```
.tl_gset:N
```

```
.tl_gset:c
```

Defines $\langle key \rangle$ to set $\langle token list variable \rangle$ to $\langle value \rangle$. If the variable does not exist, it is created globally at the point that the key is set up.

```
.tl_set_e:N <key> .tl_set_e:N = <token list variable>
```

```
.tl_set_e:c
```

```
.tl_gset_e:N
```

```
.tl_gset_e:c
```

Defines $\langle key \rangle$ to set $\langle token list variable \rangle$ to $\langle value \rangle$, which will be subjected to an e-type expansion (*i.e.* using `\tl_set:N`). If the variable does not exist, it is created globally at the point that the key is set up.

New: 2023-09-18

```
.undefine: <key> .undefine:
```

New: 2015-07-14 Removes the definition of the $\langle key \rangle$ within the current \TeX group.

```
.value_forbidden:n <key> .value_forbidden:n = true|false
```

New: 2015-07-14 Specifies that $\langle key \rangle$ cannot receive a $\langle value \rangle$ when used. If a $\langle value \rangle$ is given then an error will be issued. Setting the property “false” cancels the restriction.

```
.value_required:n <key> .value_required:n = true|false
```

New: 2015-07-14 Specifies that $\langle key \rangle$ must receive a $\langle value \rangle$ when used. If a $\langle value \rangle$ is not given then an error will be issued. Setting the property “false” cancels the restriction.

27.2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several subsets for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { mymodule / subset }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
  { subset / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is `/`. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `mymodule/subset/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

27.3 Choice and multiple choice keys

The `l3keys` system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { mymodule }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependent only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
  {
    key .choices:nn =
      { choice-a, choice-b, choice-c }
      {
        You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
        which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      }
  }
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

`\l_keys_choice_int` `\l_keys_choice_tl` Inside the code block for a choice generated using `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e. the choice name) is also available as `#1`.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
  }
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined

behaviour when used outside of code created using `.choices:nn` (*i.e.* anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special `unknown` choice. The general behavior of the `unknown` key is described in Section 27.6. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
{
  key .choice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
  key / unknown .code:n =
    \msg_error:nnee { mymodule } { unknown-choice }
    { key } % Name of choice key
    { choice-a , choice-b , choice-c } % Valid choices
    { \exp_not:n {#1} } % Invalid choice given
  %
  %
}
```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```
\keys_define:nn { mymodule }
{
  key .multichoices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
      which~is~in~position~
      \int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}
```

and

```
\keys_define:nn { mymodule }
{
  key .multichoice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}
```

are valid.

When a multiple choice key is set

```
\keys_set:nn { mymodule }
{
  key = { a , b , c } % 'key' defined as a multiple choice
}
```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```
\keys_set:nn { mymodule }
{
  key = a ,
  key = b ,
  key = c ,
}
```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

27.4 Key usage scope

Some keys will be used as settings which have a strictly limited scope of usage. Some will be only available once, others will only be valid until typesetting begins. To allow formats to support this in a structured way, `l3keys` allows this information to be specified using the `.usage:n` property.

`.usage:n` $\langle key \rangle$ `.usage:n = $\langle scope \rangle$`

New: 2022-01-10 Defines the $\langle key \rangle$ to have usage within the $\langle scope \rangle$, which should be one of **general**, **preamble** or **load**.

`\l_keys_usage_load_prop`
`\l_keys_usage_preamble_prop`

New: 2022-01-10

`l3keys` itself does *not* attempt to redefine keys based on the usage scope. Rather, this information is made available with these two property lists. These hold an entry for each module (prefix); the value of each entry is a comma-separated list of the usage-restricted key(s).

27.5 Setting keys

`\keys_set:nn` `\keys_set:nn { $\langle module \rangle$ } { $\langle keyval list \rangle$ }`

`\keys_set:(nV|nv|ne|no)` Parses the $\langle keyval list \rangle$, and sets those keys which are defined for $\langle module \rangle$. The behaviour on finding an unknown key can be set by defining a special **unknown** key: this is illustrated later.

Updated: 2017-11-14

<code>\l_keys_path_str</code> <code>\l_keys_key_str</code> <code>\l_keys_value_tl</code>	For each key processed, information of the full <i>path</i> of the key, the <i>name</i> of the key and the <i>value</i> of the key is available within two string and one token list variables. These may be used within the code of the key.
--	---

Updated: 2020-02-08

The *path* of the key is a “full” description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_str`.

The *name* of the key is the part of the path after the last /, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_str`.

The *value* is everything after the =, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

27.6 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special `unknown` key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts. The `unknown` key also supports the `.default:n` property.

```
\keys_define:nn { mymodule }
{
  unknown .code:n =
    You~tried~to~set~key~'\l_keys_key_str'~to~'#1' . . ,
  unknown .default:V = \c_novalue_tl
}
```

<code>\keys_set_known:nn</code> <code>\keys_set_known:(nV nv ne no)</code> <code>\keys_set_known:nnN</code> <code>\keys_set_known:(nVN nvN neN noN)</code> <code>\keys_set_known:nnnN</code> <code>\keys_set_known:(nVnN nvnN nenN nonN)</code>	<code>\keys_set_known:nn {<module>} {<keyval list>}</code> <code>\keys_set_known:nnN {<module>} {<keyval list>} <tl></code> <code>\keys_set_known:nnnN {<module>} {<keyval list>} {<root>} <tl></code>
--	--

New: 2011-08-23

Updated: 2019-01-29

These functions set keys which are known for the `<module>`, and simply ignore other keys. The `\keys_set_known:nn` function parses the `<keyval list>`, and sets those keys which are defined for `<module>`. Any keys which are unknown are not processed further by the parser. In addition, `\keys_set_known:nnN` stores the key–value pairs in the `<tl>` in comma-separated form (*i.e.* an edited version of the `<keyval list>`). When a `<root>` is given (`\keys_set_known:nnnN`), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

27.7 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys_define:nn { mymodule }
{
  key-one   .code:n   = { \my_func:n {#1} } ,
  key-two   .tl_set:N = \l_my_a_tl       ,
  key-three .tl_set:N = \l_my_b_tl       ,
  key-four  .fp_set:N = \l_my_a_fp       ,
}
```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys_define:nn { mymodule }
{
  key-one   .code:n   = { \my_func:n {#1} } ,
  key-one   .groups:n = { first }           ,
  key-two   .tl_set:N = \l_my_a_tl       ,
  key-two   .groups:n = { first }           ,
  key-three .tl_set:N = \l_my_b_tl       ,
  key-three .groups:n = { second }        ,
  key-four  .fp_set:N = \l_my_a_fp       ,
}
```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

<code>\keys_set_exclude_groups:nnn</code>	<code>\keys_set_exclude_groups:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_exclude_groups:(nnV nnv nno)</code>	<code>list}</code>
<code>\keys_set_exclude_groups:nnnN</code>	<code>\keys_set_exclude_groups:nnnN {<module>} {<groups>}</code>
<code>\keys_set_exclude_groups:(nnVN nnvN nnoN)</code>	<code>{<keyval list>} {<tl>}</code>
<code>\keys_set_exclude_groups:nnnnN</code>	<code>\keys_set_exclude_groups:nnnnN {<module>} {<groups>}</code>
<code>\keys_set_exclude_groups:(nnVnN nnvnN nnonN)</code>	<code>{<keyval list>} {<root>} {<tl>}</code>

New: 2024-01-10

Sets keys by excluding those in the specified `<groups>`. The `<groups>` are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set. The key–value pairs for each key which is filtered out are stored in the `<tl>` in a comma-separated form (*i.e.* an edited version of the `<keyval list>`). The `\keys_set_exclude_groups:nnn` version skips this stage.

Use of `\keys_set_exclude_groups:nnnN` can be nested, with the correct residual `<keyval list>` returned at each stage. In the version which takes a `<root>` argument, the key list is returned relative to that point in the key tree. In the cases without a `<root>` argument, only the key names and values are returned.

```

\keys_set_groups:nnn          \keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}
\keys_set_groups:(nnV|nnv|nno)
\keys_set_groups:nnnN
\keys_set_groups:(nnVN|nnvN|nnoN)
\keys_set_groups:nnnnN
\keys_set_groups:(nnVnN|nnvnN|nnonN)

```

New: 2013-07-14

Updated: 2024-05-08

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the $\langle groups \rangle$ specified are set. The $\langle groups \rangle$ are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set. The key–value pairs for each key which is filtered out are stored in the $\langle t1 \rangle$ in a comma-separated form (*i.e.* an edited version of the $\langle keyval list \rangle$). The $\backslash keys_set_groups:nnn$ version skips this stage

27.8 Digesting keys

```

\keys_precompile:nnN \keys_precompile:nnN {<module>} {<keyval list>} <t1>

```

New: 2022-03-09

Parses the $\langle keyval list \rangle$ as for $\backslash keys_set:nn$, placing the resulting code for those which set variables or functions into the $\langle t1 \rangle$. Thus this function “precompiles” the keyval list into a set of results which can be applied rapidly.

27.9 Utility functions for keys

```

\keys_if_exist_p:nn * \keys_if_exist_p:nn {<module>} {<key>}
\keys_if_exist_p:ne * \keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}
\keys_if_exist:nnTF * Tests if the <key> exists for <module>, i.e. if any code has been defined for <key>.
\keys_if_exist:neTF *

```

Updated: 2022-01-10

```

\keys_if_choice_exist_p:nnn * \keys_if_choice_exist_p:nnn {<module>} {<key>} {<choice>}
\keys_if_choice_exist:nnnTF * \keys_if_choice_exist:nnnTF {<module>} {<key>} {<choice>} {<true code>}
                               {<false code>}

```

New: 2011-08-21

Updated: 2017-11-14

Tests if the $\langle choice \rangle$ is defined for the $\langle key \rangle$ within the $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle / \langle choice \rangle$. The test is **false** if the $\langle key \rangle$ itself is not defined.

```

\keys_show:nn \keys_show:nn {<module>} {<key>}

```

Updated: 2015-08-09

Displays in the terminal the information associated to the $\langle key \rangle$ for a $\langle module \rangle$, including the function which is used to actually implement it.

```

\keys_log:nn \keys_log:nn {<module>} {<key>}

```

New: 2014-08-22

Updated: 2015-08-09

Writes in the log file the information associated to the $\langle key \rangle$ for a $\langle module \rangle$. See also $\backslash keys_show:nn$ which displays the result in the terminal.

27.10 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,  
KeyTwo = ValueTwo ,  
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a *key-value list* into *keys* and associated *values*. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double # tokens or expand any input. Active tokens = and , appearing at the outer level of braces are converted to category “other” (12) so that the parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

<code>\keyval_parse:nnn</code>	☆ <code>\keyval_parse:nnn {<code₁>} {<code₂>} {<key-value list>}</code>
<code>\keyval_parse:(nnV nnv)</code>	☆ Parses the <code><key-value list></code> into a series of <code><keys></code> and associated <code><values></code> , or keys alone (if no <code><value></code> was given). <code><code₁></code> receives each <code><key></code> (with no <code><value></code>) as a trailing brace group, whereas <code><code₂></code> is appended by two brace groups, the <code><key></code> and <code><value></code> . The order of the <code><keys></code> in the <code><key-value list></code> is preserved. Thus

New: 2020-12-19
Updated: 2021-05-10

```
\keyval_parse:nnn
  { \use_none:nn { code 1 } }
  { \use_none:nnn { code 2 } }
  { key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\use_none:nnn { code 2 } { key1 } { value1 }
\use_none:nnn { code 2 } { key2 } { value2 }
\use_none:nnn { code 2 } { key3 } { }
\use_none:nn { code 1 } { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the `<key>` and `<value>`, then one *outer* set of braces is removed from the `<key>` and `<value>` as part of the processing. If you need exactly the output shown above, you'll need to either `e`-type or `x`-type expand the function.

TeXhackers note: The result of each list element is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an `e`-type or `x`-type argument expansion.

<code>\keyval_parse:NNn</code>	☆	<code>\keyval_parse:NNn <function₁> <function₂> {<key-value list>}</code>
<code>\keyval_parse:(NNV NNv)</code>	☆	Parses the <code><key-value list></code> into a series of <code><keys></code> and associated <code><values></code> , or keys alone (if no <code><value></code> was given). <code><function₁></code> should take one argument, while <code><function₂></code> should absorb two arguments. After <code>\keyval_parse:NNn</code> has parsed the <code><key-value list></code> , <code><function₁></code> is used to process keys given with no value and <code><function₂></code> is used to process keys given with a value. The order of the <code><keys></code> in the <code><key-value list></code> is preserved. Thus

Updated: 2021-05-10

```
\keyval_parse:NNn \function:n \function:nn
  { key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the `<key>` and `<value>`, then one *outer* set of braces is removed from the `<key>` and `<value>` as part of the processing.

This shares the implementation of `\keyval_parse:nnn`, the difference is only semantically.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an **e**-type or **x**-type argument expansion.

Chapter 28

The `\intarray` module

Fast global integer arrays

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast `\l3seq`, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialisation
- The absolute value of each entry has maximum $2^{30} - 1$ (*i.e.* one power lower than the usual `\c_max_int` ceiling of $2^{31} - 1$)

The use of `\intarray` data is therefore recommended for cases where the need for fast access is of paramount importance.

28.1 Creating and initialising integer array variables

```
\intarray_new:Nn \intarray_new:Nn <intarray var> {<size>}
```

```
\intarray_new:cn
```

New: 2018-03-29

Evaluates the integer expression `<size>` and allocates an `<integer array variable>` with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

```
\intarray_const_from_clist:Nn \intarray_const_from_clist:Nn <intarray var> <int expr list>
```

```
\intarray_const_from_clist:cn
```

New: 2018-05-04

Creates a new constant `<integer array variable>` or raises an error if the name is already taken. The `<integer array variable>` is set (globally) to contain as its items the results of evaluating each `<integer expression>` in the `<comma list>`.

```
\intarray_gzero:N \intarray_gzero:N <intarray var>
```

```
\intarray_gzero:c
```

New: 2018-05-04

Sets all entries of the `<integer array variable>` to zero. Assignments are always global.

28.2 Adding data to integer arrays

`\intarray_gset:Nnn` `\intarray_gset:Nnn` \langle *intarray var* \rangle $\{$ \langle *position* \rangle $\}$ $\{$ \langle *value* \rangle $\}$

`\intarray_gset:cnn`

New: 2018-03-29

Stores the result of evaluating the integer expression \langle *value* \rangle into the \langle *integer array variable* \rangle at the (integer expression) \langle *position* \rangle . If the \langle *position* \rangle is not between 1 and the `\intarray_count:N`, or the \langle *value* \rangle 's absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.

28.3 Counting entries in integer arrays

`\intarray_count:N` \star `\intarray_count:N` \langle *intarray var* \rangle

`\intarray_count:c` \star

New: 2018-03-29

Expands to the number of entries in the \langle *integer array variable* \rangle . Contrarily to `\seq_count:N` this is performed in constant time.

28.4 Using a single entry

`\intarray_item:Nn` \star `\intarray_item:Nn` \langle *intarray var* \rangle $\{$ \langle *position* \rangle $\}$

`\intarray_item:cn` \star

New: 2018-03-29

Expands to the integer entry stored at the (integer expression) \langle *position* \rangle in the \langle *integer array variable* \rangle . If the \langle *position* \rangle is not between 1 and the `\intarray_count:N`, an error occurs.

`\intarray_rand_item:N` \star `\intarray_rand_item:N` \langle *intarray var* \rangle

`\intarray_rand_item:c` \star

New: 2018-05-05

Selects a pseudo-random item of the \langle *integer array* \rangle . If the \langle *integer array* \rangle is empty, produce an error.

28.5 Integer array conditional

`\intarray_if_exist_p:N` \star `\intarray_if_exist_p:N` \langle *intarray var* \rangle

`\intarray_if_exist_p:c` \star `\intarray_if_exist:NTF` \langle *intarray var* \rangle $\{$ \langle *true code* \rangle $\}$ $\{$ \langle *false code* \rangle $\}$

`\intarray_if_exist:NTF` \star

`\intarray_if_exist:cTF` \star

New: 2024-03-31

Tests whether the \langle *intarray var* \rangle is currently defined. This does not check that the \langle *intarray var* \rangle really is an integer array variable.

28.6 Viewing integer arrays

`\intarray_show:N` `\intarray_show:N` \langle *intarray var* \rangle

`\intarray_show:c` `\intarray_log:N` \langle *intarray var* \rangle

`\intarray_log:N`

`\intarray_log:c`

New: 2018-05-04

Displays the items in the \langle *integer array variable* \rangle in the terminal or writes them in the log file.

28.7 Implementation notes

It is a wrapper around the `\fontdimen` primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to `l3seq` sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive `\fontdimen` stores dimensions but the `l3intarray` module transparently converts these from/to integers. Assignments are always global.

While LuaTeX's memory is extensible, other engines can “only” deal with a bit less than 4×10^6 entries in all `\fontdimen` arrays combined (with default TeX Live settings).

Chapter 29

The `l3fp` module

Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. *Floating point expressions* (“`<fp expr>`”) support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x / y , square root \sqrt{x} , and parentheses.
 - Comparison operators: $x < y$, $x \leq y$, $x >? y$, $x != y$ etc.
 - Boolean logic: sign `sign x` , negation `! x` , conjunction `x && y` , disjunction `x || y` , ternary operator `x ? y : z` .
 - Exponentials: `exp x` , `ln x` , `x y` , `logb x` .
 - Integer factorial: `fact x` .
 - Trigonometry: `sin x` , `cos x` , `tan x` , `cot x` , `sec x` , `csc x` expecting their arguments in radians, and `sind x` , `cosd x` , `tand x` , `cotd x` , `secd x` , `cscd x` expecting their arguments in degrees.
 - Inverse trigonometric functions: `asin x` , `acos x` , `atan x` , `acot x` , `asec x` , `acsc x` giving a result in radians, and `asind x` , `acosd x` , `atand x` , `acotd x` , `asecd x` , `acscd x` giving a result in degrees.
- (*not yet*) Hyperbolic functions and their inverse functions: `sinh x` , `cosh x` , `tanh x` , `coth x` , `sech x` , `csch`, and `asinh x` , `acosh x` , `atanh x` , `acoth x` , `asech x` , `acsch x` .
- Extrema: `max(x_1, x_2, \dots)`, `min(x_1, x_2, \dots)`, `abs(x)`.
 - Rounding functions, controlled by two optional values, n (number of places, 0 by default) and t (behavior on a tie, `nan` by default):
 - `trunc(x, n)` rounds towards zero,
 - `floor(x, n)` rounds towards $-\infty$,

- `ceil(x, n)` rounds towards $+\infty$,
- `round(x, n, t)` rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$.

And (*not yet*) modulo, and “quantize”.

- Random numbers: `rand()`, `randint(m, n)`.
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\langle type \rangle_use:N`) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples: (x_1, \dots, x_n) that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. A “floating point” is a floating point number or a tuple thereof. See section 29.12.1 for a description of what a floating point is, section 29.12.2 for details about how an expression is parsed, and section 29.12.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 29.10.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n {\sin(3.5)/2 + 2e-3} $.
```

The operation `round` can be used to limit the result’s precision. Adding `+0` avoids the possibly undesirable output `-0`, replacing it by `+0`. However, the `l3fp` module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnun } { m }
  { \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnun { 2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

See the documentation of `siunitx` for various options of `\num`.

29.1 Creating and initialising floating point variables

<code>\fp_new:N</code>	<code>\fp_new:N <fp var></code>
<code>\fp_new:c</code>	Creates a new <code><fp var></code> or raises an error if the name is already taken. The declaration is global. The <code><fp var></code> is initially +0.

Updated: 2012-05-08

<code>\fp_const:Nn</code>	<code>\fp_const:Nn <fp var> {<fp expr>}</code>
<code>\fp_const:cn</code>	Creates a new constant <code><fp var></code> or raises an error if the name is already taken. The <code><fp var></code> is set globally equal to the result of evaluating the <code><fp expr></code> .

Updated: 2012-05-08

<code>\fp_zero:N</code>	<code>\fp_zero:N <fp var></code>
<code>\fp_zero:c</code>	Sets the <code><fp var></code> to +0.
<code>\fp_gzero:N</code>	
<code>\fp_gzero:c</code>	

Updated: 2012-05-08

<code>\fp_zero_new:N</code>	<code>\fp_zero_new:N <fp var></code>
<code>\fp_zero_new:c</code>	Ensures that the <code><fp var></code> exists globally by applying <code>\fp_new:N</code> if necessary, then applies <code>\fp_(g)zero:N</code> to leave the <code><fp var></code> set to +0.
<code>\fp_gzero_new:N</code>	
<code>\fp_gzero_new:c</code>	

Updated: 2012-05-08

29.2 Setting floating point variables

<code>\fp_set:Nn</code>	<code>\fp_set:Nn <fp var> {<fp expr>}</code>
<code>\fp_set:cn</code>	Sets <code><fp var></code> equal to the result of computing the <code><fp expr></code> .
<code>\fp_gset:Nn</code>	
<code>\fp_gset:cn</code>	

Updated: 2012-05-08

<code>\fp_set_eq:NN</code>	<code>\fp_set_eq:NN <fp var₁₂</code>
<code>\fp_set_eq:(cN Nc cc)</code>	Sets the floating point variable <code><fp var_{1 equal to the current value of <code><fp var_{2.}</code>}</code>
<code>\fp_gset_eq:NN</code>	
<code>\fp_gset_eq:(cN Nc cc)</code>	

Updated: 2012-05-08

<code>\fp_add:Nn</code>	<code>\fp_add:Nn <fp var> {<fp expr>}</code>
<code>\fp_add:cn</code>	Adds the result of computing the <code><fp expr></code> to the <code><fp var></code> . This also applies if <code><fp var></code> and <code><floating point expression></code> evaluate to tuples of the same size.
<code>\fp_gadd:Nn</code>	
<code>\fp_gadd:cn</code>	

Updated: 2012-05-08

<code>\fp_sub:Nn</code>	<code>\fp_sub:Nn <fp var> {<fp expr>}</code>
<code>\fp_sub:cn</code>	Subtracts the result of computing the <i><floating point expression></i> from the <i><fp var></i> .
<code>\fp_gsub:Nn</code>	This also applies if <i><fp var></i> and <i><floating point expression></i> evaluate to tuples of
<code>\fp_gsub:cn</code>	the same size.

Updated: 2012-05-08

29.3 Using floating points

<code>\fp_eval:n</code>	<code>\fp_eval:n {<fp expr>}</code>
New: 2012-05-08	Evaluates the <i><fp expr></i> and expresses the result as a decimal number with no exponent.
Updated: 2012-07-08	Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and <code>nan</code> trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_eval:n</code> and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to <code>\fp_to_decimal:n</code> .

<code>\fp_sign:n</code>	<code>\fp_sign:n {<fp expr>}</code>
New: 2018-11-03	Evaluates the <i><fp expr></i> and leaves its sign in the input stream using <code>\fp_eval:n</code> $\{\text{sign}(\langle result \rangle)\}$: +1 for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 . If the operand is a tuple or is <code>nan</code> , then “invalid operation” occurs and the result is 0.

<code>\fp_to_decimal:N</code>	<code>\fp_to_decimal:N <fp var></code>
<code>\fp_to_decimal:c</code>	<code>\fp_to_decimal:n {<fp expr>}</code>
<code>\fp_to_decimal:n</code>	Evaluates the <i><fp expr></i> and expresses the result as a decimal number with no exponent.
New: 2012-05-08	Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and <code>nan</code> trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_to_decimal:n</code> and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.
Updated: 2012-07-08	

<code>\fp_to_dim:N</code>	<code>\fp_to_dim:N <fp var></code>
<code>\fp_to_dim:c</code>	<code>\fp_to_dim:n {<fp expr>}</code>
<code>\fp_to_dim:n</code>	Evaluates the <i><fp expr></i> and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to <code>\fp_to_decimal:n</code> , with an additional trailing <code>pt</code> (both letter tokens). In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid TeX dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values $\pm\infty$ and <code>nan</code> , trigger an “invalid operation” exception.
Updated: 2016-03-22	

<code>\fp_to_int:N</code>	<code>\fp_to_int:N <fp var></code>
<code>\fp_to_int:c</code>	<code>\fp_to_int:n {<fp expr>}</code>
<code>\fp_to_int:n</code>	Evaluates the <i><fp expr></i> , and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid TeX integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values $\pm\infty$ and <code>nan</code> , trigger an “invalid operation” exception.
Updated: 2012-07-08	

`\fp_to_scientific:N` * `\fp_to_scientific:N <fp var>`
`\fp_to_scientific:c` * `\fp_to_scientific:n {<fp expr>}`
`\fp_to_scientific:n` * Evaluates the `<fp expr>` and expresses the result in scientific notation:
New: 2012-05-08
Updated: 2016-03-22 `<optional -><digit>.<15 digits>e<optional sign><exponent>`

The leading `<digit>` is non-zero except in the case of ± 0 . The values $\pm\infty$ and `nan` trigger an “invalid operation” exception. Normal category codes apply: thus the `e` is category code 11 (a letter). For a tuple, each item is converted using `\fp_to_scientific:n` and they are combined as `(<fp1>, <fp2>, …, <fpn>)` if $n > 1$ and `(<fp1>,)` or `()` for fewer items.

`\fp_to_tl:N` * `\fp_to_tl:N <fp var>`
`\fp_to_tl:c` * `\fp_to_tl:n {<fp expr>}`
`\fp_to_tl:n` * Evaluates the `<fp expr>` and expresses the result in (almost) the shortest possible form.
Updated: 2016-03-22 Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from `\fp_to_scientific:n`). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see `\fp_to_decimal:n`. Negative numbers start with `-`. The special values ± 0 , $\pm\infty$ and `nan` are rendered as `0`, `-0`, `inf`, `-inf`, and `nan` respectively. Normal category codes apply and thus `inf` or `nan`, if produced, are made up of letters. For a tuple, each item is converted using `\fp_to_tl:n` and they are combined as `(<fp1>, <fp2>, …, <fpn>)` if $n > 1$ and `(<fp1>,)` or `()` for fewer items.

`\fp_use:N` * `\fp_use:N <fp var>`
`\fp_use:c` * Inserts the value of the `<fp var>` into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and `nan` trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_to_decimal:n` and they are combined as `(<fp1>, <fp2>, …, <fpn>)` if $n > 1$ and `(<fp1>,)` or `()` for fewer items. This function is identical to `\fp_to_decimal:N`.
Updated: 2012-07-08

29.4 Floating point conditionals

`\fp_if_exist_p:N` * `\fp_if_exist_p:N <fp var>`
`\fp_if_exist_p:c` * `\fp_if_exist:NTF <fp var> {<true code>} {<false code>}`
`\fp_if_exist:NTF` * Tests whether the `<fp var>` is currently defined. This does not check that the `<fp var>`
`\fp_if_exist:cTF` * really is a floating point variable.
Updated: 2012-05-08

```

\fp_compare_p:nNn * \fp_compare_p:nNn {<fp expr1>} <relation> {<fp expr2>}
\fp_compare:nNnTF * \fp_compare:nNnTF {<fp expr1>} <relation> {<fp expr2>} {<true code>} {<false code>}

```

Updated: 2012-05-08 Compares the $\langle fp\ expr_1 \rangle$ and the $\langle fp\ expr_2 \rangle$, and returns **true** if the $\langle relation \rangle$ is obeyed. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is **nan** or is a tuple, unless they are equal tuples. Note that a **nan** is distinct from any value, even another **nan**, hence $x = x$ is not true for a **nan**. To test if a value is **nan**, compare it to an arbitrary number with the “not ordered” relation.

```

\fp_compare:nNnTF { <value> } ? { 0 }
{ } % <value> is nan
{ } % <value> is not nan

```

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no **nan**). At present any other comparison with tuples yields ? (not ordered). This is experimental.

This function is less flexible than `\fp_compare:nTF` but slightly faster. It is provided for consistency with `\int_compare:nNnTF` and `\dim_compare:nNnTF`.

```

\fp_compare_p:n * \fp_compare_p:n
\fp_compare:nTF * {
Updated: 2013-12-14
  \langle fp expr_1 \rangle \langle relation_1 \rangle
  ...
  \langle fp expr_N \rangle \langle relation_N \rangle
  \langle fp expr_{N+1} \rangle
}
\fp_compare:nTF
{
  \langle fp expr_1 \rangle \langle relation_1 \rangle
  ...
  \langle fp expr_N \rangle \langle relation_N \rangle
  \langle fp expr_{N+1} \rangle
}
{\langle true code \rangle} {\langle false code \rangle}

```

Evaluates the $\langle fp\ exprs \rangle$ as described for $\backslash fp_eval:n$ and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle fp\ expr_1 \rangle$ and $\langle fp\ expr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle fp\ expr_2 \rangle$ and $\langle fp\ expr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle fp\ expr_N \rangle$ and $\langle fp\ expr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields true if all comparisons are true. Each $\langle floating\ point\ expression \rangle$ is evaluated only once. Contrarily to $\backslash int_compare:nTF$, all $\langle fp\ exprs \rangle$ are computed, even if one comparison is false. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is nan or is a tuple, unless they are equal tuples. Each $\langle relation \rangle$ can be any (non-empty) combination of $<$, $=$, $>$, and $?$, plus an optional leading $!$ (which negates the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with $?$, as this symbol has a different meaning (in combination with $:$) within floating point expressions. The comparison $x \langle relation \rangle y$ is then true if the $\langle relation \rangle$ does not start with $!$ and the actual relation ($<$, $=$, $>$, or $?$) between x and y appears within the $\langle relation \rangle$, or on the contrary if the $\langle relation \rangle$ starts with $!$ and the relation between x and y does not appear within the $\langle relation \rangle$. Common choices of $\langle relation \rangle$ include \geq (greater or equal), \neq (not equal), $!?$ or $\leq\Rightarrow$ (comparable).

This function is more flexible than $\backslash fp_compare:nNnTF$ and only slightly slower.

```

\fp_if_nan_p:n * \fp_if_nan_p:n {\langle fp expr \rangle}
\fp_if_nan:nTF * \fp_if_nan:nTF {\langle fp expr \rangle} {\langle true code \rangle} {\langle false code \rangle}

```

New: 2019-08-25 Evaluates the $\langle fp\ expr \rangle$ and tests whether the result is exactly nan. The test returns false for any other result, even a tuple containing nan.

29.5 Floating point expression loops

```

\fp_do_until:nNnn ☆ \fp_do_until:nNnn {\langle fp expr_1 \rangle} \langle relation \rangle {\langle fp expr_2 \rangle} {\langle code \rangle}

```

New: 2012-08-16 Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for $\backslash fp_compare:nNnTF$. If the test is false then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is true.

<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {<fp expr₁>} <relation> {<fp expr₂>} {<code>}</code>
New: 2012-08-16	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> . If the test is <code>true</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <i><relation></i> is <code>false</code> .
<code>\fp_until_do:nNnn</code> ☆	<code>\fp_until_do:nNnn {<fp expr₁>} <relation> {<fp expr₂>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <i><relation></i> is <code>false</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\fp_while_do:nNnn</code> ☆	<code>\fp_while_do:nNnn {<fp expr₁>} <relation> {<fp expr₂>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <i><relation></i> is <code>true</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>false</code> .
<code>\fp_do_until:nn</code> ☆	<code>\fp_do_until:nn { <fp expr₁> <relation> <fp expr₂> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is <code>false</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <i><relation></i> is <code>true</code> .
<code>\fp_do_while:nn</code> ☆	<code>\fp_do_while:nn { <fp expr₁> <relation> <fp expr₂> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is <code>true</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <i><relation></i> is <code>false</code> .
<code>\fp_until_do:nn</code> ☆	<code>\fp_until_do:nn { <fp expr₁> <relation> <fp expr₂> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <i><relation></i> is <code>false</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\fp_while_do:nn</code> ☆	<code>\fp_while_do:nn { <fp expr₁> <relation> <fp expr₂> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <i><relation></i> is <code>true</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>false</code> .

`\fp_step_function:nnnN` ☆ `\fp_step_function:nnnN` {*initial value*} {*step*} {*final value*} (*function*)

`\fp_step_function:nnnc` ☆

New: 2016-11-21
Updated: 2016-12-06

This function first evaluates the *initial value*, *step* and *final value*, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The *function* is then placed in front of each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*). The *step* must be non-zero. If the *step* is positive, the loop stops when the *value* becomes larger than the *final value*. If the *step* is negative, the loop stops when the *value* becomes smaller than the *final value*. The *function* should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

```
[I saw 1.0] [I saw 1.1] [I saw 1.2] [I saw 1.3] [I saw 1.4] [I saw 1.5]
```

TpXhackers note: Due to rounding, it may happen that adding the *step* to the *value* does not change the *value*; such cases give an error, as they would otherwise lead to an infinite loop.

`\fp_step_inline:nnnn` `\fp_step_inline:nnnn` {*initial value*} {*step*} {*final value*} {*code*}

New: 2016-11-21
Updated: 2016-12-06

This function first evaluates the *initial value*, *step* and *final value*, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*), the *code* is inserted into the input stream with *#1* replaced by the current *value*. Thus the *code* should define a function of one argument (*#1*).

`\fp_step_variable:nnnNn` `\fp_step_variable:nnnNn`
{*initial value*} {*step*} {*final value*} *tl var* {*code*}

New: 2017-04-12

This function first evaluates the *initial value*, *step* and *final value*, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*), the *code* is inserted into the input stream, with the *tl var* defined as the current *value*. Thus the *code* should make use of the *tl var*.

29.6 Symbolic expressions

Floating point expressions support variables: these can only be set locally, so act like standard `\l...` variables.

```
\fp_new_variable:n { A }
\fp_set:Nn \l_tmpb_fp { 1 * sin(A) + 3**2 }
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
\fp_set_variable:nn { A } { pi/2 }
```

```

\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
\fp_set_variable:nn { A } { 0 }
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp

```

defines A to be a variable, then defines `\l_tmpb_fp` to stand for $1*\sin(A)+9$ (note that $3**2$ is evaluated, but the $1*$ product is not simplified away). Until `\l_tmpb_fp` is changed, `\fp_show:N \l_tmpb_fp` will show $((1*\sin(A))+9)$ regardless of the value of A. The next step defines A to be equal to $\pi/2$: then `\fp_show:n { \l_tmpb_fp }` will evaluate `\l_tmpb_fp` and show 10. We then redefine A to be 0: since `\l_tmpb_fp` still stands for $1*\sin(A)+9$, the value shown is then 9. Variables can be set with `\fp_set_variable:nn` to arbitrary floating point expressions including other variables.

```
\fp_new_variable:n \fp_new_variable:n {<identifier>}
```

New: 2023-10-19

Declares the `<identifier>` as a variable, which allows it to be used in floating point expressions. For instance,

```

\fp_new_variable:n { A }
\fp_show:n { A**2 - A + 1 }

```

shows $((A^2)-A)+1$. If the declaration was missing, the parser would complain about an “Unknown fp word ‘A’”. The `<identifier>` must consist entirely of Latin letters among [a-zA-Z].

```
\fp_set_variable:nn \fp_set_variable:nn {<identifier>} {<fp expr>}
```

New: 2023-10-19

Defines the `<identifier>` to stand in any further expression for the result of evaluating the `<floating point expression>` as much as possible. The result may contain other variables, which are then replaced by their values if they have any. For instance,

```

\fp_new_variable:n { A }
\fp_new_variable:n { B }
\fp_new_variable:n { C }
\fp_set_variable:nn { A } { 3 }
\fp_set_variable:nn { C } { A ** 2 + B * 1 }
\fp_show:n { C + 4 }
\fp_set_variable:nn { A } { 4 }
\fp_show:n { C + 4 }

```

shows $((9+(B*1))+4)$ twice: changing the value of A to 4 does not alter C because A was replaced by its value 3 when evaluating $A**2+B*1$.

`\fp_clear_variable:n` `\fp_clear_variable:n {⟨identifier⟩}`
New: 2023-10-19 Removes any value given by `\fp_set_variable:nn` to the variable with this `⟨identifier⟩`.
 For instance,

```
\fp_new_variable:n { A }
\fp_set_variable:nn { A } { 3 }
\fp_show:n { A ^ 2 }
\fp_clear_variable:n { A }
\fp_show:n { A ^ 2 }
```

shows 9, then (A^2) .

29.7 User-defined functions

It is possible to define new user functions which can be used inside the argument to `\fp_eval:n`, etc. These functions may take one or more named arguments, and should be implemented using expansion methods only.

`\fp_new_function:n` `\fp_new_function:n {⟨identifier⟩}`
New: 2023-10-19 Declares the `⟨identifier⟩` as a function, which allows it to be used in floating point expressions. For instance,

```
\fp_new_function:n { foo }
\fp_show:n { foo ( 1 + 2 , foo(3), A ) ** 2 } }
```

shows $(\text{foo}(3, \text{foo}(3), A))^2$. If the declaration was missing, the parser would complain about an “Unknown fp word ‘foo’”. The `⟨identifier⟩` must consist entirely of Latin letters [a-zA-Z].

`\fp_set_function:nnn` `\fp_set_function:nnn {⟨identifier⟩} {⟨vars⟩} {⟨fp expr⟩}`
New: 2023-10-19 Defines the `⟨identifier⟩` to stand in any further expression for the result of evaluating the `⟨floating point expression⟩`, with the `⟨identifier⟩` accepting the `⟨vars⟩` (a non-empty comma-separated list). The result may contain other functions, which are then replaced by their results if they have any. For instance,

```
\fp_new_function:n { foo }
\fp_set_function:nnn { npow } { a,b } { a**b }
\fp_show:n { npow(16,0.25) } }
```

shows 2. The names of the `⟨vars⟩` must consist entirely of Latin letters [a-zA-Z], but are otherwise not restricted: in particular, they are independent of any variables declared by `\fp_new_variable:n`.

`\fp_clear_function:n` `\fp_clear_function:n {⟨identifier⟩}`
New: 2023-10-19 Removes any definition given by `\fp_set_function:nnn` to the function with this `⟨identifier⟩`.

29.8 Some useful constants, and scratch variables

`\c_zero_fp` Zero, with either sign.
`\c_minus_zero_fp`

New: 2012-05-08

`\c_one_fp` One as an `fp`: useful for comparisons in some places.

New: 2012-05-08

`\c_inf_fp` Infinity, with either sign. These can be input directly in a floating point expression as
`\c_minus_inf_fp` `inf` and `-inf`.

New: 2012-05-08

`\c_nan_fp` Not a number. This can be input directly in a floating point expression as `nan`.

New: 2012-05-08

`\c_e_fp` The value of the base of the natural logarithm, $e = \exp(1)$.

Updated: 2012-05-08

`\c_pi_fp` The value of π . This can be input directly in a floating point expression as `pi`.

Updated: 2013-11-17

`\c_one_degree_fp` The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as `deg`.

New: 2012-05-08

Updated: 2013-11-17

29.9 Scratch variables

`\l_tmpa_fp` Scratch floating points for local assignment. These are never used by the kernel code, and
`\l_tmpb_fp` so are safe for use with any \LaTeX 3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_fp` Scratch floating points for global assignment. These are never used by the kernel code,
`\g_tmpb_fp` and so are safe for use with any \LaTeX 3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

29.10 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as $0 / 0$, or $10 ** 1e9999$. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0 .
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$ or $\sin(\infty)$, and results in a `nan`. It also occurs for conversion functions whose target type does not have the appropriate infinite or `nan` value (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, e.g., $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.

(*not yet*) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in L^AT_EX3.

To each exception we associate a “flag”: `\l_fp_overflow_flag`, `\l_fp_underflow_flag`, `\l_fp_invalid_operation_flag` and `\l_fp_division_by_zero_flag`. The state of these flags can be tested and modified with commands from `l3flag`

By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

`\fp_trap:nn` `\fp_trap:nn <exception> <trap type>`

New: 2012-07-19 All occurrences of the `<exception>` (`overflow`, `underflow`, `invalid_operation` or
Updated: 2017-02-13 `division_by_zero`) within the current group are treated as `<trap type>`, which can be

- **none**: the `<exception>` will be entirely ignored, and leave no trace;
- **flag**: the `<exception>` will turn the corresponding flag on when it occurs;
- **error**: additionally, the `<exception>` will halt the T_EX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

`\l_fp_overflow_flag`
`\l_fp_underflow_flag`
`\l_fp_invalid_operation_flag`
`\l_fp_division_by_zero_flag`

Flags denoting the occurrence of various floating-point exceptions.

29.11 Viewing floating points

<code>\fp_show:N</code>	<code>\fp_show:N <fp var></code>
<code>\fp_show:c</code>	<code>\fp_show:n {<fp expr>}</code>
<code>\fp_show:n</code>	Evaluates the <code><fp expr></code> and displays the result in the terminal.

New: 2012-05-08

Updated: 2021-04-29

<code>\fp_log:N</code>	<code>\fp_log:N <fp var></code>
<code>\fp_log:c</code>	<code>\fp_log:n {<fp expr>}</code>
<code>\fp_log:n</code>	Evaluates the <code><fp expr></code> and writes the result in the log file.

New: 2014-08-22

Updated: 2021-04-29

29.12 Floating point expressions

29.12.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;
- ± 0 , zero, with a given sign;
- $\pm \infty$, infinity, with a given sign;
- `nan`, is “not a number”, and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- `<sign>`: a possibly empty string of + and - characters;
- `<significand>`: a non-empty string of digits together with zero or one dot;
- `<exponent>` optionally: the character `e` or `E`, followed by a possibly empty string of + and - tokens, and a non-empty string of digits.

The sign of the resulting number is + if `<sign>` contains an even number of -, and - otherwise, hence, an empty `<sign>` denotes a non-negative input. The stored significand is obtained from `<significand>` by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input `<significand>` has at most 16 digits. The stored `<exponent>` is obtained by combining the input `<exponent>` (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting `<exponent>` is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm \infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if $\langle \textit{significand} \rangle$ contains no non-zero digit (*i.e.*, consists only in characters 0, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to $+0$, but that is not guaranteed to remain true.

The $\langle \textit{significand} \rangle$ must be non-empty, so `e1` and `e-1` are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as `e` and should be input as `exp(1)` or `\c_e_fp` (which is faster).

Special numbers are input as follows:

- `inf` represents $+\infty$, and can be preceded by any $\langle \textit{sign} \rangle$, yielding $\pm\infty$ as appropriate.
- `nan` represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a `nan`.
- Note that commands such as `\infty`, `\pi`, or `\sin` *do not* work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

29.12.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc.*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Implicit multiplication by juxtaposition (`2pi`) when neither factor is in parentheses.
- Binary `*` and `/`, implicit multiplication by juxtaposition with parentheses (for instance `3(4+5)`).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc.*
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

$$\begin{aligned} 1/2\text{pi} &= 1/(2\pi), \\ 1/2\text{pi}(\text{pi} + \text{pi}) &= (2\pi)^{-1}(\pi + \pi) \simeq 1, \\ \text{sin}2\text{pi} &= \sin(2)\pi \neq 0, \\ 2^2\text{max}(3, 5) &= 2^2 \max(3, 5) = 20, \\ 1\text{in}/1\text{cm} &= (1\text{in})/(1\text{cm}) = 2.54. \end{aligned}$$

Functions are called on the value of their argument, contrarily to $\text{T}_{\text{E}}\text{X}$ macros.

29.12.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is **false** if it is ± 0 , and **true** otherwise, including when it is **nan** or a tuple such as $(0, 0)$. Tuples are only supported to some extent by operations that work with truth values ($?:$, $||$, $\&\&$, $!$), by comparisons ($!<=>?$), and by $+$, $-$, $*$, $/$. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a **nan** result.

```
?: \fp_eval:n { <operand1> ? <operand2> : <operand3> }
```

The ternary operator $?:$ results in $\langle\text{operand}_2\rangle$ if $\langle\text{operand}_1\rangle$ is true (not ± 0), and $\langle\text{operand}_3\rangle$ if $\langle\text{operand}_1\rangle$ is false (± 0). All three $\langle\text{operands}\rangle$ are evaluated in all cases; they may be tuples. The operator is right associative, hence

```
\fp_eval:n
{
  1 + 3 > 4 ? 1 :
  2 + 4 > 5 ? 2 :
  3 + 5 > 6 ? 3 : 4
}
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following $:$ is taken, and $2 + 4 > 5$ is compared; since this is true, the branch before $:$ is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

```
|| \fp_eval:n { <operand1> || <operand2> }
```

If $\langle\text{operand}_1\rangle$ is true (not ± 0), use that value, otherwise the value of $\langle\text{operand}_2\rangle$. Both $\langle\text{operands}\rangle$ are evaluated in all cases; they may be tuples. In $\langle\text{operand}_1\rangle || \langle\text{operand}_2\rangle || \dots || \langle\text{operands}_n\rangle$, the first true (nonzero) $\langle\text{operand}\rangle$ is used and if all are zero the last one (± 0) is used.

```
&& \fp_eval:n { <operand1> && <operand2> }
```

If $\langle\text{operand}_1\rangle$ is false (equal to ± 0), use that value, otherwise the value of $\langle\text{operand}_2\rangle$. Both $\langle\text{operands}\rangle$ are evaluated in all cases; they may be tuples. In $\langle\text{operand}_1\rangle \&\& \langle\text{operand}_2\rangle \&\& \dots \&\& \langle\text{operands}_n\rangle$, the first false (± 0) $\langle\text{operand}\rangle$ is used and if none is zero the last one is used.

```

<      \fp_eval:n
=      {
>      <operand1> <relation1>
?      ...
Updated: 2013-12-14  <operandN> <relationN>
                    <operandN+1>
}

```

Each $\langle \text{relation} \rangle$ consists of a non-empty string of $<$, $=$, $>$, and $?$, optionally preceded by $!$, and may not start with $?$. This evaluates to $+1$ if all comparisons $\langle \text{operand}_i \rangle \langle \text{relation}_i \rangle \langle \text{operand}_{i+1} \rangle$ are true, and $+0$ otherwise. All $\langle \text{operands} \rangle$ are evaluated (once) in all cases. See `\fp_compare:nTF` for details.

```

-
+ \fp_eval:n { <operand1> + <operand2> }
- \fp_eval:n { <operand1> - <operand2> }
-

```

Computes the sum or the difference of its two $\langle \text{operands} \rangle$. The “invalid operation” exception occurs for $\infty - \infty$. “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is `nan`.

```

-
* \fp_eval:n { <operand1> * <operand2> }
/ \fp_eval:n { <operand1> / <operand2> }
-

```

Computes the product or the ratio of its two $\langle \text{operands} \rangle$. The “invalid operation” exception occurs for ∞/∞ , $0/0$, or $0 * \infty$. “Division by zero” occurs when dividing a finite non-zero number by ± 0 . “Underflow” and “overflow” occur when appropriate. When $\langle \text{operand}_1 \rangle$ is a tuple and $\langle \text{operand}_2 \rangle$ is a floating point number, each item of $\langle \text{operand}_1 \rangle$ is multiplied or divided by $\langle \text{operand}_2 \rangle$. Multiplication also supports the case where $\langle \text{operand}_1 \rangle$ is a floating point number and $\langle \text{operand}_2 \rangle$ a tuple. Other combinations yield an “invalid operation” exception and a `nan` result.

```

-
+ \fp_eval:n { + <operand> }
- \fp_eval:n { - <operand> }
! \fp_eval:n { ! <operand> }
-

```

The unary $+$ does nothing, the unary $-$ changes the sign of the $\langle \text{operand} \rangle$ (for a tuple, of all its components), and $!$ $\langle \text{operand} \rangle$ evaluates to 1 if $\langle \text{operand} \rangle$ is false (is ± 0) and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

```

-
** \fp_eval:n { <operand1> ** <operand2> }
^ \fp_eval:n { <operand1> ^ <operand2> }
-

```

Raises $\langle \text{operand}_1 \rangle$ to the power $\langle \text{operand}_2 \rangle$. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. If $\langle \text{operand}_1 \rangle$ is negative or -0 then: the result’s sign is $+$ if the $\langle \text{operand}_2 \rangle$ is infinite and $(-1)^p$ if the $\langle \text{operand}_2 \rangle$ is $p/5^q$ with p, q integers; the result is $+0$ if $\text{abs}(\langle \text{operand}_1 \rangle)^{\langle \text{operand}_2 \rangle}$ evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising ± 0 to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.

```

abs \fp_eval:n { abs( <fp expr> ) }

```

Computes the absolute value of the $\langle \text{fp expr} \rangle$. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also `\fp_abs:n`.

exp \fp_eval:n { exp(*fp expr*) }

Computes the exponential of the *fp expr*. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

fact \fp_eval:n { fact(*fp expr*) }

Computes the factorial of the *fp expr*. If the *fp expr* is an integer between -0 and 3248 included, the result is finite and correctly rounded. Larger positive integers give $+\infty$ with “overflow”, while $\text{fact}(+\infty) = +\infty$ and $\text{fact}(\text{nan}) = \text{nan}$ with no exception. All other inputs give nan with the “invalid operation” exception.

ln \fp_eval:n { ln(*fp expr*) }

Computes the natural logarithm of the *fp expr*. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for $\ln(-0)$. “Division by zero” occurs when evaluating $\ln(+0) = -\infty$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

logb * \fp_eval:n { logb(*fp expr*) }

New: 2018-11-03 Determines the exponent of the *fp expr*, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating $\text{logb}(\pm 0) = -\infty$. Other special values are $\text{logb}(\pm\infty) = +\infty$ and $\text{logb}(\text{nan}) = \text{nan}$. If the operand is a tuple or is nan , then “invalid operation” occurs and the result is nan .

max \fp_eval:n { max(*fp expr*₁ , *fp expr*₂ , ...) }

min \fp_eval:n { min(*fp expr*₁ , *fp expr*₂ , ...) }

Evaluates each *fp expr* and computes the largest (smallest) of those. If any of the *fp expr* is a nan or tuple, the result is nan . If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.

<code>round</code>	<code>\fp_eval:n { round (<fp expr>) }</code>
<code>trunc</code>	<code>\fp_eval:n { round (<fp expr₁> , <fp expr₂>) }</code>
<code>ceil</code>	<code>\fp_eval:n { round (<fp expr₁> , <fp expr₂> , <fp expr₃>) }</code>
<code>floor</code>	Only <code>round</code> accepts a third argument. Evaluates $\langle fp\ expr_1 \rangle = x$ and $\langle fp\ expr_2 \rangle = n$ and $\langle fp\ expr_3 \rangle = t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n} ; if $n = +\infty$, this always yields x ; if $n = -\infty$, this yields one of ± 0 , $\pm\infty$, or <code>nan</code> ; if $n = \text{nan}$, this yields <code>nan</code> ; if n is neither $\pm\infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle fp\ expr_2 \rangle$ is omitted, $n = 0$, <i>i.e.</i> , $\langle fp\ expr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.

New: 2013-12-14
Updated: 2015-08-08

- `round` yields the multiple of 10^{-n} closest to x , with ties (x half-way between two such multiples) rounded as follows. If t is `nan` (or not given) the even multiple is chosen (“ties to even”), if $t = \pm 0$ the multiple closest to 0 is chosen (“ties to zero”), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen (“ties towards positive/negative infinity”).
- `floor` yields the largest multiple of 10^{-n} smaller or equal to x (“round towards negative infinity”);
- `ceil` yields the smallest multiple of 10^{-n} greater or equal to x (“round towards positive infinity”);
- `trunc` yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less than that of x (“round towards zero”).

“Overflow” occurs if x is finite and the result is infinite (this can only happen if $\langle fp\ expr_2 \rangle < -9984$). If any operand is a tuple, “invalid operation” occurs.

<code>sign</code>	<code>\fp_eval:n { sign(<fp expr>) }</code>
-------------------	---

Evaluates the $\langle fp\ expr \rangle$ and determines its sign: +1 for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 , and `nan` for `nan`. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

<code>sin</code>	<code>\fp_eval:n { sin(<fp expr>) }</code>
<code>cos</code>	<code>\fp_eval:n { cos(<fp expr>) }</code>
<code>tan</code>	<code>\fp_eval:n { tan(<fp expr>) }</code>
<code>cot</code>	<code>\fp_eval:n { cot(<fp expr>) }</code>
<code>csc</code>	<code>\fp_eval:n { csc(<fp expr>) }</code>
<code>sec</code>	<code>\fp_eval:n { sec(<fp expr>) }</code>

Updated: 2013-11-17
Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fp\ expr \rangle$ given in radians. For arguments given in degrees, see `sind`, `cosd`, *etc.* Note that since π is irrational, `sin(8pi)` is not quite zero, while its analogue `sind(8 × 180)` is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

sind      \fp_eval:n { sind( <fp expr> ) }
cosd     \fp_eval:n { cosd( <fp expr> ) }
tand     \fp_eval:n { tand( <fp expr> ) }
cotd     \fp_eval:n { cotd( <fp expr> ) }
cscd     \fp_eval:n { cscd( <fp expr> ) }
secd     \fp_eval:n { secd( <fp expr> ) }

```

New: 2013-11-02 Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fp\ expr \rangle$ given in degrees. For arguments given in radians, see `sin`, `cos`, *etc.* Note that since π is irrational, `sin(8pi)` is not quite zero, while its analogue `sind(8 × 180)` is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

asin     \fp_eval:n { asin( <fp expr> ) }
acos     \fp_eval:n { acos( <fp expr> ) }
acsc     \fp_eval:n { acsc( <fp expr> ) }
asec     \fp_eval:n { asec( <fp expr> ) }

```

New: 2013-11-02 Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fp\ expr \rangle$ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

asind    \fp_eval:n { asind( <fp expr> ) }
acosd    \fp_eval:n { acosd( <fp expr> ) }
acscd    \fp_eval:n { acscd( <fp expr> ) }
asecd    \fp_eval:n { asecd( <fp expr> ) }

```

New: 2013-11-02 Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fp\ expr \rangle$ and returns the result in degrees, in the range $[-90, 90]$ for `asind` and `acscd` and $[0, 180]$ for `acosd` and `asecd`. For a result in radians, use `asin`, *etc.* If the argument of `asind` or `acosd` lies outside the range $[-1, 1]$, or the argument of `acscd` or `asecd` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>atan</code>	<code>\fp_eval:n { atan(<fp expr>) }</code>
<code>acot</code>	<code>\fp_eval:n { atan(<fp expr₁> , <fp expr₂>) }</code>
<code>atan</code>	<code>\fp_eval:n { acot(<fp expr>) }</code>
<code>acot</code>	<code>\fp_eval:n { acot(<fp expr₁> , <fp expr₂>) }</code>

Those functions yield an angle in radians: `atand` and `acotd` are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the `<fp expr>`: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fp expr_2 \rangle, \langle fp expr_1 \rangle)$: this is the arctangent of $\langle fp expr_1 \rangle / \langle fp expr_2 \rangle$, possibly shifted by π depending on the signs of $\langle fp expr_1 \rangle$ and $\langle fp expr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fp expr_1 \rangle, \langle fp expr_2 \rangle)$, equal to the arccotangent of $\langle fp expr_1 \rangle / \langle fp expr_2 \rangle$, possibly shifted by π . Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fp expr_1 \rangle / \langle fp expr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

<code>atand</code>	<code>\fp_eval:n { atand(<fp expr>) }</code>
<code>acotd</code>	<code>\fp_eval:n { atand(<fp expr₁> , <fp expr₂>) }</code>
<code>atand</code>	<code>\fp_eval:n { acotd(<fp expr>) }</code>
<code>acotd</code>	<code>\fp_eval:n { acotd(<fp expr₁> , <fp expr₂>) }</code>

Those functions yield an angle in degrees: `atan` and `acot` are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the `<fp expr>`: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fp expr_2 \rangle, \langle fp expr_1 \rangle)$: this is the arctangent of $\langle fp expr_1 \rangle / \langle fp expr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fp expr_1 \rangle$ and $\langle fp expr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fp expr_1 \rangle, \langle fp expr_2 \rangle)$, equal to the arccotangent of $\langle fp expr_1 \rangle / \langle fp expr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fp expr_1 \rangle / \langle fp expr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

<code>sqrt</code>	<code>\fp_eval:n { sqrt(<fp expr>) }</code>
-------------------	---

`sqrt` Computes the square root of the `<fp expr>`. The “invalid operation” is raised when the `<fp expr>` is negative or is a tuple; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{nan}} = \text{nan}$.

rand `\fp_eval:n { rand() }`

New: 2016-12-05 Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. This is not available in older versions of X_ƎTeX. The random seed can be queried using `\sys_rand_seed:` and set using `\sys_gset_rand_seed:n`.

TeXhackers note: This is based on pseudo-random numbers provided by the engine’s primitive `\pdfuniformdeviate` in pdfTeX, pTeX, upTeX and `\uniformdeviate` in LuaTeX and X_ƎTeX. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of “The Art of Computer Programming, Volume 2”.

While we are more careful than `\uniformdeviate` to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

randint `\fp_eval:n { randint(<fp expr>) }`

New: 2016-12-05 `\fp_eval:n { randint(<fp expr1> , <fp expr2>) }`

Produces a pseudo-random integer between 1 and `<fp expr>` or between `<fp expr1>` and `<fp expr2>` inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second. See **rand** for important comments on how these pseudo-random numbers are generated.

inf The special values $+\infty$, $-\infty$, and **nan** are represented as `inf`, `-inf` and `nan` (see `\c_-nan_inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

pi The value of π (see `\c_pi_fp`).

deg The value of 1° in radians (see `\c_one_degree_fp`).

<hr/>	
<code>em</code>	Those units of measurement are equal to their values in pt, namely
<code>ex</code>	
<code>in</code>	$1 \text{ in} = 72.27 \text{ pt}$
<code>pt</code>	$1 \text{ pt} = 1 \text{ pt}$
<code>pc</code>	
<code>cm</code>	$1 \text{ pc} = 12 \text{ pt}$
<code>mm</code>	
<code>dd</code>	$1 \text{ cm} = \frac{1}{2.54} \text{ in} = 28.45275590551181 \text{ pt}$
<code>cc</code>	
<code>nd</code>	$1 \text{ mm} = \frac{1}{25.4} \text{ in} = 2.845275590551181 \text{ pt}$
<code>nc</code>	
<code>bp</code>	$1 \text{ dd} = 0.376065 \text{ mm} = 1.07000856496063 \text{ pt}$
<code>sp</code>	$1 \text{ cc} = 12 \text{ dd} = 12.84010277952756 \text{ pt}$
<hr/>	
	$1 \text{ nd} = 0.375 \text{ mm} = 1.066978346456693 \text{ pt}$
	$1 \text{ nc} = 12 \text{ nd} = 12.80374015748031 \text{ pt}$
	$1 \text{ bp} = \frac{1}{72} \text{ in} = 1.00375 \text{ pt}$
	$1 \text{ sp} = 2^{-16} \text{ pt} = 1.52587890625 \times 10^{-5} \text{ pt}.$

The values of the (font-dependent) units `em` and `ex` are gathered from \TeX when the surrounding floating point expression is evaluated.

<hr/>	
<code>true</code>	Other names for 1 and +0.
<code>false</code>	
<hr/>	

<hr/>	
<code>\fp_abs:n</code>	\star <code>\fp_abs:n</code> $\langle fp \text{ expr} \rangle$
<code>New: 2012-05-14</code>	Evaluates the $\langle fp \text{ expr} \rangle$ as described for <code>\fp_eval:n</code> and leaves the absolute value of
<code>Updated: 2012-07-08</code>	the result in the input stream. If the argument is $\pm\infty$, <code>nan</code> or a tuple, “invalid operation”
	occurs. Within floating point expressions, <code>abs()</code> can be used; it accepts $\pm\infty$ and <code>nan</code> as
	arguments.

<hr/>	
<code>\fp_max:nn</code>	\star <code>\fp_max:nn</code> $\langle fp \text{ expr}_1 \rangle$ $\langle fp \text{ expr}_2 \rangle$
<code>\fp_min:nn</code>	\star Evaluates the $\langle fp \text{ exprs} \rangle$ as described for <code>\fp_eval:n</code> and leaves the resulting larger
<code>New: 2012-09-26</code>	(<code>max</code>) or smaller (<code>min</code>) value in the input stream. If the argument is a tuple, “invalid
	operation” occurs, but no other case raises exceptions. Within floating point expressions,
	<code>max()</code> and <code>min()</code> can be used.

29.13 Disclaimer and roadmap

This module may break if the escape character is among `0123456789_+`, or if it receives a \TeX primitive conditional affected by `\exp_not:N`.

The following need to be done. I’ll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.

- Support signalling `nan`.
- Modulo and remainder, and rounding function `quantize` (and its friends analogous to `trunc`, `ceil`, `floor`).
- `\fp_format:nn` $\{\langle fp\ expr\rangle\}$ $\{\langle format\rangle\}$, but what should $\langle format\rangle$ be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add $\log(x, b)$ for logarithm of x in base b .
- `hypot` (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Factorial (not with `!`), gamma function.
- Improve coefficients of the `sin` and `tan` series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an `array(1,2,3)` and `i=complex(0,1)`.
- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?
- Provide an `isnan` function analogue of `\fp_if_nan:nTF`?
- Support keyword arguments?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n` $\{\text{Opt}\}$ should return -0 , not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
- `0e9999999999` gives a `TEX` “number too large” error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection [29.12.1](#), write a grammar.

- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in `l3fp-parse`.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t . However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (`l3fp-basics`).
- Understand and document `__fp_basics_pack_weird_low:NNNNw` and `__fp_basics_pack_weird_high:NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to `l3fp-aux` under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan's articles, some previous TeX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)?

Chapter 30

The `l3fparray` module

Fast global floating point arrays

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast `l3seq`, where access time is linear). The interface is very close to that of `l3intarray`. The size of the array is fixed and must be given at point of initialisation

30.1 Creating and initialising floating point array variables

<code>\fparray_new:Nn</code>	<code>\fparray_new:Nn <fparray var> {<size>}</code>
<code>\fparray_new:cn</code>	Evaluates the integer expression <code><size></code> and allocates an <code><floating point array variable></code> with that number of (zero) entries. The variable name should start with <code>\g_</code> because assignments are always global.
<small>New: 2018-05-05</small>	

<code>\fparray_gzero:N</code>	<code>\fparray_gzero:N <fparray var></code>
<code>\fparray_gzero:c</code>	Sets all entries of the <code><floating point array variable></code> to <code>+0</code> . Assignments are always global.
<small>New: 2018-05-05</small>	

30.2 Adding data to floating point arrays

<code>\fparray_gset:Nnn</code>	<code>\fparray_gset:Nnn <fparray var> {<position>} {<value>}</code>
<code>\fparray_gset:cn</code>	Stores the result of evaluating the floating point expression <code><value></code> into the <code><floating point array variable></code> at the (integer expression) <code><position></code> . If the <code><position></code> is not between 1 and the <code>\fparray_count:N</code> , an error occurs. Assignments are always global.
<small>New: 2018-05-05</small>	

30.3 Counting entries in floating point arrays

`\fparray_count:N` * `\fparray_count:N` $\langle farray var \rangle$
`\fparray_count:c` * Expands to the number of entries in the $\langle floating point array variable \rangle$. This is performed in constant time.
New: 2018-05-05

30.4 Using a single entry

`\fparray_item:Nn` * `\fparray_item:Nn` $\langle farray var \rangle$ $\{\langle position \rangle\}$
`\fparray_item:cn` * Applies `\fp_use:N` or `\fp_to_tl:N` (respectively) to the floating point entry stored at the (integer expression) $\langle position \rangle$ in the $\langle floating point array variable \rangle$. If the $\langle position \rangle$ is not between 1 and the `\fparray_count:N` $\langle farray var \rangle$, an error occurs.
New: 2018-05-05

30.5 Floating point array conditional

`\fparray_if_exist_p:N` * `\fparray_if_exist_p:N` $\langle farray var \rangle$
`\fparray_if_exist_p:c` * `\fparray_if_exist:NTF` $\langle farray var \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
`\fparray_if_exist:NTF` * Tests whether the $\langle farray var \rangle$ is currently defined. This does not check that the $\langle farray var \rangle$ really is a floating point array variable.
`\fparray_if_exist:cTF` *
New: 2024-03-31

Chapter 31

The `l3bitset` module

Bitsets

This module defines and implements the data type `bitset`, a vector of bits. The size of the vector may grow dynamically. Individual bits can be set and unset by names pointing to an index position. The names `1`, `2`, `3`, ... are predeclared and point to the index positions `1`, `2`, `3`, ... More names can be added and existing names can be changed. The index is like all other indices in `expl3` modules *1-based*. A `bitset` can be output as binary number or—as needed e.g. in a PDF dictionary—as decimal (arabic) number. Currently only a small subset of the functions provided by the `bitset` package are implemented here, mainly the functions needed to use bitsets in PDF dictionaries.

The `bitset` is stored as a string (but one shouldn't rely on the internal representation) and so the vector size is theoretically unlimited, only restricted by `TEX`-memory. But the functions to set and clear bits use integer functions for the index so bitsets can't be longer than $2^{31} - 1$. The export function `\bitset_to_arabic:N` can use functions from the `int` module only if the largest index used for this `bitset` is smaller than `32`, for longer bitsets `fp` is used and this is slower.

31.1 Creating bitsets

```

\bitset_new:N \bitset_new:N <bitset var>
\bitset_new:c \bitset_new:Nn <bitset var>
\bitset_new:Nn {
\bitset_new:cn   <name1> = <index1> ,
                 <name2> = <index2> , ...
New: 2023-11-15 }

```

Creates a new *<bitset var>* or raises an error if the name is already taken. The declaration is global. The *<bitset var>* is initially 0.

Bitsets are implemented as string variables consisting of 1's and 0's. The rightmost number is the index position 1, so the string variable can be viewed directly as the binary number. But one shouldn't rely on the internal representation, but use the dedicated `\bitset_to_bin:N` instead to get the binary number.

The name–index pairs given in the second argument of `\bitset_new:Nn` declares names for some indices, which can be used to set and unset bits. The names 1, 2, 3, ... are predeclared and point to the index positions 1, 2, 3, ...

<index...> should be a positive number or an *<integer expression>* which evaluates to a positive number. The expression is evaluated when the index is used, not at declaration time. The names *<name...>* should be unique. Using a number as name, e.g. `10=1`, is allowed, it then overwrites the predeclared name 10, but the index position 10 can then only be reached if some other name for it exists, e.g. `ten=10`. It is not necessary to give every index a name, and an index can have more than one name. The named index can be extended or changed with the next function.

```

\bitset_addto_named_index:Nn \bitset_addto_named_index:Nn <bitset var>
New: 2023-11-15 {
                 <name1> = <index1> ,
                 <name2> = <index2> , ...
}

```

This extends or changes the name–index pairs for *<bitset var>* globally as described for `\bitset_new:Nn`.

For example after these settings

```

\bitset_new:Nn \l_pdfannot_F_bitset
{
  Invisible      = 1,
  Hidden        = 2,
  Print         = 3,
  NoZoom        = 4,
  NoRotate      = 5,
  NoView        = 6,
  ReadOnly      = 7,
  Locked        = 8,
  ToggleNoView  = 9,
  LockedContents = 10
}
\bitset_addto_named_index:Nn \l_pdfannot_F_bitset
{

```

```

    print = 3
}

```

it is possible to set bit 3 by using any of these alternatives:

```

\bitset_set_true:Nn \l_pdfannot_F_bitset {Print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {3}

```

```

\bitset_if_exist_p:N * \bitset_if_exist_p:N <bitset var>
\bitset_if_exist_p:c * \bitset_if_exist:NTF <bitset var> {(true code)} {(false code)}
\bitset_if_exist:NTF * Tests whether the <bitset var> exist.
\bitset_if_exist:cTF *

```

New: 2023-11-15

31.2 Setting and unsetting bits

```

\bitset_set_true:Nn \bitset_set_true:Nn <bitset var> {(name)}
\bitset_set_true:cn
\bitset_gset_true:Nn
\bitset_gset_true:cn

```

This sets the bit of the index position represented by `{(name)}` to 1. `(name)` should be either one of the predeclared names 1, 2, 3, ..., or one of the names added manually. Index position are 1-based. If needed the length of the bit vector is enlarged.

New: 2023-11-15

```

\bitset_set_false:Nn \bitset_set_false:Nn <bitset var> {(name)}
\bitset_set_false:cn
\bitset_gset_false:Nn
\bitset_gset_false:cn

```

This unsets the bit of the index position represented by `{(name)}` (sets it to 0). `(name)` should be either one of the predeclared names 1, 2, 3, ..., or one of the names added manually. The index is 1-based. If the index position is larger than the current length of the bit vector nothing happens. If the leading (left most) bit is unset, zeros are not trimmed but stay in the bit vector and are still shown by `\bitset_show:N`.

New: 2023-11-15

```

\bitset_clear:N \bitset_clear:N <bitset var>
\bitset_clear:c
\bitset_gclear:N
\bitset_gclear:c

```

This resets the bitset to the initial state. The declared names are not changed.

New: 2023-11-15

31.3 Using bitsets

```

\bitset_item:Nn * \bitset_item:Nn <bitset var> {(name)}
\bitset_item:cn *

```

`\bitset_item:Nn` outputs 1 if the bit with the index number represented by `(name)` is set and 0 otherwise. `(name)` is either one of the predeclared names 1, 2, 3, ..., or one of the names added manually.

New: 2023-11-15

`\bitset_to_bin:N` * `\bitset_to_bin:N` \langle *bitset var* \rangle
`\bitset_to_bin:c` * This leaves the current value of the bitset expressed as a binary (string) number in the
New: 2023-11-15 input stream. If no bit has been set yet, the output is zero.

`\bitset_to_arabic:N` * `\bitset_to_arabic:N` \langle *bitset var* \rangle
`\bitset_to_arabic:c` * This leaves the current value of the bitset expressed as a decimal number in the input
New: 2023-11-15 stream. If no bit has been set yet, the output is zero. The function uses `\int_from_`
`bin:n` if the largest index that have been set or unset is smaller than 32, and a slower
implementation based on `\fp_eval:n` otherwise.

`\bitset_show:N` * `\bitset_show:N` \langle *bitset var* \rangle
`\bitset_show:c` * Displays the binary and decimal values of the \langle *bitset var* \rangle on the terminal.
New: 2023-11-15

`\bitset_log:N` * `\bitset_log:N` \langle *bitset var* \rangle
`\bitset_log:c` * Writes the binary and decimal values of the \langle *bitset var* \rangle in the log file.
New: 2023-11-15

`\bitset_show_named_index:N` * `\bitset_show_named_index:N` \langle *bitset var* \rangle
`\bitset_show_named_index:c` * Displays declared name–index pairs of the \langle *bitset var* \rangle on the terminal.
New: 2023-11-15

`\bitset_log_named_index:N` * `\bitset_log_named_index:N` \langle *bitset var* \rangle
`\bitset_log_named_index:c` * Writes declared name–index pairs of the \langle *bitset var* \rangle in the log file.
New: 2023-12-11

Chapter 32

The `\l3cctab` module

Category code tables

A category code table enables rapid switching of all category codes in one operation. For Lua \TeX , this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables. The implementation of category code tables in `expl3` also saves and restores the \TeX `\endlinechar` primitive value, meaning they could be used for example to implement `\ExplSyntaxOn`.

32.1 Creating and initialising category code tables

<code>\cctab_new:N</code>	<code>\cctab_new:N</code> \langle <i>category code table</i> \rangle
<code>\cctab_new:c</code>	Creates a new \langle <i>category code table</i> \rangle variable or raises an error if the name is already taken. The declaration is global. The \langle <i>category code table</i> \rangle is initialised with the codes as used by <code>ini\TeX</code> .
<code>Updated: 2020-07-02</code>	

<code>\cctab_const:Nn</code>	<code>\cctab_const:Nn</code> \langle <i>category code table</i> \rangle $\{$ \langle <i>category code set up</i> \rangle $\}$
<code>\cctab_const:cn</code>	Creates a new \langle <i>category code table</i> \rangle , applies (in a group) the \langle <i>category code set up</i> \rangle on top of <code>ini\TeX</code> settings, then saves them globally as a constant table. The \langle <i>category code set up</i> \rangle can include a call to <code>\cctab_select:N</code> .
<code>Updated: 2020-07-07</code>	

<code>\cctab_gset:Nn</code>	<code>\cctab_gset:Nn</code> \langle <i>category code table</i> \rangle $\{$ \langle <i>category code set up</i> \rangle $\}$
<code>\cctab_gset:cn</code>	Starting from the <code>ini\TeX</code> category codes, applies (in a group) the \langle <i>category code set up</i> \rangle , then saves them globally in the \langle <i>category code table</i> \rangle . The \langle <i>category code set up</i> \rangle can include a call to <code>\cctab_select:N</code> .
<code>Updated: 2020-07-07</code>	

<code>\cctab_gsave_current:N</code>	<code>\cctab_gsave_current:N</code> \langle <i>category code table</i> \rangle
<code>\cctab_gsave_current:c</code>	Saves the current prevailing category codes in the \langle <i>category code table</i> \rangle .
<code>New: 2023-05-26</code>	

32.2 Using category code tables

<code>\cctab_begin:N</code>	<code>\cctab_begin:N</code> \langle category code table \rangle
<code>\cctab_begin:c</code>	Switches locally the category codes in force to those stored in the \langle category code table \rangle . The prevailing codes before the function is called are added to a stack, for use with <code>\cctab_end:</code> . This function does not start a T _E X group.
Updated: 2020-07-02	

<code>\cctab_end:</code>	<code>\cctab_end:</code>
Updated: 2020-07-02	Ends the scope of a \langle category code table \rangle started using <code>\cctab_begin:N</code> , returning the codes to those in force before the matching <code>\cctab_begin:N</code> was used. This must be used within the same T _E X group (and at the same T _E X group level) as the matching <code>\cctab_begin:N</code> .

<code>\cctab_select:N</code>	<code>\cctab_select:N</code> \langle category code table \rangle
<code>\cctab_select:c</code>	Selects the \langle category code table \rangle for the scope of the current group. This is in particular useful in the \langle setup \rangle arguments of <code>\tl_set_rescan:Nnn</code> , <code>\tl_rescan:nn</code> , <code>\cctab_const:Nn</code> , and <code>\cctab_gset:Nn</code> .
New: 2020-05-19	
Updated: 2020-07-02	

<code>\cctab_item:Nn</code> *	<code>\cctab_item:Nn</code> \langle category code table \rangle $\{\langle$ int expr $\rangle\}$
<code>\cctab_item:cn</code> *	Determines the \langle character \rangle with character code given by the \langle int expr \rangle and expands to its category code specified by the \langle category code table \rangle .
New: 2021-05-10	

32.3 Category code table conditionals

<code>\cctab_if_exist_p:N</code> *	<code>\cctab_if_exist_p:N</code> \langle category code table \rangle
<code>\cctab_if_exist_p:c</code> *	<code>\cctab_if_exist:NTF</code> \langle category code table \rangle $\{\langle$ true code $\rangle\}$ $\{\langle$ false code $\rangle\}$
<code>\cctab_if_exist:NTF</code> *	Tests whether the \langle category code table \rangle is currently defined. This does not check that the \langle category code table \rangle really is a category code table.
<code>\cctab_if_exist:cTF</code> *	

32.4 Constant and scratch category code tables

<code>\c_code_cctab</code>	Category code table for the expl3 code environment; this does <i>not</i> include <code>@</code> , which is retained as an “other” character. Sets the <code>\endlinechar</code> value to 32 (a space).
Updated: 2020-07-10	

<code>\c_document_cctab</code>	Category code table for a standard L ^A T _E X document, as set by the L ^A T _E X kernel. In particular, the upper-half of the 8-bit range will be set to “active” with pdfT _E X <i>only</i> . No babel shorthands will be activated. Sets the <code>\endlinechar</code> value to 13 (normal line ending).
Updated: 2020-07-08	

\c_initex_cctab Category code table as set up by iniT_EX.

Updated: 2020-07-02

\c_other_cctab Category code table where all characters have category code 12 (other). Sets the **\endlinechar** value to -1 .

Updated: 2020-07-02

\c_str_cctab Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space). Sets the **\endlinechar** value to -1 .

Updated: 2020-07-02

\g_tmpa_cctab Scratch category code tables.

\g_tmpb_cctab

New: 2023-05-26

Part V
Text manipulation

Chapter 33

The `l3unicode` module Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. Most of the code here is internal, but there are a small set of public functions. These work with Unicode *codepoints* and are designed to give usable results with both Unicode-aware and 8-bit engines.

`\codepoint_generate:nm` * `\codepoint_generate:nm` {`<codepoint>`} {`<catcode>`}

New: 2022-10-09
Updated: 2022-11-09

Generates one or more character tokens representing the `<codepoint>`. With Unicode engines, exactly one character token will be generated, and this will have the `<catcode>` specified as the second argument:

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the `<codepoint>`. For all codepoints outside of the classical ASCII range, the generated character tokens will be active (category code 13); for codepoints in the ASCII range, the given `<catcode>` will be used. To allow the result of this function to be used inside an expansion context, the result is protected by `\exp_not:n`.

TeXhackers note: Users of (u)pTeX note that these engines are treated as 8-bit in this context. In particular, for upTeX, irrespective of the `\kcatcode` of the `<codepoint>`, any value outside the ASCII range will result in a series of active bytes being generated.

`\codepoint_str_generate:n` * `\codepoint_str_generate:n` {`<codepoint>`}

New: 2022-10-09

Generates one or more character tokens representing the `<codepoint>`. With Unicode engines, exactly one character token will be generated. For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the `<codepoint>`. All of the generated character tokens will be of category code 12, except any spaces (codepoint 32), which will be category code 10.

`\codepoint_to_category:n` ★ `\codepoint_to_category:n` {*codepoint*}

New: 2023-06-19

Expands to the Unicode general category identifier of the *codepoint*. The general category identifier is a string made up of two letter characters, the first uppercase and the second lowercase. The uppercase letters divide codepoints into broader groups, which are then refined by the lowercase letter. For example, codepoints representing letters all have identifiers starting L, for example Lu (uppercase letter), Lt (titlecase letter), *etc.* Full details are available in the documentation provided by the Unicode Consortium: see https://www.unicode.org/reports/tr44/#General_Category_Values

`\codepoint_to_nfd:n` ★ `\codepoint_to_nfd:n` {*codepoint*}

New: 2022-10-09

Converts the *codepoint* to the Unicode Normalization Form Canonical Decomposition. The generated character(s) will have the current category code as they would if typed in directly for Unicode engines; for 8-bit engines, active characters are used for all codepoints outside of the ASCII range.

Chapter 34

The `l3text` module

Text processing

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the $\langle text \rangle$ are normalized and become $\{$ and $\}$, respectively.

34.1 Expanding text

```
\text_expand:n * \text_expand:n { $\langle text \rangle$ }
```

New: 2020-01-02
Updated: 2023-06-09

Takes user input $\langle text \rangle$ and expands the content. Protected commands (typically formatting) are left in place, and no processing takes place of math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`). Commands which are neither engine- nor \LaTeX protected are expanded exhaustively. Any commands listed in `\l_text_expand_exclude_tl` are excluded from expansion, as are those in `\l_text_case_exclude_arg_tl` and `\l_text_math_arg_tl`.

```
\text_declare_expand_equivalent:Nn \text_declare_expand_equivalent:Nn  $\langle cmd \rangle$  { $\langle replacement \rangle$ }  
\text_declare_expand_equivalent:cn
```

New: 2020-01-22

Declares that the $\langle replacement \rangle$ tokens should be used whenever the $\langle cmd \rangle$ (a single token) is encountered. The $\langle replacement \rangle$ tokens should be expandable. A token can be “replaced” by itself if the defined replacement wraps it in `\exp_not:n`, for example

```
\text_declare_expand_equivalent:Nn \' { \exp_not:n { \' } }
```


34.2 Case changing

<code>\text_lowercase:n</code>	*	<code>\text_uppercase:n</code>	{ <i>tokens</i> }
<code>\text_uppercase:n</code>	*	<code>\text_uppercase:nn</code>	{ <i>BCP-47</i> } { <i>tokens</i> }
<code>\text_titlecase_all:n</code>	*	Takes user input <i>text</i> first applies <code>\text_expand:n</code> , then transforms the case of character tokens as specified by the function name. The category code of letters are not	
<code>\text_titlecase_first:n</code>	*	changed by this process when Unicode engines are used; in 8-bit engines, case changed	
<code>\text_lowercase:nn</code>	*	characters in the ASCII range will have the current prevailing category code, while those	
<code>\text_uppercase:nn</code>	*	outside of it will be represented by active characters.	
<code>\text_titlecase_all:nn</code>	*		
<code>\text_titlecase_first:nn</code>	*		

New: 2019-11-20

Updated: 2023-07-08

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first character of the *tokens* to uppercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example *ij* in Dutch which becomes *IJ*. There are two functions available for titlecasing: one which applies the change to each “word” and a second which only applies at the start of the input. (Here, “word” boundaries are spaces: at present, full Unicode word breaking is not attempted.)

Importantly, notice that these functions are intended for working with user *text for typesetting*. For case changing programmatic data see the `l3str` module and discussion there of `\str_lowercase:n`, `\str_uppercase:n` and `\str_casefold:n`.

Case changing does not take place within math mode material so for example

```
\text_uppercase:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

The first mandatory argument of commands listed in `\l_text_case_exclude_arg_tl` is excluded from case changing; the latter are entirely non-textual content (such as labels).

The standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. For \TeX , only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Locale-sensitive conversions are enabled using the *BCP-47* argument, and follow Unicode Consortium guidelines. Currently, the locale strings recognized for special handling are as follows.

- Armenian (`hy` and `hy-x-yiwn`) The setting `hy` maps the codepoint U+0587, the ligature of letters *ech* and *yiwn*, to the codepoints for capital *ech* and *vew* when uppercasing: this follows the spelling reform which is used in Armenia. The alternative `hy-x-yiwn` maps U+0587 to capital *ech* and *yiwn* on uppercasing (also the output if Armenian is not selected at all).
- Azeri and Turkish (`az` and `tr`). The case pairs *I/i-dotless* and *I-dot/i* are activated for these languages. The combining dot mark is removed when lowercasing *I-dot* and introduced when upper casing *i-dotless*.
- German (`de-x-eszett`). An alternative mapping for German in which the lowercase *Eszett* maps to a *großes Eszett*.

- Greek (`e1`). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place. A variant `e1-x-iota` is available which converts the *ypogegrammeni* (subscript muted iota) to capital iota when uppercasing: the standard version retains the subscript versions.
- Lithuanian (`1t`). The lowercase letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Medieval Latin (`1a-x-medieval`). The characters u and V are interchanged on case changing.
- Dutch (`n1`). Capitalisation of ij at the beginning of titlecased input produces IJ rather than Ij.

Determining whether non-letter characters at the start of text should count as the uppercase element is controllable. When `\l_text_titlecase_check_letter_bool` is `true`, codepoints which are not letters (Unicode general category L) are not changed, and only the first *letter* is uppercased. When `\l_text_titlecase_check_letter_bool` is `false`, the first codepoint is uppercased, irrespective of the general code of the character.

```
\text_declare_case_equivalent:Nn \text_declare_case_equivalent:Nn <cmd> {<replacement>}
```

New: 2022-07-04

Declares that the `<replacement>` tokens should be used whenever the `<cmd>` (a single token) is encountered during case changing.

```
\text_declare_lowercase_mapping:nn \text_declare_lowercase_mapping:nn {<codepoint>} {<replacement>}
\text_declare_lowercase_mapping:nnn \text_declare_lowercase_mapping:nnn {<BCP-47>} {<codepoint>}
\text_declare_titlecase_mapping:nn {<replacement>}
\text_declare_titlecase_mapping:nnn
\text_declare_uppercase_mapping:nn
\text_declare_uppercase_mapping:nnn
```

New: 2023-04-11

Updated: 2023-04-20

Declares that the `<replacement>` tokens should be used when case mapping the `<codepoint>`, rather than the standard mapping given in the Unicode data files. The `nnn` version takes a BCP-47 tag, which can be used to specify that the customisation only applies to that locale.

```
\text_case_switch:nnnn * \text_case_switch:nnnn {<normal>} {<upper>} {<lower>} {<title>}
```

New: 2022-07-04

Context-sensitive function which will expand to one of the `<normal>`, `<upper>`, `<lower>` or `<title>` tokens depending on the current case changing operation. Outside of case changing, the `<normal>` tokens are produced. Within case changing, the appropriate mapping tokens are inserted.

34.3 Removing formatting from text

`\text_purify:n` ★ `\text_purify:n` $\langle\text{text}\rangle$

New: 2020-03-05
Updated: 2020-05-14

Takes user input $\langle\text{text}\rangle$ and expands as described for `\text_expand:n`, then removes all functions from the resulting text. Math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) is left contained in a pair of \$ delimiters. Non-expandable functions present in the $\langle\text{text}\rangle$ must either have a defined equivalent (see `\text_declare_purify_equivalent:Nn`) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.

`\text_declare_purify_equivalent:Nn` `\text_declare_purify_equivalent:Nn` $\langle\text{cmd}\rangle$ $\langle\text{replacement}\rangle$
`\text_declare_purify_equivalent:Ne`

New: 2020-03-05

Declares that the $\langle\text{replacement}\rangle$ tokens should be used whenever the $\langle\text{cmd}\rangle$ (a single token) is encountered. The $\langle\text{replacement}\rangle$ tokens should be expandable.

34.4 Control variables

`\l_text_math_arg_tl` Lists commands present in the $\langle\text{text}\rangle$ where the argument of the command should be treated as math mode material. The treatment here is similar to `\l_text_math_delims_tl` but for a command rather than paired delimiters.

`\l_text_math_delims_tl` Lists pairs of tokens which delimit (in-line) math mode content; such content *may* be excluded from processing.

`\l_text_case_exclude_arg_tl`

Lists commands where the first mandatory argument is excluded from case changing.

`\l_text_expand_exclude_tl` Lists commands which are excluded from expansion. This protection includes everything up to and including their first braced argument.

`\l_text_titlecase_check_letter_bool`

Controls how the start of titlecasing is handled: when `true`, the first *letter* in text is considered. The standard setting is `true`.

34.5 Mapping to graphemes

Grapheme splitting is implemented using the algorithm described in Unicode Standard Annex #29. This includes support for extended grapheme clusters. Text starting with a line feed or carriage return character will drop this due to standard T_EX processing. At present extended pictograms are not supported: these may be added in a future release.

`\text_map_function:nN` ☆ `\text_map_function:nN <text> {<function>}`

New: 2022-08-04

Takes user input `<text>` and expands as described for `\text_expand:n`, then maps over the *graphemes* within the result, passing each grapheme to the `<function>`. Broadly a grapheme is a “user perceived character”: the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The `<function>` should accept one argument as `<balanced text>`: this may be comprise codepoints or may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_inline:nn`.

`\text_map_inline:nn` `\text_map_inline:nn <text> {<inline function>}`

New: 2022-08-04

Takes user input `<text>` and expands as described for `\text_expand:n`, then maps over the *graphemes* within the result, passing each grapheme to the `<inline function>`. Broadly a grapheme is a “user perceived character”: the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The `<inline function>` should consist of code which receives the grapheme as `<balanced text>`: this may be comprise codepoints or may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_function:nN`.

`\text_map_break:` ☆ `\text_map_break:`

`\text_map_break:n` ☆ `\text_map_break:n {<code>}`

New: 2022-08-04

Used to terminate a `\text_map_...` function before all entries in the `<text>` have been processed. This normally takes place within a conditional statement.

Part VI
Typesetting

Chapter 35

The l3box module

Boxes

Box variables contain typeset material that can be inserted on the page or in other boxes. Their contents cannot be converted back to lists of tokens. There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`. For instance, a new box variable containing the words “Hello, world!” (in a horizontal box) can be obtained by the following code.

```
\box_new:N \l_hello_box
\hbox_set:Nn \l_hello_box { Hello, ~ world! }
```

The argument is typeset inside a \TeX group so that any variables assigned during the construction of this box restores its value afterwards.

Box variables from `l3box` are compatible with those of $\LaTeX 2_{\epsilon}$ and plain \TeX and can be used interchangeably. The `l3box` commands to construct boxes, such as `\hbox:n` or `\hbox_set:Nn`, are “color-safe”, meaning that

```
\hbox:n { \color_select:n { blue } Hello, } ~ world!
```

will result in “Hello,” taking the color blue, but “world!” remaining with the prevailing color outside the box.

35.1 Creating and initialising boxes

```
\box_new:N \box_new:N <box>
```

`\box_new:c` Creates a new `<box>` or raises an error if the name is already taken. The declaration is global. The `<box>` is initially void.

```
\box_clear:N \box_clear:N <box>
```

`\box_clear:c` Clears the content of the `<box>` by setting the box equal to `\c_empty_box`.

```
\box_gclear:N
\box_gclear:c
```

<code>\box_clear_new:N</code>	<code>\box_clear_new:N</code> $\langle box \rangle$
<code>\box_clear_new:c</code>	
<code>\box_gclear_new:N</code>	Ensures that the $\langle box \rangle$ exists globally by applying <code>\box_new:N</code> if necessary, then applies
<code>\box_gclear_new:c</code>	<code>\box_(g)clear:N</code> to leave the $\langle box \rangle$ empty.

<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_set_eq:(cN Nc cc)</code>	
<code>\box_gset_eq:NN</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$.
<code>\box_gset_eq:(cN Nc cc)</code>	

<code>\box_if_exist_p:N</code> *	<code>\box_if_exist_p:N</code> $\langle box \rangle$
<code>\box_if_exist_p:c</code> *	<code>\box_if_exist:NTF</code> $\langle box \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$
<code>\box_if_exist:NTF</code> *	Tests whether the $\langle box \rangle$ is currently defined. This does not check that the $\langle box \rangle$ really
<code>\box_if_exist:cTF</code> *	is a box.

New: 2012-03-03

35.2 Using boxes

<code>\box_use:N</code>	<code>\box_use:N</code> $\langle box \rangle$
<code>\box_use:c</code>	
	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.

T_EXhackers note: This is the T_EX primitive `\copy`.

<code>\box_move_right:nn</code>	<code>\box_move_right:nn</code> $\{ \langle dim\ expr \rangle \}$ $\{ \langle box\ function \rangle \}$
<code>\box_move_left:nn</code>	
	This function operates in vertical mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced horizontally by the given $\langle dim\ expr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N \langle box \rangle</code> or a “raw” box specification such as <code>\vbox:n { xyz }</code> .

<code>\box_move_up:nn</code>	<code>\box_move_up:nn</code> $\{ \langle dim\ expr \rangle \}$ $\{ \langle box\ function \rangle \}$
<code>\box_move_down:nn</code>	
	This function operates in horizontal mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced vertically by the given $\langle dim\ expr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N \langle box \rangle</code> or a “raw” box specification such as <code>\vbox:n { xyz }</code> .

35.3 Measuring and setting box dimensions

`\box_dp:N` `\box_dp:N` $\langle box \rangle$
`\box_dp:c` Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dim\ expr \rangle$.

TeXhackers note: This is the TeX primitive `\dp`.

`\box_ht:N` `\box_ht:N` $\langle box \rangle$
`\box_ht:c` Calculates the height (above the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dim\ expr \rangle$.

TeXhackers note: This is the TeX primitive `\ht`.

`\box_wd:N` `\box_wd:N` $\langle box \rangle$
`\box_wd:c` Calculates the width of the $\langle box \rangle$ in a form suitable for use in a $\langle dim\ expr \rangle$.

TeXhackers note: This is the TeX primitive `\wd`.

`\box_ht_plus_dp:N` `\box_ht_plus_dp:N` $\langle box \rangle$
`\box_ht_plus_dp:c` Calculates the total vertical size (height plus depth) of the $\langle box \rangle$ in a form suitable for use in a $\langle dim\ expr \rangle$.
New: 2021-05-05

`\box_set_dp:Nn` `\box_set_dp:Nn` $\langle box \rangle$ $\{ \langle dim\ expr \rangle \}$
`\box_set_dp:cn` Set the depth (below the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dim\ expr \rangle \}$.
`\box_gset_dp:Nn`
`\box_gset_dp:cn`
Updated: 2019-01-22

`\box_set_ht:Nn` `\box_set_ht:Nn` $\langle box \rangle$ $\{ \langle dim\ expr \rangle \}$
`\box_set_ht:cn` Set the height (above the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dim\ expr \rangle \}$.
`\box_gset_ht:Nn`
`\box_gset_ht:cn`
Updated: 2019-01-22

`\box_set_wd:Nn` `\box_set_wd:Nn` $\langle box \rangle$ $\{ \langle dim\ expr \rangle \}$
`\box_set_wd:cn` Set the width of the $\langle box \rangle$ to the value of the $\{ \langle dim\ expr \rangle \}$.
`\box_gset_wd:Nn`
`\box_gset_wd:cn`
Updated: 2019-01-22

35.4 Box conditionals

`\box_if_empty_p:N` * `\box_if_empty_p:N` $\langle box \rangle$
`\box_if_empty_p:c` * `\box_if_empty:NTF` $\langle box \rangle$ $\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
`\box_if_empty:NTF` * Tests if $\langle box \rangle$ is a empty (equal to `\c_empty_box`).
`\box_if_empty:cTF` *

`\box_if_horizontal_p:N` * `\box_if_horizontal_p:N` $\langle box \rangle$
`\box_if_horizontal_p:c` * `\box_if_horizontal:NTF` $\langle box \rangle$ $\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
`\box_if_horizontal:NTF` * Tests if $\langle box \rangle$ is a horizontal box.
`\box_if_horizontal:cTF` *

`\box_if_vertical_p:N` * `\box_if_vertical_p:N` $\langle box \rangle$
`\box_if_vertical_p:c` * `\box_if_vertical:NTF` $\langle box \rangle$ $\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
`\box_if_vertical:NTF` * Tests if $\langle box \rangle$ is a vertical box.
`\box_if_vertical:cTF` *

35.5 The last box inserted

`\box_set_to_last:N` `\box_set_to_last:N` $\langle box \rangle$
`\box_set_to_last:c` Sets the $\langle box \rangle$ equal to the last item (box) added to the current partial list, removing the
`\box_gset_to_last:N` item from the list at the same time. When applied to the main vertical list, the $\langle box \rangle$ is
`\box_gset_to_last:c` always void as it is not possible to recover the last added item.

35.6 Constant boxes

`\c_empty_box` This is a permanently empty box, which is neither set as horizontal nor vertical.
Updated: 2012-11-04 **TeXhackers note:** At the TeX level this is a void box.

35.7 Scratch boxes

`\l_tmpa_box` Scratch boxes for local assignment. These are never used by the kernel code, and so are
`\l_tmpb_box` safe for use with any L^AT_EX₃-defined function. However, they may be overwritten by
Updated: 2012-11-04 other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_box` Scratch boxes for global assignment. These are never used by the kernel code, and so
`\g_tmpb_box` are safe for use with any L^AT_EX₃-defined function. However, they may be overwritten by
other non-kernel code and so should only be used for short-term storage.

35.8 Viewing box contents

<code>\box_show:N</code>	<code>\box_show:N <box></code>
<code>\box_show:c</code>	Shows full details of the content of the <code><box></code> in the terminal.
<hr/> <small>Updated: 2012-05-11</small>	

<code>\box_show:Nnn</code>	<code>\box_show:Nnn <box> {(int expr1)} {(int expr2)}</code>
<code>\box_show:cnn</code>	Display the contents of <code><box></code> in the terminal, showing the first <code><int expr1></code> items of the box, and descending into <code><int expr2></code> group levels.
<hr/> <small>New: 2012-05-11</small>	

<code>\box_log:N</code>	<code>\box_log:N <box></code>
<code>\box_log:c</code>	Writes full details of the content of the <code><box></code> to the log.
<hr/> <small>New: 2012-05-11</small>	

<code>\box_log:Nnn</code>	<code>\box_log:Nnn <box> {(int expr1)} {(int expr2)}</code>
<code>\box_log:cnn</code>	Writes the contents of <code><box></code> to the log, showing the first <code><int expr1></code> items of the box, and descending into <code><int expr2></code> group levels.
<hr/> <small>New: 2012-05-11</small>	

35.9 Boxes and color

All L^AT_EX3 boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

35.10 Horizontal mode boxes

<code>\hbox:n</code>	<code>\hbox:n {<contents>}</code>
<code>\hbox:n</code>	Typesets the <code><contents></code> into a horizontal box of natural width and then includes this box in the current list for typesetting.
<hr/> <small>Updated: 2017-04-05</small>	

<code>\hbox_to_wd:nn</code>	<code>\hbox_to_wd:nn {<dim expr>} {<contents>}</code>
<code>\hbox_to_wd:nn</code>	Typesets the <code><contents></code> into a horizontal box of width <code><dim expr></code> and then includes this box in the current list for typesetting.
<hr/> <small>Updated: 2017-04-05</small>	

<code>\hbox_to_zero:n</code>	<code>\hbox_to_zero:n {<contents>}</code>
<code>\hbox_to_zero:n</code>	Typesets the <code><contents></code> into a horizontal box of zero width and then includes this box in the current list for typesetting.
<hr/> <small>Updated: 2017-04-05</small>	

<code>\hbox_set:Nn</code>	<code>\hbox_set:Nn <box> {<contents>}</code>
<code>\hbox_set:cn</code>	Typesets the <code><contents></code> at natural width and then stores the result inside the <code><box></code> .
<code>\hbox_gset:Nn</code>	
<code>\hbox_gset:cn</code>	
<hr/> <small>Updated: 2017-04-05</small>	

<code>\hbox_set_to_wd:Nnn</code>	<code>\hbox_set_to_wd:Nnn <box> {(dim expr)} {<contents>}</code>
<code>\hbox_set_to_wd:cnn</code>	Typesets the <code><contents></code> to the width given by the <code><dim expr></code> and then stores the result inside the <code><box></code> .
<code>\hbox_gset_to_wd:Nnn</code>	
<code>\hbox_gset_to_wd:cnn</code>	
Updated: 2017-04-05	

<code>\hbox_overlap_center:n</code>	<code>\hbox_overlap_center:n {<contents>}</code>
New: 2020-08-25	Typesets the <code><contents></code> into a horizontal box of zero width such that material protrudes equally to both sides of the insertion point.

<code>\hbox_overlap_right:n</code>	<code>\hbox_overlap_right:n {<contents>}</code>
Updated: 2017-04-05	Typesets the <code><contents></code> into a horizontal box of zero width such that material protrudes to the right of the insertion point.

<code>\hbox_overlap_left:n</code>	<code>\hbox_overlap_left:n {<contents>}</code>
Updated: 2017-04-05	Typesets the <code><contents></code> into a horizontal box of zero width such that material protrudes to the left of the insertion point.

<code>\hbox_set:Nw</code>	<code>\hbox_set:Nw <box> <contents> \hbox_set_end:</code>
<code>\hbox_set:cw</code>	Typesets the <code><contents></code> at natural width and then stores the result inside the <code><box></code> . In contrast to <code>\hbox_set:Nn</code> this function does not absorb the argument when finding the <code><content></code> , and so can be used in circumstances where the <code><content></code> may not be a simple argument.
<code>\hbox_set_end:</code>	
<code>\hbox_gset:Nw</code>	
<code>\hbox_gset:cw</code>	
<code>\hbox_gset_end:</code>	
Updated: 2017-04-05	

<code>\hbox_set_to_wd:Nnw</code>	<code>\hbox_set_to_wd:Nnw <box> {(dim expr)} <contents> \hbox_set_end:</code>
<code>\hbox_set_to_wd:cnw</code>	Typesets the <code><contents></code> to the width given by the <code><dim expr></code> and then stores the result inside the <code><box></code> . In contrast to <code>\hbox_set_to_wd:Nnn</code> this function does not absorb the argument when finding the <code><content></code> , and so can be used in circumstances where the <code><content></code> may not be a simple argument
<code>\hbox_gset_to_wd:Nnw</code>	
<code>\hbox_gset_to_wd:cnw</code>	
New: 2017-06-08	

<code>\hbox_unpack:N</code>	<code>\hbox_unpack:N <box></code>
<code>\hbox_unpack:c</code>	Unpacks the content of the horizontal <code><box></code> , retaining any stretching or shrinking applied when the <code><box></code> was set.

TeXhackers note: This is the TeX primitive `\unhcopy`.

35.11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are

`_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

`\vbox:n` `\vbox:n {⟨contents⟩}`
 Updated: 2017-04-05 Typesets the `⟨contents⟩` into a vertical box of natural height and includes this box in the current list for typesetting.

`\vbox_top:n` `\vbox_top:n {⟨contents⟩}`
 Updated: 2017-04-05 Typesets the `⟨contents⟩` into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the *first* item added to the box.

`\vbox_to_ht:nn` `\vbox_to_ht:nn {⟨dim expr⟩} {⟨contents⟩}`
 Updated: 2017-04-05 Typesets the `⟨contents⟩` into a vertical box of height `⟨dim expr⟩` and then includes this box in the current list for typesetting.

`\vbox_to_zero:n` `\vbox_to_zero:n {⟨contents⟩}`
 Updated: 2017-04-05 Typesets the `⟨contents⟩` into a vertical box of zero height and then includes this box in the current list for typesetting.

`\vbox_set:Nn` `\vbox_set:Nn ⟨box⟩ {⟨contents⟩}`
`\vbox_set:cn` Typesets the `⟨contents⟩` at natural height and then stores the result inside the `⟨box⟩`.
`\vbox_gset:Nn`
`\vbox_gset:cn`
 Updated: 2017-04-05

`\vbox_set_top:Nn` `\vbox_set_top:Nn ⟨box⟩ {⟨contents⟩}`
`\vbox_set_top:cn` Typesets the `⟨contents⟩` at natural height and then stores the result inside the `⟨box⟩`.
`\vbox_gset_top:Nn` The baseline of the box is equal to that of the *first* item added to the box.
`\vbox_gset_top:cn`
 Updated: 2017-04-05

`\vbox_set_to_ht:Nnn` `\vbox_set_to_ht:Nnn ⟨box⟩ {⟨dim expr⟩} {⟨contents⟩}`
`\vbox_set_to_ht:cnn` Typesets the `⟨contents⟩` to the height given by the `⟨dim expr⟩` and then stores the
`\vbox_gset_to_ht:Nnn` result inside the `⟨box⟩`.
`\vbox_gset_to_ht:cnn`
 Updated: 2017-04-05

`\vbox_set:Nw` `\vbox_set:Nw ⟨box⟩ ⟨contents⟩ \vbox_set_end:`
`\vbox_set:cw` Typesets the `⟨contents⟩` at natural height and then stores the result inside the `⟨box⟩`.
`\vbox_set_end:` In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding
`\vbox_gset:Nw` the `⟨content⟩`, and so can be used in circumstances where the `⟨content⟩` may not be a
`\vbox_gset:cw` simple argument.
`\vbox_gset_end:`
 Updated: 2017-04-05

<code>\vbox_set_to_ht:Nnw</code>	<code>\vbox_set_to_ht:Nnw <box> {<dim expr>} <contents> \vbox_set_end:</code>
<code>\vbox_set_to_ht:cnw</code>	Typesets the <code><contents></code> to the height given by the <code><dim expr></code> and then stores the result inside the <code><box></code> . In contrast to <code>\vbox_set_to_ht:Nnn</code> this function does not absorb the argument when finding the <code><content></code> , and so can be used in circumstances where the <code><content></code> may not be a simple argument
<code>\vbox_gset_to_ht:Nnw</code>	
<code>\vbox_gset_to_ht:cnw</code>	

New: 2017-06-08

<code>\vbox_set_split_to_ht:NNn</code>	<code>\vbox_set_split_to_ht:NNn <box₁₂</code>
<code>\vbox_set_split_to_ht:(cNn Ncn ccn)</code>	
<code>\vbox_gset_split_to_ht:NNn</code>	
<code>\vbox_gset_split_to_ht:(cNn Ncn ccn)</code>	

Updated: 2018-12-29

Sets `<box1 to contain material to the height given by the <dim expr> by removing content from the top of <box2 (which must be a vertical box).`

<code>\vbox_unpack:N</code>	<code>\vbox_unpack:N <box></code>
<code>\vbox_unpack:c</code>	Unpacks the content of the vertical <code><box></code> , retaining any stretching or shrinking applied when the <code><box></code> was set.

TeXhackers note: This is the TeX primitive `\unvcopy`.

35.12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other `expl3` variable. However, for efficiency reasons, it is also useful to have functions which *drop* box contents on use. When a box is dropped, the box becomes empty at the group level *where the box was originally set* rather than necessarily *at the current group level*. For example, with

```
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
  \hbox_set:Nn \l_tmpa_box { B }
  \group_begin:
    \box_use_drop:N \l_tmpa_box
  \group_end:
  \box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
```

the first use of `\box_show:N` will show an entirely cleared (void) box, and the second will show the letter A in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of `drop` functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.

<code>\box_use_drop:N</code>	<code>\box_use_drop:N</code> $\langle box \rangle$
<code>\box_use_drop:c</code>	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

T_EXhackers note: This is the T_EX primitive `\box`.

<code>\box_set_eq_drop:NN</code>	<code>\box_set_eq_drop:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_set_eq_drop:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$, then drops $\langle box_2 \rangle$.
<small>New: 2019-01-17</small>	

<code>\box_gset_eq_drop:NN</code>	<code>\box_gset_eq_drop:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_gset_eq_drop:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ globally equal to that of $\langle box_2 \rangle$, then drops $\langle box_2 \rangle$.
<small>New: 2019-01-17</small>	

<code>\hbox_unpack_drop:N</code>	<code>\hbox_unpack_drop:N</code> $\langle box \rangle$
<code>\hbox_unpack_drop:c</code>	Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The original $\langle box \rangle$ is then dropped.
<small>New: 2019-01-17</small>	

T_EXhackers note: This is the T_EX primitive `\unhbox`.

<code>\vbox_unpack_drop:N</code>	<code>\vbox_unpack_drop:N</code> $\langle box \rangle$
<code>\vbox_unpack_drop:c</code>	Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The original $\langle box \rangle$ is then dropped.
<small>New: 2019-01-17</small>	

T_EXhackers note: This is the T_EX primitive `\unvbox`.

35.13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in T_EX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

```
\box_autosize_to_wd_and_ht:Nnn \box_autosize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}
\box_autosize_to_wd_and_ht:cnn
\box_gautosize_to_wd_and_ht:Nnn
\box_gautosize_to_wd_and_ht:cnn
```

New: 2017-04-04
Updated: 2019-01-22

Resizes the $\langle box \rangle$ to fit within the given $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ is the smaller of $\{ \langle x-size \rangle \}$ and $\{ \langle y-size \rangle \}$, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_autosize_to_wd_and_ht_plus_dp:Nnn \box_autosize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>}
\box_autosize_to_wd_and_ht_plus_dp:cnn {<y-size>}
\box_gautosize_to_wd_and_ht_plus_dp:Nnn
\box_gautosize_to_wd_and_ht_plus_dp:cnn
```

New: 2017-04-04
Updated: 2019-01-22

Resizes the $\langle box \rangle$ to fit within the given $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ is the smaller of $\{ \langle x-size \rangle \}$ and $\{ \langle y-size \rangle \}$, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_ht:Nn \box_resize_to_ht:Nn <box> {<y-size>}
\box_resize_to_ht:cn
\box_gresize_to_ht:Nn
\box_gresize_to_ht:cn
```

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle y-size \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y-size \rangle$ is a dimension expression. The $\langle y-size \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_ht_plus_dp:Nn \box_resize_to_ht_plus_dp:Nn <box> {<y-size>}
\box_resize_to_ht_plus_dp:cn
\box_gresize_to_ht_plus_dp:Nn
\box_gresize_to_ht_plus_dp:cn
```

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle y-size \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y-size \rangle$ is a dimension expression. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_wd:Nn \box_resize_to_wd:Nn <box> {<x-size>}
\box_resize_to_wd:cn
\box_gresize_to_wd:Nn
\box_gresize_to_wd:cn
```

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally), scaling the vertical size by the same amount; $\langle x-size \rangle$ is a dimension expression. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle x-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle x-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_wd_and_ht:Nnn \box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}
\box_resize_to_wd_and_ht:cn
\box_gresize_to_wd_and_ht:Nnn
\box_gresize_to_wd_and_ht:cn
```

New: 2014-07-03

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the height only and does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_wd_and_ht_plus_dp:Nnn \box_resize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>} {<y-size>}
\box_resize_to_wd_and_ht_plus_dp:cn
\box_gresize_to_wd_and_ht_plus_dp:Nnn
\box_gresize_to_wd_and_ht_plus_dp:cn
```

New: 2017-04-06

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {<angle>}</code>
<code>\box_rotate:cn</code>	
<code>\box_grotate:Nn</code>	
<code>\box_grotate:cn</code>	
<code>Updated: 2019-01-22</code>	

Rotates the `<box>` by `<angle>` (a `<fp expr>` in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the rotation is applied.

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>
<code>\box_scale:cnn</code>	
<code>\box_gscale:Nnn</code>	
<code>\box_gscale:cnn</code>	
<code>Updated: 2019-01-22</code>	

Scales the `<box>` by factors `<x-scale>` and `<y-scale>` in the horizontal and vertical directions, respectively (both scales are `<fp expr>`). The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the scaling is applied. Negative scalings cause the material in the `<box>` to be reversed in direction, but the reference point of the `<box>` is unchanged. Thus a negative `<y-scale>` results in the `<box>` having a depth dependent on the height of the original and *vice versa*.

35.14 Viewing part of a box

<code>\box_set_clipped:N</code>	<code>\box_set_clipped:N <box></code>
<code>\box_set_clipped:c</code>	
<code>\box_gset_clipped:N</code>	
<code>\box_gset_clipped:c</code>	
<code>Updated: 2023-04-14</code>	

Clips the `<box>` in the output so that only material inside the bounding box is displayed in the output. The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the clipping is applied. Additional box levels are also generated by this operation.

TeXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

<code>\box_set_trim:Nnnnn</code>	<code>\box_set_trim:Nnnnn <box> {<left>} {<bottom>} {<right>} {<top>}</code>
<code>\box_set_trim:cnnnn</code>	
<code>\box_gset_trim:Nnnnn</code>	
<code>\box_gset_trim:cnnnn</code>	
<code>New: 2019-01-23</code>	

Adjusts the bounding box of the `<box>` `<left>` is removed from the left-hand edge of the bounding box, `<right>` from the right-hand edge and so fourth. All adjustments are `<dim exprs>`. Material outside of the bounding box is still displayed in the output unless `\box_set_clipped:N` is subsequently applied. The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the trim operation is applied. Additional box levels are also generated by this operation. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

<code>\box_set_viewport:Nnnnn</code>	<code>\box_set_viewport:Nnnnn <box> {<llx>} {<lly>} {<urx>} {<ury>}</code>
<code>\box_set_viewport:cnnnn</code>	
<code>\box_gset_viewport:Nnnnn</code>	
<code>\box_gset_viewport:cnnnn</code>	
<code>New: 2019-01-23</code>	

Adjusts the bounding box of the `<box>` such that it has lower-left coordinates (`<llx>`, `<lly>`) and upper-right coordinates (`<urx>`, `<ury>`). All four coordinate positions are `<dim exprs>`. Material outside of the bounding box is still displayed in the output unless `\box_set_clipped:N` is subsequently applied. The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the viewport operation is applied. Additional box levels are also generated by this operation.

35.15 Primitive box conditionals

```
\if_hbox:N * \if_hbox:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests is `<box>` is a horizontal box.

TeXhackers note: This is the TeX primitive `\ifhbox`.

```
\if_vbox:N * \if_vbox:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests is `<box>` is a vertical box.

TeXhackers note: This is the TeX primitive `\ifvbox`.

```
\if_box_empty:N * \if_box_empty:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests is `<box>` is an empty (void) box.

TeXhackers note: This is the TeX primitive `\ifvoid`.

Chapter 36

The `\l3coffins` module

Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the `xcoffins` module (in the `\l3experimental` bundle).

36.1 Creating and initialising coffins

`\coffin_new:N` `\coffin_new:N` $\langle coffin \rangle$

`\coffin_new:c` Creates a new $\langle coffin \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle coffin \rangle$ is initially empty.

New: 2011-08-17

`\coffin_clear:N` `\coffin_clear:N` $\langle coffin \rangle$

`\coffin_clear:c` Clears the content of the $\langle coffin \rangle$.

`\coffin_gclear:N`
`\coffin_gclear:c`

New: 2011-08-17

Updated: 2019-01-21

`\coffin_set_eq:NN` `\coffin_set_eq:NN` $\langle coffin_1 \rangle$ $\langle coffin_2 \rangle$

`\coffin_set_eq:(Nc|cN|cc)` Sets both the content and poles of $\langle coffin_1 \rangle$ equal to those of $\langle coffin_2 \rangle$.

`\coffin_gset_eq:NN`
`\coffin_gset_eq:(Nc|cN|cc)`

New: 2011-08-17

Updated: 2019-01-21

`\coffin_if_exist_p:N` \star `\coffin_if_exist_p:N` $\langle coffin \rangle$

`\coffin_if_exist_p:c` \star `\coffin_if_exist:NTF` $\langle coffin \rangle$ $\{(true\ code)\}$ $\{(false\ code)\}$

`\coffin_if_exist:NTF` \star Tests whether the $\langle coffin \rangle$ is currently defined.

`\coffin_if_exist:cTF` \star

New: 2012-06-20

36.2 Setting coffin content and poles

<code>\hcoffin_set:Nn</code>	<code>\hcoffin_set:Nn <coffin> {<material>}</code>
<code>\hcoffin_set:cn</code>	Typesets the <code><material></code> in horizontal mode, storing the result in the <code><coffin></code> . The standard poles for the <code><coffin></code> are then set up based on the size of the typeset material.
<code>\hcoffin_gset:Nn</code>	
<code>\hcoffin_gset:cn</code>	

New: 2011-08-17

Updated: 2019-01-21

<code>\hcoffin_set:Nw</code>	<code>\hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:</code>
<code>\hcoffin_set:cw</code>	Typesets the <code><material></code> in horizontal mode, storing the result in the <code><coffin></code> . The standard poles for the <code><coffin></code> are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.
<code>\hcoffin_set_end:</code>	
<code>\hcoffin_gset:Nw</code>	
<code>\hcoffin_gset:cw</code>	
<code>\hcoffin_gset_end:</code>	

New: 2011-09-10

Updated: 2019-01-21

<code>\vcoffin_set:Nnn</code>	<code>\vcoffin_set:Nnn <coffin> {<width>} {<material>}</code>
<code>\vcoffin_set:cnn</code>	Typesets the <code><material></code> in vertical mode constrained to the given <code><width></code> and stores the result in the <code><coffin></code> . The standard poles for the <code><coffin></code> are then set up based on the size of the typeset material.
<code>\vcoffin_gset:Nnn</code>	
<code>\vcoffin_gset:cnn</code>	

New: 2011-08-17

Updated: 2023-02-03

<code>\vcoffin_set:Nnw</code>	<code>\vcoffin_set:Nnw <coffin> {<width>} <material> \vcoffin_set_end:</code>
<code>\vcoffin_set:cnw</code>	Typesets the <code><material></code> in vertical mode constrained to the given <code><width></code> and stores the result in the <code><coffin></code> . The standard poles for the <code><coffin></code> are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.
<code>\vcoffin_set_end:</code>	
<code>\vcoffin_gset:Nnw</code>	
<code>\vcoffin_gset:cnw</code>	
<code>\vcoffin_gset_end:</code>	

New: 2011-09-10

Updated: 2023-02-03

<code>\coffin_set_horizontal_pole:Nnn</code>	<code>\coffin_set_horizontal_pole:Nnn <coffin></code>
<code>\coffin_set_horizontal_pole:cnn</code>	<code>{<pole>} {<offset>}</code>
<code>\coffin_gset_horizontal_pole:Nnn</code>	
<code>\coffin_gset_horizontal_pole:cnn</code>	

New: 2012-07-20

Updated: 2019-01-21

Sets the `<pole>` to run horizontally through the `<coffin>`. The `<pole>` is placed at the `<offset>` from the baseline of the `<coffin>`. The `<offset>` should be given as a dimension expression.

`\coffin_set_vertical_pole:Nnn` `\coffin_set_vertical_pole:Nnn` $\langle coffin \rangle$ $\{\langle pole \rangle\}$ $\{\langle offset \rangle\}$
`\coffin_set_vertical_pole:cnn`
`\coffin_gset_vertical_pole:Nnn`
`\coffin_gset_vertical_pole:cnn`

New: 2012-07-20
Updated: 2019-01-21

Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ is placed at the $\langle offset \rangle$ from the left-hand edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

`\coffin_reset_poles:N` `\coffin_reset_poles:N` $\langle coffin \rangle$
`\coffin_greset_poles:N`

New: 2023-05-17

Resets the poles of the $\langle coffin \rangle$ to the standard set, removing any custom or inherited poles. The poles will therefore be equal to those that would be obtained from `\hcoffin_set:Nn` or similar; the bounding box of the coffin is not reset, so any material outside of the formal bounding box will not influence the poles.

36.3 Coffin affine transformations

`\coffin_resize:Nnn` `\coffin_resize:Nnn` $\langle coffin \rangle$ $\{\langle width \rangle\}$ $\{\langle total-height \rangle\}$
`\coffin_resize:cnn`
`\coffin_gresize:Nnn` Resized the $\langle coffin \rangle$ to $\langle width \rangle$ and $\langle total-height \rangle$, both of which should be given as
`\coffin_gresize:cnn` dimension expressions.

Updated: 2019-01-23

`\coffin_rotate:Nn` `\coffin_rotate:Nn` $\langle coffin \rangle$ $\{\langle angle \rangle\}$
`\coffin_rotate:cn`
`\coffin_grotate:Nn` Rotates the $\langle coffin \rangle$ by the given $\langle angle \rangle$ (given in degrees counter-clockwise). This
`\coffin_grotate:cnn` process rotates both the coffin content and poles. Multiple rotations do not result in the
bounding box of the coffin growing unnecessarily.

`\coffin_scale:Nnn` `\coffin_scale:Nnn` $\langle coffin \rangle$ $\{\langle x-scale \rangle\}$ $\{\langle y-scale \rangle\}$
`\coffin_scale:cnn`
`\coffin_gscale:Nnn` Scales the $\langle coffin \rangle$ by a factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical
`\coffin_gscale:cnn` directions, respectively. The two scale factors should be given as real numbers.

Updated: 2019-01-23

36.4 Joining and using coffins

<code>\coffin_attach:NnnNnnnn</code>	<code>\coffin_attach:NnnNnnnn</code>
<code>\coffin_attach:(cnnNnnnn Nnncnnnn cnncnnnn)</code>	<code>\coffin_attach:\langle coffin_1 \rangle \{\langle coffin_1-pole_1 \rangle\} \{\langle coffin_1-pole_2 \rangle\}</code>
<code>\coffin_gattach:NnnNnnnn</code>	<code>\coffin_gattach:\langle coffin_2 \rangle \{\langle coffin_2-pole_1 \rangle\} \{\langle coffin_2-pole_2 \rangle\}</code>
<code>\coffin_gattach:(cnnNnnnn Nnncnnnn cnncnnnn)</code>	<code>\coffin_gattach:\langle x-offset \rangle \{\langle y-offset \rangle\}</code>

Updated: 2019-01-22

This function attaches $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ is not altered, *i.e.* $\langle coffin_2 \rangle$ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<code>\coffin_join:NnnNnnnn</code>	<code>\coffin_join:NnnNnnnn</code>
<code>\coffin_join:(cnnNnnnn Nnncnnnn cnncnnnn)</code>	<code>\coffin_join:\langle coffin_1 \rangle \{\langle coffin_1-pole_1 \rangle\} \{\langle coffin_1-pole_2 \rangle\}</code>
<code>\coffin_gjoin:NnnNnnnn</code>	<code>\coffin_gjoin:\langle coffin_2 \rangle \{\langle coffin_2-pole_1 \rangle\} \{\langle coffin_2-pole_2 \rangle\}</code>
<code>\coffin_gjoin:(cnnNnnnn Nnncnnnn cnncnnnn)</code>	<code>\coffin_gjoin:\langle x-offset \rangle \{\langle y-offset \rangle\}</code>

Updated: 2019-01-22

This function joins $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<code>\coffin_typeset:Nnnnn</code>	<code>\coffin_typeset:Nnnnn \langle coffin \rangle \{\langle pole_1 \rangle\} \{\langle pole_2 \rangle\}</code>
<code>\coffin_typeset:cnnnn</code>	<code>\coffin_typeset:cnnnn \{\langle x-offset \rangle\} \{\langle y-offset \rangle\}</code>

Updated: 2012-07-20

Typesetting is carried out by first calculating $\langle handle \rangle$, the point of intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the $\langle handle \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

36.5 Measuring coffins

<code>\coffin_dp:N</code>	<code>\coffin_dp:N \langle coffin \rangle</code>
<code>\coffin_dp:c</code>	Calculates the depth (below the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dim expr \rangle$.

<hr/> <code>\coffin_ht:N</code>	<code>\coffin_ht:N</code> $\langle coffin \rangle$
<hr/> <code>\coffin_ht:c</code>	Calculates the height (above the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dim expr \rangle$.
<hr/> <code>\coffin_wd:N</code>	<code>\coffin_wd:N</code> $\langle coffin \rangle$
<hr/> <code>\coffin_wd:c</code>	Calculates the width of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dim expr \rangle$.

36.6 Coffin diagnostics

<hr/> <code>\coffin_display_handles:Nn</code>	<code>\coffin_display_handles:Nn</code> $\langle coffin \rangle$ $\{\langle color \rangle\}$
<hr/> <code>\coffin_display_handles:cn</code>	This function first calculates the intersections between all of the $\langle poles \rangle$ of the $\langle coffin \rangle$ to give a set of $\langle handles \rangle$. It then prints the $\langle coffin \rangle$ at the current location in the source, with the position of the $\langle handles \rangle$ marked on the coffin. The $\langle handles \rangle$ are labelled as part of this process: the locations of the $\langle handles \rangle$ and the labels are both printed in the $\langle color \rangle$ specified.
<hr/> <code>Updated: 2011-09-02</code>	

<hr/> <code>\coffin_mark_handle:Nnnn</code>	<code>\coffin_mark_handle:Nnnn</code> $\langle coffin \rangle$ $\{\langle pole_1 \rangle\}$ $\{\langle pole_2 \rangle\}$ $\{\langle color \rangle\}$
<hr/> <code>\coffin_mark_handle:cnnn</code>	This function first calculates the $\langle handle \rangle$ for the $\langle coffin \rangle$ as defined by the intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. It then marks the position of the $\langle handle \rangle$ on the $\langle coffin \rangle$. The $\langle handle \rangle$ are labelled as part of this process: the location of the $\langle handle \rangle$ and the label are both printed in the $\langle color \rangle$ specified.
<hr/> <code>Updated: 2011-09-02</code>	

<hr/> <code>\coffin_show_structure:N</code>	<code>\coffin_show_structure:N</code> $\langle coffin \rangle$
<hr/> <code>\coffin_show_structure:c</code>	This function shows the structural information about the $\langle coffin \rangle$ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.
<hr/> <code>Updated: 2015-08-01</code>	

Notice that the poles of a coffin are defined by four values: the x and y coordinates of a point that the pole passes through and the x - and y -components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

<hr/> <code>\coffin_log_structure:N</code>	<code>\coffin_log_structure:N</code> $\langle coffin \rangle$
<hr/> <code>\coffin_log_structure:c</code>	This function writes the structural information about the $\langle coffin \rangle$ in the log file. See also <code>\coffin_show_structure:N</code> which displays the result in the terminal.
<hr/> <code>New: 2014-08-22</code>	
<hr/> <code>Updated: 2015-08-01</code>	

<hr/> <code>\coffin_show:N</code>	<code>\coffin_show:N</code> $\langle coffin \rangle$
<hr/> <code>\coffin_show:c</code>	<code>\coffin_log:N</code> $\langle coffin \rangle$
<hr/> <code>\coffin_log:N</code>	Shows full details of poles and contents of the $\langle coffin \rangle$ in the terminal or log file. See
<hr/> <code>\coffin_log:c</code>	<code>\coffin_show_structure:N</code> and <code>\box_show:N</code> to show separately the pole structure and the contents.
<hr/> <code>New: 2021-05-11</code>	

<code>\coffin_show:Nnn</code>	<code>\coffin_show:Nnn <coffin> {<int expr1>} {<int expr2>}</code>
<code>\coffin_show:cnn</code>	<code>\coffin_log:Nnn <coffin> {<int expr1>} {<int expr2>}</code>
<code>\coffin_log:Nnn</code>	Shows poles and contents of the <code><coffin></code> in the terminal or log file, showing the first <code><int expr1></code> items in the coffin, and descending into <code><int expr2></code> group levels. See <code>\coffin_show_structure:N</code> and <code>\box_show:Nnn</code> to show separately the pole structure and the contents.
<code>\coffin_log:cnn</code>	

New: 2021-05-11

36.7 Constants and variables

`\c_empty_coffin` A permanently empty coffin.

`\l_tmpa_coffin` Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

New: 2012-06-19

`\g_tmpa_coffin` Scratch coffins for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

New: 2019-01-24

Chapter 37

The `l3color` module

Color support

37.1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

<code>\color_group_begin:</code>	<code>\color_group_begin:</code>
<code>\color_group_end:</code>	<code>...</code>
<small>New: 2011-09-03</small>	<code>\color_group_end:</code>

Creates a color group: one used to “trap” color settings. This grouping is built in to for example `\hbox_set:Nn`.

<code>\color_ensure_current:</code>	<code>\color_ensure_current:</code>
<small>New: 2011-09-03</small>	

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end: group`.

37.2 Color models

A color *model* is a way to represent sets of colors. Different models are particularly suitable for different output methods, *e.g.* screen or print. Parameter-based models can describe a very large number of unique colors, and have a varying number of *axes* which define a color space. In contrast, various proprietary models are available which define *spot* colors (more formally separations).

Core models are used to pass color information to output; these are “native” to `l3color`. Core models use real numbers in the range $[0, 1]$ to represent values. The core models supported here are

- **gray** Grayscale color, with a single axis running from 0 (fully black) to 1 (fully white)
- **rgb** Red-green-blue color, with three axes, one for each of the components

- **cm_yk** Cyan-magenta-yellow-black color, with four axes, one for each of the components

There are also interface models: these are convenient for users but have to be manipulated before storing/passing to the backend. Interface models are primarily integer-based: see below for more detail. The supported interface models are

- **Gray** Grayscale color, with a single axis running from 0 (fully black) to 15 (fully white)
- **hsb** Hue-saturation-brightness color, with three axes, all real values in the range $[0, 1]$ for hue saturation and brightness
- **Hsb** Hue-saturation-brightness color, with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness
- **HSB** Hue-saturation-brightness color, with three axes, integers in the range $[0, 240]$ for hue, saturation and brightness
- **HTML** HTML format representation of RGB color given as a single six-digit hexadecimal number
- **RGB** Red-green-blue color, with three axes, one for each of the components, values as integers from 0 to 255
- **wave** Light wavelength, a real number in the range 380 to 780 (nanometres)

All interface models are internally stored as **rgb**.

Finally, there are a small number of models which are parsed to allow data transfer from **xcolor** but which should not be used by end-users. These are

- **cm_y** Cyan-magenta-yellow color with three axes, one for each of the components; converted to **cm_yk**
- **tHsb** “Tuned” hue-saturation-brightness color with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness; converted to **rgb** using the standard tuning map defined by **xcolor**
- **&spot** Spot color tint with one value; treated as a gray tint as spot color data is not available for extraction

To allow parsing of data from **xcolor**, any leading model up the first **:** will be discarded; the approach of selecting an internal form for data is *not* used in **l3color**.

Additional models may be created to allow mixing of separation colors with each other or with those from other models. See Section 37.9 for more detail of color support for additional models.

When color is selected by model, the $\langle \text{values} \rangle$ given are specified as a comma-separated list. The length of the list will therefore be determined by the detail of the model involved.

Color models (and interconversion) are complex, and more details are given in the manual to the $\text{\LaTeX} 2_{\epsilon}$ **xcolor** package and in the *PostScript Language Reference Manual*, published by Addison–Wesley.

37.3 Color expressions

In addition to allowing specification of color by model and values, `l3color` also supports color expressions. These are created by combining one or more color names, with the amount of each specified as a value in the range 0–100. The value should be given between `!` symbols in the expression. Thus for example

```
red!50!green
```

is a mixture of 50% red and 50% green. A trailing value is interpreted as implicitly followed by `!white`, and so

```
red!25
```

specifies 25% red mixed with 75% white.

Where the models for the mixed colors are different, the model of the first color is used. Thus

```
red!50!cyan
```

will result in a color specification using the `rgb` model, made up of 50% red and 50% of cyan *expressed in rgb*. This may be important as color model interconversion is not exact.

The one exception to the above is where the first model in an expression is `gray`. In this case, the order of mixing is “swapped” internally, so that for example

```
black!50!red
```

has the same result as

```
red!50!black
```

(the predefined colors `black` and `white` use the `gray` model).

Where more than two colors are mixed in an expression, evaluation takes place in a stepwise fashion. Thus in

```
cyan!50!magenta!10!yellow
```

the sub-expression

```
cyan!50!magenta
```

is first evaluated to give an intermediate color specification, before the second step

```
<intermediate>!10!yellow
```

where `<intermediate>` represents this transitory calculated value.

Within a color expression, `.` may be used to represent the color active for typesetting (the current color). This allows for example

```
!.!50
```

to mean a mixture of 50% of current color with white.

(Color expressions supported here are a subset of those provided by the `LATEX 2ε xcolor` package. At present, only such features as are clearly useful have been added here.)

37.4 Named colors

Color names are stored in a single namespace, which makes them accessible as part of color expressions. Whilst they are not reserved in a technical sense, the names **black**, **white**, **red**, **green**, **blue**, **cyan**, **magenta** and **yellow** have special meaning and should not be redefined. Color names should be made up of letters, numbers and spaces only: other characters are reserved for use in color expressions. In particular, `.` represents the current color at the start of a color expression.

`\color_set:nn` `\color_set:nn {<name>} {<color expression>}`

Evaluates the `<color expression>` and stores the resulting color specification as the `<name>`.

`\color_set:nnn` `\color_set:nnn {<name>} {<model(s)>} {<value(s)>}`

Stores the color specification equivalent to the `<model(s)>` and `<values>` as the `<name>`.

`\color_set_eq:nn` `\color_set_eq:nn {<name1>} {<name2>}`

Copies the color specification in `<name2>` to `<name1>`. The special name `.` may be used to represent the current color, allowing it to be saved to a name.

`\color_if_exist_p:n` `\color_if_exist_p:n {<name>}`

`\color_if_exist:nTF` `\color_if_exist:nTF {<name>} {<>true code>} {<>false code>}`

New: 2022-08-12 Tests whether `<name>` is currently defined to provide a color specification.

`\color_show:n` `\color_show:n {<name>}`

`\color_log:n` `\color_log:n {<name>}`

New: 2021-05-11 Displays the color specification stored in the `<name>` on the terminal or log file.

37.5 Selecting colors

General selection of color is safe when split across pages: a stack is used to ensure that the correct color is re-selected on the new page.

These commands set the current color (`.`): other more specialised functions such as fill and stroke selectors do *not* adjust this value.

`\color_select:n` `\color_select:n {<color expression>}`

Parses the `<color expression>` and then activates the resulting color specification for typeset material.

`\color_select:nn` `\color_select:nn {<model(s)>} {<value(s)>}`

Activates the color specification equivalent to the `<model(s)>` and `<value(s)>` for typeset material.

`\l_color_fixed_model_tl` When this is set to a non-empty value, colors will be converted to the specified model when they are selected. Note that included images and similar are not influenced by this setting.

37.6 Colors for fills and strokes

Colors for drawing operations and so forth are split into strokes and fills (the latter may also be referred to as non-stroke color). The fill color is used for text under normal circumstances. Depending on the backend, stroke color may use a *stack*, in which case it exhibits the same page breaking behavior as general color. However, `dvips/dvisvgn` do not support this, and so color will need to be contained within a scope, such as `\draw_begin:/\draw_end:`.

<hr/> <code>\color_fill:n</code>	<code>\color_fill:n {<color expression>}</code>
<code>\color_stroke:n</code>	Parses the <code><color expression></code> and then activates the resulting color specification for filling or stroking.
<hr/> <code>\color_fill:nn</code>	<code>\color_fill:nn {<model(s)>} {<value(s)>}</code>
<code>\color_stroke:nn</code>	Activates the color specification equivalent to the <code><model(s)></code> and <code><value(s)></code> for filling or stroking.
<hr/> <code>color.sc</code>	When using <code>dvips</code> , this PostScript variables hold the stroke color.

37.6.1 Coloring math mode material

Coloring math mode material using `\color_select:nn(n)` has some restrictions and often leads to spacing issues and/or poor input syntax. Avoiding generating `\mathord` atoms whilst coloring only those parts of the input which are required needs careful handling. The functionality here covers this important use case.

<hr/> <code>\color_math:nn</code>	<code>\color_math:nn {<color expression>}{<content>}</code>
<code>\color_math:nnn</code>	<code>\color_math:nnn {<model(s)>} {<value(s)>} {<content>}</code>
<hr/> <small>New: 2022-01-26</small>	Works as for <code>\color_select:n(n)</code> but applies color only to the math mode <code><content></code> . The function does not generate a group and the <code><content></code> therefore retains its math atom states. Sub/superscripts are also properly handled.
<hr/> <code>\l_color_math_active_tl</code>	This list controls which tokens are considered as math active and should therefore be replaced by their definition during searching for sub/superscripts.
<hr/> <small>New: 2022-01-26</small>	

37.7 Multiple color models

When selecting or setting a color with an explicit model, it is possible to give values for more than one model at one time. This is particularly useful where automated conversion between models does not give the desired outcome. To do this, the list of models and list of values are both subdivided using `/` characters (as for the similar function in `xcolor`). For example, to save a color with explicit `cmymk` and `rgb` values, one could use

```
\color_set:nnn { foo } { cmyk / rgb }
  { 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

The manually-specified conversion will be used in preference to automated calculation whenever the model(s) listed are used: both in expressions and when a fixed model is active.

Similarly, the same syntax can be applied to directly selecting a color.

```
\color_select:nn { cmyk / rgb }
  { 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

Again, this list is used when a fixed model is active: the first entry is used unless there is a fixed model matching one of the other entries.

37.8 Exporting color specifications

The major use of color expressions is in setting typesetting output, but there are other places in which some form of color information is required. These may need data in a different format or using a different model to the internal representation. Thus a set of functions are available to export colors in different formats.

Valid export targets are

- **backend** Two brace groups: the first containing the model, the second containing space-separated values appropriate for the model; this is the format required by backend functions of `expl3`
- **comma-sep-cmyk** Comma-separated cyan-magenta-yellow-black values
- **comma-sep-rgb** Comma-separated red-green-blue values suitable for use as a PDF annotation color
- **HTML** Uppercase two-digit hexadecimal values, expressing a red-green-blue color; the digits are *not* separated
- **space-sep-cmyk** Space-separated cyan-magenta-yellow-black values
- **space-sep-rgb** Space-separated red-green-blue values suitable for use as a PDF annotation color

```
\color_export:nnN \color_export:nnN {<color expression>} {<format>} {<t1>}
```

Parses the `<color expression>` as described earlier, then converts to the `<format>` specified and assigns the data to the `<t1>`.

```
\color_export:nnnN \color_export:nnnN {<model>} {<value(s)>} {<format>} {<t1>}
```

Expresses the combination of `<model>` and `<value(s)>` in an internal representation, then converts to the `<format>` specified and assigns the data to the `<t1>`.

37.9 Creating new color models

Additional color models are required to support specialist workflows, for example those involving separations (see <https://helpx.adobe.com/indesign/using/spot-process-colors.html> for details of the use of separations in print). Color models may be split into families; for the standard device-based color models (`DeviceCMYK`, `DeviceRGB`, `DeviceGray`), these are synonymous. This is not generally the case: see the PDF reference for more details. (Note that `l3color` uses the shorter names `cmyk`, etc.)

`\color_model_new:nnn` `\color_model_new:nnn` $\langle model \rangle$ $\langle family \rangle$ $\langle params \rangle$

Creates a new $\langle model \rangle$ which is derived from the color model $\langle family \rangle$. The latter should be one of

- `DeviceN`
- `ICCBased`
- `Separation`

(The $\langle family \rangle$ may be given in mixed case as in the PDF reference: internally, case of these strings is folded.) Depending on the $\langle family \rangle$, one or more $\langle params \rangle$ are mandatory or optional.

For a `Separation` space, there are three *compulsory* keys.

- **name** The name of the Separation, for example the formal name of a spot color ink. Such a $\langle name \rangle$ may contain spaces, etc., which are not permitted in the $\langle model \rangle$.
- **alternative-model** An alternative device colorspace, one of `cmyk`, `rgb`, `gray` or `CIELAB`. The three parameter-based models work as described above; see below for details of `CIELAB` colors.
- **alternative-values** A comma-separated list of values appropriate to the `alternative-model`. This information is used by the PDF application if the `Separation` is not available.

`CIELAB` color separations are created using the `alternative-model = CIELAB` setting. These colors must also have an `illuminant` key, one of `a`, `c`, `e`, `d50`, `d55`, `d65` or `d75`. The `alternative-values` in this case are the three parameters L^* , a^* and b^* of the `CIELAB` model. Full details of this device-independent color approach are given in the documentation to the `colorspace` package.

`CIELAB` colors *cannot* be converted into other device-dependent color spaces, and as such, mixing can only occur if colors set up using the `CIELAB` model are also given with an alternative parameter-based model. If that is not the case, `l3color` will fallback to using black as the colorant in any mixing.

For a `DeviceN` space, there is one *compulsory* key.

- **names** The names of the components of the `DeviceN` space. Each should be either the $\langle name \rangle$ of a `Separation` model, a process color name (`cyan`, etc.) or the special name `none`.

For a `ICCBased` space, there is one *compulsory* key.

- **file** The name of the file containing the profile.

37.9.1 Color profiles

Color profiles are used to ensure color accuracy by linking to collaboration. Applying a profile can be used to standardise color which is otherwise device-dependence.

`\color_profile_apply:nn` `\color_profile_apply:nn {<profile>} {<model>}`

New: 2021-02-23

This function applies a `<profile>` to one of the device `<models>`. The profile will then apply to all color of the selected `<model>`. The `<profile>` should specify an ICC profile file. The `<model>` has to be one the standard device models: `cmym`, `gray` or `rgb`.

Chapter 38

The l3pdf module Core PDF support

38.1 Objects

38.1.1 Named objects

An *object* name should fully expand to tokens suitable for use in a label-like context.

`\pdf_object_new:n` `\pdf_object_new:n {<object>}`

New: 2022-08-23 Declares *object* as a PDF object. The object may be referenced from this point on, and written later using `\pdf_object_write:nnn`.

`\pdf_object_write:nnn` `\pdf_object_write:nnn {<object>} {<type>} {<content>}`

`\pdf_object_write:nne` Writes the *content* as content of the *object*. Depending on the *type* declared for the object, the format required for the *data* will vary

New: 2022-08-23

`array` A space-separated list of values

`dict` Key–value pairs in the form `/<key> <value>`

`fstream` Two brace groups: `<file name>` and `<file content>`

`stream` Two brace groups: `<attributes (dictionary)>` and `<stream contents>`

`\pdf_object_ref:n` `\pdf_object_ref:n {<object>}`

New: 2021-02-10 Inserts the appropriate information to reference the *object* in for example page resource allocation. If the *object* does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

`\pdf_object_if_exist_p:n` `\pdf_object_if_exist_p:n {<object>}`

`\pdf_object_if_exist:nTF` `\pdf_object_if_exist:nTF {<object>} {<>true code>} {<>false code>}`

New: 2020-05-15 Tests whether an object with name `{<object>}` has been defined.

38.1.2 Indexed objects

Objects can also be created using a pair of $\langle class \rangle$ and $\langle index \rangle$; the $\langle class \rangle$ argument should expand to character tokens, whilst the $\langle index \rangle$ is an $\langle int\ expr \rangle$ and starts at 1. For large families of objects, this approach is more efficient than using individual names.

$\backslash pdf_object_new_indexed:nn$ $\backslash pdf_object_new_indexed:nn$ $\{\langle class \rangle\}$ $\{\langle index \rangle\}$

New: 2024-04-01

Declares a PDF object of $\langle class \rangle$ and $\langle index \rangle$. The object may be referenced from this point on, and written later using $\backslash pdf_object_write_indexed:mnnn$.

$\backslash pdf_object_write_indexed:mnnn$ $\backslash pdf_object_write_indexed:mnnn$ $\{\langle class \rangle\}$ $\{\langle index \rangle\}$ $\{\langle type \rangle\}$ $\{\langle content \rangle\}$
 $\backslash pdf_object_write_indexed:mnnn$

New: 2024-04-01

Writes the $\langle content \rangle$ as content of the object of $\langle class \rangle$ and $\langle index \rangle$. Depending on the $\langle type \rangle$ declared for the object, the format required for the $\langle content \rangle$ will vary

array A space-separated list of values

dict Key-value pairs in the form $/\langle key \rangle \langle value \rangle$

fstream Two brace groups: $\langle file\ name \rangle$ and $\langle file\ content \rangle$

stream Two brace groups: $\langle attributes\ (dictionary) \rangle$ and $\langle stream\ contents \rangle$

$\backslash pdf_object_ref_indexed:nn$ \star $\backslash pdf_object_ref_indexed:nn$ $\{\langle class \rangle\}$ $\{\langle index \rangle\}$

New: 2024-04-01

Inserts the appropriate information to reference the object of $\langle class \rangle$ and $\langle index \rangle$ in for example page resource allocation. If the $\langle class \rangle/\langle index \rangle$ combination does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

38.1.3 General functions

$\backslash pdf_object_unnamed_write:nn$ $\backslash pdf_object_unnamed_write:nn$ $\{\langle type \rangle\}$ $\{\langle content \rangle\}$
 $\backslash pdf_object_unnamed_write:nn$

New: 2021-02-10

Writes the $\langle content \rangle$ as content of an anonymous object. Depending on the $\langle type \rangle$, the format required for the $\langle data \rangle$ will vary

array A space-separated list of values

dict Key-value pairs in the form $/\langle key \rangle \langle value \rangle$

fstream Two brace groups: $\langle attributes\ (dictionary) \rangle$ and $\langle file\ name \rangle$

stream Two brace groups: $\langle attributes\ (dictionary) \rangle$ and $\langle stream\ contents \rangle$

<code>\pdf_object_ref_last: *</code>	<code>\pdf_object_ref_last:</code>
<small>New: 2021-02-10</small>	Inserts the appropriate information to reference the last <code><object></code> created. This is particularly useful for anonymous objects.
<code>\pdf_pageobject_ref:n *</code>	<code>\pdf_pageobject_ref:n {<abspage>}</code>
<small>New: 2021-02-10 Updated: 2024-04-22</small>	Inserts the appropriate information to reference the <code><abspage></code> ; the latter is expanded fully before further processing.

38.2 Version

<code>\pdf_version_compare_p:Nn *</code>	<code>\pdf_version_compare_p:Nn <comparator> {<version>}</code>
<code>\pdf_version_compare:NnTF *</code>	<code>\pdf_version_compare:NnTF <comparator> {<version>} {<true code>} {<false code>}</code>
<small>New: 2021-02-10</small>	

Compares the version of the PDF being created with the `<version>` string specified, using the `<comparator>`. Either the `<true code>` or `<false code>` will be left in the output stream.

<code>\pdf_version_gset:n</code>	<code>\pdf_version_gset:n {<version>}</code>
<code>\pdf_version_min_gset:n</code>	Sets the <code><version></code> of the PDF being created. The <code>min</code> version will not alter the output version unless it is currently lower than the <code><version></code> requested.
<small>New: 2021-02-10</small>	

This function may only be used up to the point where the PDF file is initialised. With dvips it sets `\pdf_version_major:` and `\pdf_version_minor:` and allows to compare the values with `\pdf_version_compare:Nn`, but the PDF version itself still has to be set with the command line option `-dCompatibilityLevel` of `ps2pdf`.

<code>\pdf_version:</code>	<code>\pdf_version:</code>
<code>\pdf_version_major: *</code>	Expands to the currently-active PDF version.
<code>\pdf_version_minor: *</code>	
<small>New: 2021-02-10</small>	

38.3 Page (media) size

<code>\pdf_pagesize_gset:nm</code>	<code>\pdf_pagesize_gset:nm {<width>} {<height>}</code>
<small>New: 2023-01-14</small>	Sets the page size (mediabox) of the PDF being created to the <code><width></code> and <code><height></code> , both of which are <code><dimexpr></code> . The page size can only be set at the start of the output with dvips; with other backends, this can be adjusted on a per-page basis.

38.4 Compression

<code>\pdf_uncompress:</code>	<code>\pdf_uncompress:</code>
<small>New: 2021-02-10</small>	Disables any compression of the PDF, where possible. This function may only be used up to the point where the PDF file is initialised.

38.5 Destinations

Destinations are the places a link jumped too. Unlike the name may suggest they don't described an exact location in the PDF. Instead a destination contains a reference to a page along with an instruction how to display this page. The normally used “XYZ *top left zoom*” for example instructs the viewer to show the page with the given *zoom* and the top left corner at the *top left* coordinates—which then gives the impression that there is an anchor at this position.

If an instruction takes a coordinate, it is calculated by the following commands relative to the location the command is issued. So to get a specific coordinate one has to move the command to the right place.

`\pdf_destination:nn` `\pdf_destination:nn {<name>} {<type or integer>}`

New: 2021-01-03

This creates a destination. `{<type or integer>}` can be one of `fit`, `fith`, `fitv`, `fitb`, `fitbh`, `fitbv`, `fitr`, `xyz` or an integer representing a scale factor in percent. `fitr` here gives only a lightweight version of `/FitR`: The backend code defines `fitr` so that it will with pdfL^AT_EX and LuaL^AT_EX use the coordinates of the surrounding box, with dvips and dvipdfmx it falls back to `fit`. For full control use `\pdf_destination:nnnn`.

The keywords match to the PDF names as described in the following tabular.

Keyword	PDF	Remarks
<code>fit</code>	<code>/Fit</code>	Fits the page to the window
<code>fith</code>	<code>/FitH top</code>	Fits the width of the page to the window
<code>fitv</code>	<code>/FitV left</code>	Fits the height of the page to the window
<code>fitb</code>	<code>/FitB</code>	Fits the page bounding box to the window
<code>fitbh</code>	<code>/FitBH top</code>	Fits the width of the page bounding box to the window.
<code>fitbv</code>	<code>/FitBV left</code>	Fits the height of the page bounding box to the window.
<code>fitr</code>	<code>/FitR left bottom right top</code>	Fits the rectangle specified by the four coordinates to the window (see above for the restrictions)
<code>xyz</code>	<code>/XYZ left top null</code>	Sets a coordinate but doesn't change the zoom.
<code>{<integer>}</code>	<code>/XYZ left top zoom</code>	Sets a coordinate and a zoom meaning <code>{<integer>}%</code> .

`\pdf_destination:nnnn` `\pdf_destination:nnnn {<name>} {<width>} {<height>} {<depth>}`

New: 2021-01-17

This creates a destination with `/FitR` type with the given dimensions relative to the current location. The destination is in a box of size zero, but it doesn't switch to horizontal mode.

Part VII
Implementation

Chapter 39

l3bootstrap implementation

```
1 <*package>
2 <@@=kernel>
```

39.1 The `\pdfstrcmp` primitive in X_{Γ} TeX

Only pdfTeX has a primitive called `\pdfstrcmp`. The X_{Γ} TeX version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the pdfTeX name is “safe”.

```
3 \begingroup\expandafter\expandafter\expandafter\endgroup
4 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
5 \let\pdfstrcmp\strcmp
6 \fi
```

39.2 Loading support Lua code

When LuaTeX is used there are various pieces of Lua code which need to be loaded. The code itself is defined in `l3luatex` and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.

```
7 \begingroup\expandafter\expandafter\expandafter\endgroup
8 \expandafter\ifx\csname directlua\endcsname\relax
9 \else
10 \ifnum\luatexversion<110 %
11 \else
```

For LuaTeX we make sure the basic support is loaded: this is only necessary in plain.

```
12 \begingroup\expandafter\expandafter\expandafter\endgroup
13 \expandafter\ifx\csname newcatcodetable\endcsname\relax
14 \input{ltluatex}%
15 \fi
16 \begingroup\expandafter\expandafter\expandafter\endgroup
17 \expandafter\ifx\csname newluabytecode\endcsname\relax
18 \else
19 \newluabytecode\@expl@luadata@bytecode
20 \fi
21 \directlua{require("expl3")}%
```

As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

```

22 \ifnum 0%
23 \directlua{
24   if status.ini_version then
25     tex.write("1")
26   end
27 }>0 %
28 \everyjob\expandafter{%
29   \the\expandafter\everyjob
30   \csname\detokenize{lua_now:n}\endcsname{require("expl3")}%
31 }%
32 \fi
33 \fi
34 \fi

```

39.3 Engine requirements

The code currently requires ε -TeX, the set of “pdfTeX extensions” *including* `\expanded`, and for Unicode engines the ability to generate arbitrary character tokens by expansion. That is covered by all supported engines since TeX Live 2019, which we therefore use as a baseline for engine and L^AT_EX format support. For LuaTeX, we require at least Lua 5.3 and the `token.set_lua` function. This is available at least since LuaTeX 1.10, which again is the one in TeX Live 2019. (u)pTeX only gained `\ifincname` for TeX Live 2020, but at present that primitive is unused in expl3 so for the present it’s not tested. If and when that changes, we will need to revisit the code here.

```

35 \begingroup
36 \def\next{\endgroup}%
37 \def\ShortText{Required primitives not found}%
38 \def\LongText%
39   {%
40     The L3 programming layer requires the e-TeX primitives and the
41     \LineBreak 'pdfTeX utilities' as described in the README file.
42     \LineBreak
43     These are available in the engines\LineBreak
44     - pdfTeX v1.40.20\LineBreak
45     - XeTeX v0.999991\LineBreak
46     - LuaTeX v1.10\LineBreak
47     - e-(u)pTeX v3.8.2\LineBreak
48     - Prote (2021)\LineBreak
49     or later.\LineBreak
50     \LineBreak
51   }%
52 \ifnum0%
53 \expandafter\ifx\csname luatexversion\endcsname\relax
54 \expandafter\ifx\csname expanded\endcsname\relax\else 1\fi
55 \else
56 \ifnum\luatexversion<110 \else 1\fi
57 \fi
58 =0 %
59 \newlinechar‘^^J %

```

```

60 \def\LineBreak{\noexpand\MessageBreak}%
61 \expandafter\ifx\csname PackageError\endcsname\relax
62 \def\LineBreak{^^J}%
63 \begingroup
64 \lccode'\~='\ \lccode'\}=\' %
65 \lccode'\T='\T\lccode'\H='\H%
66 \catcode'\ =11 %
67 \lowercase{\endgroup\def\PackageError#1#2#3{%
68 \begingroup\errorcontextlines-1\immediate\write0{\errhelp{#3}\def%
69 \      {#1 Error: #2.^^J^^J
70 Type H <return> for immediate help}\def~{\errmessage{%
71 \      }}~\endgroup}}%
72 \fi
73 \edef\next
74 {%
75 \noexpand\PackageError{expl3}{\ShortText}
76 \{\LongText Loading of expl3 will abort!}%
77 \endgroup
78 \noexpand\endinput
79 }%
80 \fi
81 \next

```

39.4 The L^AT_EX3 code environment

The code environment is now set up.

\ExplSyntaxOff Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied **\ExplSyntaxOff** becomes a “do nothing” command until **\ExplSyntaxOn** is used.

```

82 \protected\edef\ExplSyntaxOff
83 {%
84 \protected\def\noexpand\ExplSyntaxOff{%
85 \catcode 9 = \the\catcode 9\relax
86 \catcode 32 = \the\catcode 32\relax
87 \catcode 34 = \the\catcode 34\relax
88 \catcode 58 = \the\catcode 58\relax
89 \catcode 94 = \the\catcode 94\relax
90 \catcode 95 = \the\catcode 95\relax
91 \catcode 124 = \the\catcode 124\relax
92 \catcode 126 = \the\catcode 126\relax
93 \endlinechar = \the\endlinechar\relax
94 \chardef\csname\detokenize{1__kernel_expl_bool}\endcsname = 0\relax
95 }%

```

(End of definition for \ExplSyntaxOff. This function is documented on page 9.)

The code environment is now set up.

```

96 \catcode 9 = 9\relax
97 \catcode 32 = 9\relax
98 \catcode 34 = 12\relax
99 \catcode 58 = 11\relax
100 \catcode 94 = 7\relax
101 \catcode 95 = 11\relax

```



```

102 \catcode 124 = 12\relax
103 \catcode 126 = 10\relax
104 \endlinechar = 32\relax

```

`\l__kernel_expl_bool` The status for code syntax: this is on at present.

```

105 \global\chardef\l__kernel_expl_bool = 1\relax

```

(End of definition for \l__kernel_expl_bool.)

\ExplSyntaxOn The idea here is that multiple `\ExplSyntaxOn` calls are not going to mess up category codes, and that multiple calls to `\ExplSyntaxOff` are also not wasting time. Applying `\ExplSyntaxOn` alters the definition of `\ExplSyntaxOff` and so in package mode this function should not be used until after the end of the loading process!

```

106 \protected \def \ExplSyntaxOn
107 {
108   \bool_if:NF \l__kernel_expl_bool
109   {
110     \cs_set_protected:Npe \ExplSyntaxOff
111     {
112       \char_set_catcode:n { 9 } { \char_value_catcode:n { 9 } }
113       \char_set_catcode:n { 32 } { \char_value_catcode:n { 32 } }
114       \char_set_catcode:n { 34 } { \char_value_catcode:n { 34 } }
115       \char_set_catcode:n { 58 } { \char_value_catcode:n { 58 } }
116       \char_set_catcode:n { 94 } { \char_value_catcode:n { 94 } }
117       \char_set_catcode:n { 95 } { \char_value_catcode:n { 95 } }
118       \char_set_catcode:n { 124 } { \char_value_catcode:n { 124 } }
119       \char_set_catcode:n { 126 } { \char_value_catcode:n { 126 } }
120       \tex_endlinechar:D =
121         \tex_the:D \tex_endlinechar:D \scan_stop:
122       \bool_set_false:N \l__kernel_expl_bool
123       \cs_set_protected:Npn \ExplSyntaxOff { }
124     }
125   }
126   \char_set_catcode_ignore:n { 9 } % tab
127   \char_set_catcode_ignore:n { 32 } % space
128   \char_set_catcode_other:n { 34 } % double quote
129   \char_set_catcode_letter:n { 58 } % colon
130   \char_set_catcode_math_superscript:n { 94 } % circumflex
131   \char_set_catcode_letter:n { 95 } % underscore
132   \char_set_catcode_other:n { 124 } % pipe
133   \char_set_catcode_space:n { 126 } % tilde
134   \tex_endlinechar:D = 32 \scan_stop:
135   \bool_set_true:N \l__kernel_expl_bool
136 }

```

(End of definition for \ExplSyntaxOn. This function is documented on page 9.)

```

137 </package>

```

Chapter 40

l3names implementation

```
138 <*package & tex>
```

The prefix here is `kernel`. A few places need `@@` to be left as is; this is obtained as `@@@`.

```
139 <@@=kernel>
```

The code here simply renames all of the primitives to new, internal, names.

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded `csname` in the hash table.

```
140 \let \tex_global:D \global
```

```
141 \let \tex_let:D \let
```

Everything is inside a (rather long) group, which keeps `__kernel_primitive:NN` trapped.

```
142 \begingroup
```

`__kernel_primitive:NN` A temporary function to actually do the renaming.

```
143 \long \def \__kernel_primitive:NN #1#2
```

```
144 { \tex_global:D \tex_let:D #2 #1 }
```

(End of definition for `__kernel_primitive:NN`.)

To allow extracting “just the names”, a bit of `DocStrip` fiddling.

```
145 </package & tex>
```

```
146 <*names | tex>
```

```
147 <*names | package>
```

In the current incarnation of this module, all `TEX` primitives are given a new name of the form `\tex_oldname:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```
148 \__kernel_primitive:NN \ \tex_space:D
```

```
149 \__kernel_primitive:NN \ / \tex_italiccorrection:D
```

```
150 \__kernel_primitive:NN \- \tex_hyphen:D
```

Now all the other primitives.

```
151 \__kernel_primitive:NN \above \tex_above:D
```

```
152 \__kernel_primitive:NN \abovedisplayshortskip \tex_abovedisplayshortskip:D
```

```
153 \__kernel_primitive:NN \abovedisplayskip \tex_abovedisplayskip:D
```

```
154 \__kernel_primitive:NN \abovewithdelims \tex_abovewithdelims:D
```

155	<code>_kernel_primitive:NN</code>	<code>\accent</code>	<code>\tex_accent:D</code>
156	<code>_kernel_primitive:NN</code>	<code>\adjdemerits</code>	<code>\tex_adjdemerits:D</code>
157	<code>_kernel_primitive:NN</code>	<code>\advance</code>	<code>\tex_advance:D</code>
158	<code>_kernel_primitive:NN</code>	<code>\afterassignment</code>	<code>\tex_afterassignment:D</code>
159	<code>_kernel_primitive:NN</code>	<code>\aftergroup</code>	<code>\tex_aftergroup:D</code>
160	<code>_kernel_primitive:NN</code>	<code>\atop</code>	<code>\tex_atop:D</code>
161	<code>_kernel_primitive:NN</code>	<code>\atopwithdelims</code>	<code>\tex_atopwithdelims:D</code>
162	<code>_kernel_primitive:NN</code>	<code>\badness</code>	<code>\tex_badness:D</code>
163	<code>_kernel_primitive:NN</code>	<code>\baselineskip</code>	<code>\tex_baselineskip:D</code>
164	<code>_kernel_primitive:NN</code>	<code>\batchmode</code>	<code>\tex_batchmode:D</code>
165	<code>_kernel_primitive:NN</code>	<code>\begingroup</code>	<code>\tex_begingroup:D</code>
166	<code>_kernel_primitive:NN</code>	<code>\belowdisplayshortskip</code>	<code>\tex_belowdisplayshortskip:D</code>
167	<code>_kernel_primitive:NN</code>	<code>\belowdisplayskip</code>	<code>\tex_belowdisplayskip:D</code>
168	<code>_kernel_primitive:NN</code>	<code>\binoppenalty</code>	<code>\tex_binoppenalty:D</code>
169	<code>_kernel_primitive:NN</code>	<code>\botmark</code>	<code>\tex_botmark:D</code>
170	<code>_kernel_primitive:NN</code>	<code>\box</code>	<code>\tex_box:D</code>
171	<code>_kernel_primitive:NN</code>	<code>\boxmaxdepth</code>	<code>\tex_boxmaxdepth:D</code>
172	<code>_kernel_primitive:NN</code>	<code>\brokenpenalty</code>	<code>\tex_brokenpenalty:D</code>
173	<code>_kernel_primitive:NN</code>	<code>\catcode</code>	<code>\tex_catcode:D</code>
174	<code>_kernel_primitive:NN</code>	<code>\char</code>	<code>\tex_char:D</code>
175	<code>_kernel_primitive:NN</code>	<code>\chardef</code>	<code>\tex_chardef:D</code>
176	<code>_kernel_primitive:NN</code>	<code>\cleaders</code>	<code>\tex_cleaders:D</code>
177	<code>_kernel_primitive:NN</code>	<code>\closein</code>	<code>\tex_closein:D</code>
178	<code>_kernel_primitive:NN</code>	<code>\closeout</code>	<code>\tex_closeout:D</code>
179	<code>_kernel_primitive:NN</code>	<code>\clubpenalty</code>	<code>\tex_clubpenalty:D</code>
180	<code>_kernel_primitive:NN</code>	<code>\copy</code>	<code>\tex_copy:D</code>
181	<code>_kernel_primitive:NN</code>	<code>\count</code>	<code>\tex_count:D</code>
182	<code>_kernel_primitive:NN</code>	<code>\countdef</code>	<code>\tex_countdef:D</code>
183	<code>_kernel_primitive:NN</code>	<code>\cr</code>	<code>\tex_cr:D</code>
184	<code>_kernel_primitive:NN</code>	<code>\crrcr</code>	<code>\tex_crrcr:D</code>
185	<code>_kernel_primitive:NN</code>	<code>\csname</code>	<code>\tex_csname:D</code>
186	<code>_kernel_primitive:NN</code>	<code>\day</code>	<code>\tex_day:D</code>
187	<code>_kernel_primitive:NN</code>	<code>\deadcycles</code>	<code>\tex_deadcycles:D</code>
188	<code>_kernel_primitive:NN</code>	<code>\def</code>	<code>\tex_def:D</code>
189	<code>_kernel_primitive:NN</code>	<code>\defaultthyphenchar</code>	<code>\tex_defaultthyphenchar:D</code>
190	<code>_kernel_primitive:NN</code>	<code>\defaultskewchar</code>	<code>\tex_defaultskewchar:D</code>
191	<code>_kernel_primitive:NN</code>	<code>\delcode</code>	<code>\tex_delcode:D</code>
192	<code>_kernel_primitive:NN</code>	<code>\delimiter</code>	<code>\tex_delimiter:D</code>
193	<code>_kernel_primitive:NN</code>	<code>\delimiterfactor</code>	<code>\tex_delimiterfactor:D</code>
194	<code>_kernel_primitive:NN</code>	<code>\delimitershortfall</code>	<code>\tex_delimitershortfall:D</code>
195	<code>_kernel_primitive:NN</code>	<code>\dimen</code>	<code>\tex_dimen:D</code>
196	<code>_kernel_primitive:NN</code>	<code>\dimendef</code>	<code>\tex_dimendef:D</code>
197	<code>_kernel_primitive:NN</code>	<code>\discretionary</code>	<code>\tex_discretionary:D</code>
198	<code>_kernel_primitive:NN</code>	<code>\displayindent</code>	<code>\tex_displayindent:D</code>
199	<code>_kernel_primitive:NN</code>	<code>\displaylimits</code>	<code>\tex_displaylimits:D</code>
200	<code>_kernel_primitive:NN</code>	<code>\displaystyle</code>	<code>\tex_displaystyle:D</code>
201	<code>_kernel_primitive:NN</code>	<code>\displaywidowpenalty</code>	<code>\tex_displaywidowpenalty:D</code>
202	<code>_kernel_primitive:NN</code>	<code>\displaywidth</code>	<code>\tex_displaywidth:D</code>
203	<code>_kernel_primitive:NN</code>	<code>\divide</code>	<code>\tex_divide:D</code>
204	<code>_kernel_primitive:NN</code>	<code>\doublehyphendemerits</code>	<code>\tex_doublehyphendemerits:D</code>
205	<code>_kernel_primitive:NN</code>	<code>\dp</code>	<code>\tex_dp:D</code>
206	<code>_kernel_primitive:NN</code>	<code>\dump</code>	<code>\tex_dump:D</code>
207	<code>_kernel_primitive:NN</code>	<code>\edef</code>	<code>\tex_edef:D</code>
208	<code>_kernel_primitive:NN</code>	<code>\else</code>	<code>\tex_else:D</code>

209	<code>_kernel_primitive:NN \emergencystretch</code>	<code>\tex_emergencystretch:D</code>
210	<code>_kernel_primitive:NN \end</code>	<code>\tex_end:D</code>
211	<code>_kernel_primitive:NN \endcsname</code>	<code>\tex_endcsname:D</code>
212	<code>_kernel_primitive:NN \endgroup</code>	<code>\tex_endgroup:D</code>
213	<code>_kernel_primitive:NN \endinput</code>	<code>\tex_endinput:D</code>
214	<code>_kernel_primitive:NN \endlinechar</code>	<code>\tex_endlinechar:D</code>
215	<code>_kernel_primitive:NN \eqno</code>	<code>\tex_eqno:D</code>
216	<code>_kernel_primitive:NN \errhelp</code>	<code>\tex_errhelp:D</code>
217	<code>_kernel_primitive:NN \errmessage</code>	<code>\tex_errmessage:D</code>
218	<code>_kernel_primitive:NN \errorcontextlines</code>	<code>\tex_errorcontextlines:D</code>
219	<code>_kernel_primitive:NN \errorstopmode</code>	<code>\tex_errorstopmode:D</code>
220	<code>_kernel_primitive:NN \escapechar</code>	<code>\tex_escapechar:D</code>
221	<code>_kernel_primitive:NN \everycr</code>	<code>\tex_everycr:D</code>
222	<code>_kernel_primitive:NN \everydisplay</code>	<code>\tex_everydisplay:D</code>
223	<code>_kernel_primitive:NN \everyhbox</code>	<code>\tex_everyhbox:D</code>
224	<code>_kernel_primitive:NN \everyjob</code>	<code>\tex_everyjob:D</code>
225	<code>_kernel_primitive:NN \everymath</code>	<code>\tex_everymath:D</code>
226	<code>_kernel_primitive:NN \everypar</code>	<code>\tex_everypar:D</code>
227	<code>_kernel_primitive:NN \everyvbox</code>	<code>\tex_everyvbox:D</code>
228	<code>_kernel_primitive:NN \exhyphenpenalty</code>	<code>\tex_exhyphenpenalty:D</code>
229	<code>_kernel_primitive:NN \expandafter</code>	<code>\tex_expandafter:D</code>
230	<code>_kernel_primitive:NN \fam</code>	<code>\tex_fam:D</code>
231	<code>_kernel_primitive:NN \fi</code>	<code>\tex_fi:D</code>
232	<code>_kernel_primitive:NN \finalhyphendemerits</code>	<code>\tex_finalhyphendemerits:D</code>
233	<code>_kernel_primitive:NN \firstmark</code>	<code>\tex_firstmark:D</code>
234	<code>_kernel_primitive:NN \floatingpenalty</code>	<code>\tex_floatingpenalty:D</code>
235	<code>_kernel_primitive:NN \font</code>	<code>\tex_font:D</code>
236	<code>_kernel_primitive:NN \fontdimen</code>	<code>\tex_fontdimen:D</code>
237	<code>_kernel_primitive:NN \fontname</code>	<code>\tex_fontname:D</code>
238	<code>_kernel_primitive:NN \futurelet</code>	<code>\tex_futurelet:D</code>
239	<code>_kernel_primitive:NN \gdef</code>	<code>\tex_gdef:D</code>
240	<code>_kernel_primitive:NN \global</code>	<code>\tex_global:D</code>
241	<code>_kernel_primitive:NN \globaldefs</code>	<code>\tex_globaldefs:D</code>
242	<code>_kernel_primitive:NN \halign</code>	<code>\tex_halign:D</code>
243	<code>_kernel_primitive:NN \hangafter</code>	<code>\tex_hangafter:D</code>
244	<code>_kernel_primitive:NN \hangindent</code>	<code>\tex_hangindent:D</code>
245	<code>_kernel_primitive:NN \hbadness</code>	<code>\tex_hbadness:D</code>
246	<code>_kernel_primitive:NN \hbox</code>	<code>\tex_hbox:D</code>
247	<code>_kernel_primitive:NN \hfil</code>	<code>\tex_hfil:D</code>
248	<code>_kernel_primitive:NN \hfill</code>	<code>\tex_hfill:D</code>
249	<code>_kernel_primitive:NN \hfilneg</code>	<code>\tex_hfilneg:D</code>
250	<code>_kernel_primitive:NN \hfuzz</code>	<code>\tex_hfuzz:D</code>
251	<code>_kernel_primitive:NN \hoffset</code>	<code>\tex_hoffset:D</code>
252	<code>_kernel_primitive:NN \holdinginserts</code>	<code>\tex_holdinginserts:D</code>
253	<code>_kernel_primitive:NN \hrule</code>	<code>\tex_hrule:D</code>
254	<code>_kernel_primitive:NN \hsize</code>	<code>\tex_hsize:D</code>
255	<code>_kernel_primitive:NN \hskip</code>	<code>\tex_hskip:D</code>
256	<code>_kernel_primitive:NN \hss</code>	<code>\tex_hss:D</code>
257	<code>_kernel_primitive:NN \ht</code>	<code>\tex_ht:D</code>
258	<code>_kernel_primitive:NN \hyphenation</code>	<code>\tex_hyphenation:D</code>
259	<code>_kernel_primitive:NN \hyphenchar</code>	<code>\tex_hyphenchar:D</code>
260	<code>_kernel_primitive:NN \hyphenpenalty</code>	<code>\tex_hyphenpenalty:D</code>
261	<code>_kernel_primitive:NN \if</code>	<code>\tex_if:D</code>
262	<code>_kernel_primitive:NN \ifcase</code>	<code>\tex_ifcase:D</code>

263	<code>_kernel_primitive:NN \ifcat</code>	<code>\tex_ifcat:D</code>
264	<code>_kernel_primitive:NN \ifdim</code>	<code>\tex_ifdim:D</code>
265	<code>_kernel_primitive:NN \ifeof</code>	<code>\tex_ifeof:D</code>
266	<code>_kernel_primitive:NN \iffalse</code>	<code>\tex_iffalse:D</code>
267	<code>_kernel_primitive:NN \ifhbox</code>	<code>\tex_ifhbox:D</code>
268	<code>_kernel_primitive:NN \ifhmode</code>	<code>\tex_ifhmode:D</code>
269	<code>_kernel_primitive:NN \ifinner</code>	<code>\tex_ifinner:D</code>
270	<code>_kernel_primitive:NN \ifmmode</code>	<code>\tex_ifmmode:D</code>
271	<code>_kernel_primitive:NN \ifnum</code>	<code>\tex_ifnum:D</code>
272	<code>_kernel_primitive:NN \ifodd</code>	<code>\tex_ifodd:D</code>
273	<code>_kernel_primitive:NN \iftrue</code>	<code>\tex_iftrue:D</code>
274	<code>_kernel_primitive:NN \ifvbox</code>	<code>\tex_ifvbox:D</code>
275	<code>_kernel_primitive:NN \ifvmode</code>	<code>\tex_ifvmode:D</code>
276	<code>_kernel_primitive:NN \ifvoid</code>	<code>\tex_ifvoid:D</code>
277	<code>_kernel_primitive:NN \ifx</code>	<code>\tex_ifx:D</code>
278	<code>_kernel_primitive:NN \ignorespaces</code>	<code>\tex_ignorespaces:D</code>
279	<code>_kernel_primitive:NN \immediate</code>	<code>\tex_immediate:D</code>
280	<code>_kernel_primitive:NN \indent</code>	<code>\tex_indent:D</code>
281	<code>_kernel_primitive:NN \input</code>	<code>\tex_input:D</code>
282	<code>_kernel_primitive:NN \inputlineno</code>	<code>\tex_inputlineno:D</code>
283	<code>_kernel_primitive:NN \insert</code>	<code>\tex_insert:D</code>
284	<code>_kernel_primitive:NN \insertpenalties</code>	<code>\tex_insertpenalties:D</code>
285	<code>_kernel_primitive:NN \interlinepenalty</code>	<code>\tex_interlinepenalty:D</code>
286	<code>_kernel_primitive:NN \jobname</code>	<code>\tex_jobname:D</code>
287	<code>_kernel_primitive:NN \kern</code>	<code>\tex_kern:D</code>
288	<code>_kernel_primitive:NN \language</code>	<code>\tex_language:D</code>
289	<code>_kernel_primitive:NN \lastbox</code>	<code>\tex_lastbox:D</code>
290	<code>_kernel_primitive:NN \lastkern</code>	<code>\tex_lastkern:D</code>
291	<code>_kernel_primitive:NN \lastpenalty</code>	<code>\tex_lastpenalty:D</code>
292	<code>_kernel_primitive:NN \lastskip</code>	<code>\tex_lastskip:D</code>
293	<code>_kernel_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
294	<code>_kernel_primitive:NN \leaders</code>	<code>\tex_leaders:D</code>
295	<code>_kernel_primitive:NN \left</code>	<code>\tex_left:D</code>
296	<code>_kernel_primitive:NN \leftthyphenmin</code>	<code>\tex_leftthyphenmin:D</code>
297	<code>_kernel_primitive:NN \leftskip</code>	<code>\tex_leftskip:D</code>
298	<code>_kernel_primitive:NN \leqno</code>	<code>\tex_leqno:D</code>
299	<code>_kernel_primitive:NN \let</code>	<code>\tex_let:D</code>
300	<code>_kernel_primitive:NN \limits</code>	<code>\tex_limits:D</code>
301	<code>_kernel_primitive:NN \linepenalty</code>	<code>\tex_linepenalty:D</code>
302	<code>_kernel_primitive:NN \lineskip</code>	<code>\tex_lineskip:D</code>
303	<code>_kernel_primitive:NN \lineskiplimit</code>	<code>\tex_lineskiplimit:D</code>
304	<code>_kernel_primitive:NN \long</code>	<code>\tex_long:D</code>
305	<code>_kernel_primitive:NN \looseness</code>	<code>\tex_looseness:D</code>
306	<code>_kernel_primitive:NN \lower</code>	<code>\tex_lower:D</code>
307	<code>_kernel_primitive:NN \lowercase</code>	<code>\tex_lowercase:D</code>
308	<code>_kernel_primitive:NN \mag</code>	<code>\tex_mag:D</code>
309	<code>_kernel_primitive:NN \mark</code>	<code>\tex_mark:D</code>
310	<code>_kernel_primitive:NN \mathaccent</code>	<code>\tex_mathaccent:D</code>
311	<code>_kernel_primitive:NN \mathbin</code>	<code>\tex_mathbin:D</code>
312	<code>_kernel_primitive:NN \mathchar</code>	<code>\tex_mathchar:D</code>
313	<code>_kernel_primitive:NN \mathchardef</code>	<code>\tex_mathchardef:D</code>
314	<code>_kernel_primitive:NN \mathchoice</code>	<code>\tex_mathchoice:D</code>
315	<code>_kernel_primitive:NN \mathclose</code>	<code>\tex_mathclose:D</code>
316	<code>_kernel_primitive:NN \mathcode</code>	<code>\tex_mathcode:D</code>

317	<code>__kernel_primitive:NN \mathinner</code>	<code>\tex_mathinner:D</code>
318	<code>__kernel_primitive:NN \mathop</code>	<code>\tex_mathop:D</code>
319	<code>__kernel_primitive:NN \mathopen</code>	<code>\tex_mathopen:D</code>
320	<code>__kernel_primitive:NN \mathord</code>	<code>\tex_mathord:D</code>
321	<code>__kernel_primitive:NN \mathpunct</code>	<code>\tex_mathpunct:D</code>
322	<code>__kernel_primitive:NN \mathrel</code>	<code>\tex_mathrel:D</code>
323	<code>__kernel_primitive:NN \mathsurround</code>	<code>\tex_mathsurround:D</code>
324	<code>__kernel_primitive:NN \maxdeadcycles</code>	<code>\tex_maxdeadcycles:D</code>
325	<code>__kernel_primitive:NN \maxdepth</code>	<code>\tex_maxdepth:D</code>
326	<code>__kernel_primitive:NN \meaning</code>	<code>\tex_meaning:D</code>
327	<code>__kernel_primitive:NN \medmuskip</code>	<code>\tex_medmuskip:D</code>
328	<code>__kernel_primitive:NN \message</code>	<code>\tex_message:D</code>
329	<code>__kernel_primitive:NN \mkern</code>	<code>\tex_mkern:D</code>
330	<code>__kernel_primitive:NN \month</code>	<code>\tex_month:D</code>
331	<code>__kernel_primitive:NN \moveleft</code>	<code>\tex_moveleft:D</code>
332	<code>__kernel_primitive:NN \moveright</code>	<code>\tex_moveright:D</code>
333	<code>__kernel_primitive:NN \mskip</code>	<code>\tex_mskip:D</code>
334	<code>__kernel_primitive:NN \multiply</code>	<code>\tex_multiply:D</code>
335	<code>__kernel_primitive:NN \muskip</code>	<code>\tex_muskip:D</code>
336	<code>__kernel_primitive:NN \muskipdef</code>	<code>\tex_muskipdef:D</code>
337	<code>__kernel_primitive:NN \newlinechar</code>	<code>\tex_newlinechar:D</code>
338	<code>__kernel_primitive:NN \noalign</code>	<code>\tex_noalign:D</code>
339	<code>__kernel_primitive:NN \noboundary</code>	<code>\tex_noboundary:D</code>
340	<code>__kernel_primitive:NN \noexpand</code>	<code>\tex_noexpand:D</code>
341	<code>__kernel_primitive:NN \noindent</code>	<code>\tex_noindent:D</code>
342	<code>__kernel_primitive:NN \nolimits</code>	<code>\tex_nolimits:D</code>
343	<code>__kernel_primitive:NN \nonscript</code>	<code>\tex_nonscript:D</code>
344	<code>__kernel_primitive:NN \nonstopmode</code>	<code>\tex_nonstopmode:D</code>
345	<code>__kernel_primitive:NN \nulldelimiterspace</code>	<code>\tex_nulldelimiterspace:D</code>
346	<code>__kernel_primitive:NN \nullfont</code>	<code>\tex_nullfont:D</code>
347	<code>__kernel_primitive:NN \number</code>	<code>\tex_number:D</code>
348	<code>__kernel_primitive:NN \omit</code>	<code>\tex_omit:D</code>
349	<code>__kernel_primitive:NN \openin</code>	<code>\tex_openin:D</code>
350	<code>__kernel_primitive:NN \openout</code>	<code>\tex_openout:D</code>
351	<code>__kernel_primitive:NN \or</code>	<code>\tex_or:D</code>
352	<code>__kernel_primitive:NN \outer</code>	<code>\tex_outer:D</code>
353	<code>__kernel_primitive:NN \output</code>	<code>\tex_output:D</code>
354	<code>__kernel_primitive:NN \outputpenalty</code>	<code>\tex_outputpenalty:D</code>
355	<code>__kernel_primitive:NN \over</code>	<code>\tex_over:D</code>
356	<code>__kernel_primitive:NN \overfullrule</code>	<code>\tex_overfullrule:D</code>
357	<code>__kernel_primitive:NN \overline</code>	<code>\tex_overline:D</code>
358	<code>__kernel_primitive:NN \overwithdelims</code>	<code>\tex_overwithdelims:D</code>
359	<code>__kernel_primitive:NN \pagedepth</code>	<code>\tex_pagedepth:D</code>
360	<code>__kernel_primitive:NN \pagefilllstretch</code>	<code>\tex_pagefilllstretch:D</code>
361	<code>__kernel_primitive:NN \pagefillstretch</code>	<code>\tex_pagefillstretch:D</code>
362	<code>__kernel_primitive:NN \pagefilstretch</code>	<code>\tex_pagefilstretch:D</code>
363	<code>__kernel_primitive:NN \pagegoal</code>	<code>\tex_pagegoal:D</code>
364	<code>__kernel_primitive:NN \pageshrink</code>	<code>\tex_pageshrink:D</code>
365	<code>__kernel_primitive:NN \pagestretch</code>	<code>\tex_pagestretch:D</code>
366	<code>__kernel_primitive:NN \pagetotal</code>	<code>\tex_pagetotal:D</code>
367	<code>__kernel_primitive:NN \par</code>	<code>\tex_par:D</code>
368	<code>__kernel_primitive:NN \parfillskip</code>	<code>\tex_parfillskip:D</code>
369	<code>__kernel_primitive:NN \parindent</code>	<code>\tex_parindent:D</code>
370	<code>__kernel_primitive:NN \parshape</code>	<code>\tex_parshape:D</code>

371	<code>_kernel_primitive:NN \parskip</code>	<code>\tex_parskip:D</code>
372	<code>_kernel_primitive:NN \patterns</code>	<code>\tex_patterns:D</code>
373	<code>_kernel_primitive:NN \pausing</code>	<code>\tex_pausing:D</code>
374	<code>_kernel_primitive:NN \penalty</code>	<code>\tex_penalty:D</code>
375	<code>_kernel_primitive:NN \postdisplaypenalty</code>	<code>\tex_postdisplaypenalty:D</code>
376	<code>_kernel_primitive:NN \predisplaypenalty</code>	<code>\tex_predisplaypenalty:D</code>
377	<code>_kernel_primitive:NN \preplaysize</code>	<code>\tex_preplaysize:D</code>
378	<code>_kernel_primitive:NN \pretolerance</code>	<code>\tex_pretolerance:D</code>
379	<code>_kernel_primitive:NN \prevdepth</code>	<code>\tex_prevdepth:D</code>
380	<code>_kernel_primitive:NN \prevgraf</code>	<code>\tex_prevgraf:D</code>
381	<code>_kernel_primitive:NN \radical</code>	<code>\tex_radical:D</code>
382	<code>_kernel_primitive:NN \raise</code>	<code>\tex_raise:D</code>
383	<code>_kernel_primitive:NN \read</code>	<code>\tex_read:D</code>
384	<code>_kernel_primitive:NN \relax</code>	<code>\tex_relax:D</code>
385	<code>_kernel_primitive:NN \relpenalty</code>	<code>\tex_relpenalty:D</code>
386	<code>_kernel_primitive:NN \right</code>	<code>\tex_right:D</code>
387	<code>_kernel_primitive:NN \righthyphenmin</code>	<code>\tex_righthyphenmin:D</code>
388	<code>_kernel_primitive:NN \rightskip</code>	<code>\tex_rightskip:D</code>
389	<code>_kernel_primitive:NN \romannumeral</code>	<code>\tex_romannumeral:D</code>
390	<code>_kernel_primitive:NN \scriptfont</code>	<code>\tex_scriptfont:D</code>
391	<code>_kernel_primitive:NN \scriptscriptfont</code>	<code>\tex_scriptscriptfont:D</code>
392	<code>_kernel_primitive:NN \scriptscriptstyle</code>	<code>\tex_scriptscriptstyle:D</code>
393	<code>_kernel_primitive:NN \scriptspace</code>	<code>\tex_scriptspace:D</code>
394	<code>_kernel_primitive:NN \scriptstyle</code>	<code>\tex_scriptstyle:D</code>
395	<code>_kernel_primitive:NN \scrollmode</code>	<code>\tex_scrollmode:D</code>
396	<code>_kernel_primitive:NN \setbox</code>	<code>\tex_setbox:D</code>
397	<code>_kernel_primitive:NN \setlanguage</code>	<code>\tex_setlanguage:D</code>
398	<code>_kernel_primitive:NN \sfcode</code>	<code>\tex_sfcode:D</code>
399	<code>_kernel_primitive:NN \shipout</code>	<code>\tex_shipout:D</code>
400	<code>_kernel_primitive:NN \show</code>	<code>\tex_show:D</code>
401	<code>_kernel_primitive:NN \showbox</code>	<code>\tex_showbox:D</code>
402	<code>_kernel_primitive:NN \showboxbreadth</code>	<code>\tex_showboxbreadth:D</code>
403	<code>_kernel_primitive:NN \showboxdepth</code>	<code>\tex_showboxdepth:D</code>
404	<code>_kernel_primitive:NN \showlists</code>	<code>\tex_showlists:D</code>
405	<code>_kernel_primitive:NN \showthe</code>	<code>\tex_showthe:D</code>
406	<code>_kernel_primitive:NN \skewchar</code>	<code>\tex_skewchar:D</code>
407	<code>_kernel_primitive:NN \skip</code>	<code>\tex_skip:D</code>
408	<code>_kernel_primitive:NN \skipdef</code>	<code>\tex_skipdef:D</code>
409	<code>_kernel_primitive:NN \spacefactor</code>	<code>\tex_spacefactor:D</code>
410	<code>_kernel_primitive:NN \spaceskip</code>	<code>\tex_spaceskip:D</code>
411	<code>_kernel_primitive:NN \span</code>	<code>\tex_span:D</code>
412	<code>_kernel_primitive:NN \special</code>	<code>\tex_special:D</code>
413	<code>_kernel_primitive:NN \splitbotmark</code>	<code>\tex_splitbotmark:D</code>
414	<code>_kernel_primitive:NN \splitfirstmark</code>	<code>\tex_splitfirstmark:D</code>
415	<code>_kernel_primitive:NN \splitmaxdepth</code>	<code>\tex_splitmaxdepth:D</code>
416	<code>_kernel_primitive:NN \splittopskip</code>	<code>\tex_splittopskip:D</code>
417	<code>_kernel_primitive:NN \string</code>	<code>\tex_string:D</code>
418	<code>_kernel_primitive:NN \tabskip</code>	<code>\tex_tabskip:D</code>
419	<code>_kernel_primitive:NN \textfont</code>	<code>\tex_textfont:D</code>
420	<code>_kernel_primitive:NN \textstyle</code>	<code>\tex_textstyle:D</code>
421	<code>_kernel_primitive:NN \the</code>	<code>\tex_the:D</code>
422	<code>_kernel_primitive:NN \thickmuskip</code>	<code>\tex_thickmuskip:D</code>
423	<code>_kernel_primitive:NN \thinmuskip</code>	<code>\tex_thinmuskip:D</code>
424	<code>_kernel_primitive:NN \time</code>	<code>\tex_time:D</code>

425	<code>_kernel_primitive:NN \toks</code>	<code>\tex_toks:D</code>
426	<code>_kernel_primitive:NN \toksdef</code>	<code>\tex_toksdef:D</code>
427	<code>_kernel_primitive:NN \tolerance</code>	<code>\tex_tolerance:D</code>
428	<code>_kernel_primitive:NN \topmark</code>	<code>\tex_topmark:D</code>
429	<code>_kernel_primitive:NN \topskip</code>	<code>\tex_topskip:D</code>
430	<code>_kernel_primitive:NN \tracingcommands</code>	<code>\tex_tracingcommands:D</code>
431	<code>_kernel_primitive:NN \tracinglostchars</code>	<code>\tex_tracinglostchars:D</code>
432	<code>_kernel_primitive:NN \tracingmacros</code>	<code>\tex_tracingmacros:D</code>
433	<code>_kernel_primitive:NN \tracingonline</code>	<code>\tex_tracingonline:D</code>
434	<code>_kernel_primitive:NN \tracingoutput</code>	<code>\tex_tracingoutput:D</code>
435	<code>_kernel_primitive:NN \tracingpages</code>	<code>\tex_tracingpages:D</code>
436	<code>_kernel_primitive:NN \tracingparagraphs</code>	<code>\tex_tracingparagraphs:D</code>
437	<code>_kernel_primitive:NN \tracingrestores</code>	<code>\tex_tracingrestores:D</code>
438	<code>_kernel_primitive:NN \tracingstats</code>	<code>\tex_tracingstats:D</code>
439	<code>_kernel_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
440	<code>_kernel_primitive:NN \uchyph</code>	<code>\tex_uchyph:D</code>
441	<code>_kernel_primitive:NN \underline</code>	<code>\tex_underline:D</code>
442	<code>_kernel_primitive:NN \unhbox</code>	<code>\tex_unhbox:D</code>
443	<code>_kernel_primitive:NN \unhcopy</code>	<code>\tex_unhcopy:D</code>
444	<code>_kernel_primitive:NN \unkern</code>	<code>\tex_unkern:D</code>
445	<code>_kernel_primitive:NN \unpenalty</code>	<code>\tex_unpenalty:D</code>
446	<code>_kernel_primitive:NN \unskip</code>	<code>\tex_unskip:D</code>
447	<code>_kernel_primitive:NN \unvbox</code>	<code>\tex_unvbox:D</code>
448	<code>_kernel_primitive:NN \unvcopy</code>	<code>\tex_unvcopy:D</code>
449	<code>_kernel_primitive:NN \uppercase</code>	<code>\tex_uppercase:D</code>
450	<code>_kernel_primitive:NN \vadjust</code>	<code>\tex_vadjust:D</code>
451	<code>_kernel_primitive:NN \valign</code>	<code>\tex_valign:D</code>
452	<code>_kernel_primitive:NN \vbadness</code>	<code>\tex_vbadness:D</code>
453	<code>_kernel_primitive:NN \vbox</code>	<code>\tex_vbox:D</code>
454	<code>_kernel_primitive:NN \vcenter</code>	<code>\tex_vcenter:D</code>
455	<code>_kernel_primitive:NN \vfil</code>	<code>\tex_vfil:D</code>
456	<code>_kernel_primitive:NN \vfill</code>	<code>\tex_vfill:D</code>
457	<code>_kernel_primitive:NN \vfilneg</code>	<code>\tex_vfilneg:D</code>
458	<code>_kernel_primitive:NN \vfuzz</code>	<code>\tex_vfuzz:D</code>
459	<code>_kernel_primitive:NN \voffset</code>	<code>\tex_voffset:D</code>
460	<code>_kernel_primitive:NN \vrule</code>	<code>\tex_vrule:D</code>
461	<code>_kernel_primitive:NN \vsize</code>	<code>\tex_vsize:D</code>
462	<code>_kernel_primitive:NN \vskip</code>	<code>\tex_vskip:D</code>
463	<code>_kernel_primitive:NN \vsplit</code>	<code>\tex_vsplit:D</code>
464	<code>_kernel_primitive:NN \vss</code>	<code>\tex_vss:D</code>
465	<code>_kernel_primitive:NN \vtop</code>	<code>\tex_vtop:D</code>
466	<code>_kernel_primitive:NN \wd</code>	<code>\tex_wd:D</code>
467	<code>_kernel_primitive:NN \widowpenalty</code>	<code>\tex_widowpenalty:D</code>
468	<code>_kernel_primitive:NN \write</code>	<code>\tex_write:D</code>
469	<code>_kernel_primitive:NN \xdef</code>	<code>\tex_xdef:D</code>
470	<code>_kernel_primitive:NN \xleaders</code>	<code>\tex_xleaders:D</code>
471	<code>_kernel_primitive:NN \xspaceskip</code>	<code>\tex_xspaceskip:D</code>
472	<code>_kernel_primitive:NN \year</code>	<code>\tex_year:D</code>

Primitives introduced by ϵ -TeX.

473	<code>_kernel_primitive:NN \beginL</code>	<code>\tex_beginL:D</code>
474	<code>_kernel_primitive:NN \beginR</code>	<code>\tex_beginR:D</code>
475	<code>_kernel_primitive:NN \botmarks</code>	<code>\tex_botmarks:D</code>
476	<code>_kernel_primitive:NN \clubpenalties</code>	<code>\tex_clubpenalties:D</code>
477	<code>_kernel_primitive:NN \currentgrouplevel</code>	<code>\tex_currentgrouplevel:D</code>

478	<code>_kernel_primitive:NN</code>	<code>\currentgrouptype</code>	<code>\tex_currentgrouptype:D</code>
479	<code>_kernel_primitive:NN</code>	<code>\currentifbranch</code>	<code>\tex_currentifbranch:D</code>
480	<code>_kernel_primitive:NN</code>	<code>\currentiflevel</code>	<code>\tex_currentiflevel:D</code>
481	<code>_kernel_primitive:NN</code>	<code>\currentifttype</code>	<code>\tex_currentifttype:D</code>
482	<code>_kernel_primitive:NN</code>	<code>\detokenize</code>	<code>\tex_detokenize:D</code>
483	<code>_kernel_primitive:NN</code>	<code>\dimexpr</code>	<code>\tex_dimexpr:D</code>
484	<code>_kernel_primitive:NN</code>	<code>\displaywidowpenalties</code>	<code>\tex_displaywidowpenalties:D</code>
485	<code>_kernel_primitive:NN</code>	<code>\endL</code>	<code>\tex_endL:D</code>
486	<code>_kernel_primitive:NN</code>	<code>\endR</code>	<code>\tex_endR:D</code>
487	<code>_kernel_primitive:NN</code>	<code>\eTeXrevision</code>	<code>\tex_eTeXrevision:D</code>
488	<code>_kernel_primitive:NN</code>	<code>\eTeXversion</code>	<code>\tex_eTeXversion:D</code>
489	<code>_kernel_primitive:NN</code>	<code>\everyeof</code>	<code>\tex_everyeof:D</code>
490	<code>_kernel_primitive:NN</code>	<code>\firstmarks</code>	<code>\tex_firstmarks:D</code>
491	<code>_kernel_primitive:NN</code>	<code>\fontcharpd</code>	<code>\tex_fontcharpd:D</code>
492	<code>_kernel_primitive:NN</code>	<code>\fontcharht</code>	<code>\tex_fontcharht:D</code>
493	<code>_kernel_primitive:NN</code>	<code>\fontcharic</code>	<code>\tex_fontcharic:D</code>
494	<code>_kernel_primitive:NN</code>	<code>\fontcharwd</code>	<code>\tex_fontcharwd:D</code>
495	<code>_kernel_primitive:NN</code>	<code>\glueexpr</code>	<code>\tex_glueexpr:D</code>
496	<code>_kernel_primitive:NN</code>	<code>\glueshrink</code>	<code>\tex_glueshrink:D</code>
497	<code>_kernel_primitive:NN</code>	<code>\glueshrinkorder</code>	<code>\tex_glueshrinkorder:D</code>
498	<code>_kernel_primitive:NN</code>	<code>\gluestretch</code>	<code>\tex_gluestretch:D</code>
499	<code>_kernel_primitive:NN</code>	<code>\gluestretchorder</code>	<code>\tex_gluestretchorder:D</code>
500	<code>_kernel_primitive:NN</code>	<code>\gluetomu</code>	<code>\tex_gluetomu:D</code>
501	<code>_kernel_primitive:NN</code>	<code>\ifcsname</code>	<code>\tex_ifcsname:D</code>
502	<code>_kernel_primitive:NN</code>	<code>\ifdefined</code>	<code>\tex_ifdefined:D</code>
503	<code>_kernel_primitive:NN</code>	<code>\iffontchar</code>	<code>\tex_iffontchar:D</code>
504	<code>_kernel_primitive:NN</code>	<code>\interactionmode</code>	<code>\tex_interactionmode:D</code>
505	<code>_kernel_primitive:NN</code>	<code>\interlinepenalties</code>	<code>\tex_interlinepenalties:D</code>
506	<code>_kernel_primitive:NN</code>	<code>\lastlinefit</code>	<code>\tex_lastlinefit:D</code>
507	<code>_kernel_primitive:NN</code>	<code>\lastnodetype</code>	<code>\tex_lastnodetype:D</code>
508	<code>_kernel_primitive:NN</code>	<code>\marks</code>	<code>\tex_marks:D</code>
509	<code>_kernel_primitive:NN</code>	<code>\middle</code>	<code>\tex_middle:D</code>
510	<code>_kernel_primitive:NN</code>	<code>\muexpr</code>	<code>\tex_muexpr:D</code>
511	<code>_kernel_primitive:NN</code>	<code>\mutoglua</code>	<code>\tex_mutoglua:D</code>
512	<code>_kernel_primitive:NN</code>	<code>\numexpr</code>	<code>\tex_numexpr:D</code>
513	<code>_kernel_primitive:NN</code>	<code>\pagediscards</code>	<code>\tex_pagediscards:D</code>
514	<code>_kernel_primitive:NN</code>	<code>\parshapedimen</code>	<code>\tex_parshapedimen:D</code>
515	<code>_kernel_primitive:NN</code>	<code>\parshapeindent</code>	<code>\tex_parshapeindent:D</code>
516	<code>_kernel_primitive:NN</code>	<code>\parshaplength</code>	<code>\tex_parshaplength:D</code>
517	<code>_kernel_primitive:NN</code>	<code>\predisplaydirection</code>	<code>\tex_predisplaydirection:D</code>
518	<code>_kernel_primitive:NN</code>	<code>\protected</code>	<code>\tex_protected:D</code>
519	<code>_kernel_primitive:NN</code>	<code>\readline</code>	<code>\tex_readline:D</code>
520	<code>_kernel_primitive:NN</code>	<code>\savinghyphcodes</code>	<code>\tex_savinghyphcodes:D</code>
521	<code>_kernel_primitive:NN</code>	<code>\savingvdiscards</code>	<code>\tex_savingvdiscards:D</code>
522	<code>_kernel_primitive:NN</code>	<code>\scantokens</code>	<code>\tex_scantokens:D</code>
523	<code>_kernel_primitive:NN</code>	<code>\showgroups</code>	<code>\tex_showgroups:D</code>
524	<code>_kernel_primitive:NN</code>	<code>\showifs</code>	<code>\tex_showifs:D</code>
525	<code>_kernel_primitive:NN</code>	<code>\showtokens</code>	<code>\tex_showtokens:D</code>
526	<code>_kernel_primitive:NN</code>	<code>\splitbotmarks</code>	<code>\tex_splitbotmarks:D</code>
527	<code>_kernel_primitive:NN</code>	<code>\splitdiscards</code>	<code>\tex_splitdiscards:D</code>
528	<code>_kernel_primitive:NN</code>	<code>\splitfirstmarks</code>	<code>\tex_splitfirstmarks:D</code>
529	<code>_kernel_primitive:NN</code>	<code>\TeXeTstate</code>	<code>\tex_TeXeTstate:D</code>
530	<code>_kernel_primitive:NN</code>	<code>\topmarks</code>	<code>\tex_topmarks:D</code>
531	<code>_kernel_primitive:NN</code>	<code>\tracingassigns</code>	<code>\tex_tracingassigns:D</code>

```

532 \__kernel_primitive:NN \tracinggroups \tex_tracinggroups:D
533 \__kernel_primitive:NN \tracingifs \tex_tracingifs:D
534 \__kernel_primitive:NN \tracingnesting \tex_tracingnesting:D
535 \__kernel_primitive:NN \tracingscantokens \tex_tracingscantokens:D
536 \__kernel_primitive:NN \unexpanded \tex_unexpanded:D
537 \__kernel_primitive:NN \unless \tex_unless:D
538 \__kernel_primitive:NN \widowpenalties \tex_widowpenalties:D

```

Post- ϵ -TeX primitives do not always end up with the same name in all engines, if indeed they are available cross-engine anyway. We therefore take the approach of preferring the shortest name that makes sense. First, we deal with the primitives introduced by pdfTeX which directly relate to PDF output: these are copied with the names unchanged.

```

539 \__kernel_primitive:NN \pdfannot \tex_pdfannot:D
540 \__kernel_primitive:NN \pdfcatalog \tex_pdfcatalog:D
541 \__kernel_primitive:NN \pdfcompresslevel \tex_pdfcompresslevel:D
542 \__kernel_primitive:NN \pdfcolorstack \tex_pdfcolorstack:D
543 \__kernel_primitive:NN \pdfcolorstackinit \tex_pdfcolorstackinit:D
544 \__kernel_primitive:NN \pdfdecimaldigits \tex_pdfdecimaldigits:D
545 \__kernel_primitive:NN \pdfdest \tex_pdfdest:D
546 \__kernel_primitive:NN \pdfdestmargin \tex_pdfdestmargin:D
547 \__kernel_primitive:NN \pdfendlink \tex_pdfendlink:D
548 \__kernel_primitive:NN \pdfendthread \tex_pdfendthread:D
549 \__kernel_primitive:NN \pdffakespace \tex_pdffakespace:D
550 \__kernel_primitive:NN \pdffontattr \tex_pdffontattr:D
551 \__kernel_primitive:NN \pdffontname \tex_pdffontname:D
552 \__kernel_primitive:NN \pdffontobjnum \tex_pdffontobjnum:D
553 \__kernel_primitive:NN \pdfgamma \tex_pdfgamma:D
554 \__kernel_primitive:NN \pdfgentounicode \tex_pdfgentounicode:D
555 \__kernel_primitive:NN \pdfglyphtounicode \tex_pdfglyphtounicode:D
556 \__kernel_primitive:NN \pdfhorigin \tex_pdfhorigin:D
557 \__kernel_primitive:NN \pdfimageapplygamma \tex_pdfimageapplygamma:D
558 \__kernel_primitive:NN \pdfimagegamma \tex_pdfimagegamma:D
559 \__kernel_primitive:NN \pdfimagehicolor \tex_pdfimagehicolor:D
560 \__kernel_primitive:NN \pdfimageresolution \tex_pdfimageresolution:D
561 \__kernel_primitive:NN \pdfincludechars \tex_pdfincludechars:D
562 \__kernel_primitive:NN \pdfinclusioncopyfonts \tex_pdfinclusioncopyfonts:D
563 \__kernel_primitive:NN \pdfinclusionerrorlevel
564 \tex_pdfinclusionerrorlevel:D
565 \__kernel_primitive:NN \pdfinfo \tex_pdfinfo:D
566 \__kernel_primitive:NN \pdfinfoomitdate \tex_pdfinfoomitdate:D
567 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
568 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
569 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
570 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
571 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
572 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
573 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
574 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
575 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
576 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
577 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
578 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
579 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D
580 \__kernel_primitive:NN \pdfinterwordsoff \tex_pdfinterwordsoff:D

```

```

581 \__kernel_primitive:NN \pdfmajorversion \tex_pdfmajorversion:D
582 \__kernel_primitive:NN \pdfminorversion \tex_pdfminorversion:D
583 \__kernel_primitive:NN \pdfnames \tex_pdfnames:D
584 \__kernel_primitive:NN \pdfnobluiltintounicode \tex_pdfnobluiltintounicode:D
585 \__kernel_primitive:NN \pdfobj \tex_pdfobj:D
586 \__kernel_primitive:NN \pdfobjcompresslevel \tex_pdfobjcompresslevel:D
587 \__kernel_primitive:NN \pdfomitcharset \tex_pdfomitcharset:D
588 \__kernel_primitive:NN \pdfoutline \tex_pdfoutline:D
589 \__kernel_primitive:NN \pdfoutput \tex_pdfoutput:D
590 \__kernel_primitive:NN \pdfpageattr \tex_pdfpageattr:D
591 \__kernel_primitive:NN \pdfpagesattr \tex_pdfpagesattr:D
592 \__kernel_primitive:NN \pdfpagebox \tex_pdfpagebox:D
593 \__kernel_primitive:NN \pdfpageref \tex_pdfpageref:D
594 \__kernel_primitive:NN \pdfpageresources \tex_pdfpageresources:D
595 \__kernel_primitive:NN \pdfpagesattr \tex_pdfpagesattr:D
596 \__kernel_primitive:NN \pdfrefobj \tex_pdfrefobj:D
597 \__kernel_primitive:NN \pdfrefxform \tex_pdfrefxform:D
598 \__kernel_primitive:NN \pdfrefximage \tex_pdfrefximage:D
599 \__kernel_primitive:NN \pdfrestore \tex_pdfrestore:D
600 \__kernel_primitive:NN \pdfretval \tex_pdfretval:D
601 \__kernel_primitive:NN \pdfrunninglinkoff \tex_pdfrunninglinkoff:D
602 \__kernel_primitive:NN \pdfrunninglinkon \tex_pdfrunninglinkon:D
603 \__kernel_primitive:NN \pdfsave \tex_pdfsave:D
604 \__kernel_primitive:NN \pdfsetmatrix \tex_pdfsetmatrix:D
605 \__kernel_primitive:NN \pdfstartlink \tex_pdfstartlink:D
606 \__kernel_primitive:NN \pdfstartthread \tex_pdfstartthread:D
607 \__kernel_primitive:NN \pdfsuppressptexinfo \tex_pdfsuppressptexinfo:D
608 \__kernel_primitive:NN \pdfsuppresswarningdupdest
609 \tex_pdfsuppresswarningdupdest:D
610 \__kernel_primitive:NN \pdfsuppresswarningdupmap
611 \tex_pdfsuppresswarningdupmap:D
612 \__kernel_primitive:NN \pdfsuppresswarningpagegroup
613 \tex_pdfsuppresswarningpagegroup:D
614 \__kernel_primitive:NN \pdfthread \tex_pdfthread:D
615 \__kernel_primitive:NN \pdfthreadmargin \tex_pdfthreadmargin:D
616 \__kernel_primitive:NN \pdftrailer \tex_pdftrailer:D
617 \__kernel_primitive:NN \pdftrailerid \tex_pdftrailerid:D
618 \__kernel_primitive:NN \pdfuniqueresname \tex_pdfuniqueresname:D
619 \__kernel_primitive:NN \pdfvorigin \tex_pdfvorigin:D
620 \__kernel_primitive:NN \pdfxform \tex_pdfxform:D
621 \__kernel_primitive:NN \pdfxformname \tex_pdfxformname:D
622 \__kernel_primitive:NN \pdfximage \tex_pdfximage:D
623 \__kernel_primitive:NN \pdfximagebbox \tex_pdfximagebbox:D

```

These are not related to PDF output and either already appear in other engines without the `\pdf` prefix, or might reasonably do so at some future stage. We therefore drop the leading `pdf` here.

```

624 \__kernel_primitive:NN \ifpdfabsdim \tex_ifabsdim:D
625 \__kernel_primitive:NN \ifpdfabsnum \tex_ifabsnum:D
626 \__kernel_primitive:NN \ifpdfprimitive \tex_ifprimitive:D
627 \__kernel_primitive:NN \pdfadjustinterwordglue
628 \tex_adjustinterwordglue:D
629 \__kernel_primitive:NN \pdfadjustspacing \tex_adjustspacing:D
630 \__kernel_primitive:NN \pdfappendkern \tex_appendkern:D

```

631	<code>_kernel_primitive:NN</code>	<code>\pdfcopyfont</code>	<code>\tex_copyfont:D</code>
632	<code>_kernel_primitive:NN</code>	<code>\pdfcreationdate</code>	<code>\tex_creationdate:D</code>
633	<code>_kernel_primitive:NN</code>	<code>\pdfdraftmode</code>	<code>\tex_draftmode:D</code>
634	<code>_kernel_primitive:NN</code>	<code>\pdfeachlinedepth</code>	<code>\tex_eachlinedepth:D</code>
635	<code>_kernel_primitive:NN</code>	<code>\pdfeachlineheight</code>	<code>\tex_eachlineheight:D</code>
636	<code>_kernel_primitive:NN</code>	<code>\pdfelapsedtime</code>	<code>\tex_elapsedtime:D</code>
637	<code>_kernel_primitive:NN</code>	<code>\pdfescapehex</code>	<code>\tex_escapehex:D</code>
638	<code>_kernel_primitive:NN</code>	<code>\pdfescapename</code>	<code>\tex_escapename:D</code>
639	<code>_kernel_primitive:NN</code>	<code>\pdfescapestring</code>	<code>\tex_escapestring:D</code>
640	<code>_kernel_primitive:NN</code>	<code>\pdffirstlineheight</code>	<code>\tex_firstlineheight:D</code>
641	<code>_kernel_primitive:NN</code>	<code>\pdffontexpand</code>	<code>\tex_fontexpand:D</code>
642	<code>_kernel_primitive:NN</code>	<code>\pdffontsize</code>	<code>\tex_fontsize:D</code>
643	<code>_kernel_primitive:NN</code>	<code>\pdfignoreddimen</code>	<code>\tex_ignoreddimen:D</code>
644	<code>_kernel_primitive:NN</code>	<code>\pdfinsertht</code>	<code>\tex_insertht:D</code>
645	<code>_kernel_primitive:NN</code>	<code>\pdfastlinedepth</code>	<code>\tex_astlinedepth:D</code>
646	<code>_kernel_primitive:NN</code>	<code>\pdfastmatch</code>	<code>\tex_astmatch:D</code>
647	<code>_kernel_primitive:NN</code>	<code>\pdfastxpos</code>	<code>\tex_astxpos:D</code>
648	<code>_kernel_primitive:NN</code>	<code>\pdfastypos</code>	<code>\tex_astypos:D</code>
649	<code>_kernel_primitive:NN</code>	<code>\pdfmatch</code>	<code>\tex_match:D</code>
650	<code>_kernel_primitive:NN</code>	<code>\pdfnoligatures</code>	<code>\tex_noligatures:D</code>
651	<code>_kernel_primitive:NN</code>	<code>\pdfnormaldeviate</code>	<code>\tex_normaldeviate:D</code>
652	<code>_kernel_primitive:NN</code>	<code>\pdfpageheight</code>	<code>\tex_pageheight:D</code>
653	<code>_kernel_primitive:NN</code>	<code>\pdfpagewidth</code>	<code>\tex_pagewidth:D</code>
654	<code>_kernel_primitive:NN</code>	<code>\pdfpkmode</code>	<code>\tex_pkmode:D</code>
655	<code>_kernel_primitive:NN</code>	<code>\pdfpkresolution</code>	<code>\tex_pkresolution:D</code>
656	<code>_kernel_primitive:NN</code>	<code>\pdfprimitive</code>	<code>\tex_primitive:D</code>
657	<code>_kernel_primitive:NN</code>	<code>\pdfprependkern</code>	<code>\tex_prependkern:D</code>
658	<code>_kernel_primitive:NN</code>	<code>\pdfprotrudechars</code>	<code>\tex_protrudechars:D</code>
659	<code>_kernel_primitive:NN</code>	<code>\pdfpxdimen</code>	<code>\tex_pxdimen:D</code>
660	<code>_kernel_primitive:NN</code>	<code>\pdfrandomseed</code>	<code>\tex_randomseed:D</code>
661	<code>_kernel_primitive:NN</code>	<code>\pdfresettimer</code>	<code>\tex_resettimer:D</code>
662	<code>_kernel_primitive:NN</code>	<code>\pdfsavepos</code>	<code>\tex_savepos:D</code>
663	<code>_kernel_primitive:NN</code>	<code>\pdfsetrandomseed</code>	<code>\tex_setrandomseed:D</code>
664	<code>_kernel_primitive:NN</code>	<code>\pdfshellescape</code>	<code>\tex_shellescape:D</code>
665	<code>_kernel_primitive:NN</code>	<code>\pdftracingfonts</code>	<code>\tex_tracingfonts:D</code>
666	<code>_kernel_primitive:NN</code>	<code>\pdfunescapehex</code>	<code>\tex_unescapehex:D</code>
667	<code>_kernel_primitive:NN</code>	<code>\pdfuniformdeviate</code>	<code>\tex_uniformdeviate:D</code>

The version primitives are not related to PDF mode but are pdfTeX-specific, so again are carried forward unchanged.

668	<code>_kernel_primitive:NN</code>	<code>\pdfptextbanner</code>	<code>\tex_pdfptextbanner:D</code>
669	<code>_kernel_primitive:NN</code>	<code>\pdfptextrevision</code>	<code>\tex_pdfptextrevision:D</code>
670	<code>_kernel_primitive:NN</code>	<code>\pdfptextversion</code>	<code>\tex_pdfptextversion:D</code>

These ones appear in pdfTeX but don't have pdf in the name at all: no decisions to make.

671	<code>_kernel_primitive:NN</code>	<code>\efcode</code>	<code>\tex_efcode:D</code>
672	<code>_kernel_primitive:NN</code>	<code>\ifincsname</code>	<code>\tex_ifincsname:D</code>
673	<code>_kernel_primitive:NN</code>	<code>\knaccode</code>	<code>\tex_knaccode:D</code>
674	<code>_kernel_primitive:NN</code>	<code>\knbccode</code>	<code>\tex_knbccode:D</code>
675	<code>_kernel_primitive:NN</code>	<code>\knbscode</code>	<code>\tex_knbscode:D</code>
676	<code>_kernel_primitive:NN</code>	<code>\leftmarginkern</code>	<code>\tex_leftmarginkern:D</code>
677	<code>_kernel_primitive:NN</code>	<code>\letterspacefont</code>	<code>\tex_letterspacefont:D</code>
678	<code>_kernel_primitive:NN</code>	<code>\lpcode</code>	<code>\tex_lpcode:D</code>
679	<code>_kernel_primitive:NN</code>	<code>\quitvmode</code>	<code>\tex_quitvmode:D</code>

```

680 \__kernel_primitive:NN \rightmarginkern \tex_rightmarginkern:D
681 \__kernel_primitive:NN \rpxcode \tex_rpxcode:D
682 \__kernel_primitive:NN \shbscode \tex_shbscode:D
683 \__kernel_primitive:NN \stbscode \tex_stbscode:D
684 \__kernel_primitive:NN \synctex \tex_synctex:D
685 \__kernel_primitive:NN \tagcode \tex_tagcode:D

```

Post pdfTeX primitive availability gets more complex. Both XeTeX and LuaTeX have varying names for some primitives from pdfTeX. Particularly for LuaTeX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```

686 </names | package>
687 <*package>
688 \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
689 \tex_long:D \tex_def:D \use_none:n #1 { }
690 \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
691 {
692   \tex_ifdefined:D #1
693   \tex_expandafter:D \use_ii:nn
694   \tex_fi:D
695   \use_none:n { \tex_global:D \tex_let:D #2 #1 }
696 }
697 </package>
698 <*names | package>

```

Some pdfTeX primitives are handled here because they got dropped in LuaTeX but the corresponding internal names are emulated later. The Lua code is already loaded at this point, so we shouldn't overwrite them.

```

699 \__kernel_primitive:NN \pdfstrcmp \tex_strcmp:D
700 \__kernel_primitive:NN \pdffilesize \tex_filesize:D
701 \__kernel_primitive:NN \pdfmdfivesum \tex_mdfivesum:D
702 \__kernel_primitive:NN \pdffilemoddate \tex_filemoddate:D
703 \__kernel_primitive:NN \pdffiledump \tex_filedump:D

```

XeTeX-specific primitives. Note that XeTeX's `\strcmp` is handled earlier and is “rolled up” into `\pdfstrcmp`. A few cross-compatibility names which lack the pdf of the original are handled later.

```

704 \__kernel_primitive:NN \suppressfontnotfounderror
705 \tex_suppressfontnotfounderror:D
706 \__kernel_primitive:NN \XeTeXcharclass \tex_XeTeXcharclass:D
707 \__kernel_primitive:NN \XeTeXcharglyph \tex_XeTeXcharglyph:D
708 \__kernel_primitive:NN \XeTeXcountfeatures \tex_XeTeXcountfeatures:D
709 \__kernel_primitive:NN \XeTeXcountglyphs \tex_XeTeXcountglyphs:D
710 \__kernel_primitive:NN \XeTeXcountselectors \tex_XeTeXcountselectors:D
711 \__kernel_primitive:NN \XeTeXcountvariations \tex_XeTeXcountvariations:D
712 \__kernel_primitive:NN \XeTeXdefaultencoding \tex_XeTeXdefaultencoding:D
713 \__kernel_primitive:NN \XeTeXdashbreakstate \tex_XeTeXdashbreakstate:D
714 \__kernel_primitive:NN \XeTeXfeaturecode \tex_XeTeXfeaturecode:D
715 \__kernel_primitive:NN \XeTeXfeaturename \tex_XeTeXfeaturename:D
716 \__kernel_primitive:NN \XeTeXfindfeaturebyname
717 \tex_XeTeXfindfeaturebyname:D
718 \__kernel_primitive:NN \XeTeXfindselectorbyname
719 \tex_XeTeXfindselectorbyname:D
720 \__kernel_primitive:NN \XeTeXfindvariationbyname
721 \tex_XeTeXfindvariationbyname:D

```

```

722 \__kernel_primitive:NN \XeTeXfirstfontchar \tex_XeTeXfirstfontchar:D
723 \__kernel_primitive:NN \XeTeXfonttype \tex_XeTeXfonttype:D
724 \__kernel_primitive:NN \XeTeXgenerateactualtext
725 \tex_XeTeXgenerateactualtext:D
726 \__kernel_primitive:NN \XeTeXglyph \tex_XeTeXglyph:D
727 \__kernel_primitive:NN \XeTeXglyphbounds \tex_XeTeXglyphbounds:D
728 \__kernel_primitive:NN \XeTeXglyphindex \tex_XeTeXglyphindex:D
729 \__kernel_primitive:NN \XeTeXglyphname \tex_XeTeXglyphname:D
730 \__kernel_primitive:NN \XeTeXinputencoding \tex_XeTeXinputencoding:D
731 \__kernel_primitive:NN \XeTeXinputnormalization
732 \tex_XeTeXinputnormalization:D
733 \__kernel_primitive:NN \XeTeXinterchartokenstate
734 \tex_XeTeXinterchartokenstate:D
735 \__kernel_primitive:NN \XeTeXinterchartoks \tex_XeTeXinterchartoks:D
736 \__kernel_primitive:NN \XeTeXisdefaultselector
737 \tex_XeTeXisdefaultselector:D
738 \__kernel_primitive:NN \XeTeXisexclusivefeature
739 \tex_XeTeXisexclusivefeature:D
740 \__kernel_primitive:NN \XeTeXlastfontchar \tex_XeTeXlastfontchar:D
741 \__kernel_primitive:NN \XeTeXlinebreakskip \tex_XeTeXlinebreakskip:D
742 \__kernel_primitive:NN \XeTeXlinebreaklocale \tex_XeTeXlinebreaklocale:D
743 \__kernel_primitive:NN \XeTeXlinebreakpenalty \tex_XeTeXlinebreakpenalty:D
744 \__kernel_primitive:NN \XeTeXOTcountfeatures \tex_XeTeXOTcountfeatures:D
745 \__kernel_primitive:NN \XeTeXOTcountlanguages \tex_XeTeXOTcountlanguages:D
746 \__kernel_primitive:NN \XeTeXOTcountscripts \tex_XeTeXOTcountscripts:D
747 \__kernel_primitive:NN \XeTeXOTfeaturetag \tex_XeTeXOTfeaturetag:D
748 \__kernel_primitive:NN \XeTeXOTlanguagetag \tex_XeTeXOTlanguagetag:D
749 \__kernel_primitive:NN \XeTeXOTscripttag \tex_XeTeXOTscripttag:D
750 \__kernel_primitive:NN \XeTeXpdffile \tex_XeTeXpdffile:D
751 \__kernel_primitive:NN \XeTeXpdfpagecount \tex_XeTeXpdfpagecount:D
752 \__kernel_primitive:NN \XeTeXpicfile \tex_XeTeXpicfile:D
753 \__kernel_primitive:NN \XeTeXrevision \tex_XeTeXrevision:D
754 \__kernel_primitive:NN \XeTeXselectorname \tex_XeTeXselectorname:D
755 \__kernel_primitive:NN \XeTeXtracingfonts \tex_XeTeXtracingfonts:D
756 \__kernel_primitive:NN \XeTeXupwardsmode \tex_XeTeXupwardsmode:D
757 \__kernel_primitive:NN \XeTeXuseglyphmetrics \tex_XeTeXuseglyphmetrics:D
758 \__kernel_primitive:NN \XeTeXvariation \tex_XeTeXvariation:D
759 \__kernel_primitive:NN \XeTeXvariationdefault \tex_XeTeXvariationdefault:D
760 \__kernel_primitive:NN \XeTeXvariationmax \tex_XeTeXvariationmax:D
761 \__kernel_primitive:NN \XeTeXvariationmin \tex_XeTeXvariationmin:D
762 \__kernel_primitive:NN \XeTeXvariationname \tex_XeTeXvariationname:D
763 \__kernel_primitive:NN \XeTeXversion \tex_XeTeXversion:D
764 \__kernel_primitive:NN \XeTeXselectorcode \tex_XeTeXselectorcode:D
765 \__kernel_primitive:NN \XeTeXinterwordspaceshaping
766 \tex_XeTeXinterwordspaceshaping:D
767 \__kernel_primitive:NN \XeTeXhyphenatablelength
768 \tex_XeTeXhyphenatablelength:D

```

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

```

769 \__kernel_primitive:NN \creationdate \tex_creationdate:D
770 \__kernel_primitive:NN \elapsedtime \tex_elapsedtime:D
771 \__kernel_primitive:NN \filedump \tex_filedump:D
772 \__kernel_primitive:NN \filemoddate \tex_filemoddate:D
773 \__kernel_primitive:NN \filesize \tex_filesize:D
774 \__kernel_primitive:NN \mdfivesum \tex_mdfivesum:D

```

```

775 \__kernel_primitive:NN \ifprimitive          \tex_ifprimitive:D
776 \__kernel_primitive:NN \primitive          \tex_primitive:D
777 \__kernel_primitive:NN \resettimer        \tex_resettimer:D
778 \__kernel_primitive:NN \shellescape      \tex_shellescape:D
779 \__kernel_primitive:NN \XeTeXprotrudechars \tex_protrudechars:D

```

Primitives from LuaTeX, some of which have been ported back to X_YTeX.

```

780 \__kernel_primitive:NN \alignmark         \tex_alignmark:D
781 \__kernel_primitive:NN \aligntab        \tex_aligntab:D
782 \__kernel_primitive:NN \attribute       \tex_attribute:D
783 \__kernel_primitive:NN \attributedef    \tex_attributedef:D
784 \__kernel_primitive:NN \automaticdiscretionary
785 \tex_automaticdiscretionary:D
786 \__kernel_primitive:NN \automatichyphenmode \tex_automatichyphenmode:D
787 \__kernel_primitive:NN \automatichyphenpenalty
788 \tex_automatichyphenpenalty:D
789 \__kernel_primitive:NN \begincsname     \tex_begincsname:D
790 \__kernel_primitive:NN \bodydir        \tex_bodydir:D
791 \__kernel_primitive:NN \bodydirection  \tex_bodydirection:D
792 \__kernel_primitive:NN \boundary       \tex_boundary:D
793 \__kernel_primitive:NN \boxdir        \tex_boxdir:D
794 \__kernel_primitive:NN \boxdirection  \tex_boxdirection:D
795 \__kernel_primitive:NN \breakafterdirmode \tex_breakafterdirmode:D
796 \__kernel_primitive:NN \catcodetable  \tex_catcodetable:D
797 \__kernel_primitive:NN \clearmarks    \tex_clearmarks:D
798 % \__kernel_primitive:NN \compoundhyphenmode
799 % \tex_compoundhyphenmode:D % not documented in manual
800 \__kernel_primitive:NN \crampeddisplaystyle \tex_crampeddisplaystyle:D
801 \__kernel_primitive:NN \crampedscriptscriptstyle
802 \tex_crampedscriptscriptstyle:D
803 \__kernel_primitive:NN \crampedscriptstyle \tex_crampedscriptstyle:D
804 \__kernel_primitive:NN \crampedtextstyle \tex_crampedtextstyle:D
805 \__kernel_primitive:NN \csstring      \tex_csstring:D
806 \__kernel_primitive:NN \deferred     \tex_deferred:D
807 \__kernel_primitive:NN \discretionaryligaturemode
808 \tex_discretionaryligaturemode:D
809 \__kernel_primitive:NN \directlua     \tex_directlua:D
810 \__kernel_primitive:NN \dviextension  \tex_dviextension:D
811 \__kernel_primitive:NN \dvifedback   \tex_dvifedback:D
812 \__kernel_primitive:NN \dvivariable  \tex_dvivariable:D
813 \__kernel_primitive:NN \eTeXglueshrinkorder \tex_eTeXglueshrinkorder:D
814 \__kernel_primitive:NN \eTeXgluestretchorder \tex_eTeXgluestretchorder:D
815 \__kernel_primitive:NN \endlocalcontrol \tex_endlocalcontrol:D
816 \__kernel_primitive:NN \etoksapp     \tex_etoksapp:D
817 \__kernel_primitive:NN \etokspre     \tex_etokspre:D
818 \__kernel_primitive:NN \exceptionpenalty \tex_exceptionpenalty:D
819 \__kernel_primitive:NN \exhyphenchar \tex_exhyphenchar:D
820 \__kernel_primitive:NN \explicitlyhyphenpenalty \tex_explicitlyhyphenpenalty:D
821 \__kernel_primitive:NN \expanded     \tex_expanded:D
822 \__kernel_primitive:NN \explicitdiscretionary \tex_explicitdiscretionary:D
823 \__kernel_primitive:NN \firstvalidlanguage \tex_firstvalidlanguage:D
824 % \__kernel_primitive:NN \fixupboxesmode
825 % \tex_fixupboxesmode:D % not documented in manual
826 \__kernel_primitive:NN \fontid      \tex_fontid:D
827 \__kernel_primitive:NN \formatname   \tex_formatname:D

```

828	<code>__kernel_primitive:NN \hjcode</code>	<code>\tex_hjcode:D</code>
829	<code>__kernel_primitive:NN \hpack</code>	<code>\tex_hpack:D</code>
830	<code>__kernel_primitive:NN \hyphenationbounds</code>	<code>\tex_hyphenationbounds:D</code>
831	<code>__kernel_primitive:NN \hyphenationmin</code>	<code>\tex_hyphenationmin:D</code>
832	<code>__kernel_primitive:NN \hyphenpenaltymode</code>	<code>\tex_hyphenpenaltymode:D</code>
833	<code>__kernel_primitive:NN \gleaders</code>	<code>\tex_gleaders:D</code>
834	<code>__kernel_primitive:NN \glet</code>	<code>\tex_glet:D</code>
835	<code>__kernel_primitive:NN \glyphdimensionsmode</code>	<code>\tex_glyphdimensionsmode:D</code>
836	<code>__kernel_primitive:NN \gtoksapp</code>	<code>\tex_gtoksapp:D</code>
837	<code>__kernel_primitive:NN \gtokspre</code>	<code>\tex_gtokspre:D</code>
838	<code>__kernel_primitive:NN \ifcondition</code>	<code>\tex_ifcondition:D</code>
839	<code>__kernel_primitive:NN \immediateassigned</code>	<code>\tex_immediateassigned:D</code>
840	<code>__kernel_primitive:NN \immediateassignment</code>	<code>\tex_immediateassignment:D</code>
841	<code>__kernel_primitive:NN \initcatcodetable</code>	<code>\tex_initcatcodetable:D</code>
842	<code>__kernel_primitive:NN \lastnamedcs</code>	<code>\tex_lastnamedcs:D</code>
843	<code>__kernel_primitive:NN \latelua</code>	<code>\tex_latelua:D</code>
844	<code>__kernel_primitive:NN \lateluafunction</code>	<code>\tex_lateluafunction:D</code>
845	<code>__kernel_primitive:NN \leftghost</code>	<code>\tex_leftghost:D</code>
846	<code>__kernel_primitive:NN \letcharcode</code>	<code>\tex_letcharcode:D</code>
847	<code>__kernel_primitive:NN \linedir</code>	<code>\tex_linedir:D</code>
848	<code>__kernel_primitive:NN \linedirection</code>	<code>\tex_linedirection:D</code>
849	<code>__kernel_primitive:NN \localbrokenpenalty</code>	<code>\tex_localbrokenpenalty:D</code>
850	<code>__kernel_primitive:NN \localinterlinepenalty</code>	<code>\tex_localinterlinepenalty:D</code>
851	<code>__kernel_primitive:NN \luabytecode</code>	<code>\tex_luabytecode:D</code>
852	<code>__kernel_primitive:NN \luabytecodecall</code>	<code>\tex_luabytecodecall:D</code>
853	<code>__kernel_primitive:NN \luacopyinputnodes</code>	<code>\tex_luacopyinputnodes:D</code>
854	<code>__kernel_primitive:NN \luaedef</code>	<code>\tex_luaedef:D</code>
855	<code>__kernel_primitive:NN \llocalleftbox</code>	<code>\tex_llocalleftbox:D</code>
856	<code>__kernel_primitive:NN \llocalrightbox</code>	<code>\tex_llocalrightbox:D</code>
857	<code>__kernel_primitive:NN \luaescapestring</code>	<code>\tex_luaescapestring:D</code>
858	<code>__kernel_primitive:NN \luafunction</code>	<code>\tex_luafunction:D</code>
859	<code>__kernel_primitive:NN \luafunctioncall</code>	<code>\tex_luafunctioncall:D</code>
860	<code>__kernel_primitive:NN \luatexbanner</code>	<code>\tex_luatexbanner:D</code>
861	<code>__kernel_primitive:NN \luatexrevision</code>	<code>\tex_luatexrevision:D</code>
862	<code>__kernel_primitive:NN \luatexversion</code>	<code>\tex_luatexversion:D</code>
863	<code>__kernel_primitive:NN \mathdefaultsmode</code>	<code>\tex_mathdefaultsmode:D</code>
864	<code>__kernel_primitive:NN \mathdelimitersmode</code>	<code>\tex_mathdelimitersmode:D</code>
865	<code>__kernel_primitive:NN \mathdir</code>	<code>\tex_mathdir:D</code>
866	<code>__kernel_primitive:NN \mathdirection</code>	<code>\tex_mathdirection:D</code>
867	<code>__kernel_primitive:NN \mathdisplayskipmode</code>	<code>\tex_mathdisplayskipmode:D</code>
868	<code>__kernel_primitive:NN \matheqdirmode</code>	<code>\tex_matheqdirmode:D</code>
869	<code>__kernel_primitive:NN \matheqnogapstep</code>	<code>\tex_matheqnogapstep:D</code>
870	<code>__kernel_primitive:NN \mathflattenmode</code>	<code>\tex_mathflattenmode:D</code>
871	<code>__kernel_primitive:NN \mathitalicsmode</code>	<code>\tex_mathitalicsmode:D</code>
872	<code>__kernel_primitive:NN \mathnolimitsmode</code>	<code>\tex_mathnolimitsmode:D</code>
873	<code>__kernel_primitive:NN \mathoption</code>	<code>\tex_mathoption:D</code>
874	<code>__kernel_primitive:NN \mathpenaltiesmode</code>	<code>\tex_mathpenaltiesmode:D</code>
875	<code>__kernel_primitive:NN \mathrulesfam</code>	<code>\tex_mathrulesfam:D</code>
876	<code>% __kernel_primitive:NN \mathrulesmode</code>	
877	<code>% \tex_mathrulesmode:D % not documented in manual</code>	
878	<code>% __kernel_primitive:NN \mathrulethicknessmode</code>	
879	<code>% \tex_mathrulethicknessmode:D % not documented in manual</code>	
880	<code>__kernel_primitive:NN \mathscriptsmode</code>	<code>\tex_mathscriptsmode:D</code>
881	<code>__kernel_primitive:NN \mathscriptboxmode</code>	<code>\tex_mathscriptboxmode:D</code>

882	<code>__kernel_primitive:NN \mathscriptcharmode</code>	<code>\tex_mathscriptcharmode:D</code>
883	<code>__kernel_primitive:NN \mathstyle</code>	<code>\tex_mathstyle:D</code>
884	<code>__kernel_primitive:NN \mathsurroundmode</code>	<code>\tex_mathsurroundmode:D</code>
885	<code>__kernel_primitive:NN \mathsurroundskip</code>	<code>\tex_mathsurroundskip:D</code>
886	<code>__kernel_primitive:NN \nohrule</code>	<code>\tex_nohrule:D</code>
887	<code>__kernel_primitive:NN \nokerns</code>	<code>\tex_nokerns:D</code>
888	<code>__kernel_primitive:NN \noligs</code>	<code>\tex_noligs:D</code>
889	<code>__kernel_primitive:NN \nospaces</code>	<code>\tex_nospaces:D</code>
890	<code>__kernel_primitive:NN \novrule</code>	<code>\tex_novrule:D</code>
891	<code>__kernel_primitive:NN \outputbox</code>	<code>\tex_outputbox:D</code>
892	<code>__kernel_primitive:NN \pagebottomoffset</code>	<code>\tex_pagebottomoffset:D</code>
893	<code>__kernel_primitive:NN \pagedir</code>	<code>\tex_pagedir:D</code>
894	<code>__kernel_primitive:NN \pagedirection</code>	<code>\tex_pagedirection:D</code>
895	<code>__kernel_primitive:NN \pageleftoffset</code>	<code>\tex_pageleftoffset:D</code>
896	<code>__kernel_primitive:NN \pagerightoffset</code>	<code>\tex_pagerightoffset:D</code>
897	<code>__kernel_primitive:NN \pagetopoffset</code>	<code>\tex_pagetopoffset:D</code>
898	<code>__kernel_primitive:NN \pardir</code>	<code>\tex_pardir:D</code>
899	<code>__kernel_primitive:NN \pardirection</code>	<code>\tex_pardirection:D</code>
900	<code>__kernel_primitive:NN \pdfextension</code>	<code>\tex_pdfextension:D</code>
901	<code>__kernel_primitive:NN \pdffeedback</code>	<code>\tex_pdffeedback:D</code>
902	<code>__kernel_primitive:NN \pdfvariable</code>	<code>\tex_pdfvariable:D</code>
903	<code>__kernel_primitive:NN \postexhyphenchar</code>	<code>\tex_postexhyphenchar:D</code>
904	<code>__kernel_primitive:NN \posthyphenchar</code>	<code>\tex_posthyphenchar:D</code>
905	<code>__kernel_primitive:NN \prebinoppenalty</code>	<code>\tex_prebinoppenalty:D</code>
906	<code>__kernel_primitive:NN \predisplaygapfactor</code>	<code>\tex_predisplaygapfactor:D</code>
907	<code>__kernel_primitive:NN \preexhyphenchar</code>	<code>\tex_preexhyphenchar:D</code>
908	<code>__kernel_primitive:NN \prehyphenchar</code>	<code>\tex_prehyphenchar:D</code>
909	<code>__kernel_primitive:NN \prerelpenalty</code>	<code>\tex_prerelpenalty:D</code>
910	<code>__kernel_primitive:NN \protrusionboundary</code>	<code>\tex_protrusionboundary:D</code>
911	<code>__kernel_primitive:NN \rightghost</code>	<code>\tex_rightghost:D</code>
912	<code>__kernel_primitive:NN \savecatcodetable</code>	<code>\tex_savecatcodetable:D</code>
913	<code>__kernel_primitive:NN \scantexttokens</code>	<code>\tex_scantexttokens:D</code>
914	<code>__kernel_primitive:NN \setfontid</code>	<code>\tex_setfontid:D</code>
915	<code>__kernel_primitive:NN \shapemode</code>	<code>\tex_shapemode:D</code>
916	<code>__kernel_primitive:NN \suppressifcsnameerror</code>	<code>\tex_suppressifcsnameerror:D</code>
917	<code>__kernel_primitive:NN \suppresslongerror</code>	<code>\tex_suppresslongerror:D</code>
918	<code>__kernel_primitive:NN \suppressmathparerror</code>	<code>\tex_suppressmathparerror:D</code>
919	<code>__kernel_primitive:NN \suppressoutererror</code>	<code>\tex_suppressoutererror:D</code>
920	<code>__kernel_primitive:NN \suppressprimitiveerror</code>	
921	<code>\tex_suppressprimitiveerror:D</code>	
922	<code>__kernel_primitive:NN \textdir</code>	<code>\tex_textdir:D</code>
923	<code>__kernel_primitive:NN \textdirection</code>	<code>\tex_textdirection:D</code>
924	<code>__kernel_primitive:NN \toksapp</code>	<code>\tex_toksapp:D</code>
925	<code>__kernel_primitive:NN \tokspre</code>	<code>\tex_tokspre:D</code>
926	<code>__kernel_primitive:NN \tpack</code>	<code>\tex_tpack:D</code>
927	<code>__kernel_primitive:NN \variablefam</code>	<code>\tex_variablefam:D</code>
928	<code>__kernel_primitive:NN \vpack</code>	<code>\tex_vpack:D</code>
929	<code>__kernel_primitive:NN \wordboundary</code>	<code>\tex_wordboundary:D</code>
930	<code>__kernel_primitive:NN \xtoksapp</code>	<code>\tex_xtoksapp:D</code>
931	<code>__kernel_primitive:NN \xtokspre</code>	<code>\tex_xtokspre:D</code>
Primitives from pdfTeX that LuaTeX renames.		
932	<code>__kernel_primitive:NN \adjustspacing</code>	<code>\tex_adjustspacing:D</code>
933	<code>__kernel_primitive:NN \copyfont</code>	<code>\tex_copyfont:D</code>
934	<code>__kernel_primitive:NN \draftmode</code>	<code>\tex_draftmode:D</code>

```

935 \__kernel_primitive:NN \expandglyphsinfont \tex_fontexpand:D
936 \__kernel_primitive:NN \ifabsdim \tex_ifabsdim:D
937 \__kernel_primitive:NN \ifabsnum \tex_ifabsnum:D
938 \__kernel_primitive:NN \ignoreligaturesinfont \tex_ignoreligaturesinfont:D
939 \__kernel_primitive:NN \insertht \tex_insertht:D
940 \__kernel_primitive:NN \lastsavedboxresourceindex
941 \tex_pdflastxform:D
942 \__kernel_primitive:NN \lastsavedimageresourceindex
943 \tex_pdflastximage:D
944 \__kernel_primitive:NN \lastsavedimageresourcepages
945 \tex_pdflastximagepages:D
946 \__kernel_primitive:NN \lastxpos \tex_lastxpos:D
947 \__kernel_primitive:NN \lastypos \tex_lastypos:D
948 \__kernel_primitive:NN \normaldeviate \tex_normaldeviate:D
949 \__kernel_primitive:NN \outputmode \tex_outputmode:D
950 \__kernel_primitive:NN \pageheight \tex_pageheight:D
951 \__kernel_primitive:NN \pagewidth \tex_pagewidth:D
952 \__kernel_primitive:NN \protrudechars \tex_protrudechars:D
953 \__kernel_primitive:NN \pxdimen \tex_pxdimen:D
954 \__kernel_primitive:NN \randomseed \tex_randomseed:D
955 \__kernel_primitive:NN \useboxresource \tex_pdfrefxform:D
956 \__kernel_primitive:NN \useimageresource \tex_pdfrefximage:D
957 \__kernel_primitive:NN \savepos \tex_savepos:D
958 \__kernel_primitive:NN \saveboxresource \tex_pdfxform:D
959 \__kernel_primitive:NN \saveimageresource \tex_pdfximage:D
960 \__kernel_primitive:NN \setrandomseed \tex_setrandomseed:D
961 \__kernel_primitive:NN \tracingfonts \tex_tracingfonts:D
962 \__kernel_primitive:NN \uniformdeviate \tex_uniformdeviate:D

```

The set of Unicode math primitives were introduced by X_YTeX and LuaTeX in a somewhat complex fashion: a few first as XeTeX... which were then renamed with LuaTeX having a lot more. These names now all start \U... and mainly \Umath....

```

963 \__kernel_primitive:NN \Uchar \tex_Uchar:D
964 \__kernel_primitive:NN \Ucharcat \tex_Ucharcat:D
965 \__kernel_primitive:NN \Udelcode \tex_Udelcode:D
966 \__kernel_primitive:NN \Udelcodenum \tex_Udelcodenum:D
967 \__kernel_primitive:NN \Udelimiter \tex_Udelimiter:D
968 \__kernel_primitive:NN \Udelimiterover \tex_Udelimiterover:D
969 \__kernel_primitive:NN \Udelimiterunder \tex_Udelimiterunder:D
970 \__kernel_primitive:NN \Uhexextensible \tex_Uhexextensible:D
971 \__kernel_primitive:NN \Uleft \tex_Uleft:D
972 \__kernel_primitive:NN \Umathaccent \tex_Umathaccent:D
973 \__kernel_primitive:NN \Umathaxis \tex_Umathaxis:D
974 \__kernel_primitive:NN \Umathbinbinspacing \tex_Umathbinbinspacing:D
975 \__kernel_primitive:NN \Umathbinclonespacing \tex_Umathbinclonespacing:D
976 \__kernel_primitive:NN \Umathbininnerspacing \tex_Umathbininnerspacing:D
977 \__kernel_primitive:NN \Umathbinopenspacing \tex_Umathbinopenspacing:D
978 \__kernel_primitive:NN \Umathbinopenspacing \tex_Umathbinopenspacing:D
979 \__kernel_primitive:NN \Umathbinordspacing \tex_Umathbinordspacing:D
980 \__kernel_primitive:NN \Umathbinpunctspacing \tex_Umathbinpunctspacing:D
981 \__kernel_primitive:NN \Umathbinrelspacing \tex_Umathbinrelspacing:D
982 \__kernel_primitive:NN \Umathchar \tex_Umathchar:D
983 \__kernel_primitive:NN \Umathcharclass \tex_Umathcharclass:D
984 \__kernel_primitive:NN \Umathchardef \tex_Umathchardef:D

```

985 __kernel_primitive:NN \Umathcharfam \tex_Umathcharfam:D
986 __kernel_primitive:NN \Umathcharnum \tex_Umathcharnum:D
987 __kernel_primitive:NN \Umathcharnumdef \tex_Umathcharnumdef:D
988 __kernel_primitive:NN \Umathcharslot \tex_Umathcharslot:D
989 __kernel_primitive:NN \Umathclosebinspacing \tex_Umathclosebinspacing:D
990 __kernel_primitive:NN \Umathcloseclosespacing
991 \tex_Umathcloseclosespacing:D
992 __kernel_primitive:NN \Umathcloseinnerspacing
993 \tex_Umathcloseinnerspacing:D
994 __kernel_primitive:NN \Umathcloseopenspacing \tex_Umathcloseopenspacing:D
995 __kernel_primitive:NN \Umathcloseopspacing \tex_Umathcloseopspacing:D
996 __kernel_primitive:NN \Umathcloseordspacing \tex_Umathcloseordspacing:D
997 __kernel_primitive:NN \Umathclosepunctspacing
998 \tex_Umathclosepunctspacing:D
999 __kernel_primitive:NN \Umathcloserelspacing \tex_Umathcloserelspacing:D
1000 __kernel_primitive:NN \Umathcode \tex_Umathcode:D
1001 __kernel_primitive:NN \Umathcodenum \tex_Umathcodenum:D
1002 __kernel_primitive:NN \Umathconnectoroverlapmin
1003 \tex_Umathconnectoroverlapmin:D
1004 __kernel_primitive:NN \Umathfractiondelsize \tex_Umathfractiondelsize:D
1005 __kernel_primitive:NN \Umathfractiondenomdown
1006 \tex_Umathfractiondenomdown:D
1007 __kernel_primitive:NN \Umathfractiondenomvgap
1008 \tex_Umathfractiondenomvgap:D
1009 __kernel_primitive:NN \Umathfractionnumup \tex_Umathfractionnumup:D
1010 __kernel_primitive:NN \Umathfractionnumvgap \tex_Umathfractionnumvgap:D
1011 __kernel_primitive:NN \Umathfractionrule \tex_Umathfractionrule:D
1012 __kernel_primitive:NN \Umathinnerbinspacing \tex_Umathinnerbinspacing:D
1013 __kernel_primitive:NN \Umathinnerclosespacing
1014 \tex_Umathinnerclosespacing:D
1015 __kernel_primitive:NN \Umathinnerinnerspacing
1016 \tex_Umathinnerinnerspacing:D
1017 __kernel_primitive:NN \Umathinneropenspacing \tex_Umathinneropenspacing:D
1018 __kernel_primitive:NN \Umathinneropspacing \tex_Umathinneropspacing:D
1019 __kernel_primitive:NN \Umathinnerordspacing \tex_Umathinnerordspacing:D
1020 __kernel_primitive:NN \Umathinnerpunctspacing
1021 \tex_Umathinnerpunctspacing:D
1022 __kernel_primitive:NN \Umathinnerrelspacing \tex_Umathinnerrelspacing:D
1023 __kernel_primitive:NN \Umathlimitabovebgap \tex_Umathlimitabovebgap:D
1024 __kernel_primitive:NN \Umathlimitabovekern \tex_Umathlimitabovekern:D
1025 __kernel_primitive:NN \Umathlimitabovevgap \tex_Umathlimitabovevgap:D
1026 __kernel_primitive:NN \Umathlimitbelowbgap \tex_Umathlimitbelowbgap:D
1027 __kernel_primitive:NN \Umathlimitbelowkern \tex_Umathlimitbelowkern:D
1028 __kernel_primitive:NN \Umathlimitbelowvgap \tex_Umathlimitbelowvgap:D
1029 __kernel_primitive:NN \Umathnolimitsubfactor \tex_Umathnolimitsubfactor:D
1030 __kernel_primitive:NN \Umathnolimitsupfactor \tex_Umathnolimitsupfactor:D
1031 __kernel_primitive:NN \Umathopbinspacing \tex_Umathopbinspacing:D
1032 __kernel_primitive:NN \Umathopclosespacing \tex_Umathopclosespacing:D
1033 __kernel_primitive:NN \Umathopenbinspacing \tex_Umathopenbinspacing:D
1034 __kernel_primitive:NN \Umathopenclosespacing \tex_Umathopenclosespacing:D
1035 __kernel_primitive:NN \Umathopeninnerspacing \tex_Umathopeninnerspacing:D
1036 __kernel_primitive:NN \Umathopenopenspacing \tex_Umathopenopenspacing:D
1037 __kernel_primitive:NN \Umathopenopspacing \tex_Umathopenopspacing:D
1038 __kernel_primitive:NN \Umathopenordspacing \tex_Umathopenordspacing:D

1039 __kernel_primitive:NN \Umathopenpunctspacing \tex_Umathopenpunctspacing:D
1040 __kernel_primitive:NN \Umathopenrelspacing \tex_Umathopenrelspacing:D
1041 __kernel_primitive:NN \Umathoperatorsize \tex_Umathoperatorsize:D
1042 __kernel_primitive:NN \Umathopinnerspacing \tex_Umathopinnerspacing:D
1043 __kernel_primitive:NN \Umathopopenspacing \tex_Umathopopenspacing:D
1044 __kernel_primitive:NN \Umathopopspacing \tex_Umathopopspacing:D
1045 __kernel_primitive:NN \Umathopordspacing \tex_Umathopordspacing:D
1046 __kernel_primitive:NN \Umathoppunctspacing \tex_Umathoppunctspacing:D
1047 __kernel_primitive:NN \Umathoprelspacing \tex_Umathoprelspacing:D
1048 __kernel_primitive:NN \Umathordbinspacing \tex_Umathordbinspacing:D
1049 __kernel_primitive:NN \Umathordclosespacing \tex_Umathordclosespacing:D
1050 __kernel_primitive:NN \Umathordinnerspacing \tex_Umathordinnerspacing:D
1051 __kernel_primitive:NN \Umathordopenspacing \tex_Umathordopenspacing:D
1052 __kernel_primitive:NN \Umathordopspacing \tex_Umathordopspacing:D
1053 __kernel_primitive:NN \Umathordordspacing \tex_Umathordordspacing:D
1054 __kernel_primitive:NN \Umathordpunctspacing \tex_Umathordpunctspacing:D
1055 __kernel_primitive:NN \Umathordrelspacing \tex_Umathordrelspacing:D
1056 __kernel_primitive:NN \Umathoverbarkern \tex_Umathoverbarkern:D
1057 __kernel_primitive:NN \Umathoverbarrule \tex_Umathoverbarrule:D
1058 __kernel_primitive:NN \Umathoverbarvgap \tex_Umathoverbarvgap:D
1059 __kernel_primitive:NN \Umathoverdelimitergap
1060 \tex_Umathoverdelimitergap:D
1061 __kernel_primitive:NN \Umathoverdelimitervgap
1062 \tex_Umathoverdelimitervgap:D
1063 __kernel_primitive:NN \Umathpunctbinspacing \tex_Umathpunctbinspacing:D
1064 __kernel_primitive:NN \Umathpunctclosespacing
1065 \tex_Umathpunctclosespacing:D
1066 __kernel_primitive:NN \Umathpunctinnerspacing
1067 \tex_Umathpunctinnerspacing:D
1068 __kernel_primitive:NN \Umathpunctopenspacing \tex_Umathpunctopenspacing:D
1069 __kernel_primitive:NN \Umathpunctopspacing \tex_Umathpunctopspacing:D
1070 __kernel_primitive:NN \Umathpunctordspacing \tex_Umathpunctordspacing:D
1071 __kernel_primitive:NN \Umathpunctpunctspacing
1072 \tex_Umathpunctpunctspacing:D
1073 __kernel_primitive:NN \Umathpunctrelspacing \tex_Umathpunctrelspacing:D
1074 __kernel_primitive:NN \Umathquad \tex_Umathquad:D
1075 __kernel_primitive:NN \Umathradicaldegreeafter
1076 \tex_Umathradicaldegreeafter:D
1077 __kernel_primitive:NN \Umathradicaldegreebefore
1078 \tex_Umathradicaldegreebefore:D
1079 __kernel_primitive:NN \Umathradicaldegreeraise
1080 \tex_Umathradicaldegreeraise:D
1081 __kernel_primitive:NN \Umathradicalkern \tex_Umathradicalkern:D
1082 __kernel_primitive:NN \Umathradicalrule \tex_Umathradicalrule:D
1083 __kernel_primitive:NN \Umathradicalvgap \tex_Umathradicalvgap:D
1084 __kernel_primitive:NN \Umathrelbinspacing \tex_Umathrelbinspacing:D
1085 __kernel_primitive:NN \Umathrelclosespacing \tex_Umathrelclosespacing:D
1086 __kernel_primitive:NN \Umathrelinnerspacing \tex_Umathrelinnerspacing:D
1087 __kernel_primitive:NN \Umathrelopenspacing \tex_Umathrelopenspacing:D
1088 __kernel_primitive:NN \Umathrelopspacing \tex_Umathrelopspacing:D
1089 __kernel_primitive:NN \Umathrelordspacing \tex_Umathrelordspacing:D
1090 __kernel_primitive:NN \Umathrelpunctspacing \tex_Umathrelpunctspacing:D
1091 __kernel_primitive:NN \Umathrelrelspacing \tex_Umathrelrelspacing:D
1092 __kernel_primitive:NN \Umathskewedfractionhgap

1093	<code>\tex_Umathskewedfractionhgap:D</code>	
1094	<code>__kernel_primitive:NN \Umathskewedfractionvgap</code>	
1095	<code>\tex_Umathskewedfractionvgap:D</code>	
1096	<code>__kernel_primitive:NN \Umathspaceafterscript</code>	<code>\tex_Umathspaceafterscript:D</code>
1097	<code>__kernel_primitive:NN \Umathstackdenomdown</code>	<code>\tex_Umathstackdenomdown:D</code>
1098	<code>__kernel_primitive:NN \Umathstacknumup</code>	<code>\tex_Umathstacknumup:D</code>
1099	<code>__kernel_primitive:NN \Umathstackvgap</code>	<code>\tex_Umathstackvgap:D</code>
1100	<code>__kernel_primitive:NN \Umathsubshiftdown</code>	<code>\tex_Umathsubshiftdown:D</code>
1101	<code>__kernel_primitive:NN \Umathsubshiftdrop</code>	<code>\tex_Umathsubshiftdrop:D</code>
1102	<code>__kernel_primitive:NN \Umathsubsupshiftdown</code>	<code>\tex_Umathsubsupshiftdown:D</code>
1103	<code>__kernel_primitive:NN \Umathsubsupvgap</code>	<code>\tex_Umathsubsupvgap:D</code>
1104	<code>__kernel_primitive:NN \Umathsubtopmax</code>	<code>\tex_Umathsubtopmax:D</code>
1105	<code>__kernel_primitive:NN \Umathsupbottommin</code>	<code>\tex_Umathsupbottommin:D</code>
1106	<code>__kernel_primitive:NN \Umathsupshiftdrop</code>	<code>\tex_Umathsupshiftdrop:D</code>
1107	<code>__kernel_primitive:NN \Umathsupshiftdown</code>	<code>\tex_Umathsupshiftdown:D</code>
1108	<code>__kernel_primitive:NN \Umathsupsubbottommax</code>	<code>\tex_Umathsupsubbottommax:D</code>
1109	<code>__kernel_primitive:NN \Umathunderbarkern</code>	<code>\tex_Umathunderbarkern:D</code>
1110	<code>__kernel_primitive:NN \Umathunderbarrule</code>	<code>\tex_Umathunderbarrule:D</code>
1111	<code>__kernel_primitive:NN \Umathunderbarvgap</code>	<code>\tex_Umathunderbarvgap:D</code>
1112	<code>__kernel_primitive:NN \Umathunderdelimitervgap</code>	
1113	<code>\tex_Umathunderdelimitervgap:D</code>	
1114	<code>__kernel_primitive:NN \Umathunderdelimitervgap</code>	
1115	<code>\tex_Umathunderdelimitervgap:D</code>	
1116	<code>__kernel_primitive:NN \Umiddle</code>	<code>\tex_Umiddle:D</code>
1117	<code>__kernel_primitive:NN \Unosubscript</code>	<code>\tex_Unosubscript:D</code>
1118	<code>__kernel_primitive:NN \Unosuperscript</code>	<code>\tex_Unosuperscript:D</code>
1119	<code>__kernel_primitive:NN \Uoverdelimiterv</code>	<code>\tex_Uoverdelimiterv:D</code>
1120	<code>__kernel_primitive:NN \Uradical</code>	<code>\tex_Uradical:D</code>
1121	<code>__kernel_primitive:NN \Uright</code>	<code>\tex_Uright:D</code>
1122	<code>__kernel_primitive:NN \Uroot</code>	<code>\tex_Uroot:D</code>
1123	<code>__kernel_primitive:NN \Uskewed</code>	<code>\tex_Uskewed:D</code>
1124	<code>__kernel_primitive:NN \Uskewedwithdelims</code>	<code>\tex_Uskewedwithdelims:D</code>
1125	<code>__kernel_primitive:NN \Ustack</code>	<code>\tex_Ustack:D</code>
1126	<code>__kernel_primitive:NN \Ustartdisplaymath</code>	<code>\tex_Ustartdisplaymath:D</code>
1127	<code>__kernel_primitive:NN \Ustartmath</code>	<code>\tex_Ustartmath:D</code>
1128	<code>__kernel_primitive:NN \Ustopdisplaymath</code>	<code>\tex_Ustopdisplaymath:D</code>
1129	<code>__kernel_primitive:NN \Ustopmath</code>	<code>\tex_Ustopmath:D</code>
1130	<code>__kernel_primitive:NN \Usubscript</code>	<code>\tex_Usubscript:D</code>
1131	<code>__kernel_primitive:NN \Usuperscript</code>	<code>\tex_Usuperscript:D</code>
1132	<code>__kernel_primitive:NN \Uunderdelimiterv</code>	<code>\tex_Uunderdelimiterv:D</code>
1133	<code>__kernel_primitive:NN \Uvextensible</code>	<code>\tex_Uvextensible:D</code>

Primitives from pTeX.

1134	<code>__kernel_primitive:NN \autospaceing</code>	<code>\tex_autospaceing:D</code>
1135	<code>__kernel_primitive:NN \autoxspaceing</code>	<code>\tex_autoxspaceing:D</code>
1136	<code>__kernel_primitive:NN \currentcjktoken</code>	<code>\tex_currentcjktoken:D</code>
1137	<code>__kernel_primitive:NN \currentspacingmode</code>	<code>\tex_currentspacingmode:D</code>
1138	<code>__kernel_primitive:NN \currentxspacingmode</code>	<code>\tex_currentxspacingmode:D</code>
1139	<code>__kernel_primitive:NN \disinhibitglue</code>	<code>\tex_disinhibitglue:D</code>
1140	<code>__kernel_primitive:NN \dtou</code>	<code>\tex_dtou:D</code>
1141	<code>__kernel_primitive:NN \epTeXinputencoding</code>	<code>\tex_epTeXinputencoding:D</code>
1142	<code>__kernel_primitive:NN \epTeXversion</code>	<code>\tex_epTeXversion:D</code>
1143	<code>__kernel_primitive:NN \euc</code>	<code>\tex_euc:D</code>
1144	<code>__kernel_primitive:NN \hfi</code>	<code>\tex_hfi:D</code>
1145	<code>__kernel_primitive:NN \ifdbox</code>	<code>\tex_ifdbox:D</code>

1146	<code>_kernel_primitive:NN \ifddir</code>	<code>\tex_ifddir:D</code>
1147	<code>_kernel_primitive:NN \ifjfont</code>	<code>\tex_ifjfont:D</code>
1148	<code>_kernel_primitive:NN \ifmbox</code>	<code>\tex_ifmbox:D</code>
1149	<code>_kernel_primitive:NN \ifmdir</code>	<code>\tex_ifmdir:D</code>
1150	<code>_kernel_primitive:NN \iftbox</code>	<code>\tex_iftbox:D</code>
1151	<code>_kernel_primitive:NN \iftfont</code>	<code>\tex_iftfont:D</code>
1152	<code>_kernel_primitive:NN \iftdir</code>	<code>\tex_iftdir:D</code>
1153	<code>_kernel_primitive:NN \ifybox</code>	<code>\tex_ifybox:D</code>
1154	<code>_kernel_primitive:NN \ifydir</code>	<code>\tex_ifydir:D</code>
1155	<code>_kernel_primitive:NN \inhibitglue</code>	<code>\tex_inhibitglue:D</code>
1156	<code>_kernel_primitive:NN \inhibitxspcode</code>	<code>\tex_inhibitxspcode:D</code>
1157	<code>_kernel_primitive:NN \jcharwidowpenalty</code>	<code>\tex_jcharwidowpenalty:D</code>
1158	<code>_kernel_primitive:NN \jfam</code>	<code>\tex_jfam:D</code>
1159	<code>_kernel_primitive:NN \jfont</code>	<code>\tex_jfont:D</code>
1160	<code>_kernel_primitive:NN \jis</code>	<code>\tex_jis:D</code>
1161	<code>_kernel_primitive:NN \kanjiskip</code>	<code>\tex_kanjiskip:D</code>
1162	<code>_kernel_primitive:NN \kansuji</code>	<code>\tex_kansuji:D</code>
1163	<code>_kernel_primitive:NN \kansujichar</code>	<code>\tex_kansujichar:D</code>
1164	<code>_kernel_primitive:NN \kcatcode</code>	<code>\tex_kcatcode:D</code>
1165	<code>_kernel_primitive:NN \kuten</code>	<code>\tex_kuten:D</code>
1166	<code>_kernel_primitive:NN \lastnodechar</code>	<code>\tex_lastnodechar:D</code>
1167	<code>_kernel_primitive:NN \lastnodefont</code>	<code>\tex_lastnodefont:D</code>
1168	<code>_kernel_primitive:NN \lastnodesubtype</code>	<code>\tex_lastnodesubtype:D</code>
1169	<code>_kernel_primitive:NN \noautospaceing</code>	<code>\tex_noautospaceing:D</code>
1170	<code>_kernel_primitive:NN \noautoxspaceing</code>	<code>\tex_noautoxspaceing:D</code>
1171	<code>_kernel_primitive:NN \pagefistretch</code>	<code>\tex_pagefistretch:D</code>
1172	<code>_kernel_primitive:NN \postbreakpenalty</code>	<code>\tex_postbreakpenalty:D</code>
1173	<code>_kernel_primitive:NN \prebreakpenalty</code>	<code>\tex_prebreakpenalty:D</code>
1174	<code>_kernel_primitive:NN \ptexfontname</code>	<code>\tex_ptexfontname:D</code>
1175	<code>_kernel_primitive:NN \ptexlineendmode</code>	<code>\tex_lineendmode:D</code>
1176	<code>_kernel_primitive:NN \ptexminorversion</code>	<code>\tex_ptexminorversion:D</code>
1177	<code>_kernel_primitive:NN \ptexrevision</code>	<code>\tex_ptexrevision:D</code>
1178	<code>_kernel_primitive:NN \ptextracingfonts</code>	<code>\tex_ptextracingfonts:D</code>
1179	<code>_kernel_primitive:NN \ptexversion</code>	<code>\tex_ptexversion:D</code>
1180	<code>_kernel_primitive:NN \readpapersizespecial</code>	<code>\tex_readpapersizespecial:D</code>
1181	<code>_kernel_primitive:NN \scriptbaselineshiftfactor</code>	
1182	<code>\tex_scriptbaselineshiftfactor:D</code>	
1183	<code>_kernel_primitive:NN \scriptscriptbaselineshiftfactor</code>	
1184	<code>\tex_scriptscriptbaselineshiftfactor:D</code>	
1185	<code>_kernel_primitive:NN \showmode</code>	<code>\tex_showmode:D</code>
1186	<code>_kernel_primitive:NN \sjis</code>	<code>\tex_sjis:D</code>
1187	<code>_kernel_primitive:NN \tate</code>	<code>\tex_tate:D</code>
1188	<code>_kernel_primitive:NN \tbaselineshift</code>	<code>\tex_tbaselineshift:D</code>
1189	<code>_kernel_primitive:NN \textbaselineshiftfactor</code>	
1190	<code>\tex_textbaselineshiftfactor:D</code>	
1191	<code>_kernel_primitive:NN \tfont</code>	<code>\tex_tfont:D</code>
1192	<code>_kernel_primitive:NN \tojis</code>	<code>\tex_tojis:D</code>
1193	<code>_kernel_primitive:NN \toucs</code>	<code>\tex_toucs:D</code>
1194	<code>_kernel_primitive:NN \ucs</code>	<code>\tex_ucs:D</code>
1195	<code>_kernel_primitive:NN \xkanjiskip</code>	<code>\tex_xkanjiskip:D</code>
1196	<code>_kernel_primitive:NN \xspcode</code>	<code>\tex_xspcode:D</code>
1197	<code>_kernel_primitive:NN \ybaselineshift</code>	<code>\tex_ybaselineshift:D</code>
1198	<code>_kernel_primitive:NN \yoko</code>	<code>\tex_yoko:D</code>
1199	<code>_kernel_primitive:NN \vfi</code>	<code>\tex_vfi:D</code>

Primitives from upTeX.

```

1200 \__kernel_primitive:NN \currentcjktoken \tex_currentcjktoken:D
1201 \__kernel_primitive:NN \disablecjktoken \tex_disablecjktoken:D
1202 \__kernel_primitive:NN \enablecjktoken \tex_enablecjktoken:D
1203 \__kernel_primitive:NN \forcecjktoken \tex_forcecjktoken:D
1204 \__kernel_primitive:NN \kchar \tex_kchar:D
1205 \__kernel_primitive:NN \kchardef \tex_kchardef:D
1206 \__kernel_primitive:NN \kuten \tex_kuten:D
1207 \__kernel_primitive:NN \uptexrevision \tex_uptexrevision:D
1208 \__kernel_primitive:NN \uptexversion \tex_uptexversion:D

```

Omega primitives provided by pTeX (listed separately mainly to allow understanding of their source).

```

1209 \__kernel_primitive:NN \odelcode \tex_odelcode:D
1210 \__kernel_primitive:NN \odelimiter \tex_odelimiter:D
1211 \__kernel_primitive:NN \omathaccent \tex_omathaccent:D
1212 \__kernel_primitive:NN \omathchar \tex_omathchar:D
1213 \__kernel_primitive:NN \omathchardef \tex_omathchardef:D
1214 \__kernel_primitive:NN \omathcode \tex_omathcode:D
1215 \__kernel_primitive:NN \oradical \tex_oradical:D

```

Newer cross-engine primitives.

```

1216 \__kernel_primitive:NN \partokencontext \tex_partokencontext:D
1217 \__kernel_primitive:NN \partokenname \tex_partokenname:D
1218 \__kernel_primitive:NN \showstream \tex_showstream:D
1219 \__kernel_primitive:NN \tracingstacklevels \tex_tracingstacklevels:D

```

End of the “just the names” part of the source.

```

1220 </names | package>
1221 </names | tex>
1222 <*package>
1223 <*tex>

```

The job is done: close the group (using the primitive renamed!).

```

1224 \tex_endgroup:D

```

L^AT_εEX 2_ε moves a few primitives, so these are sorted out. In newer versions of L^AT_εEX 2_ε, expl3 is loaded rather early, so only some primitives are already renamed, so we need two tests here. At the beginning of the L^AT_εEX 2_ε format, the primitives `\end` and `\input` are renamed, and only later on the other ones.

```

1225 \tex_ifdefined:D \@@end
1226 \tex_let:D \tex_end:D \@@end
1227 \tex_let:D \tex_input:D \@@input
1228 \tex_fi:D

```

If `\@@@hyph` is defined, we are loading expl3 in a pre-2020/10/01 release of L^AT_εEX 2_ε, so a few other primitives have to be tested as well.

```

1229 \tex_ifdefined:D \@@hyph
1230 \tex_let:D \tex_everydisplay:D \frozen@everydisplay
1231 \tex_let:D \tex_everymath:D \frozen@everymath
1232 \tex_let:D \tex_hyphen:D \@@hyph
1233 \tex_let:D \tex_italiccorrection:D \@@italiccorr
1234 \tex_let:D \tex_underline:D \@@underline

```

The `\shipout` primitive is particularly tricky as a number of packages want to hook in here. First, we see if a sufficiently-new kernel has saved a copy: if it has, just use that. Otherwise, we need to check each of the possible packages/classes that might move it: here, we are looking for those which do *not* delay action to the `\AtBeginDocument` hook. (We cannot use `\primitive` as that doesn't allow us to make a direct copy of the primitive *itself*.) As we know that L^AT_εX is in use, we use its `\@tfor` loop here.

```

1235 \tex_ifdefined:D \@@shipout
1236 \tex_let:D \tex_shipout:D \@@shipout
1237 \tex_fi:D
1238 \tex_begingroup:D
1239 \tex_edef:D \l_tmpa_tl { \tex_string:D \shipout }
1240 \tex_edef:D \l_tmpb_tl { \tex_meaning:D \shipout }
1241 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1242 \tex_else:D
1243 \tex_expandafter:D \@tfor \tex_expandafter:D \@tempa \tex_string:D :=
1244 \CROP@shipout
1245 \dup@shipout
1246 \GPTorg@shipout
1247 \LL@shipout
1248 \mem@oldshipout
1249 \opem@shipout
1250 \pgfpages@originalshipout
1251 \pr@shipout
1252 \Shipout
1253 \verso@orig@shipout
1254 \do
1255 {
1256 \tex_edef:D \l_tmpb_tl
1257 { \tex_expandafter:D \tex_meaning:D \@tempa }
1258 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1259 \tex_global:D \tex_expandafter:D \tex_let:D
1260 \tex_expandafter:D \tex_shipout:D \@tempa
1261 \tex_fi:D
1262 }
1263 \tex_fi:D
1264 \tex_endgroup:D

```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer Lua^T_εX has this simply as `\tracingfonts`, but that is overwritten by the L^AT_εX kernel. So any spurious definition has to be removed, then the real version saved either from the pdf^T_εX name or from Lua^T_εX. In the latter case, we leave `\@@tracingfonts` available: this might be useful and almost all L^AT_εX users will have expl3 loaded by fontspec. (We follow the usual kernel convention that `@@` is used for saved primitives.)

```

1265 \tex_let:D \tex_tracingfonts:D \tex_undefined:D
1266 \tex_ifdefined:D \pdftracingfonts
1267 \tex_let:D \tex_tracingfonts:D \pdftracingfonts
1268 \tex_else:D
1269 \tex_ifdefined:D \tex_directlua:D
1270 \tex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1271 \tex_let:D \tex_tracingfonts:D \@@tracingfonts
1272 \tex_fi:D
1273 \tex_fi:D
1274 \tex_fi:D

```


Only pdfTeX and LuaTeX define `\pdfmapfile` and `\pdfmapline`: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```

1275 \tex_ifnum:D 0
1276 \tex_ifdefined:D \tex_pdftexversion:D 1 \tex_fi:D
1277 \tex_ifdefined:D \tex luatexversion:D 1 \tex_fi:D
1278 = 0 %
1279 \tex_let:D \tex_pdfmapfile:D \tex_undefined:D
1280 \tex_let:D \tex_pdfmapline:D \tex_undefined:D
1281 \tex_fi:D

```

A few packages do unfortunate things to date-related primitives.

```

1282 \tex_begingroup:D
1283 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_time:D }
1284 \tex_edef:D \l_tmpb_tl { \tex_string:D \time }
1285 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1286 \tex_else:D
1287 \tex_global:D \tex_let:D \tex_time:D \tex_undefined:D
1288 \tex_fi:D
1289 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_day:D }
1290 \tex_edef:D \l_tmpb_tl { \tex_string:D \day }
1291 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1292 \tex_else:D
1293 \tex_global:D \tex_let:D \tex_day:D \tex_undefined:D
1294 \tex_fi:D
1295 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_month:D }
1296 \tex_edef:D \l_tmpb_tl { \tex_string:D \month }
1297 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1298 \tex_else:D
1299 \tex_global:D \tex_let:D \tex_month:D \tex_undefined:D
1300 \tex_fi:D
1301 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_year:D }
1302 \tex_edef:D \l_tmpb_tl { \tex_string:D \year }
1303 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1304 \tex_else:D
1305 \tex_global:D \tex_let:D \tex_year:D \tex_undefined:D
1306 \tex_fi:D
1307 \tex_endgroup:D

```

cslatex moves a couple of primitives which we recover here; as there is no other marker, we can only work by looking for the names.

```

1308 \tex_ifdefined:D \orieveryjob
1309 \tex_let:D \tex_everyjob:D \orieveryjob
1310 \tex_fi:D
1311 \tex_ifdefined:D \oripdfoutput
1312 \tex_let:D \tex_pdfoutput:D \oripdfoutput
1313 \tex_fi:D

```

For ConTeXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using `\end` as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConTeXt.

```

1314 \tex_ifdefined:D \normalend
1315 \tex_let:D \tex_end:D \normalend
1316 \tex_let:D \tex_everyjob:D \normaleveryjob
1317 \tex_let:D \tex_input:D \normalinput

```

```

1318 \tex_let:D \tex_language:D \normallanguage
1319 \tex_let:D \tex_mathop:D \normalmathop
1320 \tex_let:D \tex_month:D \normalmonth
1321 \tex_let:D \tex_outer:D \normalouter
1322 \tex_let:D \tex_over:D \normalover
1323 \tex_let:D \tex_vcenter:D \normalvcenter
1324 \tex_let:D \tex_unexpanded:D \normalunexpanded
1325 \tex_let:D \tex_expanded:D \normalexpanded
1326 \tex_fi:D
1327 \tex_ifdefined:D \normalitaliccorrection
1328 \tex_let:D \tex_hoffset:D \normalhoffset
1329 \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1330 \tex_let:D \tex_voffset:D \normalvoffset
1331 \tex_let:D \tex_showtokens:D \normalshowtokens
1332 \tex_let:D \tex_bodydir:D \spac_directions_normal_body_dir
1333 \tex_let:D \tex_pagedir:D \spac_directions_normal_page_dir
1334 \tex_fi:D
1335 \tex_ifdefined:D \normalleft
1336 \tex_let:D \tex_left:D \normalleft
1337 \tex_let:D \tex_middle:D \normalmiddle
1338 \tex_let:D \tex_right:D \normalright
1339 \tex_fi:D
1340 </tex>

```

In LuaTeX, we additionally emulate some primitives using Lua code.

```

1341 (*lua)

```

`\tex_strcmp:D` Compare two strings, expanding to 0 if they are equal, -1 if the first one is smaller and 1 if the second one is smaller. Here “smaller” refers to codepoint order which does not correspond to the user expected order for most non-ASCII strings.

```

1342 local minus_tok = token_new(string.byte'-' , 12)
1343 local zero_tok = token_new(string.byte'0' , 12)
1344 local one_tok = token_new(string.byte'1' , 12)
1345 luacmd('tex_strcmp:D', function()
1346   local first = scan_string()
1347   local second = scan_string()
1348   if first < second then
1349     put_next(minus_tok, one_tok)
1350   else
1351     put_next(first == second and zero_tok or one_tok)
1352   end
1353 end, 'global')

```

(End of definition for `\tex_strcmp:D`.)

`\tex_Ucharcat:D` Creating arbitrary chars using `tex.cprint`. The alternative approach using `token.new(...)` is about 10% slower but needed to create arbitrary space tokens.

```

1354 local sprint = tex.sprint
1355 local cprint = tex.cprint
1356 luacmd('tex_Ucharcat:D', function()
1357   local charcode = scan_int()
1358   local catcode = scan_int()
1359   if catcode == 10 then
1360     sprint(token_new(charcode, 10))

```

```

1361 else
1362   cprint(catcode, utf8_char(charcode))
1363 end
1364 end, 'global')

```

(End of definition for `\tex_Ucharcat:D`.)

`\tex_filesize:D` Wrap the function from `ltxutils`.

```

1365 luacmd('tex_filesize:D', function()
1366   local size = filesize(scan_string())
1367   if size then write(size) end
1368 end, 'global')

```

(End of definition for `\tex_filesize:D`.)

`\tex_md5sum:D` There are two cases: Either hash a file or a string. Both are already implemented in `l3luatex` or built-in.

```

1369 luacmd('tex_md5sum:D', function()
1370   local hash
1371   if scan_keyword"file" then
1372     hash = filemd5sum(scan_string())
1373   else
1374     hash = md5_HEX(scan_string())
1375   end
1376   if hash then write(hash) end
1377 end, 'global')

```

(End of definition for `\tex_md5sum:D`.)

`\tex_filemoddate:D` A primitive for getting the modification date of a file.

```

1378 luacmd('tex_filemoddate:D', function()
1379   local date = filemoddate(scan_string())
1380   if date then write(date) end
1381 end, 'global')

```

(End of definition for `\tex_filemoddate:D`.)

`\tex_filedump:D` An emulated primitive for getting a hexdump from a (partial) file. The length has a default of 0. This is consistent with `pdfTeX`, but it effectively makes the primitive useless without an explicit `length`. Therefore we allow the keyword `whole` to be used instead of a length, indicating that the whole remaining file should be read.

```

1382 luacmd('tex_filedump:D', function()
1383   local offset = scan_keyword'offset' and scan_int() or nil
1384   local length = scan_keyword'length' and scan_int()
1385                 or not scan_keyword'whole' and 0 or nil
1386   local data = filedump(scan_string(), offset, length)
1387   if data then write(data) end
1388 end, 'global')

```

(End of definition for `\tex_filedump:D`.)

```

1389 </lua>
1390 </package>

```

Chapter 41

I3kernel-functions: kernel-reserved functions

41.1 Internal I3debug kernel functions

These function are only created if debugging is enabled, hence they are actually defined in I3debug.

`_kernel_chk_var_local:N` `_kernel_chk_var_local:N` $\langle var \rangle$
`_kernel_chk_var_global:N` `_kernel_chk_var_global:N` $\langle var \rangle$

Applies `_kernel_chk_var_exist:N` $\langle var \rangle$ as well as `_kernel_chk_var_scope:NN` $\langle scope \rangle$ $\langle var \rangle$, where $\langle scope \rangle$ is l or g.

`_kernel_chk_var_scope:NN` `_kernel_chk_var_scope:NN` $\langle scope \rangle$ $\langle var \rangle$

Checks the $\langle var \rangle$ has the correct $\langle scope \rangle$, and if not raises a kernel-level error. The $\langle scope \rangle$ is a single letter l, g, c denoting local variables, global variables, or constants. More precisely, if the variable name starts with a letter and an underscore (normal `expl3` convention) the function checks that this single letter matches the $\langle scope \rangle$. Otherwise the function cannot know the scope $\langle var \rangle$ the first time: instead, it defines `_debug_chk_/ $\langle var \text{ name} \rangle$` to store that information for the next call. Thus, if a given $\langle var \rangle$ is subject to assignments of different scopes a kernel error will result.

`_kernel_chk_cs_exist:N` `_kernel_chk_cs_exist:N` $\langle cs \rangle$
`_kernel_chk_cs_exist:c` `_kernel_chk_var_exist:N` $\langle var \rangle$

`_kernel_chk_var_exist:N` Checks that their argument is defined according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error. Error messages are different.

`_kernel_chk_flag_exist:NN` * `_kernel_chk_flag_exist:NN`
 $\langle function \rangle$ $\langle flag \rangle$

Checks that the $\langle flag \rangle$ is defined according to the criterion for `\flag_if_exist_p:N`, and if not raises a kernel-level error and calls the function with the argument `\l_tmpa_flag` to proceed somehow without producing too many errors.

`_kernel_debug_log:e` `_kernel_debug_log:e` $\langle message\ text\rangle$

If the `log-functions` option is active, this function writes the $\langle message\ text\rangle$ to the log file using `\iow_log:e`. Otherwise, the $\langle message\ text\rangle$ is ignored using `\use_none:n`.

41.2 Internal kernel functions

`_kernel_chk_defined:NT` `_kernel_chk_defined:NT` $\langle variable\rangle$ $\langle true\ code\rangle$

If $\langle variable\rangle$ is not defined (according to `\cs_if_exist:NTF`), this triggers an error, otherwise the $\langle true\ code\rangle$ is run.

`_kernel_chk_expr:nNn` `_kernel_chk_expr:nNn` $\langle expr\rangle$ $\langle eval\rangle$ $\langle convert\rangle$ $\langle caller\rangle$

This function is only created if debugging is enabled. By default it is equivalent to `\use_i:nmnn`. When expression checking is enabled, it leaves in the input stream the result of `\tex_the:D` $\langle eval\rangle$ $\langle expr\rangle$ `\tex_relax:D` after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the $\langle caller\rangle$. For instance $\langle eval\rangle$ can be `_int_eval:w` and $\langle caller\rangle$ can be `\int_eval:n` or `\int_set:Nn`. The argument $\langle convert\rangle$ is empty except for mu expressions where it is `\tex_mutogluae:D`, used for internal purposes.

`_kernel_chk_tl_type:NnnT` `_kernel_chk_tl_type:NnnT` $\langle control\ sequence\rangle$ $\langle specific\ type\rangle$
 $\langle reconstruction\rangle$ $\langle true\ code\rangle$

Helper to test that the $\langle control\ sequence\rangle$ is a variable of the given $\langle specific\ type\rangle$ of token list. Produces suitable error messages if the $\langle control\ sequence\rangle$ does not exist, or if it is not a token list variable at all, or if the $\langle control\ sequence\rangle$ differs from the result of e-expanding $\langle reconstruction\rangle$. If all of these tests succeed then the $\langle true\ code\rangle$ is run.

`_kernel_codepoint_to_bytes:n` `_kernel_codepoint_to_bytes:n` $\langle codepoint\rangle$

Converts the $\langle codepoint\rangle$ to UTF-8 bytes. The expansion of this function comprises four brace groups, each of which will contain a hexadecimal value: the appropriate byte. As UTF-8 is a variable-length, one or more of the groups may be empty: the bytes read in the logical order, such that a two-byte codepoint will have groups #1 and #2 filled and #3 and #4 empty.

`_kernel_cs_parm_from_arg_count:nnF` `_kernel_cs_parm_from_arg_count:nnF` $\langle follow-on\rangle$ $\langle args\rangle$
 $\langle false\ code\rangle$

Evaluates the number of $\langle args\rangle$ and leaves the $\langle follow-on\rangle$ code followed by a brace group containing the required number of primitive parameter markers (#1, etc.). If the number of $\langle args\rangle$ is outside the range [0,9], the $\langle false\ code\rangle$ is inserted *instead* of the $\langle follow-on\rangle$.

`__kernel_dependency_version_check:Nn` `__kernel_dependency_version_check:Nn` $\langle\text{date}\rangle$ $\langle\text{file}\rangle$
`__kernel_dependency_version_check:nn` `__kernel_dependency_version_check:nn` $\langle\text{date}\rangle$ $\langle\text{file}\rangle$

Checks if the loaded version of the expl3 kernel is at least $\langle\text{date}\rangle$, required by $\langle\text{file}\rangle$. If the kernel date is older than $\langle\text{date}\rangle$, the loading of $\langle\text{file}\rangle$ is aborted and an error is raised.

`__kernel_deprecation_code:nn` `__kernel_deprecation_code:nn` $\langle\text{error code}\rangle$ $\langle\text{working code}\rangle$

Stores both an $\langle\text{error}\rangle$ and $\langle\text{working}\rangle$ definition for given material such that they can be exchanged by `\debug_on:n` and `\debug_off:n`.

`__kernel_exp_not:w` \star `__kernel_exp_not:w` $\langle\text{expandable tokens}\rangle$ $\langle\text{content}\rangle$

Carries out expansion on the $\langle\text{expandable tokens}\rangle$ before preventing further expansion of the $\langle\text{content}\rangle$ as for `\exp_not:n`. Typically, the $\langle\text{expandable tokens}\rangle$ will alter the nature of the $\langle\text{content}\rangle$, *i.e.* allow it to be generated in some way.

`\l__kernel_expl_bool` A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by `\ExplSyntaxOn/\ExplSyntaxOff`.

(End of definition for `\l__kernel_expl_bool`.)

`\c__kernel_expl_date_tl` A token list containing the release date of the l3kernel preloaded in L^AT_EX 2_ε used to check if dependencies match.

(End of definition for `\c__kernel_expl_date_tl`.)

`__kernel_file_missing:n` `__kernel_file_missing:n` $\langle\text{name}\rangle$

Expands the $\langle\text{name}\rangle$ as per `__kernel_file_name_sanitize:n` then produces an error message indicating that this file was not found.

`__kernel_file_name_sanitize:n` \star `__kernel_file_name_sanitize:n` $\langle\text{name}\rangle$

Updated: 2021-04-17

Expands the file name using a `\csname`-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using `\ifcsname`. This is less conservative than the token-by-token approach used before, but it is much faster.

`__kernel_file_input_push:n` `__kernel_file_input_push:n` $\langle\text{name}\rangle$

`__kernel_file_input_pop:` `__kernel_file_input_pop:`

Used to push and pop data from the internal file stack: needed only in package mode, where interfacing with the L^AT_EX 2_ε kernel is necessary.

`__kernel_int_add:nnn` \star `__kernel_int_add:nnn` $\langle\text{integer}_1\rangle$ $\langle\text{integer}_2\rangle$ $\langle\text{integer}_3\rangle$

Expands to the result of adding the three $\langle\text{integers}\rangle$ (which must be suitable input for `\int_eval:w`), avoiding intermediate overflow. Overflow occurs only if the overall result is outside $[-2^{31} + 1, 2^{31} - 1]$. The $\langle\text{integers}\rangle$ may be of the form `\int_eval:w` ... `\scan_stop:` but may be evaluated more than once.

`__kernel_intarray_gset:Nnn` `__kernel_intarray_gset:Nnn` $\langle intarray\ var\rangle$ $\{\langle index\rangle\}$ $\{\langle value\rangle\}$

New: 2018-03-31

Faster version of `\intarray_gset:Nnn`. Stores the $\langle value\rangle$ into the $\langle integer\ array\ variable\rangle$ at the $\langle position\rangle$. The $\langle index\rangle$ and $\langle value\rangle$ must be suitable for a direct assignment to a TeX count register, for instance expanding to an integer denotation or obtained through the primitive `\numexpr` (which may be un-terminated). No bound checking is performed: the caller is responsible for ensuring that the $\langle position\rangle$ is between 1 and the `\intarray_count:N`, and the $\langle value\rangle$'s absolute value is at most $2^{30}-1$. Assignments are always global.

`__kernel_intarray_item:Nn` \star `__kernel_intarray_item:Nn` $\langle intarray\ var\rangle$ $\{\langle index\rangle\}$

New: 2018-03-31

Faster version of `\intarray_item:Nn`. Expands to the integer entry stored at the $\langle index\rangle$ in the $\langle integer\ array\ variable\rangle$. The $\langle index\rangle$ must be suitable for a direct assignment to a TeX count register and must be between 1 and the `\intarray_count:N`, lest a low-level TeX error occur.

`__kernel_intarray_range_to_clist:Nnn` \star `__kernel_intarray_range_to_clist:Nnn` $\langle intarray\ var\rangle$ $\{\langle start\ index\rangle\}$ $\{\langle end\ index\rangle\}$

New: 2020-07-12

Converts to integer denotations separated by commas the entries of the $\langle intarray\rangle$ from positions $\langle start\ index\rangle$ to $\langle end\ index\rangle$ included. The $\langle start\ index\rangle$ and $\langle end\ index\rangle$ must be suitable for a direct assignment to a TeX count register, must be between 1 and the `\intarray_count:N`, and be suitably ordered. All tokens have category code other.

`__kernel_intarray_gset_range_from_clist:Nnn` `__kernel_intarray_gset_range_from_clist:Nnn` $\langle intarray\ var\rangle$ $\{\langle start\ index\rangle\}$ $\{\langle integer\ clist\rangle\}$

New: 2020-07-12

Stores the entries of the $\langle clist\rangle$ as entries of the $\langle intarray\ var\rangle$ starting from the $\langle start\ index\rangle$, upwards. This is done without any bound checking. The $\langle start\ index\rangle$ and all entries of the $\langle integer\ comma\ list\rangle$ (which do not undergo space trimming and brace stripping as in normal clist mappings) must be suitable for a direct assignment to a TeX count register. An empty entry may stop the loop.

`__kernel_ior_open:Nn` `__kernel_ior_open:Nn` $\langle stream\rangle$ $\{\langle file\ name\rangle\}$

`__kernel_ior_open:No`

This function has identical syntax to the public version. However, it does not take precautions against active characters in the $\langle file\ name\rangle$, and it does not attempt to add a $\langle path\rangle$ to the $\langle file\ name\rangle$: it is therefore intended to be used by higher-level functions which have already fully expanded the $\langle file\ name\rangle$ and which need to perform multiple open or close operations. See for example the implementation of `\ior_shell_open:Nn`.

`__kernel_iow_open:Nn` `__kernel_iow_open:Nn` $\langle stream\rangle$ $\{\langle file\ name\rangle\}$

`__kernel_iow_open:No`

This function has identical syntax to the public version. However, it does not take precautions against active characters in the $\langle file\ name\rangle$, and it does not attempt to add a $\langle path\rangle$ to the $\langle file\ name\rangle$: it is therefore intended to be used by higher-level functions which have already fully expanded the $\langle file\ name\rangle$ and which need to perform multiple open or close operations. See for example the implementation of `\iow_shell_open:Nn`.

`_kernel_iow_with:Nnn` `_kernel_iow_with:Nnn` $\langle integer \rangle$ $\{\langle value \rangle\}$ $\{\langle code \rangle\}$

If the $\langle integer \rangle$ is equal to the $\langle value \rangle$ then this function simply runs the $\langle code \rangle$. Otherwise it saves the current value of the $\langle integer \rangle$, sets it to the $\langle value \rangle$, runs the $\langle code \rangle$, and restores the $\langle integer \rangle$ to its former value. This is used to ensure that the `\newlinechar` is 10 when writing to a stream, which lets `\iow_newline:` work, and that `\errorcontextlines` is -1 when displaying a message.

`_kernel_kern:n` `_kernel_kern:n` $\{\langle length \rangle\}$

Inserts a kern of the specified $\langle length \rangle$, a dimension expression.

(End of definition for `_kernel_kern:n`.)

`_kernel_msg_show_eval:Nn` `_kernel_msg_show_eval:Nn` $\langle function \rangle$ $\{\langle expression \rangle\}$

`_kernel_msg_log_eval:Nn` Shows or logs the $\langle expression \rangle$ (turned into a string), an equal sign, and the result of applying the $\langle function \rangle$ to the $\{\langle expression \rangle\}$ (with f-expansion). For instance, if the $\langle function \rangle$ is `\int_eval:n` and the $\langle expression \rangle$ is `1+2` then this logs `> 1+2=3`.

`_kernel_pdf_object_id:n` \star `_kernel_pdf_object_id:n` $\{\langle object \rangle\}$

`_kernel_pdf_object_id_indexed:nn` \star `_kernel_pdf_object_id_indexed:nn` $\{\langle class \rangle\}$ $\{\langle number \rangle\}$

Expands to the ID of $\langle object \rangle$ (or object of $\langle number \rangle$ within the $\langle class \rangle$), in for example page resource allocation. Depending on the backend, the result may be the same as `\pdf_object_id:n/\pdf_object_id_indexed:nn`.

`\g__kernel_prg_map_int` This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions `\langle type \rangle_map_1:w`, `\langle type \rangle_map_2:w`, etc., labelled by `\g__kernel_prg_map_int` hold functions to be mapped over various list datatypes in inline and variable mappings.

(End of definition for `\g__kernel_prg_map_int`.)

`__kernel_quark_new_test:N` `__kernel_quark_new_test:N` $\langle name \rangle : \langle arg\ spec \rangle$

Defines a quark-test function $\langle name \rangle : \langle arg\ spec \rangle$ which tests if its argument is `\q__ $\langle namespace \rangle$ _recursion_tail`, then acts accordingly, as described below for each possible $\langle arg\ spec \rangle$.

The $\langle namespace \rangle$ is determined as the first (nonempty) `_`-delimited word in $\langle name \rangle$ and is used internally in the definition of auxiliaries. The function `__kernel_quark_new_test:N` does *not* define the `\q__ $\langle namespace \rangle$ _recursion_tail` and `\q__ $\langle namespace \rangle$ _recursion_stop` quarks. They should be manually defined with `\quark_new:N`.

There are 6 different types of quark-test functions. Which one is defined depends on the $\langle arg\ spec \rangle$, which *must* be one of the options listed now. Four of them are modeled after `\quark_if_recursion_tail:(N|n)` and `\quark_if_recursion_tail_do:(N|n)n`.

`n` defines $\langle name \rangle : n$ such that it checks if #1 contains only `\q__ $\langle namespace \rangle$ _recursion_tail`, and if so consumes all tokens up to `\q__ $\langle namespace \rangle$ _recursion_stop` (*c.f.* `\quark_if_recursion_tail_stop:n`).

`nn` defines $\langle name \rangle : nn$ such that it checks if #1 contains only `\q__ $\langle namespace \rangle$ _recursion_tail`, and if so consumes all tokens up to `\q__ $\langle namespace \rangle$ _recursion_stop`, then executes the code #2 after that (*c.f.* `\quark_if_recursion_tail_stop_do:nn`).

`N` defines $\langle name \rangle : N$ such that it checks if #1 is `\q__ $\langle namespace \rangle$ _recursion_tail`, and if so consumes all tokens up to `\q__ $\langle namespace \rangle$ _recursion_stop` (*c.f.* `\quark_if_recursion_tail_stop:N`).

`Nn` defines $\langle name \rangle : Nn$ such that it checks if #1 is `\q__ $\langle namespace \rangle$ _recursion_tail`, and if so consumes all tokens up to `\q__ $\langle namespace \rangle$ _recursion_stop`, then executes the code #2 after that (*c.f.* `\quark_if_recursion_tail_stop_do:Nn`).

The last two are modeled after `\quark_if_recursion_tail_break:(n|N)N`, and in those cases the quark `\q__ $\langle namespace \rangle$ _recursion_stop` is not used (and thus needs not be defined).

`nN` defines $\langle name \rangle : nN$ such that it checks if #1 contains only `\q__ $\langle namespace \rangle$ _recursion_tail`, and if so uses the `\langle type \rangle_map_break:` function #2.

`NN` defines $\langle name \rangle : NN$ such that it checks if #1 is `\q__ $\langle namespace \rangle$ _recursion_tail`, and if so uses the `\langle type \rangle_map_break:` function #2.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

_kernel_quark_new_conditional:Nn _kernel_quark_new_conditional:Nn
 _*namespace*_quark_if_*name*\:(*arg spec*) {\i>conditions}

Defines a collection of quark conditionals that test if their argument is the quark `\q_\namespace_\name` and perform suitable actions. The *conditions* are a comma-separated list of one or more of p, T, F, and TF, and one conditional is defined for each *condition* in the list, as described for `\prg_new_conditional:Npnn`. The conditionals are defined using `\prg_new_conditional:Npnn`, so that their name is obtained by adding p, T, F, or TF to the base name `_\namespace_quark_if_\name\:(arg spec)`.

The first argument of `_kernel_quark_new_conditional:Nn` must contain `_quark_if_` and `:`, as these markers are used to determine the *name* of the quark `\q_\namespace_\name` to be tested. This quark should be manually defined with `\quark_new:N`, as `_kernel_quark_new_conditional:Nn` does *not* define it.

The function `_kernel_quark_new_conditional:Nn` can define 2 different types of quark conditionals. Which one is defined depends on the *arg spec*, which *must* be one of the following options, modeled after `\quark_if_nil:(N|n)(TF)`.

n defines `_\namespace_quark_if_\name\:n(TF)` such that it checks if #1 contains only `\q_\namespace_\name`, and executes the proper conditional branch.

N defines `_\namespace_quark_if_\name\:N(TF)` such that it checks if #1 is `\q_\namespace_\name`, and executes the proper conditional branch.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

_kernel_sys_everyjob: _kernel_sys_everyjob:

Inserts the internal token list required at the start of every run (job).

`\c_kernel_randint_max_int` Maximal allowed argument to `_kernel_randint:n`. Equal to $2^{17} - 1$.

(End of definition for `\c_kernel_randint_max_int`.)

_kernel_randint:n _kernel_randint:n {\i>max}

Used in an integer expression this gives a pseudo-random number between 1 and *max* included. One must have $\langle max \rangle \leq 2^{17} - 1$. The *max* must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent).

_kernel_randint:nn _kernel_randint:nn {\i>min} {\i>max}

Used in an integer expression this gives a pseudo-random number between *min* and *max* included. The *min* and *max* must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent). For small ranges $R = \langle max \rangle - \langle min \rangle + 1 \leq 2^{17} - 1$, `\langle min \rangle - 1 + _kernel_randint:n\{R\}` is faster.

_kernel_register_show:N _kernel_register_show:N (*register*)

_kernel_register_show:c Used to show the contents of a T_EX register at the terminal, formatted such that internal parts of the mechanism are not visible.

_kernel_register_log:N _kernel_register_log:N (*register*)

_kernel_register_log:c Used to write the contents of a T_EX register to the log file in a form similar to `_kernel_register_show:N`.

`_kernel_str_to_other:n` * `_kernel_str_to_other:n` $\{ \langle token\ list \rangle \}$

Converts the $\langle token\ list \rangle$ to a $\langle other\ string \rangle$, where spaces have category code “other”. This function can be f-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

`_kernel_str_to_other_fast:n` * `_kernel_str_to_other_fast:n` $\{ \langle token\ list \rangle \}$

Same behaviour `_kernel_str_to_other:n` but only restricted-expandable. It takes a time linear in the character count of the string.

`_kernel_tl_to_str:w` * `_kernel_tl_to_str:w` $\langle expandable\ tokens \rangle$ $\{ \langle tokens \rangle \}$

Carries out expansion on the $\langle expandable\ tokens \rangle$ before conversion of the $\langle tokens \rangle$ to a string as describe for `\tl_to_str:n`. Typically, the $\langle expandable\ tokens \rangle$ will alter the nature of the $\langle tokens \rangle$, *i.e.* allow it to be generated in some way. This function requires only a single expansion.

`_kernel_tl_set:Nx` `_kernel_tl_set:Nx` $\langle tl\ var \rangle$ $\{ \langle tokens \rangle \}$

`_kernel_tl_gset:Nx`

Fully expands $\langle tokens \rangle$ and assigns the result to $\langle tl\ var \rangle$. $\langle tokens \rangle$ must be given in braces and there must be no token between $\langle tl\ var \rangle$ and $\langle tokens \rangle$.

`_kernel_codepoint_data:nn` * `_kernel_codepoint_data:nn` $\{ \langle type \rangle \}$ $\{ \langle codepoint \rangle \}$

Expands to the appropriate value for the $\langle type \rangle$ of data requested for a $\langle codepoint \rangle$. The current list of $\langle types \rangle$ and results are

lowercase The *single* codepoint specified by `UnicodeData.txt` for lowercase mapping of the codepoint: will be equal to the input $\langle codepoint \rangle$ if there is no mapping specified in `UnicodeData.txt`

uppercase The *single* codepoint specified by `UnicodeData.txt` for uppercase mapping of the codepoint: will be equal to the input $\langle codepoint \rangle$ if there is no mapping specified in `UnicodeData.txt`

`_kernel_codepoint_case:nn` * `_kernel_codepoint_case:nn` $\{ \langle mapping \rangle \}$ $\{ \langle codepoint \rangle \}$

Expands to a list of three balanced text, of which at least the first will contain a codepoint. This list of up to three codepoints specifies the full case mapping for the input $\langle codepoint \rangle$. The $\langle mapping \rangle$ should be one of

- `casefold`
- `lowercase`
- `titlecase`
- `uppercase`

41.3 Kernel backend functions

These functions are required to pass information to the backend. The nature of these means that they are defined only when the relevant backend is in use.

```
\__kernel_backend_literal:n \__kernel_backend_literal:n {<content>}  
\__kernel_backend_literal:(e|e)
```

Adds the $\langle content \rangle$ literally to the current vertical list as a whatsit. The nature of the $\langle content \rangle$ will depend on the backend in use.

```
\__kernel_backend_literal_postscript:n \__kernel_backend_literal_postscript:n {<PostScript>}  
\__kernel_backend_literal_postscript:e
```

Adds the $\langle PostScript \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

```
\__kernel_backend_literal_pdf:n \__kernel_backend_literal_pdf:n {<PDF instructions>}  
\__kernel_backend_literal_pdf:e
```

Adds the $\langle PDF instructions \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

```
\__kernel_backend_literal_svg:n \__kernel_backend_literal_svg:n {<SVG instructions>}  
\__kernel_backend_literal_svg:e
```

Adds the $\langle SVG instructions \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

```
\__kernel_backend_postscript:n \__kernel_backend_postscript:n {<PostScript>}  
\__kernel_backend_postscript:e
```

Adds the $\langle PostScript \rangle$ to the current vertical list as a whatsit. The PostScript reference point is adjusted to match the current position. The PostScript is inserted inside a SDict begin/end pair.

```
\__kernel_backend_align_begin: \__kernel_backend_align_begin:  
\__kernel_backend_align_end: <PostScript literals>  
\__kernel_backend_align_end:
```

Arranges to align the PostScript and DVI current positions and scales.

```
\__kernel_backend_scope_begin: \__kernel_backend_scope_begin:  
\__kernel_backend_scope_end: <content>  
\__kernel_backend_scope_end:
```

Creates a scope for instructions at the backend level.

```
\__kernel_backend_matrix:n \__kernel_backend_matrix:n {<matrix>}  
\__kernel_backend_matrix:e
```

Applies the $\langle matrix \rangle$ to the current transformation matrix.

```
\g__kernel_backend_header_bool
```

Specifies whether to write headers for the backend.

`\l__kernel_color_stack_int` The color stack used in pdfTeX and LuaTeX for the main color.

Chapter 42

l3basics implementation

1391 (*package)

42.1 Renaming some T_EX primitives (again)

Having given all the T_EX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.⁸

```
\if_true: Then some conditionals.
\if_false: 1392 \tex_global:D \tex_let:D \if_true:          \tex_iftrue:D
\or:       1393 \tex_global:D \tex_let:D \if_false:       \tex_iffalse:D
\else:     1394 \tex_global:D \tex_let:D \or:                \tex_or:D
\fi:       1395 \tex_global:D \tex_let:D \else:           \tex_else:D
\reverse_if:N 1396 \tex_global:D \tex_let:D \fi:            \tex_fi:D
\if:w      1397 \tex_global:D \tex_let:D \reverse_if:N      \tex_unless:D
\if_charcode:w 1398 \tex_global:D \tex_let:D \if:w              \tex_if:D
\if_catcode:w 1399 \tex_global:D \tex_let:D \if_charcode:w      \tex_if:D
\if_meaning:w 1400 \tex_global:D \tex_let:D \if_catcode:w          \tex_ifcat:D
\if_meaning:w 1401 \tex_global:D \tex_let:D \if_meaning:w      \tex_ifx:D
\if_meaning:w 1402 \tex_global:D \tex_let:D \if_bool:N       \tex_ifodd:D
```

(End of definition for `\if_true:` and others. These functions are documented on page 28.)

```
\if_mode_math: TEX lets us detect some if its modes.
\if_mode_horizontal: 1403 \tex_global:D \tex_let:D \if_mode_math:          \tex_ifmmode:D
\if_mode_vertical:   1404 \tex_global:D \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\if_mode_inner:      1405 \tex_global:D \tex_let:D \if_mode_vertical:   \tex_ifvmode:D
\if_mode_inner:      1406 \tex_global:D \tex_let:D \if_mode_inner:      \tex_ifinner:D
```

(End of definition for `\if_mode_math:` and others. These functions are documented on page 29.)

```
\if_cs_exist:N Building csnames and testing if control sequences exist.
\if_cs_exist:w 1407 \tex_global:D \tex_let:D \if_cs_exist:N      \tex_ifdefined:D
\cs:w          1408 \tex_global:D \tex_let:D \if_cs_exist:w      \tex_ifcurname:D
\cs_end:       1409 \tex_global:D \tex_let:D \cs:w                \tex_csname:D
\cs_end:       1410 \tex_global:D \tex_let:D \cs_end:                \tex_endcurname:D
```

⁸This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the `\tex_...:D` name in the cases where no good alternative exists.

(End of definition for `\if_cs_exist:N` and others. These functions are documented on page 29.)

`\exp_after:wN` The five `\exp_` functions are used in the `l3expan` module where they are described.

```
\exp_not:N 1411 \tex_global:D \tex_let:D \exp_after:wN \tex_expandafter:D
\exp_not:n 1412 \tex_global:D \tex_let:D \exp_not:N \tex_noexpand:D
1413 \tex_global:D \tex_let:D \exp_not:n \tex_unexpanded:D
1414 \tex_global:D \tex_let:D \exp:w \tex_romannumeral:D
1415 \tex_global:D \tex_chardef:D \exp_end: = 0 ~
```

(End of definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 39.)

`\token_to_meaning:N` Examining a control sequence or token.

```
\cs_meaning:N 1416 \tex_global:D \tex_let:D \token_to_meaning:N \tex_meaning:D
1417 \tex_global:D \tex_let:D \cs_meaning:N \tex_meaning:D
```

(End of definition for `\token_to_meaning:N` and `\cs_meaning:N`. These functions are documented on page 201.)

`\tl_to_str:n` Making strings.

```
\token_to_str:N 1418 \tex_global:D \tex_let:D \tl_to_str:n \tex_detokenize:D
\__kernel_tl_to_str:w 1419 \tex_global:D \tex_let:D \token_to_str:N \tex_string:D
1420 \tex_global:D \tex_let:D \__kernel_tl_to_str:w \tex_detokenize:D
```

(End of definition for `\tl_to_str:n`, `\token_to_str:N`, and `__kernel_tl_to_str:w`. These functions are documented on page 115.)

`\scan_stop:` The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the `l3prg` module.

```
\group_begin: 1421 \tex_global:D \tex_let:D \scan_stop: \tex_relax:D
\group_end: 1422 \tex_global:D \tex_let:D \group_begin: \tex_begingroup:D
1423 \tex_global:D \tex_let:D \group_end: \tex_endgroup:D
```

(End of definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 13.)

```
1424 (@@=int)
```

`\if_int_compare:w` For integers.

```
\__int_to_roman:w 1425 \tex_global:D \tex_let:D \if_int_compare:w \tex_ifnum:D
1426 \tex_global:D \tex_let:D \__int_to_roman:w \tex_romannumeral:D
```

(End of definition for `\if_int_compare:w` and `__int_to_roman:w`. This function is documented on page 179.)

`\group_insert_after:N` Adding material after the end of a group.

```
1427 \tex_global:D \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(End of definition for `\group_insert_after:N`. This function is documented on page 14.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

```
\exp_args:cc 1428 \tex_long:D \tex_gdef:D \exp_args:Nc #1#2
1429 { \exp_after:wN #1 \cs:w #2 \cs_end: }
1430 \tex_long:D \tex_gdef:D \exp_args:cc #1#2
1431 { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
```

(End of definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 36.)

`\token_to_meaning:c` A small number of variants defined by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:NNc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```

1432 \tex_gdef:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1433 \tex_long:D \tex_gdef:D \cs_meaning:c #1
1434   {
1435     \if_cs_exist:w #1 \cs_end:
1436       \exp_after:wN \use_i:nn
1437     \else:
1438       \exp_after:wN \use_ii:nn
1439     \fi:
1440     { \exp_args:Nc \cs_meaning:N {#1} }
1441     { \tl_to_str:n {undefined} }
1442   }
1443 \tex_global:D \tex_let:D \token_to_meaning:c = \cs_meaning:c

```

(End of definition for `\token_to_meaning:N`. This function is documented on page 201.)

42.2 Defining some constants

`\c_zero_int` We need the constant `\c_zero_int` which is used by some functions in current module. The rest are defined in the `l3int` module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is required but it can't be used until the allocation has been set up properly!

```

1444 \tex_global:D \tex_chardef:D \c_zero_int = 0 ~

```

(End of definition for `\c_zero_int`. This variable is documented on page 178.)

`\c_max_register_int` This is here as this particular integer is needed in modules loaded before `l3int`, and is documented in `l3int`. Lua \TeX and those which contain parts of the Omega extensions have more registers available than $\varepsilon\text{-}\TeX$.

```

1445 \tex_ifdefined:D \tex_luatexversion:D
1446   \tex_global:D \tex_chardef:D \c_max_register_int = 65 535 ~
1447 \tex_else:D
1448   \tex_ifdefined:D \tex_omathchardef:D
1449     \tex_global:D \tex_omathchardef:D \c_max_register_int = 65535 ~
1450   \tex_else:D
1451     \tex_global:D \tex_mathchardef:D \c_max_register_int = 32767 ~
1452   \tex_fi:D
1453 \tex_fi:D

```

(End of definition for `\c_max_register_int`. This variable is documented on page 178.)

42.3 Defining functions

We start by providing functions for the typical definition functions. First the global ones.

`\cs_gset_nopar:Npn` All assignment functions in L \TeX 3 should be naturally protected; after all, the \TeX primitives for assignments are and it can be a cause of problems if others aren't.

```

\cs_gset_nopar:Npe
\cs_gset_nopar:Npx
\cs_gset:Npn
\cs_gset:Npe
\cs_gset:Npx
1454 \tex_global:D \tex_let:D \cs_gset_nopar:Npn          \tex_gdef:D
1455 \tex_global:D \tex_let:D \cs_gset_nopar:Npe          \tex_xdef:D

```

```

\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npe
\cs_gset_protected_nopar:Npx
\cs_gset_protected:Npn
\cs_gset_protected:Npe
\cs_gset_protected:Npx

```



```

1456 \tex_global:D \tex_let:D \cs_gset_nopar:Npx          \tex_xdef:D
1457 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset:Npn
1458   { \tex_long:D \tex_gdef:D }
1459 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset:Npe
1460   { \tex_long:D \tex_xdef:D }
1461 \tex_global:D \tex_let:D \cs_gset:Npx \cs_gset:Npe
1462 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected_nopar:Npn
1463   { \tex_protected:D \tex_gdef:D }
1464 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected_nopar:Npe
1465   { \tex_protected:D \tex_xdef:D }
1466 \tex_global:D \tex_let:D \cs_gset_protected_nopar:Npx \cs_gset_protected_nopar:Npe
1467 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected:Npn
1468   { \tex_protected:D \tex_long:D \tex_gdef:D }
1469 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected:Npe
1470   { \tex_protected:D \tex_long:D \tex_xdef:D }
1471 \tex_global:D \tex_let:D \cs_gset_protected:Npx \cs_gset_protected:Npe

```

(End of definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 17.)

`\cs_set_nopar:Npn` Local versions of the above functions.

```

\cs_set_nopar:Npe
\cs_set_nopar:Npx
  \cs_set:Npn
  \cs_set:Npe
  \cs_set:Npx
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npe
\cs_set_protected_nopar:Npx
  \cs_set_protected:Npn
  \cs_set_protected:Npe
  \cs_set_protected:Npx
1472 \tex_global:D \tex_let:D \cs_set_nopar:Npn          \tex_def:D
1473 \tex_global:D \tex_let:D \cs_set_nopar:Npe          \tex_edef:D
1474 \tex_global:D \tex_let:D \cs_set_nopar:Npx          \tex_edef:D
1475 \cs_gset_protected:Npn \cs_set:Npn
1476   { \tex_long:D \tex_def:D }
1477 \cs_gset_protected:Npn \cs_set:Npe
1478   { \tex_long:D \tex_edef:D }
1479 \tex_global:D \tex_let:D \cs_set:Npx \cs_set:Npe
1480 \cs_gset_protected:Npn \cs_set_protected_nopar:Npn
1481   { \tex_protected:D \tex_def:D }
1482 \cs_gset_protected:Npn \cs_set_protected_nopar:Npe
1483   { \tex_protected:D \tex_edef:D }
1484 \tex_global:D \tex_let:D \cs_set_protected_nopar:Npx \cs_set_protected_nopar:Npe
1485 \cs_gset_protected:Npn \cs_set_protected:Npn
1486   { \tex_protected:D \tex_long:D \tex_def:D }
1487 \cs_gset_protected:Npn \cs_set_protected:Npe
1488   { \tex_protected:D \tex_long:D \tex_edef:D }
1489 \tex_global:D \tex_let:D \cs_set_protected:Npx \cs_set_protected:Npe

```

(End of definition for `\cs_set_nopar:Npn` and others. These functions are documented on page 16.)

42.4 Selecting tokens

```

1490 <@@=exp>

```

`\l__exp_internal_tl` Scratch token list variable for `l3expan`, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because `l3basics` is loaded earlier.

```

1491 \cs_gset_nopar:Npn \l__exp_internal_tl { }

```

(End of definition for `\l__exp_internal_tl`.)

`\use:c` This macro grabs its argument and returns a csname from it.

```

1492 \cs_gset:Npn \use:c #1 { \cs:w #1 \cs_end: }

```

(End of definition for `\use:c`. This function is documented on page 21.)

`\use:x` Fully expands its argument and passes it to the input stream. Uses the reserved `\l__-exp_internal_tl` which we've set up above.

```

1493 \cs_gset_protected:Npn \use:x #1
1494   {
1495     \cs_set_nopar:Npx \l__exp_internal_tl {#1}
1496     \l__exp_internal_tl
1497   }

```

(End of definition for `\use:x`.)

```

1498 <@@=use>

```

`\use:e`

```

1499 \cs_gset:Npn \use:e #1 { \tex_expanded:D {#1} }

```

(End of definition for `\use:e`. This function is documented on page 26.)

```

1500 <@@=exp>

```

`\use:n` These macros grab their arguments and return them back to the input (with outer braces removed).

`\use:nn`

`\use:nnn`

`\use:nnnn`

```

1501 \cs_gset:Npn \use:n #1 {#1}
1502 \cs_gset:Npn \use:nn #1#2 {#1#2}
1503 \cs_gset:Npn \use:nnn #1#2#3 {#1#2#3}
1504 \cs_gset:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}

```

(End of definition for `\use:n` and others. These functions are documented on page 24.)

`\use_i:nn`

`\use_ii:nn`

The equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

```

1505 \cs_gset:Npn \use_i:nn #1#2 {#1}
1506 \cs_gset:Npn \use_ii:nn #1#2 {#2}

```

(End of definition for `\use_i:nn` and `\use_ii:nn`. These functions are documented on page 25.)

`\use_i:nnn`

`\use_ii:nnn`

`\use_iii:nnn`

`\use_i:nnnn`

`\use_ii:nnnn`

`\use_iii:nnnn`

`\use_iv:nnnn`

`\use_i:nnnnn`

`\use_ii:nnnnn`

`\use_iii:nnnnn`

`\use_iv:nnnnn`

`\use_v:nnnnn`

`\use_i:nnnnnn`

`\use_ii:nnnnnn`

`\use_iii:nnnnnn`

`\use_iv:nnnnnn`

`\use_v:nnnnnn`

`\use_vi:nnnnnn`

`\use_i:nnnnnnn`

`\use_ii:nnnnnnn`

`\use_iii:nnnnnnn`

`\use_iv:nnnnnnn`

`\use_v:nnnnnnn`

`\use_vi:nnnnnnn`

`\use_vii:nnnnnnn`

`\use_i:nnnnnnnn`

`\use_ii:nnnnnnnn`

`\use_iii:nnnnnnnn`

`\use_iv:nnnnnnnn`

`\use_v:nnnnnnnn`

We also need something for picking up arguments from a longer list.

```

1507 \cs_gset:Npn \use_i:nnn #1#2#3 {#1}
1508 \cs_gset:Npn \use_ii:nnn #1#2#3 {#2}
1509 \cs_gset:Npn \use_iii:nnn #1#2#3 {#3}
1510 \cs_gset:Npn \use_i:nnnn #1#2#3#4 {#1}
1511 \cs_gset:Npn \use_ii:nnnn #1#2#3#4 {#2}
1512 \cs_gset:Npn \use_iii:nnnn #1#2#3#4 {#3}
1513 \cs_gset:Npn \use_iv:nnnn #1#2#3#4 {#4}
1514 \cs_gset:Npn \use_i:nnnnn #1#2#3#4#5 {#1}
1515 \cs_gset:Npn \use_ii:nnnnn #1#2#3#4#5 {#2}
1516 \cs_gset:Npn \use_iii:nnnnn #1#2#3#4#5 {#3}
1517 \cs_gset:Npn \use_iv:nnnnn #1#2#3#4#5 {#4}
1518 \cs_gset:Npn \use_v:nnnnn #1#2#3#4#5 {#5}
1519 \cs_gset:Npn \use_i:nnnnnn #1#2#3#4#5#6 {#1}
1520 \cs_gset:Npn \use_ii:nnnnnn #1#2#3#4#5#6 {#2}
1521 \cs_gset:Npn \use_iii:nnnnnn #1#2#3#4#5#6 {#3}
1522 \cs_gset:Npn \use_iv:nnnnnn #1#2#3#4#5#6 {#4}
1523 \cs_gset:Npn \use_v:nnnnnn #1#2#3#4#5#6 {#5}
1524 \cs_gset:Npn \use_vi:nnnnnn #1#2#3#4#5#6 {#6}
1525 \cs_gset:Npn \use_i:nnnnnnn #1#2#3#4#5#6#7 {#1}
1526 \cs_gset:Npn \use_ii:nnnnnnn #1#2#3#4#5#6#7 {#2}
1527 \cs_gset:Npn \use_iii:nnnnnnn #1#2#3#4#5#6#7 {#3}

```

```

1528 \cs_gset:Npn \use_iv:nnnnnnn #1#2#3#4#5#6#7 {#4}
1529 \cs_gset:Npn \use_v:nnnnnnn #1#2#3#4#5#6#7 {#5}
1530 \cs_gset:Npn \use_vi:nnnnnnn #1#2#3#4#5#6#7 {#6}
1531 \cs_gset:Npn \use_vii:nnnnnnn #1#2#3#4#5#6#7 {#7}
1532 \cs_gset:Npn \use_i:nnnnnnnn #1#2#3#4#5#6#7#8 {#1}
1533 \cs_gset:Npn \use_ii:nnnnnnnn #1#2#3#4#5#6#7#8 {#2}
1534 \cs_gset:Npn \use_iii:nnnnnnnn #1#2#3#4#5#6#7#8 {#3}
1535 \cs_gset:Npn \use_iv:nnnnnnnn #1#2#3#4#5#6#7#8 {#4}
1536 \cs_gset:Npn \use_v:nnnnnnnn #1#2#3#4#5#6#7#8 {#5}
1537 \cs_gset:Npn \use_vi:nnnnnnnn #1#2#3#4#5#6#7#8 {#6}
1538 \cs_gset:Npn \use_vii:nnnnnnnn #1#2#3#4#5#6#7#8 {#7}
1539 \cs_gset:Npn \use_viii:nnnnnnnn #1#2#3#4#5#6#7#8 {#8}
1540 \cs_gset:Npn \use_i:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#1}
1541 \cs_gset:Npn \use_ii:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#2}
1542 \cs_gset:Npn \use_iii:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#3}
1543 \cs_gset:Npn \use_iv:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#4}
1544 \cs_gset:Npn \use_v:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#5}
1545 \cs_gset:Npn \use_vi:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#6}
1546 \cs_gset:Npn \use_vii:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#7}
1547 \cs_gset:Npn \use_viii:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#8}
1548 \cs_gset:Npn \use_ix:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#9}

```

(End of definition for `\use_i:nnn` and others. These functions are documented on page 25.)

`\use_i_ii:nnn`

```

1549 \cs_gset:Npn \use_i_ii:nnn #1#2#3 {#1#2}

```

(End of definition for `\use_i_ii:nnn`. This function is documented on page 26.)

`\use_ii_i:nn`

```

1550 \cs_gset:Npn \use_ii_i:nn #1#2 { #2 #1 }

```

(End of definition for `\use_ii_i:nn`. This function is documented on page 26.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_`
`\use_none_delimit_by_q_stop:w` recursion_stop, respectively.
`\use_none_delimit_by_q_recursion_stop:w`

```

1551 \cs_gset:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
1552 \cs_gset:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
1553 \cs_gset:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }

```

(End of definition for `\use_none_delimit_by_q_nil:w`, `\use_none_delimit_by_q_stop:w`, and `\use_`
`none_delimit_by_q_recursion_stop:w`. These functions are documented on page 26.)

`\use_i_delimit_by_q_nil:nw` Same as above but execute first argument after gobbling. Very useful when you need to
`\use_i_delimit_by_q_stop:nw` skip the rest of a mapping sequence but want an easy way to control what should be
`\use_i_delimit_by_q_recursion_stop:nw` expanded next.

```

1554 \cs_gset:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
1555 \cs_gset:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
1556 \cs_gset:Npn \use_i_delimit_by_q_recursion_stop:nw
1557 #1#2 \q_recursion_stop {#1}

```

(End of definition for `\use_i_delimit_by_q_nil:nw`, `\use_i_delimit_by_q_stop:nw`, and `\use_i_`
`delimit_by_q_recursion_stop:nw`. These functions are documented on page 27.)

42.5 Gobbling tokens from input

```

\use_none:n
\use_none:nn
\use_none:nnn
\use_none:nnnn
\use_none:nnnnn
\use_none:nnnnnn
\use_none:nnnnnnn
\use_none:nnnnnnnn

```

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

```

1558 \cs_gset:Npn \use_none:n #1 { }
1559 \cs_gset:Npn \use_none:nn #1#2 { }
1560 \cs_gset:Npn \use_none:nnn #1#2#3 { }
1561 \cs_gset:Npn \use_none:nnnn #1#2#3#4 { }
1562 \cs_gset:Npn \use_none:nnnnn #1#2#3#4#5 { }
1563 \cs_gset:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
1564 \cs_gset:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
1565 \cs_gset:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
1566 \cs_gset:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }

```

(End of definition for `\use_none:n` and others. These functions are documented on page 26.)

42.6 Debugging and patching later definitions

```

1567 <@@=debug>

```

`__kernel_if_debug:TF` A more meaningful test of whether debugging is enabled than messing up with guards. We can also more easily change the logic in one place then. This is needed primarily for deprecations.

```

1568 \cs_gset_protected:Npn \__kernel_if_debug:TF #1#2 {#2}

```

(End of definition for `__kernel_if_debug:TF`.)

`\debug_on:n` Stubs.

```

\debug_off:n

```

```

1569 \cs_gset_protected:Npn \debug_on:n #1
1570 {
1571   \sys_load_debug:
1572   \cs_if_exist:NT \__debug_all_on:
1573     { \debug_on:n {#1} }
1574 }
1575 \cs_gset_protected:Npn \debug_off:n #1
1576 {
1577   \sys_load_debug:
1578   \cs_if_exist:NT \__debug_all_on:
1579     { \debug_off:n {#1} }
1580 }

```

(End of definition for `\debug_on:n` and `\debug_off:n`. These functions are documented on page 30.)

`\debug_suspend:`

```

\debug_resume:

```

```

1581 \cs_gset_protected:Npn \debug_suspend: { }
1582 \cs_gset_protected:Npn \debug_resume: { }

```

(End of definition for `\debug_suspend:` and `\debug_resume:`. These functions are documented on page 30.)

`_kernel_deprecation_code:nn` Make deprecated commands throw errors if the user requests it. This relies on two token lists, filled up in `l3deprecation`.

```

\g__debug_deprecation_on_tl
\g__debug_deprecation_off_tl
1583 \cs_gset_nopar:Npn \g__debug_deprecation_on_tl { }
1584 \cs_gset_nopar:Npn \g__debug_deprecation_off_tl { }
1585 \cs_gset_protected:Npn \_kernel_deprecation_code:nn #1#2
1586 {
1587   \tl_gput_right:Nn \g__debug_deprecation_on_tl {#1}
1588   \tl_gput_right:Nn \g__debug_deprecation_off_tl {#2}
1589 }

```

(End of definition for `_kernel_deprecation_code:nn`, `\g__debug_deprecation_on_tl`, and `\g__debug_deprecation_off_tl`.)

42.7 Conditional processing and definitions

```
1590 (@@=prg)
```

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the `(state)` this leaves `TeX` in. Therefore, a simple user interface could be something like

```

\if_meaning:w #1#2
  \prg_return_true:
\else:
  \if_meaning:w #1#3
    \prg_return_true:
  \else:
    \prg_return_false:
\fi:
\fi:

```

Usually, a `TeX` programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the `TeX` programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` The idea here is that `\exp:w` expands fully any `\else:` and `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```

1591 \cs_gset:Npn \prg_return_true:
1592 { \exp_after:wN \use_i:nn \exp:w }
1593 \cs_gset:Npn \prg_return_false:
1594 { \exp_after:wN \use_ii:nn \exp:w}

```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn/\use_ii:nn`. Provided two arguments are absorbed then the code would work.

(End of definition for `\prg_return_true:` and `\prg_return_false:.` These functions are documented on page 65.)

```

\prg_use_none_delimit_by_q_recursion_stop:w Private version of \use_none_delimit_by_q_recursion_stop:w.
1595 \cs_gset:Npn \__prg_use_none_delimit_by_q_recursion_stop:w
1596 #1 \q__prg_recursion_stop { }

```

(End of definition for __prg_use_none_delimit_by_q_recursion_stop:w.)

```

\prg_set_conditional:Npnn The user functions for the types using parameter text from the programmer. The various
\prg_gset_conditional:Npnn functions only differ by which function is used for the assignment. For those Npnn type
\prg_new_conditional:Npnn functions, we must grab the parameter text, reading everything up to a left brace before
\prg_set_protected_conditional:Npnn continuing. Then split the base function into name and signature, and feed {\langle name\rangle}
\prg_gset_protected_conditional:Npnn {\langle signature\rangle} \langle boolean\rangle {\langle set or new\rangle} {\langle maybe protected\rangle} {\langle parameters\rangle} {TF,...}
\prg_new_protected_conditional:Npnn {\langle code\rangle} to the auxiliary function responsible for defining all conditionals. Note that e
\__prg_generate_conditional_parm:NNNpnn stands for expandable and p for protected.

```

```

1597 \cs_gset_protected:Npn \prg_set_conditional:Npnn
1598 { \__prg_generate_conditional_parm:NNNpnn \cs_set:Npn e }
1599 \cs_gset_protected:Npn \prg_gset_conditional:Npnn
1600 { \__prg_generate_conditional_parm:NNNpnn \cs_gset:Npn e }
1601 \cs_gset_protected:Npn \prg_new_conditional:Npnn
1602 { \__prg_generate_conditional_parm:NNNpnn \cs_new:Npn e }
1603 \cs_gset_protected:Npn \prg_set_protected_conditional:Npnn
1604 { \__prg_generate_conditional_parm:NNNpnn \cs_set_protected:Npn p }
1605 \cs_gset_protected:Npn \prg_gset_protected_conditional:Npnn
1606 { \__prg_generate_conditional_parm:NNNpnn \cs_gset_protected:Npn p }
1607 \cs_gset_protected:Npn \prg_new_protected_conditional:Npnn
1608 { \__prg_generate_conditional_parm:NNNpnn \cs_new_protected:Npn p }
1609 \cs_gset_protected:Npn \__prg_generate_conditional_parm:NNNpnn #1#2#3#4#
1610 {
1611   \use:e
1612   {
1613     \__prg_generate_conditional:nnNNNnnn
1614     \cs_split_function:N #3
1615   }
1616   #1 #2 {#4}
1617 }

```

(End of definition for \prg_set_conditional:Npnn and others. These functions are documented on page 63.)

```

\prg_set_conditional:Nnn The user functions for the types automatically inserting the correct parameter text based
\prg_gset_conditional:Nnn on the signature. The various functions only differ by which function is used for the
\prg_new_conditional:Nnn assignment. Split the base function into name and signature. The second auxiliary
\prg_set_protected_conditional:Nnn generates the parameter text from the number of letters in the signature. Then feed
\prg_gset_protected_conditional:Nnn {\langle name\rangle} {\langle signature\rangle} \langle boolean\rangle {\langle set or new\rangle} {\langle maybe protected\rangle} {\langle parameters\rangle}
\prg_new_protected_conditional:Nnn {TF,...} {\langle code\rangle} to the auxiliary function responsible for defining all conditionals. If
\__prg_generate_conditional_count:NNNnn the \langle signature\rangle has more than 9 letters, the definition is aborted since TEX macros have
\__prg_generate_conditional_count:nnNNNnn at most 9 arguments. The erroneous case where the function name contains no colon is
captured later.

```

```

1618 \cs_gset_protected:Npn \prg_set_conditional:Nnn
1619 { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
1620 \cs_gset_protected:Npn \prg_gset_conditional:Nnn
1621 { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
1622 \cs_gset_protected:Npn \prg_new_conditional:Nnn
1623 { \__prg_generate_conditional_count:NNNnn \cs_new:Npn e }

```

```

1624 \cs_gset_protected:Npn \prg_set_protected_conditional:Nnn
1625   { \prg_generate_conditional_count:NNNnn \cs_set_protected:Npn p }
1626 \cs_gset_protected:Npn \prg_gset_protected_conditional:Nnn
1627   { \prg_generate_conditional_count:NNNnn \cs_gset_protected:Npn p }
1628 \cs_gset_protected:Npn \prg_new_protected_conditional:Nnn
1629   { \prg_generate_conditional_count:NNNnn \cs_new_protected:Npn p }
1630 \cs_gset_protected:Npn \prg_generate_conditional_count:NNNnn #1#2#3
1631   {
1632     \use:e
1633     {
1634       \prg_generate_conditional_count:nnNNNnn
1635       \cs_split_function:N #3
1636     }
1637     #1 #2
1638   }
1639 \cs_gset_protected:Npn \prg_generate_conditional_count:nnNNNnn #1#2#3#4#5
1640   {
1641     \kernel_cs_parm_from_arg_count:nnF
1642     { \prg_generate_conditional:nnNNNnnn {#1} {#2} #3 #4 #5 }
1643     { \tl_count:n {#2} }
1644     {
1645       \msg_error:nnee { kernel } { bad-number-of-arguments }
1646       { \token_to_str:c { #1 : #2 } }
1647       { \tl_count:n {#2} }
1648     }
1649     \use_none:nn
1650   }

```

(End of definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 63.)

```

\prg_generate_conditional:nnNNNnn
\prg_generate_conditional:NNnnnnNw
\prg_generate_conditional_test:w
\prg_generate_conditional_fast:nw

```

The workhorse here is going through a list of desired forms, *i.e.*, p, TF, T and F. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. In the absence of a colon, we throw an error and don't define any conditional. The fourth and fifth arguments build up the defining function. The sixth is the parameters to use (possibly empty), the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms. The use of `\tl_to_str:n` makes the later loop more robust.

A large number of our low-level conditionals look like `\prg_return_true: \else: \prg_return_false: \fi:` so we optimize this special case by calling `\prg_generate_conditional_fast:nw {<code>}`. This passes `\use_i:nn` instead of `\use_i_ii:nnn` to functions such as `\prg_generate_p_form:wNNnnnnN`.

```

1651 \cs_gset_protected:Npn \prg_generate_conditional:nnNNNnnn #1#2#3#4#5#6#7#8
1652   {
1653     \if_meaning:w \c_false_bool #3
1654     \msg_error:nne { kernel } { missing-colon }
1655     { \token_to_str:c {#1} }
1656     \exp_after:wN \use_none:nn
1657     \fi:
1658     \use:e
1659     {
1660       \exp_not:N \prg_generate_conditional:NNnnnnNw

```

```

1661     \exp_not:n { #4 #5 {#1} {#2} {#6} }
1662     \__prg_generate_conditional_test:w
1663     #8 \s__prg_mark
1664     \__prg_generate_conditional_fast:nw
1665     \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark
1666     \use_none:n
1667     \exp_not:n { {#8} \use_i_ii:nnn }
1668     \tl_to_str:n {#7}
1669     \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
1670   }
1671 }
1672 \cs_gset:Npn \__prg_generate_conditional_test:w
1673   #1 \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark #2
1674   { #2 {#1} }
1675 \cs_gset:Npn \__prg_generate_conditional_fast:nw #1#2 \exp_not:n #3
1676   { \exp_not:n { {#1} \use_i:nn } }

```

Looping through the list of desired forms. First are six arguments and seventh is the form. Use the form to call the correct type. If the form does not exist, the `\use:c` construction results in `\relax`, and the error message is displayed (unless the form is empty, to allow for `{T, , F}`), then `\use_none:nnnnnnnn` cleans up. Otherwise, the error message is removed by the variant form.

```

1677 \cs_gset_protected:Npn \__prg_generate_conditional:NNnnnnNw #1#2#3#4#5#6#7#8 ,
1678   {
1679     \if_meaning:w \q__prg_recursion_tail #8
1680     \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1681     \fi:
1682     \use:c { __prg_generate_ #8 _form:wNNnnnnN }
1683     \tl_if_empty:nF {#8}
1684     {
1685       \msg_error:nnee
1686       { kernel } { conditional-form-unknown }
1687       {#8} { \token_to_str:c { #3 : #4 } }
1688     }
1689     \use_none:nnnnnnnn
1690     \s__prg_stop
1691     #1 #2 {#3} {#4} {#5} {#6} #7
1692     \__prg_generate_conditional:NNnnnnNw #1 #2 {#3} {#4} {#5} {#6} #7
1693   }

```

(End of definition for `__prg_generate_conditional:nnNNnnnn` and others.)

```

\__prg_generate_p_form:wNNnnnnN
\__prg_generate_TF_form:wNNnnnnN
\__prg_generate_T_form:wNNnnnnN
\__prg_generate_F_form:wNNnnnnN
\__prg_p_true:w
\__prg_T_true:w
\__prg_F_true:w
\__prg_TF_true:w

```

How to generate the various forms. Those functions take the following arguments: 1: junk, 2: `\cs_set:Npn` or similar, 3: `p` (for protected conditionals) or `e`, 4: function name, 5: signature, 6: parameter text, 7: replacement (possibly trimmed by `__prg_generate_conditional_fast:nw`), 8: `\use_i_ii:nnn` or `\use_i:nn` (for “fast” conditionals). Remember that the logic-returning functions expect two arguments to be present after `\exp_end::`: notice the construction of the different variants relies on this, and that the TF and F variants will be slightly faster than the T version. The `p` form is only valid for expandable tests, we check for that by making sure that the second argument is empty. For “fast” conditionals, #7 has an extra `\if_...`. To optimize a bit further we don’t use `\exp_after:wN \use_ii:nnn` and similar but instead use `__prg_TF_true:w` and similar to swap out the macro after `\fi:`. It would be a tiny bit faster if we directly

grabbed the T and F arguments there, but if those are actually missing, the recovery from the runaway argument would not insert `\fi:` back, messing up nesting of conditionals.

```

1694 \cs_gset_protected:Npn \__prg_generate_p_form:wNNnnnnN
1695   #1 \s__prg_stop #2#3#4#5#6#7#8
1696   {
1697     \if_meaning:w e #3
1698     \exp_after:wN \use_i:nn
1699     \else:
1700     \exp_after:wN \use_ii:nn
1701     \fi:
1702     {
1703       #8
1704       { \exp_args:Nc #2 { #4 _p: #5 } #6 }
1705       { { #7 \exp_end: \c_true_bool \c_false_bool } }
1706       { #7 \__prg_p_true:w \fi: \c_false_bool }
1707     }
1708     {
1709     \msg_error:nne { kernel } { protected-predicate }
1710     { \token_to_str:c { #4 _p: #5 } }
1711     }
1712   }
1713 \cs_gset_protected:Npn \__prg_generate_T_form:wNNnnnnN
1714   #1 \s__prg_stop #2#3#4#5#6#7#8
1715   {
1716   #8
1717   { \exp_args:Nc #2 { #4 : #5 T } #6 }
1718   { { #7 \exp_end: \use:n \use_none:n } }
1719   { #7 \__prg_T_true:w \fi: \use_none:n }
1720   }
1721 \cs_gset_protected:Npn \__prg_generate_F_form:wNNnnnnN
1722   #1 \s__prg_stop #2#3#4#5#6#7#8
1723   {
1724   #8
1725   { \exp_args:Nc #2 { #4 : #5 F } #6 }
1726   { { #7 \exp_end: { } } }
1727   { #7 \__prg_F_true:w \fi: \use:n }
1728   }
1729 \cs_gset_protected:Npn \__prg_generate_TF_form:wNNnnnnN
1730   #1 \s__prg_stop #2#3#4#5#6#7#8
1731   {
1732   #8
1733   { \exp_args:Nc #2 { #4 : #5 TF } #6 }
1734   { { #7 \exp_end: } }
1735   { #7 \__prg_TF_true:w \fi: \use_ii:nn }
1736   }
1737 \cs_gset:Npn \__prg_p_true:w \fi: \c_false_bool { \fi: \c_true_bool }
1738 \cs_gset:Npn \__prg_T_true:w \fi: \use_none:n { \fi: \use:n }
1739 \cs_gset:Npn \__prg_F_true:w \fi: \use:n { \fi: \use_none:n }
1740 \cs_gset:Npn \__prg_TF_true:w \fi: \use_ii:nn { \fi: \use_i:nn }

```

(End of definition for `__prg_generate_p_form:wNNnnnnN` and others.)

```

\prg_set_eq_conditional:NNn The setting-equal functions. Split both functions and feed {⟨name1⟩} {⟨signature1⟩}
\prg_gset_eq_conditional:NNn ⟨boolean1⟩ {⟨name2⟩} {⟨signature2⟩} ⟨boolean2⟩ ⟨copying function⟩ ⟨conditions⟩ ,
\prg_new_eq_conditional:NNn
  \__prg_set_eq_conditional:NNn

```

\q__prg_recursion_tail , \q__prg_recursion_stop to a first auxiliary.

```

1741 \cs_gset_protected:Npn \prg_set_eq_conditional:NNn
1742   { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1743 \cs_gset_protected:Npn \prg_gset_eq_conditional:NNn
1744   { \__prg_set_eq_conditional:NNNn \cs_gset_eq:cc }
1745 \cs_gset_protected:Npn \prg_new_eq_conditional:NNn
1746   { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1747 \cs_gset_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
1748   {
1749     \use:e
1750     {
1751       \exp_not:N \__prg_set_eq_conditional:nnNnnNW
1752       \cs_split_function:N #2
1753       \cs_split_function:N #3
1754       \exp_not:N #1
1755       \tl_to_str:n {#4}
1756       \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
1757     }
1758   }

```

(End of definition for \prg_set_eq_conditional:NNn and others. These functions are documented on page 65.)

__prg_set_eq_conditional:nnNnnNW
 __prg_set_eq_conditional_loop:nnnnNW
 __prg_set_eq_conditional_p_form:nnn
 __prg_set_eq_conditional_TF_form:nnn
 __prg_set_eq_conditional_T_form:nnn
 __prg_set_eq_conditional_F_form:nnn

Split the function to be defined, and setup a manual clist loop over argument #6 of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\} \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\}$ $\langle copying\ function \rangle$ and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```

1759 \cs_gset_protected:Npn \__prg_set_eq_conditional:nnNnnNW #1#2#3#4#5#6
1760   {
1761     \if_meaning:w \c_false_bool #3
1762     \msg_error:nne { kernel } { missing-colon }
1763     { \token_to_str:c {#1} }
1764     \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1765     \fi:
1766     \if_meaning:w \c_false_bool #6
1767     \msg_error:nne { kernel } { missing-colon }
1768     { \token_to_str:c {#4} }
1769     \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1770     \fi:
1771     \__prg_set_eq_conditional_loop:nnnnNW {#1} {#2} {#4} {#5}
1772   }
1773 \cs_gset_protected:Npn \__prg_set_eq_conditional_loop:nnnnNW #1#2#3#4#5#6 ,
1774   {
1775     \if_meaning:w \q__prg_recursion_tail #6
1776     \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1777     \fi:
1778     \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
1779     \tl_if_empty:nF {#6}
1780     {
1781       \msg_error:nnee
1782       { kernel } { conditional-form-unknown }

```

```

1783         {#6} { \token_to_str:c { #1 : #2 } }
1784     }
1785     \use_none:nnnnnn
1786     \s__prg_stop
1787     #5 {#1} {#2} {#3} {#4}
1788     \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1789 }
1790 \cs_gset:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1791 { #2 { #3_p : #4 } { #5_p : #6 } }
1792 \cs_gset:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1793 { #2 { #3 : #4 TF } { #5 : #6 TF } }
1794 \cs_gset:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1795 { #2 { #3 : #4 T } { #5 : #6 T } }
1796 \cs_gset:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1797 { #2 { #3 : #4 F } { #5 : #6 F } }

```

(End of definition for `__prg_set_eq_conditional:nnNnnNNw` and others.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using `\if:w` but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

```

\c_true_bool Here are the canonical boolean values.
\c_false_bool
1798 \tex_global:D \tex_chardef:D \c_true_bool = 1 ~
1799 \tex_global:D \tex_chardef:D \c_false_bool = 0 ~

```

(End of definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 67.)

42.8 Dissecting a control sequence

```

1800 <@@=cs>

```

```

\underline{\_cs_count_signature:N} * \underline{\_cs_count_signature:N} <function>

```

Splits the `<function>` into the `<name>` (*i.e.* the part before the colon) and the `<signature>` (*i.e.* after the colon). The `<number>` of tokens in the `<signature>` is then left in the input stream. If there was no `<signature>` then the result is the marker value `-1`.

```

\underline{\_cs_tmp:w}

```

Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

```

\cs_to_str:N This converts a control sequence into the character string of its name, removing the
\underline{\_cs_to_str:N} leading escape character. This turns out to be a non-trivial matter as there a different
\underline{\_cs_to_str:w} cases:

```

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the `\escapechar` is negative;

- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N__` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and T_EX reads `__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_romannumeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N__`, and the auxiliary `__cs_to_str:w` is expanded, feeding - as a second character for the test; the test is false, and T_EX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_romannumeral:D` with `\c_zero_int`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `__cs_to_str:w` comes into play, inserting `-\int_value:w`, which expands `\c_zero_int` to the character 0. The initial `\tex_romannumeral:D` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_romannumeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
1801 \cs_gset:Npn \cs_to_str:N
1802   {
```

We implement the expansion scheme using `\tex_romannumeral:D` terminating it with `\c_zero_int` rather than using `\exp:w` and `\exp_end:` as we normally do. The reason is that the code heavily depends on terminating the expansion with `\c_zero_int` so we make this dependency explicit.

```
1803   \tex_romannumeral:D
1804   \if:w \token_to_str:N \ \__cs_to_str:w \fi:
1805   \exp_after:wN \__cs_to_str:N \token_to_str:N
1806   }
1807 \cs_gset:Npn \__cs_to_str:N #1 { \c_zero_int }
1808 \cs_gset:Npn \__cs_to_str:w #1 \__cs_to_str:N
1809   { - \int_value:w \fi: \exp_after:wN \c_zero_int }
```

If speed is a concern we could use `\csstring` in LuaT_EX. For the empty csname that primitive gives an empty result while the current `\cs_to_str:N` gives incorrect results in all engines (this is impossible to fix without huge performance hit).

(End of definition for `\cs_to_str:N`, `__cs_to_str:N`, and `__cs_to_str:w`. This function is documented on page 22.)

`\cs_split_function:N`

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean `<true>` or `<false>` is returned with `<true>` for when there is a colon in the function and `<false>` if there is not.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as `#1` the function name, delimited by the first colon, then the signature `#2`, delimited by `\s__cs_mark`, then `\c_true_bool` as `#3`, and `#4` cleans up until `\s__cs_stop`. Otherwise, the `#1` contains the function name and `\s__cs_mark \c_true_bool`, `#2` is empty, `#3` is `\c_false_bool`,

and #4 cleans up. The second auxiliary trims the trailing `\s__cs_mark` from the function name if present (that is, if the original function had no colon).

```

1810 \cs_gset_protected:Npn \__cs_tmp:w #1
1811 {
1812   \cs_gset:Npn \cs_split_function:N ##1
1813   {
1814     \exp_after:wN \exp_after:wN \exp_after:wN
1815     \__cs_split_function_auxi:w
1816     \cs_to_str:N ##1 \s__cs_mark \c_true_bool
1817     #1 \s__cs_mark \c_false_bool \s__cs_stop
1818   }
1819   \cs_gset:Npn \__cs_split_function_auxi:w
1820   ##1 #1 ##2 \s__cs_mark ##3##4 \s__cs_stop
1821   { \__cs_split_function_auxii:w ##1 \s__cs_mark \s__cs_stop {##2} ##3 }
1822   \cs_gset:Npn \__cs_split_function_auxii:w ##1 \s__cs_mark ##2 \s__cs_stop
1823   { {##1} }
1824 }
1825 \exp_after:wN \__cs_tmp:w \token_to_str:N :

```

(End of definition for `\cs_split_function:N`, `__cs_split_function_auxi:w`, and `__cs_split_function_auxii:w`. This function is documented on page 22.)

42.9 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be *free* (to be defined) if it does not already exist.

<pre> \cs_if_exist_p:N \cs_if_exist_p:c \cs_if_exist:NTF \cs_if_exist:cTF __cs_if_exist_c_aux: __cs_if_exist_c_aux:w </pre>	<p>Two versions for checking existence. For the <code>N</code> form we firstly check for <code>\scan_stop:</code> and then if it is in the hash table. There is no problem when inputting something like <code>\else:</code> or <code>\fi:</code> as <code>T_EX</code> will only ever skip input in case the token tested against is <code>\scan_stop:</code>.</p> <p>In both the <code>N</code> and <code>c</code> form we use the way <code>\prg_set_conditional:Npnn</code> optimizes the conditionals to negate the tests using <code>\else:</code> (the <code>\else:</code> in the top level functions will be removed by the optimization, and this usage of <code>\else:</code> will be fine).</p>
---	---

```

1826 \prg_gset_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1827 {
1828   \if_meaning:w #1 \scan_stop:
1829   \use_i:nnnn
1830   \else:
1831   \fi:
1832   \if_cs_exist:N #1
1833   \prg_return_true:
1834   \else:
1835   \prg_return_false:
1836   \fi:
1837 }

```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop:` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

1838 \cs_if_exist:NTF \tex_lastnamedcs:D
1839 {
1840   \prg_gset_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1841   {
1842     \if_cs_exist:w #1 \cs_end:
1843     \__cs_if_exist_c_aux:
1844     \prg_return_true:
1845     \else:
1846     \prg_return_false:
1847     \fi:
1848   }
1849   \cs_gset:Npn \__cs_if_exist_c_aux:
1850   { \fi: \exp_after:wN \if_meaning:w \tex_lastnamedcs:D \scan_stop: \else: }
1851 }
1852 {
1853   \prg_gset_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1854   {
1855     \if_cs_exist:w #1 \cs_end:
1856     \__cs_if_exist_c_aux:w
1857     \fi:
1858     \use_none:n {#1}
1859     \if_false:
1860     \prg_return_true:
1861     \else:
1862     \prg_return_false:
1863     \fi:
1864   }
1865   \cs_gset:Npn \__cs_if_exist_c_aux:w \fi: \use_none:n #1 \if_false:
1866   { \fi: \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop: \else: }
1867 }

```

(End of definition for `\cs_if_exist:NTF`, `__cs_if_exist_c_aux:`, and `__cs_if_exist_c_aux:w`. This function is documented on page 28.)

`\cs_if_free_p:N`
`\cs_if_free_p:c`
`\cs_if_free:NTF`
`\cs_if_free:cTF`

The logical reversal of the above.

```

1868 \prg_gset_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
1869 {
1870   \if_cs_exist:N #1
1871   \else:
1872   \use_none:nnnn
1873   \fi:
1874   \if_meaning:w #1 \scan_stop:
1875   \prg_return_true:
1876   \else:
1877   \prg_return_false:
1878   \fi:
1879 }
1880 \cs_if_exist:NTF \tex_lastnamedcs:D
1881 {
1882   \prg_gset_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1883   {
1884     \if_cs_exist:w #1 \cs_end:
1885     \__cs_if_free_c_aux:w
1886     \fi:

```

```

1887     \if_true:
1888     \prg_return_true:
1889     \else:
1890     \prg_return_false:
1891     \fi:
1892   }
1893   \cs_gset:Npn \__cs_if_free_c_aux:w \fi: \if_true:
1894   { \fi: \exp_after:wN \if_meaning:w \tex_lastnamedcs:D \scan_stop: }
1895 }
1896 {
1897   \prg_gset_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1898   {
1899     \if_cs_exist:w #1 \cs_end:
1900     \__cs_if_free_c_aux:w
1901     \fi:
1902     \use_none:n {#1}
1903     \if_true:
1904     \prg_return_true:
1905     \else:
1906     \prg_return_false:
1907     \fi:
1908   }
1909   \cs_gset:Npn \__cs_if_free_c_aux:w \fi: \use_none:n #1 \if_true:
1910   { \fi: \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop: }
1911 }

```

(End of definition for `\cs_if_free:NTF`. This function is documented on page 28.)

```

\cs_if_exist_use:N The \cs_if_exist_use:... functions cannot be implemented as conditionals because
\cs_if_exist_use:c the true branch must leave both the control sequence itself and the true code in the input
\cs_if_exist_use:NTF stream. For the c variants, we are careful not to put the control sequence in the hash
\cs_if_exist_use:cTF table if it does not exist. If available we use the \lastnamedcs primitive.
__cs_if_exist_use_aux:w
__cs_if_exist_use_aux:Nnn
1912 \cs_gset:Npn \cs_if_exist_use:NTF #1#2
1913 { \cs_if_exist:NTF #1 { #1 #2 } }
1914 \cs_gset:Npn \cs_if_exist_use:NF #1
1915 { \cs_if_exist:NTF #1 #1 }
1916 \cs_gset:Npn \cs_if_exist_use:NT #1 #2
1917 { \cs_if_exist:NT #1 { #1 #2 } }
1918 \cs_gset:Npn \cs_if_exist_use:N #1
1919 { \cs_if_exist:NT #1 #1 }
1920 \cs_if_exist:NTF \tex_lastnamedcs:D
1921 {
1922   \cs_gset:Npn \cs_if_exist_use:cTF #1
1923   {
1924     \if_cs_exist:w #1 \cs_end:
1925     \__cs_if_exist_use_aux:w
1926     \fi:
1927     \use_ii:nn
1928   }
1929   \cs_gset:Npn \__cs_if_exist_use_aux:w \fi: \use_ii:nn
1930   { \fi: \exp_after:wN \__cs_if_exist_use_aux:Nnn \tex_lastnamedcs:D }
1931 }
1932 {
1933   \cs_gset:Npn \cs_if_exist_use:cTF #1

```

```

1934 {
1935   \if_cs_exist:w #1 \cs_end:
1936   \__cs_if_exist_use_aux:w
1937   \fi:
1938   \use_iii:nnn {#1}
1939 }
1940 \cs_gset:Npn \__cs_if_exist_use_aux:w \fi: \use_iii:nnn #1
1941 { \fi: \exp_after:wN \__cs_if_exist_use_aux:Nnn \cs:w #1 \cs_end: }
1942 }
1943 \cs_gset:Npn \__cs_if_exist_use_aux:Nnn #1#2
1944 {
1945   \if_meaning:w #1 \scan_stop:
1946   \exp_after:wN \use_iii:nnn
1947   \fi:
1948   \use_i:nn { #1 #2 }
1949 }
1950 \cs_gset:Npn \cs_if_exist_use:cF #1
1951 { \cs_if_exist_use:cTF {#1} {} }
1952 \cs_gset:Npn \cs_if_exist_use:cT #1#2
1953 { \cs_if_exist_use:cTF {#1} {#2} {} }
1954 \cs_gset:Npn \cs_if_exist_use:c #1
1955 { \cs_if_exist_use:cTF {#1} {} {} }

```

(End of definition for `\cs_if_exist_use:NTF`, `__cs_if_exist_use_aux:w`, and `__cs_if_exist_use_aux:Nnn`. This function is documented on page 21.)

42.10 Preliminaries for new functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The next few definitions here are only temporary, they will be redefined later on.

```
\msg_error:nnee
```

If an internal error occurs before $\LaTeX\text{3}$ has loaded `l3msg` then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the `\newlinechar` is needed, to turn `^^J` into a proper line break in plain $\text{T}_{\text{E}}\text{X}$.

```
\msg_error:nne
\msg_error:nn
```

```

1956 \cs_gset_protected:Npn \msg_error:nnee #1#2#3#4
1957 {
1958   \tex_newlinechar:D = '\^^J \scan_stop:
1959   \tex_errmessage:D
1960   {
1961     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1962     Argh,~internal~LaTeX3~error! ^^J ^^J
1963     Module ~ #1 , ~ message~name~"#"#2": ^^J
1964     Arguments~'#"#3'~and~'#"#4' ^^J ^^J
1965     This~is~one~for~The~LaTeX3~Project::~bailing~out
1966   }
1967   \tex_end:D
1968 }

```



```

1969 \cs_gset_protected:Npn \msg_error:nne #1#2#3
1970   { \msg_error:nnee {#1} {#2} {#3} { } }
1971 \cs_gset_protected:Npn \msg_error:nn #1#2
1972   { \msg_error:nnee {#1} {#2} { } { } }

```

(End of definition for `\msg_error:nenn`. This function is documented on page 84.)

`\msg_line_context:` Another one from `l3msg` which will be altered later.

```

1973 \cs_gset:Npn \msg_line_context:
1974   { on~line~ \tex_the:D \tex_inputlineno:D }

```

(End of definition for `\msg_line_context:.` This function is documented on page 82.)

`\iow_log:e` We define a routine to write only to the log file. And a similar one for writing to both
`\iow_term:e` the log file and the terminal. These will be redefined later by `l3file`.

```

1975 \cs_gset_protected:Npn \iow_log:e
1976   { \tex_immediate:D \tex_write:D -1 }
1977 \cs_gset_protected:Npn \iow_term:e
1978   { \tex_immediate:D \tex_write:D 16 }

```

(End of definition for `\iow_log:n`. This function is documented on page 96.)

`__kernel_chk_if_free_cs:N` This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` etc. to make sure
`__kernel_chk_if_free_cs:c` that the argument sequence is not already in use. If it is, an error is signalled. It checks
if `<csname>` is undefined or `\scan_stop:.` Otherwise an error message is issued. We have
to make sure we don't put the argument into the conditional processing since it may be
an `\if...` type function!

```

1979 \cs_gset_protected:Npn \__kernel_chk_if_free_cs:N #1
1980   {
1981     \cs_if_free:NF #1
1982     {
1983       \msg_error:nnee { kernel } { command-already-defined }
1984       { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1985     }
1986   }
1987 \cs_gset_protected:Npn \__kernel_chk_if_free_cs:c
1988   { \exp_args:Nc \__kernel_chk_if_free_cs:N }

```

(End of definition for `__kernel_chk_if_free_cs:N`.)

42.11 Defining new functions

```

1989 (@@=cs)

```

`\cs_new_nopar:Npn` Function which check that the control sequence is free before defining it.

```

\cs_new_nopar:Npe 1990 \cs_set:Npn \__cs_tmp:w #1#2
\cs_new_nopar:Npx 1991   {
  \cs_new:Npn      1992     \cs_gset_protected:Npn #1 ##1
  \cs_new:Npe     1993     {
  \cs_new:Npx     1994       \__kernel_chk_if_free_cs:N ##1
\cs_new_protected_nopar:Npn 1995       #2 ##1
\cs_new_protected_nopar:Npe 1996     }
\cs_new_protected_nopar:Npx 1997   }
\cs_new_protected:Npn 1998 \__cs_tmp:w \cs_new_nopar:Npn          \cs_gset_nopar:Npn
\cs_new_protected:Npe
\cs_new_protected:Npx

```

```

\__cs_tmp:w

```

```

1999 \__cs_tmp:w \cs_new_nopar:Npe \cs_gset_nopar:Npe
2000 \__cs_tmp:w \cs_new_nopar:Npx \cs_gset_nopar:Npx
2001 \__cs_tmp:w \cs_new:Npn \cs_gset:Npn
2002 \__cs_tmp:w \cs_new:Npe \cs_gset:Npe
2003 \__cs_tmp:w \cs_new:Npx \cs_gset:Npx
2004 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
2005 \__cs_tmp:w \cs_new_protected_nopar:Npe \cs_gset_protected_nopar:Npe
2006 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
2007 \__cs_tmp:w \cs_new_protected:Npn \cs_gset_protected:Npn
2008 \__cs_tmp:w \cs_new_protected:Npe \cs_gset_protected:Npe
2009 \__cs_tmp:w \cs_new_protected:Npx \cs_gset_protected:Npx

```

(End of definition for `\cs_new_nopar:Npn` and others. These functions are documented on page 15.)

`\cs_set_nopar:cpn` Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the `c` stands for `csname` argument, see the expansion module). Global versions are also provided.

`\cs_gset_nopar:cpn` `\cs_set_nopar:cpn` $\langle string \rangle \langle rep-text \rangle$ turns $\langle string \rangle$ into a `csname` and then assigns $\langle rep-text \rangle$ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

\cs_new_nopar:cpn
\cs_new_nopar:cpe
\cs_new_nopar:cpx
2010 \cs_set:Npn \__cs_tmp:w #1#2
2011 { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
2012 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
2013 \__cs_tmp:w \cs_set_nopar:cpe \cs_set_nopar:Npe
2014 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
2015 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
2016 \__cs_tmp:w \cs_gset_nopar:cpe \cs_gset_nopar:Npe
2017 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
2018 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
2019 \__cs_tmp:w \cs_new_nopar:cpe \cs_new_nopar:Npe
2020 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End of definition for `\cs_set_nopar:Npn`. This function is documented on page 16.)

`\cs_set:cpn` Variants of the `\cs_set:Npn` versions which make a `csname` out of the first arguments. We may also do this globally.

```

\cs_set:cpe
\cs_set:cpx
2021 \__cs_tmp:w \cs_set:cpn \cs_set:Npn
\cs_gset:cpn
2022 \__cs_tmp:w \cs_set:cpe \cs_set:Npe
\cs_gset:cpe
2023 \__cs_tmp:w \cs_set:cpx \cs_set:Npx
\cs_gset:cpx
2024 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
\cs_new:cpn
2025 \__cs_tmp:w \cs_gset:cpe \cs_gset:Npe
\cs_new:cpe
2026 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
\cs_new:cpx
2027 \__cs_tmp:w \cs_new:cpn \cs_new:Npn
2028 \__cs_tmp:w \cs_new:cpe \cs_new:Npe
2029 \__cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End of definition for `\cs_set:Npn`. This function is documented on page 16.)

`\cs_set_protected_nopar:cpn` Variants of the `\cs_set_protected_nopar:Npn` versions which make a `csname` out of the first arguments. We may also do this globally.

```

\cs_set_protected_nopar:cpe
\cs_set_protected_nopar:cpx
2030 \__cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
\cs_gset_protected_nopar:cpn
2031 \__cs_tmp:w \cs_set_protected_nopar:cpe \cs_set_protected_nopar:Npe
\cs_gset_protected_nopar:cpe
2032 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
\cs_gset_protected_nopar:cpx
\cs_new_protected_nopar:cpn
\cs_new_protected_nopar:cpe
\cs_new_protected_nopar:cpx

```

```

2033 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
2034 \__cs_tmp:w \cs_gset_protected_nopar:cpe \cs_gset_protected_nopar:Npe
2035 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
2036 \__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
2037 \__cs_tmp:w \cs_new_protected_nopar:cpe \cs_new_protected_nopar:Npe
2038 \__cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx

```

(End of definition for `\cs_set_protected_nopar:Npn`. This function is documented on page 16.)

`\cs_set_protected:cpn` Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

\cs_set_protected:cpe
\cs_set_protected:cpx
\cs_gset_protected:cpn
\cs_gset_protected:cpe
\cs_gset_protected:cpx
\cs_new_protected:cpn
\cs_new_protected:cpe
\cs_new_protected:cpx
2039 \__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
2040 \__cs_tmp:w \cs_set_protected:cpe \cs_set_protected:Npe
2041 \__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
2042 \__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
2043 \__cs_tmp:w \cs_gset_protected:cpe \cs_gset_protected:Npe
2044 \__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
2045 \__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
2046 \__cs_tmp:w \cs_new_protected:cpe \cs_new_protected:Npe
2047 \__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End of definition for `\cs_set_protected:Npn`. This function is documented on page 16.)

42.12 Copying definitions

`\cs_set_eq:NN` These macros allow us to copy the definition of a control sequence to another control sequence.

`\cs_set_eq:cN` The = sign allows us to define funny char tokens like = itself or `_` with this function.

`\cs_set_eq:Nc` For the definition of `\c_space_char{~}` to work we need the `~` after the =.

`\cs_set_eq:cc`

`\cs_gset_eq:NN` `\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While

`\cs_gset_eq:cN` `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it

`\cs_gset_eq:Nc` long in order to throw an “already defined” error rather than “runaway argument”.

`\cs_gset_eq:cc`

```

\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:Nc
\cs_new_eq:cc
2048 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
2049 \cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
2050 \cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
2051 \cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
2052 \cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
2053 \cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
2054 \cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
2055 \cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
2056 \cs_new_protected:Npn \cs_new_eq:NN #1
2057 {
2058   \__kernel_chk_if_free_cs:N #1
2059   \tex_global:D \cs_set_eq:NN #1
2060 }
2061 \cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
2062 \cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
2063 \cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }

```

(End of definition for `\cs_set_eq:NN`, `\cs_gset_eq:NN`, and `\cs_new_eq:NN`. These functions are documented on page 20.)

42.13 Undefining functions

`\cs_undefine:N` The following function is used to free the main memory from the definition of some function that isn't in use any longer. The `c` variant is careful not to add the control sequence to the hash table if it isn't there yet, and it also avoids nesting TeX conditionals in case `#1` is unbalanced in this matter. We optimize the case where the command exists by reducing as much as possible the tokens in the conditional.

```

2064 \cs_new_protected:Npn \cs_undefine:N #1
2065   { \cs_gset_eq:NN #1 \tex_undefined:D }
2066 \cs_new_protected:Npn \cs_undefine:c #1
2067   {
2068     \if_cs_exist:w #1 \cs_end:
2069     \else:
2070       \use_i:nnnn
2071     \fi:
2072     \exp_args:Nc \cs_undefine:N {#1}
2073   }

```

(End of definition for `\cs_undefine:N`. This function is documented on page 20.)

42.14 Generating parameter text from argument count

```

2074 <@@=cs>

```

```

\_kernel_cs_parm_from_arg_count:nnF
\_cs_parm_from_arg_count_test:nnF

```

L^AT_EX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#n}`, where `n` is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range `[0, 9]`, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```

2075 \cs_new_protected:Npn \_kernel_cs_parm_from_arg_count:nnF #1#2
2076   {
2077     \exp_args:Ne \_cs_parm_from_arg_count_test:nnF
2078     {
2079       \exp_after:wN \exp_not:n
2080       \if_case:w \int_eval:n {#2}
2081         { }
2082         \or: { ##1 }
2083         \or: { ##1##2 }
2084         \or: { ##1##2##3 }
2085         \or: { ##1##2##3##4 }
2086         \or: { ##1##2##3##4##5 }
2087         \or: { ##1##2##3##4##5##6 }
2088         \or: { ##1##2##3##4##5##6##7 }
2089         \or: { ##1##2##3##4##5##6##7##8 }
2090         \or: { ##1##2##3##4##5##6##7##8##9 }
2091         \else: { \c_false_bool }
2092       \fi:
2093     }
2094     {#1}

```

```

2095 }
2096 \cs_new_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
2097 {
2098   \if_meaning:w \c_false_bool #1
2099     \exp_after:wN \use_ii:nn
2100   \else:
2101     \exp_after:wN \use_i:nn
2102   \fi:
2103   { #2 {#1} }
2104 }

```

(End of definition for `__kernel_cs_parm_from_arg_count:nnF` and `__cs_parm_from_arg_count_test:nnF`.)

42.15 Defining functions from a given number of arguments

```

2105 <@@=cs>

```

`__cs_count_signature:N` Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `\tl_count:n` if there is a signature, otherwise `-1` arguments to signal an error. We need a variant form right away.

```

2106 \cs_new:Npn \__cs_count_signature:N #1
2107   { \exp_args:Nf \__cs_count_signature:n { \cs_split_function:N #1 } }
2108 \cs_new:Npn \__cs_count_signature:n #1
2109   { \int_eval:n { \__cs_count_signature:nnN #1 } }
2110 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
2111   {
2112     \if_meaning:w \c_true_bool #3
2113       \tl_count:n {#2}
2114     \else:
2115       -1
2116     \fi:
2117   }
2118 \cs_new:Npn \__cs_count_signature:c
2119   { \exp_args:Nc \__cs_count_signature:N }

```

(End of definition for `__cs_count_signature:N`, `__cs_count_signature:n`, and `__cs_count_signature:nnN`.)

```

\cs_generate_from_arg_count:NNnn
\cs_generate_from_arg_count:cNnn
\cs_generate_from_arg_count:Ncnn

```

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since \TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```

2120 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
2121   {
2122     \__kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
2123     {
2124       \msg_error:nnee { kernel } { bad-number-of-arguments }
2125       { \token_to_str:N #1 } { \int_eval:n {#3} }

```

```

2126     \use_none:n
2127   }
2128   {#4}
2129 }

```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

2130 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
2131   { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
2132 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
2133   { \exp_args:Nnc \cs_generate_from_arg_count:NNnn }

```

(End of definition for `\cs_generate_from_arg_count:NNnn`. This function is documented on page 19.)

42.16 Using the signature to define functions

```

2134 <@@=cs>

```

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, `\cs_set:Nn \foo_bar:nn {#1,#2}`.

We want to define `\cs_set:Nn` as

```

\cs_set:Nn
\cs_set:Ne
\cs_set:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Ne
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Ne
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Ne
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Ne
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Ne
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Ne
\cs_gset_protected:Nx
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:Ne
\cs_gset_protected_nopar:Nx
\cs_new:Nn
\cs_new:Ne
\cs_new:Nx
\cs_new_nopar:Nn
\cs_new_nopar:Ne
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Ne
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Ne
\cs_new_protected_nopar:Nx

```

```

\cs_set_protected:Npn \cs_set:Nn #1#2
{
  \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
  { \@@_count_signature:N #1 } {#2}
}

```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```

2135 \cs_set:Npn \__cs_tmp:w #1#2#3
2136   {
2137     \cs_new_protected:cpx { cs_ #1 : #2 }
2138     {
2139       \exp_not:N \__cs_generate_from_signature:NNn
2140       \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
2141     }
2142   }
2143 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
2144   {
2145     \use:e
2146     {
2147       \__cs_generate_from_signature:nnNNNn
2148       \cs_split_function:N #2
2149     }
2150     #1 #2
2151   }
2152 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
2153   {
2154     \bool_if:NTF #3
2155     {
2156       \cs_set_nopar:Npx \__cs_tmp:w

```

```

2157     { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
2158 \tl_if_empty:oF \__cs_tmp:w
2159     {
2160         \msg_error:nnee { kernel } { non-base-function }
2161         { \token_to_str:N #5 } {#2} { \__cs_tmp:w }
2162     }
2163     \cs_generate_from_arg_count:NNnn
2164     #5 #4 { \tl_count:n {#2} } {#6}
2165 }
2166 {
2167     \msg_error:nne { kernel } { missing-colon }
2168     { \token_to_str:N #5 }
2169 }
2170 }
2171 \cs_new:Npn \__cs_generate_from_signature:n #1
2172 {
2173     \if:w n #1 \else: \if:w N #1 \else:
2174     \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
2175 }

```

Then we define the 24 variants beginning with N.

```

2176 \__cs_tmp:w { set } { Nn } { Npn }
2177 \__cs_tmp:w { set } { Ne } { Npe }
2178 \__cs_tmp:w { set } { Nx } { Npx }
2179 \__cs_tmp:w { set_nopar } { Nn } { Npn }
2180 \__cs_tmp:w { set_nopar } { Ne } { Npe }
2181 \__cs_tmp:w { set_nopar } { Nx } { Npx }
2182 \__cs_tmp:w { set_protected } { Nn } { Npn }
2183 \__cs_tmp:w { set_protected } { Ne } { Npe }
2184 \__cs_tmp:w { set_protected } { Nx } { Npx }
2185 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
2186 \__cs_tmp:w { set_protected_nopar } { Ne } { Npe }
2187 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
2188 \__cs_tmp:w { gset } { Nn } { Npn }
2189 \__cs_tmp:w { gset } { Ne } { Npe }
2190 \__cs_tmp:w { gset } { Nx } { Npx }
2191 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
2192 \__cs_tmp:w { gset_nopar } { Ne } { Npe }
2193 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
2194 \__cs_tmp:w { gset_protected } { Nn } { Npn }
2195 \__cs_tmp:w { gset_protected } { Ne } { Npe }
2196 \__cs_tmp:w { gset_protected } { Nx } { Npx }
2197 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
2198 \__cs_tmp:w { gset_protected_nopar } { Ne } { Npe }
2199 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
2200 \__cs_tmp:w { new } { Nn } { Npn }
2201 \__cs_tmp:w { new } { Ne } { Npe }
2202 \__cs_tmp:w { new } { Nx } { Npx }
2203 \__cs_tmp:w { new_nopar } { Nn } { Npn }
2204 \__cs_tmp:w { new_nopar } { Ne } { Npe }
2205 \__cs_tmp:w { new_nopar } { Nx } { Npx }
2206 \__cs_tmp:w { new_protected } { Nn } { Npn }
2207 \__cs_tmp:w { new_protected } { Ne } { Npe }
2208 \__cs_tmp:w { new_protected } { Nx } { Npx }
2209 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }

```

```

2210 \__cs_tmp:w { new_protected_nopar } { Ne } { Npe }
2211 \__cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End of definition for \cs_set:Nn and others. These functions are documented on page 18.)

\cs_set:cn The 24 c variants simply use \exp_args:Nc.

```

\cs_set:ce 2212 \cs_set:Npn \__cs_tmp:w #1#2
\cs_set:cx 2213 {

```

```

\cs_set_nopar:cn 2214 \cs_new_protected:cpx { cs_ #1 : c #2 }
\cs_set_nopar:ce 2215 {

```

```

\cs_set_nopar:cx 2216 \exp_not:N \exp_args:Nc
\cs_set_protected:cn 2217 \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
\cs_set_protected:ce 2218 }
\cs_set_protected:cx 2219 }

```

```

\cs_set_protected_nopar:cn 2220 \__cs_tmp:w { set } { n }
\cs_set_protected_nopar:ce 2221 \__cs_tmp:w { set } { e }
\cs_set_protected_nopar:cx 2222 \__cs_tmp:w { set } { x }

```

```

\cs_set_protected_nopar:cn 2223 \__cs_tmp:w { set_nopar } { n }
\cs_gset:cn 2224 \__cs_tmp:w { set_nopar } { e }
\cs_gset:ce 2225 \__cs_tmp:w { set_nopar } { x }

```

```

\cs_gset:cx 2226 \__cs_tmp:w { set_protected } { n }
\cs_gset_nopar:cn 2227 \__cs_tmp:w { set_protected } { e }
\cs_gset_nopar:ce 2228 \__cs_tmp:w { set_protected } { x }

```

```

\cs_gset_nopar:cx 2229 \__cs_tmp:w { set_protected_nopar } { n }
\cs_gset_protected:cn 2230 \__cs_tmp:w { set_protected_nopar } { e }
\cs_gset_protected:ce 2231 \__cs_tmp:w { set_protected_nopar } { x }

```

```

\cs_gset_protected:cx 2232 \__cs_tmp:w { gset } { n }
\cs_gset_protected_nopar:cn 2233 \__cs_tmp:w { gset } { e }
\cs_gset_protected_nopar:ce 2234 \__cs_tmp:w { gset } { x }

```

```

\cs_gset_protected_nopar:cn 2235 \__cs_tmp:w { gset_nopar } { n }
\cs_gset_protected_nopar:ce 2236 \__cs_tmp:w { gset_nopar } { e }
\cs_gset_protected_nopar:cx 2237 \__cs_tmp:w { gset_nopar } { x }

```

```

\cs_new:cn 2238 \__cs_tmp:w { gset_protected } { n }
\cs_new:ce 2239 \__cs_tmp:w { gset_protected } { e }
\cs_new:cx 2240 \__cs_tmp:w { gset_protected } { x }

```

```

\cs_new_nopar:cn 2241 \__cs_tmp:w { gset_protected_nopar } { n }
\cs_new_nopar:ce 2242 \__cs_tmp:w { gset_protected_nopar } { e }
\cs_new_nopar:cx 2243 \__cs_tmp:w { gset_protected_nopar } { x }

```

```

\cs_new_protected:cn 2244 \__cs_tmp:w { new } { n }
\cs_new_protected:ce 2245 \__cs_tmp:w { new } { e }
\cs_new_protected:cx 2246 \__cs_tmp:w { new } { x }

```

```

\cs_new_protected_nopar:cn 2247 \__cs_tmp:w { new_nopar } { n }
\cs_new_protected_nopar:ce 2248 \__cs_tmp:w { new_nopar } { e }
\cs_new_protected_nopar:cx 2249 \__cs_tmp:w { new_nopar } { x }

```

```

\cs_new_protected_nopar:cn 2250 \__cs_tmp:w { new_protected } { n }
\cs_new_protected_nopar:ce 2251 \__cs_tmp:w { new_protected } { e }
\cs_new_protected_nopar:cx 2252 \__cs_tmp:w { new_protected } { x }

```

```

\cs_new_protected_nopar:cn 2253 \__cs_tmp:w { new_protected_nopar } { n }
\cs_new_protected_nopar:ce 2254 \__cs_tmp:w { new_protected_nopar } { e }
\cs_new_protected_nopar:cx 2255 \__cs_tmp:w { new_protected_nopar } { x }

```

(End of definition for \cs_set:Nn. This function is documented on page 18.)

42.17 Checking control sequence equality

`\cs_if_eq_p:NN` Check if two control sequences are identical.

```

\cs_if_eq_p:cN 2256 \prg_new_conditional:Npnm \cs_if_eq:NN #1#2 { p , T , F , TF }
\cs_if_eq_p:Nc 2257 {
\cs_if_eq_p:cc 2258   \if_meaning:w #1#2
\cs_if_eq:NNTF 2259   \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF 2260 }
\cs_if_eq:NcTF 2261 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_if_eq:NcTF 2262 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
\cs_if_eq:ccTF 2263 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
\cs_if_eq:ccTF 2264 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
\cs_if_eq_p:Nc 2265 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
\cs_if_eq:NcTF 2266 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
\cs_if_eq:NcT 2267 \cs_new:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
\cs_if_eq:NcF 2268 \cs_new:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
\cs_if_eq_p:cc 2269 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
\cs_if_eq:ccTF 2270 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_if_eq:ccT 2271 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
\cs_if_eq:ccF 2272 \cs_new:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End of definition for `\cs_if_eq:NNTF`. This function is documented on page 28.)

42.18 Diagnostic functions

```

2273 (@@=kernel)
\__kernel_chk_defined:NT Error if the variable #1 is not defined.
2274 \cs_new_protected:Npn \__kernel_chk_defined:NT #1#2
2275 {
2276   \cs_if_exist:NTF #1
2277     {#2}
2278     {
2279       \msg_error:nne { kernel } { variable-not-defined }
2280       { \token_to_str:N #1 }
2281     }
2282 }

```

(End of definition for `__kernel_chk_defined:NT`.)

`__kernel_register_show:N` Simply using the `\showthe` primitive does not allow for line-wrapping, so instead use `\tl_show:n` and `\tl_log:n` (defined in `l3tl` and that performs line-wrapping). This displays `>~⟨variable⟩=⟨value⟩`. We expand the value before-hand as otherwise some integers (such as `\currentgrouplevel` or `\currentgrouptype`) altered by the line-wrapping code would show wrong values.

```

\__kernel_register_show_aux:NN
\__kernel_register_show_aux:nNN
2283 \cs_new_protected:Npn \__kernel_register_show:N
2284   { \__kernel_register_show_aux:NN \tl_show:n }
2285 \cs_new_protected:Npn \__kernel_register_show:c
2286   { \exp_args:Nc \__kernel_register_show:N }
2287 \cs_new_protected:Npn \__kernel_register_log:N
2288   { \__kernel_register_show_aux:NN \tl_log:n }
2289 \cs_new_protected:Npn \__kernel_register_log:c
2290   { \exp_args:Nc \__kernel_register_log:N }

```

```

2291 \cs_new_protected:Npn \__kernel_register_show_aux:NN #1#2
2292 {
2293   \__kernel_chk_defined:NT #2
2294   {
2295     \exp_args:No \__kernel_register_show_aux:nNN
2296     { \tex_the:D #2 } #2 #1
2297   }
2298 }
2299 \cs_new_protected:Npn \__kernel_register_show_aux:nNN #1#2#3
2300 { \exp_args:No #3 { \token_to_str:N #2 = #1 } }

```

(End of definition for `__kernel_register_show:N` and others.)

`\cs_show:N` Some control sequences have a very long name or meaning. Thus, simply using TeX's primitive `\show` could lead to overlong lines. The output of this primitive is mimicked to some extent, then the re-built string is given to `\tl_show:n` or `\tl_log:n` for line-wrapping. We must expand the meaning before passing it to the wrapping code as otherwise we would wrongly see the definitions that are in place there. To get correct escape characters, set the `\escapechar` in a group; this also localizes the assignment performed by e-expansion. The `\cs_show:c` and `\cs_log:c` commands convert their argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```

2301 \cs_new_protected:Npn \cs_show:N { \__kernel_show:NN \tl_show:n }
2302 \cs_new_protected:Npn \cs_show:c
2303 { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
2304 \cs_new_protected:Npn \cs_log:N { \__kernel_show:NN \tl_log:n }
2305 \cs_new_protected:Npn \cs_log:c
2306 { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
2307 \cs_new_protected:Npn \__kernel_show:NN #1#2
2308 {
2309   \group_begin:
2310   \int_set:Nn \tex_escapechar:D { '\ }
2311   \exp_args:NNe
2312   \group_end:
2313   #1 { \token_to_str:N #2 = \cs_meaning:N #2 }
2314 }

```

(End of definition for `\cs_show:N`, `\cs_log:N`, and `__kernel_show:NN`. These functions are documented on page 20.)

`\group_show_list:` Wrapper around `\showgroups`. Getting TeX to write to the log without interruption the run is done by altering the interaction mode.

```

\__kernel_group_show:NN
2315 \cs_new_protected:Npn \group_show_list:
2316 { \__kernel_group_show:NN \use_none:n 1 }
2317 \cs_new_protected:Npn \group_log_list:
2318 { \__kernel_group_show:NN \int_gzero:N 0 }
2319 \cs_new_protected:Npn \__kernel_group_show:NN #1#2
2320 {
2321   \use:e
2322   {
2323     #1 \tex_interactionmode:D
2324     \int_set:Nn \tex_tracingonline:D {#2}
2325     \int_set:Nn \tex_errorcontextlines:D { -1 }
2326     \exp_not:N \exp_after:wN \scan_stop:

```

```

2327     \tex_showgroups:D
2328     \int_gset:Nn \tex_interactionmode:D
2329       { \int_use:N \tex_interactionmode:D }
2330     \int_set:Nn \tex_tracingonline:D
2331       { \int_use:N \tex_tracingonline:D }
2332     \int_set:Nn \tex_errorcontextlines:D
2333       { \int_use:N \tex_errorcontextlines:D }
2334   }
2335 }

```

(End of definition for `\group_show_list:`, `\group_log_list:`, and `_kernel_group_show:NN`. These functions are documented on page 14.)

42.19 Decomposing a macro definition

`\cs_prefix_spec:N` We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the parameter specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```

2336 \use:e
2337 {
2338   \exp_not:n { \cs_new:Npn \_kernel_prefix_arg_replacement:wN #1 }
2339   \tl_to_str:n { macro : } \exp_not:n { #2 -> #3 \s__kernel_stop #4 }
2340 }
2341 { #4 {#1} {#2} {#3} }
2342 \cs_new:Npn \cs_prefix_spec:N #1
2343 {
2344   \token_if_macro:NTF #1
2345   {
2346     \exp_after:wN \_kernel_prefix_arg_replacement:wN
2347     \token_to_meaning:N #1 \s__kernel_stop \use_i:nnn
2348   }
2349   { \scan_stop: }
2350 }
2351 \cs_new:Npn \cs_parameter_spec:N #1
2352 {
2353   \token_if_macro:NTF #1
2354   {
2355     \exp_after:wN \_kernel_prefix_arg_replacement:wN
2356     \token_to_meaning:N #1 \s__kernel_stop \use_ii:nnn
2357   }
2358   { \scan_stop: }
2359 }
2360 \cs_new:Npn \cs_replacement_spec:N #1
2361 {
2362   \token_if_macro:NTF #1
2363   {
2364     \exp_after:wN \_kernel_prefix_arg_replacement:wN
2365     \token_to_meaning:N #1 \s__kernel_stop \use_iii:nnn
2366   }
2367   { \scan_stop: }

```

```
2368 }
```

(End of definition for `\cs_prefix_spec:N` and others. These functions are documented on page 22.)

42.20 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```
2369 \cs_new:Npn \prg_do_nothing: { }
```

(End of definition for `\prg_do_nothing:.` This function is documented on page 13.)

42.21 Breaking out of mapping functions

```
2370 <@@=prg>
```

`\prg_break_point:Nn`
`\prg_map_break:Nn` In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of `__prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `__prg_break_point:Nn`, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```
2371 \cs_new_eq:NN \prg_break_point:Nn \use_ii:nn
2372 \cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
2373 {
2374   #5
2375   \if_meaning:w #1 #4
2376     \exp_after:wN \use_iii:nnn
2377   \fi:
2378   \prg_map_break:Nn #1 {#2}
2379 }
```

(End of definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 72.)

`\prg_break_point:` Very simple analogues of `\prg_break_point:Nn` and `\prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

`\prg_break:`
`\prg_break:n`

```
2380 \cs_new_eq:NN \prg_break_point: \prg_do_nothing:
2381 \cs_new:Npn \prg_break: #1 \prg_break_point: { }
2382 \cs_new:Npn \prg_break:n #1#2 \prg_break_point: {#1}
```

(End of definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 73.)

42.22 Starting a paragraph

`\mode_leave_vertical:` The approach here is different to that used by L^AT_EX 2_ε or plain T_EX, which unbox a void box to force horizontal mode. That inserts the `\everypar` tokens *before* the re-inserted unboxing tokens. The approach here uses a protected macro, equivalent to the `\quitvmode` primitive. In vertical mode, the `\indent` primitive is inserted: this will switch to horizontal mode and insert `\everypar` tokens and nothing else. Unlike the

L^AT_EX 2_ε version, the availability of ϵ -T_EX means using a mode test can be done at for example the start of an `\halign`.

```
2383 \cs_new_protected:Npn \mode_leave_vertical:
2384 {
2385   \if_mode_vertical:
2386     \exp_after:wN \tex_indent:D
2387   \fi:
2388 }
```

(End of definition for `\mode_leave_vertical`:: This function is documented on page 30.)

```
2389 </package>
```

Chapter 43

l3expan implementation

```
2390 (*package)
```

```
2391 (@@=exp)
```

`\l__exp_internal_tl` The `\exp_` module has its private variable to temporarily store the result of `x`-type argument expansion. This is done to avoid interference with other functions using temporary variables.

(End of definition for `\l__exp_internal_tl`.)

`\exp_after:wN` These are defined in `l3basics`, as they are needed “early”. This is just a reminder of that fact!

`\exp_not:N`

`\exp_not:n`

(End of definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 39.)

43.1 General expansion

In this section a general mechanism for defining functions that handle arguments is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the $\text{\LaTeX}3$ names for `\cs_set:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 43.7. In section 43.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l__exp_internal_tl` This scratch token list variable is defined in `l3basics`.

(End of definition for `\l__exp_internal_tl`.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::⟨Z⟩` always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed. One exception to this rule is `\::p`, which has to grab an argument delimited by a left brace.

`_exp_arg_next:nnn` #1 is the result of an expansion step, #2 is the remaining argument manipulations and
`_exp_arg_next:Nnn` #3 is the current result of the expansion chain. This auxiliary function moves #1 back
after #3 in the input stream and checks if any expansion is left to be done by calling
#2. In by far the most cases we need to add a set of braces to the result of an argument
manipulation so it is more effective to do it directly here. Actually, so far only the `c` of
the final argument manipulation variants does not require a set of braces.

```
2392 \cs_new:Npn \_exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
2393 \cs_new:Npn \_exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(End of definition for `_exp_arg_next:nnn` and `_exp_arg_next:Nnn`.)

`\:::` The end marker is just another name for the identity function.

```
2394 \cs_new:Npn \::: #1 {#1}
```

(End of definition for `\:::`. This function is documented on page 43.)

`\::n` This function is used to skip an argument that doesn't need to be expanded.

```
2395 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(End of definition for `\::n`. This function is documented on page 43.)

`\::N` This function is used to skip an argument that consists of a single token and doesn't need
to be expanded.

```
2396 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(End of definition for `\::N`. This function is documented on page 43.)

`\::p` This function is used to skip an argument that is delimited by a left brace and doesn't
need to be expanded. It is not wrapped in braces in the result.

```
2397 \cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(End of definition for `\::p`. This function is documented on page 43.)

`\::c` This function is used to skip an argument that is turned into a control sequence without
expansion.

```
2398 \cs_new:Npn \::c #1 \::: #2#3
2399 { \exp_after:wN \_exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(End of definition for `\::c`. This function is documented on page 43.)

`\::o` This function is used to expand an argument once.

```
2400 \cs_new:Npn \::o #1 \::: #2#3
2401 { \exp_after:wN \_exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(End of definition for `\::o`. This function is documented on page 43.)

`\::e` With the `\expanded` primitive available, just expand.

```
2402 \cs_new:Npn \::e #1 \::: #2#3
2403 { \tex_expanded:D { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } } }
```

(End of definition for `\::e`. This function is documented on page 43.)

\::f This function is used to expand a token list until the first unexpandable token is found. This is achieved through `\exp:w \exp_end_continue_f:w` that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable (if that is a space it is removed). We introduce `\exp_stop_f:` to mark such an end-of-expansion marker. For example, `f`-expanding `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` where `\l_tmpa_tl` contains the characters `lur` gives `\tex_let:D \aaa = \blurb` which then turns out to start with the non-expandable token `\tex_let:D`. Since the expansion of `\exp:w \exp_end_continue_f:w` is empty, we wind up with a fully expanded list, only `TeX` has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `e` and `x` argument type.

```

2404 \cs_new:Npn \::f #1 \::: #2#3
2405 {
2406   \exp_after:wN \__exp_arg_next:nnn
2407   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2408   {#1} {#2}
2409 }
2410 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }

```

(End of definition for `\::f` and `\exp_stop_f:`. These functions are documented on page 43.)

\::x This function is used to expand an argument fully. We build in the expansion of `__exp_arg_next:nnn`.

```

2411 \cs_new_protected:Npn \::x #1 \::: #2#3
2412 {
2413   \cs_set_nopar:Npe \l__exp_internal_tl
2414   { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
2415   \l__exp_internal_tl
2416 }

```

(End of definition for `\::x`. This function is documented on page 43.)

\::v These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim`, `muskip`, or built-in `TeX` register. The `V` version expects a single token whereas `v` like `c` creates a csname from its argument given in braces and then evaluates it as if it was a `V`. The `\exp:w` sets off an expansion similar to an `f`-type expansion, which we terminate using `\exp_end:`. The argument is returned in braces.

```

2417 \cs_new:Npn \::V #1 \::: #2#3
2418 {
2419   \exp_after:wN \__exp_arg_next:nnn
2420   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2421   {#1} {#2}
2422 }
2423 \cs_new:Npn \::v #1 \::: #2#3
2424 {
2425   \exp_after:wN \__exp_arg_next:nnn
2426   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2427   {#1} {#2}
2428 }

```

(End of definition for `\::v` and `\::V`. These functions are documented on page 43.)

`_exp_eval_register:N` This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in T_EX register such as `\count`. For the T_EX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we want here is to find out whether the token expands to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the token in question when it has been prefixed with `\exp_not:N` and the token itself. If it is a macro, the prefixed `\exp_not:N` temporarily turns it into the primitive `\scan_stop:.`

```

2429 \cs_new:Npn \_exp_eval_register:N #1
2430 {
2431   \exp_after:wN \if_meaning:w \exp_not:N #1 #1

```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:.` In that case we throw an error. We could let T_EX do it for us but that would result in the rather obscure

```
! You can't use '\relax' after \the.
```

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```

2432   \if_meaning:w \scan_stop: #1
2433   \_exp_eval_error_msg:w
2434   \fi:

```

The next bit requires some explanation. The function must be initiated by `\exp:w` and we want to terminate this expansion chain by inserting the `\exp_end:` token. However, we have to expand the register `#1` before we do that. If it is a T_EX register, we need to execute the sequence `\exp_after:wN \exp_end: \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \exp_end: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```

2435   \else:
2436     \exp_after:wN \use_i_ii:nnn
2437   \fi:
2438   \exp_after:wN \exp_end: \tex_the:D #1
2439 }
2440 \cs_new:Npn \_exp_eval_register:c #1
2441 { \exp_after:wN \_exp_eval_register:N \cs:w #1 \cs_end: }

```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```

! Undefined control sequence.
<argument> \LaTeX3 error:
                               Erroneous variable used!
1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}

```

```

2442 \cs_new:Npn \_exp_eval_error_msg:w #1 \tex_the:D #2
2443 {
2444   \fi:
2445   \fi:
2446   \msg_expandable_error:nnn { kernel } { bad-variable } {#2}
2447   \exp_end:
2448 }

```

(End of definition for `_exp_eval_register:N` and `_exp_eval_error_msg:w`.)

43.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

`\exp_args:Nc` In l3basics.
`\exp_args:cc`

(End of definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 36.)

`\exp_args:NNc` Here are the functions that turn their argument into csnames but are expandable.
`\exp_args:Ncc`
`\exp_args:Nccc`

```
2449 \cs_new:Npn \exp_args:NNc #1#2#3
2450 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
2451 \cs_new:Npn \exp_args:Ncc #1#2#3
2452 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2453 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2454 {
2455   \exp_after:wN #1
2456   \cs:w #2 \exp_after:wN \cs_end:
2457   \cs:w #3 \exp_after:wN \cs_end:
2458   \cs:w #4 \cs_end:
2459 }
```

(End of definition for `\exp_args:NNc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 37.)

`\exp_args:No` Those lovely runs of expansion!
`\exp_args:NNo`
`\exp_args:NNNo`

```
2460 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
2461 \cs_new:Npn \exp_args:NNo #1#2#3
2462 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
2463 \cs_new:Npn \exp_args:NNNo #1#2#3#4
2464 { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(End of definition for `\exp_args:No`, `\exp_args:NNo`, and `\exp_args:NNNo`. These functions are documented on page 36.)

`\exp_args:Ne` When the `\expanded` primitive is available, use it.

```
2465 \cs_new:Npn \exp_args:Ne #1#2
2466 { \exp_after:wN #1 \tex_expanded:D { {#2} } }
```

(End of definition for `\exp_args:Ne`. This function is documented on page 36.)

`\exp_args:Nf`
`\exp_args:NV`
`\exp_args:Nv`

```
2467 \cs_new:Npn \exp_args:Nf #1#2
2468 { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2469 \cs_new:Npn \exp_args:Nv #1#2
2470 {
2471   \exp_after:wN #1 \exp_after:wN
2472   { \exp:w \__exp_eval_register:c {#2} }
2473 }
2474 \cs_new:Npn \exp_args:NV #1#2
2475 {
2476   \exp_after:wN #1 \exp_after:wN
2477   { \exp:w \__exp_eval_register:N #2 }
2478 }
```

(End of definition for `\exp_args:Nf`, `\exp_args:NV`, and `\exp_args:Nv`. These functions are documented on page 36.)

`\exp_args:NNV` Some more hand-tuned function with three arguments. If we forced that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```

2479 \cs_new:Npn \exp_args:NNV #1#2#3
2480 {
2481   \exp_after:wN #1
2482   \exp_after:wN #2
2483   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2484 }
2485 \cs_new:Npn \exp_args:NNv #1#2#3
2486 {
2487   \exp_after:wN #1
2488   \exp_after:wN #2
2489   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2490 }
2491 \cs_new:Npn \exp_args:NNe #1#2#3
2492 {
2493   \exp_after:wN #1
2494   \exp_after:wN #2
2495   \tex_expanded:D { {#3} }
2496 }
2497 \cs_new:Npn \exp_args:NNf #1#2#3
2498 {
2499   \exp_after:wN #1
2500   \exp_after:wN #2
2501   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2502 }
2503 \cs_new:Npn \exp_args:Nco #1#2#3
2504 {
2505   \exp_after:wN #1
2506   \cs:w #2 \exp_after:wN \cs_end:
2507   \exp_after:wN {#3}
2508 }
2509 \cs_new:Npn \exp_args:NcV #1#2#3
2510 {
2511   \exp_after:wN #1
2512   \cs:w #2 \exp_after:wN \cs_end:
2513   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2514 }
2515 \cs_new:Npn \exp_args:Ncv #1#2#3
2516 {
2517   \exp_after:wN #1
2518   \cs:w #2 \exp_after:wN \cs_end:
2519   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2520 }
2521 \cs_new:Npn \exp_args:Ncf #1#2#3
2522 {
2523   \exp_after:wN #1
2524   \cs:w #2 \exp_after:wN \cs_end:
2525   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2526 }
2527 \cs_new:Npn \exp_args:NVV #1#2#3
2528 {
2529   \exp_after:wN #1

```

```

2530 \exp_after:wN { \exp:w \exp_after:wN
2531 \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2532 \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2533 }

```

(End of definition for `\exp_args:NNV` and others. These functions are documented on page 37.)

`\exp_args:NNNV`
`\exp_args:NNNv`
`\exp_args:NNNe`
`\exp_args:NcNc`
`\exp_args:NcNo`
`\exp_args:Ncco`

A few more that we can hand-tune.

```

2534 \cs_new:Npn \exp_args:NNNV #1#2#3#4
2535 {
2536 \exp_after:wN #1
2537 \exp_after:wN #2
2538 \exp_after:wN #3
2539 \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
2540 }
2541 \cs_new:Npn \exp_args:NNNv #1#2#3#4
2542 {
2543 \exp_after:wN #1
2544 \exp_after:wN #2
2545 \exp_after:wN #3
2546 \exp_after:wN { \exp:w \__exp_eval_register:c {#4} }
2547 }
2548 \cs_new:Npn \exp_args:NNNe #1#2#3#4
2549 {
2550 \exp_after:wN #1
2551 \exp_after:wN #2
2552 \exp_after:wN #3
2553 \tex_expanded:D { {#4} }
2554 }
2555 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2556 {
2557 \exp_after:wN #1
2558 \cs:w #2 \exp_after:wN \cs_end:
2559 \exp_after:wN #3
2560 \cs:w #4 \cs_end:
2561 }
2562 \cs_new:Npn \exp_args:NcNo #1#2#3#4
2563 {
2564 \exp_after:wN #1
2565 \cs:w #2 \exp_after:wN \cs_end:
2566 \exp_after:wN #3
2567 \exp_after:wN {#4}
2568 }
2569 \cs_new:Npn \exp_args:Ncco #1#2#3#4
2570 {
2571 \exp_after:wN #1
2572 \cs:w #2 \exp_after:wN \cs_end:
2573 \cs:w #3 \exp_after:wN \cs_end:
2574 \exp_after:wN {#4}
2575 }

```

(End of definition for `\exp_args:NNNV` and others. These functions are documented on page 37.)

`\exp_args:Nx`

```

2576 \cs_new_protected:Npn \exp_args:Nx #1#2
2577   { \use:x { \exp_not:N #1 {#2} } }

```

(End of definition for `\exp_args:Nx`. This function is documented on page 36.)

43.3 Last-unbraced versions

`_exp_arg_last_unbraced:nn` There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\::o_unbraced
\::V_unbraced
\::v_unbraced
\::e_unbraced
\::f_unbraced
\::x_unbraced
2578 \cs_new:Npn \_exp_arg_last_unbraced:nn #1#2 { #2#1 }
2579 \cs_new:Npn \::o_unbraced \::: #1#2
2580   { \exp_after:wN \_exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2581 \cs_new:Npn \::V_unbraced \::: #1#2
2582   {
2583     \exp_after:wN \_exp_arg_last_unbraced:nn
2584     \exp_after:wN { \exp:w \_exp_eval_register:N #2 } {#1}
2585   }
2586 \cs_new:Npn \::v_unbraced \::: #1#2
2587   {
2588     \exp_after:wN \_exp_arg_last_unbraced:nn
2589     \exp_after:wN { \exp:w \_exp_eval_register:c {#2} } {#1}
2590   }
2591 \cs_new:Npn \::e_unbraced \::: #1#2
2592   { \tex_expanded:D { \exp_not:n {#1} #2 } }
2593 \cs_new:Npn \::f_unbraced \::: #1#2
2594   {
2595     \exp_after:wN \_exp_arg_last_unbraced:nn
2596     \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2597   }
2598 \cs_new_protected:Npn \::x_unbraced \::: #1#2
2599   {
2600     \cs_set_nopar:Npe \l__exp_internal_tl { \exp_not:n {#1} #2 }
2601     \l__exp_internal_tl
2602   }

```

(End of definition for `_exp_arg_last_unbraced:nn` and others. These functions are documented on page 43.)

`\exp_last_unbraced:No` Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

\exp_last_unbraced:NV
\exp_last_unbraced:Nv
\exp_last_unbraced:Ne
\exp_last_unbraced:Nf
\exp_last_unbraced:NNo
\exp_last_unbraced:NNV
\exp_last_unbraced:NNf
\exp_last_unbraced:Nco
\exp_last_unbraced:NcV
\exp_last_unbraced:NNNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNf
\exp_last_unbraced:Nno
\exp_last_unbraced:Nnf
\exp_last_unbraced:Noo
\exp_last_unbraced:Nfo
\exp_last_unbraced:NnNo
\exp_last_unbraced:NNNNo
\exp_last_unbraced:NNNNf
\exp_last_unbraced:Nx
2603 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
2604 \cs_new:Npn \exp_last_unbraced:NV #1#2
2605   { \exp_after:wN #1 \exp:w \_exp_eval_register:N #2 }
2606 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2607   { \exp_after:wN #1 \exp:w \_exp_eval_register:c {#2} }
2608 \cs_new:Npn \exp_last_unbraced:Ne #1#2
2609   { \exp_after:wN #1 \tex_expanded:D {#2} }
2610 \cs_new:Npn \exp_last_unbraced:Nf #1#2
2611   { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2612 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2613   { \exp_after:wN #1 \exp_after:wN #2 #3 }
2614 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
2615   {

```

```

2616     \exp_after:wN #1
2617     \exp_after:wN #2
2618     \exp:w \_exp_eval_register:N #3
2619   }
2620 \cs_new:Npn \exp_last_unbraced:NNf #1#2#3
2621 {
2622     \exp_after:wN #1
2623     \exp_after:wN #2
2624     \exp:w \exp_end_continue_f:w #3
2625 }
2626 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
2627 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2628 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2629 {
2630     \exp_after:wN #1
2631     \cs:w #2 \exp_after:wN \cs_end:
2632     \exp:w \_exp_eval_register:N #3
2633 }
2634 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2635 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2636 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2637 {
2638     \exp_after:wN #1
2639     \exp_after:wN #2
2640     \exp_after:wN #3
2641     \exp:w \_exp_eval_register:N #4
2642 }
2643 \cs_new:Npn \exp_last_unbraced:NNNf #1#2#3#4
2644 {
2645     \exp_after:wN #1
2646     \exp_after:wN #2
2647     \exp_after:wN #3
2648     \exp:w \exp_end_continue_f:w #4
2649 }
2650 \cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }
2651 \cs_new:Npn \exp_last_unbraced:Nnf { \::n \::f_unbraced \::: }
2652 \cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
2653 \cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
2654 \cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \::: }
2655 \cs_new:Npn \exp_last_unbraced:NNNNo #1#2#3#4#5
2656 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 \exp_after:wN #4 #5 }
2657 \cs_new:Npn \exp_last_unbraced:NNNf #1#2#3#4#5
2658 {
2659     \exp_after:wN #1
2660     \exp_after:wN #2
2661     \exp_after:wN #3
2662     \exp_after:wN #4
2663     \exp:w \exp_end_continue_f:w #5
2664 }
2665 \cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \::: }

```

(End of definition for `\exp_last_unbraced:No` and others. These functions are documented on page 39.)

`\exp_last_two_unbraced:Noo` If #2 is a single token then this can be implemented as
`_exp_last_two_unbraced:noN`

```

\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
  { \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }

```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```

2666 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2667   { \exp_after:wN \_exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2668 \cs_new:Npn \_exp_last_two_unbraced:noN #1#2#3
2669   { \exp_after:wN #3 #2 #1 }

```

(End of definition for `\exp_last_two_unbraced:Noo` and `_exp_last_two_unbraced:noN`. This function is documented on page 39.)

43.4 Preventing expansion

`_kernel_exp_not:w` At the kernel level, we need the primitive behaviour to allow expansion *before* the brace group.

```

2670 \cs_new_eq:NN \_kernel_exp_not:w \tex_unexpanded:D

```

(End of definition for `_kernel_exp_not:w`.)

`\exp_not:c` All these except `\exp_not:c` call the kernel-internal `_kernel_exp_not:w` namely `\tex_unexpanded:D`.

```

2671 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
2672 \cs_new:Npn \exp_not:o #1 { \_kernel_exp_not:w \exp_after:wN {#1} }
2673 \cs_new:Npn \exp_not:e #1
2674   { \_kernel_exp_not:w \tex_expanded:D { {#1} } }
2675 \cs_new:Npn \exp_not:f #1
2676   { \_kernel_exp_not:w \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2677 \cs_new:Npn \exp_not:V #1
2678   {
2679     \_kernel_exp_not:w \exp_after:wN
2680     { \exp:w \_exp_eval_register:N #1 }
2681   }
2682 \cs_new:Npn \exp_not:v #1
2683   {
2684     \_kernel_exp_not:w \exp_after:wN
2685     { \exp:w \_exp_eval_register:c {#1} }
2686   }

```

(End of definition for `\exp_not:c` and others. These functions are documented on page 40.)

43.5 Controlled expansion

`\exp:w` To trigger a sequence of “arbitrarily” many expansions we need a method to invoke TeX’s
`\exp_end:` expansion mechanism in such a way that (a) we are able to stop it in a controlled manner
`\exp_end_continue_f:w` and (b) the result of what triggered the expansion in the first place is null, i.e., that we
`\exp_end_continue_f:nw` do not get any unwanted side effects. There aren’t that many possibilities in TeX; in fact the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that `\tex_romannumeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number

turns out to be zero or negative. So we use that to start the expansion sequence: `\exp:w` is set equal to `\tex_romannumeral:D` in `l3basics`. To stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops `\tex_romannumeral:D`'s search for a number. Again, the definition of `\exp_end:` as the integer constant zero is in `l3basics`. (Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a “number” that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an `f`-type expansion we provide the alphabetic constant `'^^@` that also represents 0 but this time `TEX`'s syntax for a *(number)* continues searching for an optional space (and it continues expansion doing that) — see `TEXbook` page 269 for details.

```
2687 \group_begin:
2688   \tex_catcode:D '\^^@ = 13
2689   \cs_new_protected:Npn \exp_end_continue_f:w { '^^@ }
```

If the above definition ever appears outside its proper context the active character `^^@` will be executed so we turn this into an error. The test for existence covers the (unlikely) case that some other code has already defined `^^@`: this is true for example for `xmltex.tex`.

```
2690   \if_cs_exist:N ^^@
2691   \else:
2692     \cs_new:Npn ^^@
2693       { \msg_expandable_error:nn { kernel } { bad-exp-end-f } }
2694   \fi:
```

The same but grabbing an argument to remove spaces and braces.

```
2695   \cs_new:Npn \exp_end_continue_f:nw #1 { '^^@ #1 }
2696 \group_end:
```

(End of definition for `\exp:w` and others. These functions are documented on page 42.)

43.6 Defining function variants

```
2697 <@@=cs>
```

`\s__cs_mark` Internal scan marks. No `l3quark` yet, so do things by hand.

```
\s__cs_stop 2698 \cs_new_eq:NN \s__cs_mark \scan_stop:
2699 \cs_new_eq:NN \s__cs_stop \scan_stop:
```

(End of definition for `\s__cs_mark` and `\s__cs_stop`.)

`\q__cs_recursion_stop` Internal recursion quarks. No `l3quark` yet, so do things by hand.

```
2700 \cs_new:Npn \q__cs_recursion_stop { \q__cs_recursion_stop }
```

(End of definition for `\q__cs_recursion_stop`.)

`__cs_use_none_delimit_by_s_stop:w` Internal scan marks.

```
\__cs_use_i_delimit_by_s_stop:nw 2701 \cs_new:Npn \__cs_use_none_delimit_by_s_stop:w #1 \s__cs_stop { }
__cs_use_none_delimit_by_q_recursion_stop:w 2702 \cs_new:Npn \__cs_use_i_delimit_by_s_stop:nw #1 #2 \s__cs_stop {#1}
2703 \cs_new:Npn \__cs_use_none_delimit_by_q_recursion_stop:w
2704   #1 \q__cs_recursion_stop { }
```


(End of definition for `__cs_use_none_delimit_by_s_stop:w`, `__cs_use_i_delimit_by_s_stop:nw`, and `__cs_use_none_delimit_by_q_recursion_stop:w`.)

`\cs_generate_variant:Nn` #1 : Base form of a function; e.g., `\tl_set:Nn`
`\cs_generate_variant:cn` #2 : One or more variant argument specifiers; e.g., `{Nx,c,cx}`

After making sure that the base form exists, test whether it is protected or not and define `__cs_tmp:w` as either `\cs_new:Npe` or `\cs_new_protected:Npe`, which is then used to define all the variants (except those involving x-expansion, always protected). Split up the original base function only once, to grab its name and signature. Then we wish to iterate through the comma list of variant argument specifiers, which we first convert to a string: the reason is explained later.

```

2705 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
2706 {
2707   \__cs_generate_variant:N #1
2708   \use:e
2709   {
2710     \__cs_generate_variant:nnNN
2711     \cs_split_function:N #1
2712     \exp_not:N #1
2713     \tl_to_str:n {#2} ,
2714     \exp_not:N \scan_stop: ,
2715     \exp_not:N \q__cs_recursion_stop
2716   }
2717 }
2718 \cs_new_protected:Npn \cs_generate_variant:cn
2719 { \exp_args:Nc \cs_generate_variant:Nn }

```

(End of definition for `\cs_generate_variant:Nn`. This function is documented on page 33.)

`__cs_generate_variant:N`
`__cs_generate_variant:ww`
`__cs_generate_variant:wwNw`

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because non-expandable primitives cannot be \TeX conditionals.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and four others) can contain an arbitrary string after a leading `firstmark:.` Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long_`, `\protected_`, `\protected\long_`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\s__cs_mark` is taken as an argument of the `wwNw` auxiliary, and #3 is `\cs_new_protected:Npe`, otherwise it is `\cs_new:Npe`.

```

2720 \cs_new_protected:Npe \__cs_generate_variant:N #1
2721 {
2722   \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2723   \exp_not:N \exp_not:N #1 #1
2724   \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npe
2725   \exp_not:N \else:
2726   \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2727   \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2728   \s__cs_mark

```

```

2729     \s__cs_mark \cs_new_protected:Npe
2730     \tl_to_str:n { pr }
2731     \s__cs_mark \cs_new:Npe
2732     \s__cs_stop
2733 \exp_not:N \fi:
2734 }
2735 \exp_last_unbraced:NNNNo
2736 \cs_new_protected:Npn \__cs_generate_variant:ww
2737 #1 { \tl_to_str:n { ma } } #2 \s__cs_mark
2738 { \__cs_generate_variant:wwNw #1 }
2739 \exp_last_unbraced:NNNNo
2740 \cs_new_protected:Npn \__cs_generate_variant:wwNw
2741 #1 { \tl_to_str:n { pr } } #2 \s__cs_mark #3 #4 \s__cs_stop
2742 { \cs_set_eq:NN \__cs_tmp:w #3 }

```

(End of definition for `__cs_generate_variant:N`, `__cs_generate_variant:ww`, and `__cs_generate_variant:wwNw`.)

```

\__cs_generate_variant:nnNN #1 : Base name.
                             #2 : Base signature.
                             #3 : Boolean.
                             #4 : Base function.

```

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as `#4` for efficiency.

```

2743 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2744 {
2745   \if_meaning:w \c_false_bool #3
2746   \msg_error:nne { kernel } { missing-colon }
2747   { \token_to_str:c {#1} }
2748   \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2749   \fi:
2750   \__cs_generate_variant:Nnnw #4 {#1}{#2}
2751 }

```

(End of definition for `__cs_generate_variant:nnNN`.)

```

\__cs_generate_variant:Nnnw #1 : Base function.
                             #2 : Base name.
                             #3 : Base signature.
                             #4 : Beginning of variant signature.

```

First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature consisting of l letters and the last $k - l$ letters of the base signature (of length k). For example, for a base function `\prop_put:Nnn` which needs a cV variant form, we want the new signature to be cVn .

There are further subtleties:

- In `\cs_generate_variant:Nn \foo:nnTF {xxTF}`, we must define `\foo:xxTF` using `\exp_args:Nxx`, rather than a hypothetical `\exp_args:NxxTF`. Thus, we wish to trim a common trailing part from the base signature and the variant signature.
- In `\cs_generate_variant:Nn \foo:on {ox}`, the function `\foo:ox` must be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double `o` expansion.

- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` must trigger an error, because we do not have a means to replace o-expansion by x-expansion. More generally, we can only convert N to c, or convert n to V, v, o, e, f, or x.

All this boils down to a few rules. Only n and N-type arguments can be replaced by `\cs_generate_variant:Nn`. Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to n except for N and p-type arguments. A common trailing part is ignored.

We compare the base and variant signatures one character at a time within e-expansion. The result is given to `__cs_generate_variant:wwNN` (defined later) in the form `\processed variant signature__cs_mark errors__cs_stop base function new function`. If all went well, `errors` is empty; otherwise, it is a kernel error message and some clean-up code.

Note the space after #3 and after the following brace group. Those are ignored by \TeX when fetching the last argument for `__cs_generate_variant_loop:nNwN`, but can be used as a delimiter for `__cs_generate_variant_loop_end:nwwwNNnn`.

```

2752 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
2753 {
2754   \if_meaning:w \scan_stop: #4
2755   \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2756   \fi:
2757   \use:e
2758   {
2759     \exp_not:N \__cs_generate_variant:wwNN
2760     \__cs_generate_variant_loop:nNwN { }
2761     #4
2762     \__cs_generate_variant_loop_end:nwwwNNnn
2763     \s__cs_mark
2764     #3 ~
2765     { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
2766     { }
2767     \s__cs_stop
2768     \exp_not:N #1 {#2} {#4}
2769   }
2770   \__cs_generate_variant:Nnnw #1 {#2} {#3}
2771 }

```

(End of definition for `__cs_generate_variant:Nnnw`.)

<code>__cs_generate_variant_loop:nNwN</code>	#1 : Last few consecutive letters common between the base and variant (more precisely,
<code>__cs_generate_variant_loop_base:N</code>	<code>__cs_generate_variant_same:N</code> <i>letter</i> for each letter).
<code>__cs_generate_variant_loop_same:w</code>	#2 : Next variant letter.
<code>__cs_generate_variant_loop_end:nwwwNNnn</code>	#3 : Remainder of variant form.
<code>__cs_generate_variant_loop_long:wNNnn</code>	#4 : Next base letter.

The first argument is populated by `__cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed if the base is N and the variant is c, or when the base is n and the variant is V, v, o, e, f, or x. Otherwise, call `__cs_generate_variant_loop_invalid:NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second

argument of `__cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed `n` or `N`, leave in the input stream whatever argument `#1` was collected, and the next variant letter `#2`, then loop by calling `__cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, `#2` is `__cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the `<base name>` as `#7`, the `<variant signature>` `#8`, the `<next base letter>` `#1` and the part `#3` of the base signature that wasn't read yet; and combines those into the `<new function>` to be defined.
- If the end of the base form is encountered first, `#4` is `~{} \fi:` which ends the conditional (with an empty expansion), followed by `__cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `__cs_generate_variant:wwNN`.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is not the right one (`n` or `N` to support the variant). In that case too an error is placed as the second argument of `__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, `#2` is as described in the first point, and `#4` as described in the second point above. The `__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in `#4` as its first argument: this empty brace group produces the correct signature for the full variant.

```

2772 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \s__cs_mark #4
2773 {
2774   \if:w #2 #4
2775     \exp_after:wN \__cs_generate_variant_loop_same:w
2776   \else:
2777     \if:w #4 \__cs_generate_variant_loop_base:N #2 \else:
2778       \if:w 0
2779         \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
2780         \if:w \scan_stop: \__cs_generate_variant_loop_base:N #2 1 \fi:
2781         0
2782         \__cs_generate_variant_loop_special:NNwNNnn #4#2
2783       \else:
2784         \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
2785       \fi:
2786     \fi:
2787   \fi:
2788   #1
2789   \prg_do_nothing:
2790   #2
2791   \__cs_generate_variant_loop:nNwN { } #3 \s__cs_mark
2792 }
2793 \cs_new:Npn \__cs_generate_variant_loop_base:N #1
2794 {
2795   \if:w c #1 N \else:
2796     \if:w o #1 n \else:
2797       \if:w V #1 n \else:
2798         \if:w v #1 n \else:

```

```

2799         \if:w f #1 n \else:
2800         \if:w e #1 n \else:
2801         \if:w x #1 n \else:
2802         \if:w n #1 n \else:
2803         \if:w N #1 N \else:
2804         \scan_stop:
2805         \fi:
2806         \fi:
2807         \fi:
2808         \fi:
2809         \fi:
2810         \fi:
2811         \fi:
2812         \fi:
2813     \fi:
2814 }
2815 \cs_new:Npn \__cs_generate_variant_loop_same:w
2816     #1 \prg_do_nothing: #2#3#4
2817     { #3 { #1 \__cs_generate_variant_same:N #2 } }
2818 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
2819     #1#2 \s__cs_mark #3 ~ #4 \s__cs_stop #5#6#7#8
2820     {
2821     \scan_stop: \scan_stop: \fi:
2822     \s__cs_mark \s__cs_stop
2823     \exp_not:N #6
2824     \exp_not:c { #7 : #8 #1 #3 }
2825     }
2826 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \s__cs_stop #2#3#4#5
2827     {
2828     \exp_not:n
2829     {
2830     \s__cs_mark
2831     \msg_error:nnee { kernel } { variant-too-long }
2832     {#5} { \token_to_str:N #3 }
2833     \use_none:nnn
2834     \s__cs_stop
2835     #3
2836     #3
2837     }
2838     }
2839 \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
2840     #1#2 \fi: \fi: \fi: #3 \s__cs_stop #4#5#6#7
2841     {
2842     \fi: \fi: \fi:
2843     \exp_not:n
2844     {
2845     \s__cs_mark
2846     \msg_error:nneeee { kernel } { invalid-variant }
2847     {#7} { \token_to_str:N #5 } {#1} {#2}
2848     \use_none:nnn
2849     \s__cs_stop
2850     #5
2851     #5
2852     }

```

```

2853 }
2854 \cs_new:Npn \__cs_generate_variant_loop_special:NNwNNnn
2855 #1#2#3 \s__cs_stop #4#5#6#7
2856 {
2857   #3 \s__cs_stop #4 #5 {#6} {#7}
2858   \exp_not:n
2859   {
2860     \msg_error:nneeee
2861     { kernel } { deprecated-variant }
2862     {#7} { \token_to_str:N #5 } {#1} {#2}
2863   }
2864 }

```

(End of definition for `__cs_generate_variant_loop:nNwN` and others.)

`__cs_generate_variant_same:N` When the base and variant letters are identical, don't do any expansion. For most argument types, we can use the n-type no-expansion, but the N and p types require a slightly different behaviour with respect to braces. For V-type this function could output N to avoid adding useless braces but that is not a problem.

```

2865 \cs_new:Npn \__cs_generate_variant_same:N #1
2866 {
2867   \if:w N #1 #1 \else:
2868     \if:w p #1 #1 \else:
2869       \token_to_str:N n
2870     \if:w n #1 \else:
2871       \__cs_generate_variant_loop_special:NNwNNnn #1#1
2872   \fi:
2873   \fi:
2874   \fi:
2875 }

```

(End of definition for `__cs_generate_variant_same:N`.)

`__cs_generate_variant:wwNN` If the variant form has already been defined, log its existence (provided `log-functions` is active). Otherwise, make sure that the `\exp_args:N #3` form is defined, and if it contains x, change `__cs_tmp:w` locally to `\cs_new_protected:Npe`. Then define the variant by combining the `\exp_args:N #3` variant and the base function.

```

2876 \cs_new_protected:Npn \__cs_generate_variant:wwNN
2877 #1 \s__cs_mark #2 \s__cs_stop #3#4
2878 {
2879   #2
2880   \cs_if_free:NT #4
2881   {
2882     \group_begin:
2883     \__cs_generate_internal_variant:n {#1}
2884     \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
2885     \group_end:
2886   }
2887 }

```

(End of definition for `__cs_generate_variant:wwNN`.)

`__cs_generate_internal_variant:n`
`__cs_generate_internal_variant_loop:n` First test for the presence of x (this is where working with strings makes our lives easier), as the result should be protected, and the next variant to be defined using that internal variant should be protected (done by setting `__cs_tmp:w`). Then

call `__cs_generate_internal_variant:NNn` with arguments `\cs_new_protected:cpn` `\use:x` (for protected) or `\cs_new:cpn` `\tex_expanded:D` (expandable) and the signature. If `p` appears in the signature, or if the function to be defined is expandable and the primitive `\expanded` is not available, or if there are more than 8 arguments, call some fall-back code that just puts the appropriate `\::` commands. Otherwise, call `__cs_generate_internal_one_go:NNn` to construct the `\exp_args:N...` function as a macro taking up to 9 arguments and expanding them using `\use:x` or `\tex_expanded:D`.

```

2888 \cs_new_protected:Npe \__cs_generate_internal_variant:n #1
2889 {
2890   \exp_not:N \__cs_generate_internal_variant:wNnNwn
2891   #1 \s__cs_mark
2892   { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npe }
2893   \cs_new_protected:cpn
2894   \use:x
2895   \token_to_str:N x \s__cs_mark
2896   { }
2897   \cs_new:cpn
2898   \exp_not:N \tex_expanded:D
2899   \s__cs_stop
2900   {#1}
2901 }
2902 \exp_last_unbraced:NNNNo
2903 \cs_new_protected:Npn \__cs_generate_internal_variant:wNnNwn #1
2904 { \token_to_str:N x } #2 \s__cs_mark #3#4#5#6 \s__cs_stop #7
2905 {
2906   #3
2907   \cs_if_free:cT { exp_args:N #7 }
2908   { \__cs_generate_internal_variant:NNn #4 #5 {#7} }
2909 }
2910 \cs_set_protected:Npn \__cs_tmp:w #1
2911 {
2912   \cs_new_protected:Npn \__cs_generate_internal_variant:NNn ##1##2##3
2913   {
2914     \if_catcode:w X \use_none:nnnnnnnn ##3
2915     \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
2916     \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
2917     \prg_do_nothing: \prg_do_nothing: X
2918     \exp_after:wN \__cs_generate_internal_test:Nw \exp_after:wN ##2
2919     \else:
2920     \exp_after:wN \__cs_generate_internal_test_aux:w \exp_after:wN #1
2921     \fi:
2922     ##3
2923     \s__cs_mark
2924     {
2925       \use:e
2926       {
2927         ##1 { exp_args:N ##3 }
2928         { \__cs_generate_internal_variant_loop:n ##3 { : \use_i:nn } }
2929       }
2930     }
2931     #1
2932     \s__cs_mark
2933     { \exp_not:n { \__cs_generate_internal_one_go:NNn ##1 ##2 {##3} } }

```

```

2934     \s__cs_stop
2935   }
2936   \cs_new_protected:Npn \__cs_generate_internal_test_aux:w
2937     ##1 #1 ##2 \s__cs_mark ##3 ##4 \s__cs_stop {##3}
2938   \cs_new_eq:NN \__cs_generate_internal_test:Nw
2939     \__cs_generate_internal_test_aux:w
2940   }
2941 \exp_args:No \__cs_tmp:w { \token_to_str:N p }
2942 \cs_new_protected:Npn \__cs_generate_internal_one_go:NNn #1#2#3
2943   {
2944     \__cs_generate_internal_loop:nwnnw
2945     { \exp_not:N ##1 } 1 . { } { }
2946     #3 { ? \__cs_generate_internal_end:w } X ;
2947     23456789 { ? \__cs_generate_internal_long:w } ;
2948     #1 #2 {#3}
2949   }
2950 \cs_new_protected:Npn \__cs_generate_internal_loop:nwnnw #1#2 . #3#4#5#6 ; #7
2951   {
2952     \use_none:n #5
2953     \use_none:n #7
2954     \cs_if_exist_use:cF { __cs_generate_internal_#5:NN }
2955     { \__cs_generate_internal_other:NN }
2956     #5 #7
2957     #7 .
2958     { #3 #1 } { #4 ## #2 }
2959     #6 ;
2960   }
2961 \cs_new_protected:Npn \__cs_generate_internal_N:NN #1#2
2962   { \__cs_generate_internal_loop:nwnnw { \exp_not:N ###2 } }
2963 \cs_new_protected:Npn \__cs_generate_internal_c:NN #1#2
2964   { \exp_args:No \__cs_generate_internal_loop:nwnnw { \exp_not:c {###2} } }
2965 \cs_new_protected:Npn \__cs_generate_internal_n:NN #1#2
2966   { \__cs_generate_internal_loop:nwnnw { { \exp_not:n {###2} } } }
2967 \cs_new_protected:Npn \__cs_generate_internal_x:NN #1#2
2968   { \__cs_generate_internal_loop:nwnnw { {###2} } }
2969 \cs_new_protected:Npn \__cs_generate_internal_other:NN #1#2
2970   {
2971     \exp_args:No \__cs_generate_internal_loop:nwnnw
2972     {
2973       \exp_after:wN
2974       {
2975         \exp:w \exp_args:NNc \exp_after:wN \exp_end:
2976         { exp_not:#1 } {###2}
2977       }
2978     }
2979   }
2980 \cs_new_protected:Npn \__cs_generate_internal_end:w #1 . #2#3#4 ; #5 ; #6#7#8
2981   { #6 { exp_args:N #8 } #3 { #7 {#2} } }
2982 \cs_new_protected:Npn \__cs_generate_internal_long:w #1 N #2#3 . #4#5#6#
2983   {
2984     \exp_args:Nx \__cs_generate_internal_long:nnnNNn
2985     { \__cs_generate_internal_variant_loop:n #2 #6 { : \use_i:nn } }
2986     {#4} {#5}
2987   }

```



```

2988 \cs_new:Npn \__cs_generate_internal_long:nnnNn #1#2#3#4 ; ; #5#6#7
2989 { #5 { exp_args:N #7 } #3 { #6 { \exp_not:n {#1} {#2} } } }

```

This command grabs char by char outputting \::#1 (not expanded further). We avoid tests by putting a trailing : \use_i:nn, which leaves \cs_end: and removes the looping macro. The colon is in fact also turned into \::: so that the required structure for \exp_args:N... commands is correctly terminated.

```

2990 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
2991 {
2992   \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
2993   \__cs_generate_internal_variant_loop:n
2994 }

```

(End of definition for __cs_generate_internal_variant:n and __cs_generate_internal_variant_loop:n.)

```
\prg_generate_conditional_variant:Nnn
```

```

\__cs_generate_variant:nnNnn
\__cs_generate_variant:w
\__cs_generate_variant:n
\__cs_generate_variant_p_form:nnn
\__cs_generate_variant_T_form:nnn
\__cs_generate_variant_F_form:nnn
\__cs_generate_variant_TF_form:nnn

```

```

2995 \cs_new_protected:Npn \prg_generate_conditional_variant:Nnn #1
2996 {
2997   \use:e
2998   {
2999     \__cs_generate_variant:nnNnn
3000     \cs_split_function:N #1
3001   }
3002 }
3003 \cs_new_protected:Npn \__cs_generate_variant:nnNnn #1#2#3#4#5
3004 {
3005   \if_meaning:w \c_false_bool #3
3006   \msg_error:nne { kernel } { missing-colon }
3007   { \token_to_str:c {#1} }
3008   \__cs_use_i_delimit_by_s_stop:nw
3009   \fi:
3010   \exp_after:wN \__cs_generate_variant:w
3011   \tl_to_str:n {#5} , \scan_stop: , \q__cs_recursion_stop
3012   \__cs_use_none_delimit_by_s_stop:w \s__cs_mark {#1} {#2} {#4} \s__cs_stop
3013 }
3014 \cs_new_protected:Npn \__cs_generate_variant:w
3015 #1 , #2 \s__cs_mark #3#4#5
3016 {
3017   \if_meaning:w \scan_stop: #1 \scan_stop:
3018   \if_meaning:w \q__cs_nil #1 \q__cs_nil
3019   \use_i:nnn
3020   \fi:
3021   \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
3022   \else:
3023   \cs_if_exist_use:cTF { __cs_generate_variant_#1_form:nnn }
3024   { {#3} {#4} {#5} }
3025   {
3026     \msg_error:nnee
3027     { kernel } { conditional-form-unknown }
3028     {#1} { \token_to_str:c { #3 : #4 } }
3029   }
3030   \fi:
3031   \__cs_generate_variant:w #2 \s__cs_mark {#3} {#4} {#5}
3032 }

```

```

3033 \cs_new_protected:Npn \__cs_generate_variant_p_form:nnn #1#2
3034   { \cs_generate_variant:cn { #1_p : #2 } }
3035 \cs_new_protected:Npn \__cs_generate_variant_T_form:nnn #1#2
3036   { \cs_generate_variant:cn { #1 : #2 T } }
3037 \cs_new_protected:Npn \__cs_generate_variant_F_form:nnn #1#2
3038   { \cs_generate_variant:cn { #1 : #2 F } }
3039 \cs_new_protected:Npn \__cs_generate_variant_TF_form:nnn #1#2
3040   { \cs_generate_variant:cn { #1 : #2 TF } }

```

(End of definition for `\prg_generate_conditional_variant:Nnn` and others. This function is documented on page 65.)

`\exp_args_generate:n`

This function is not used in the kernel hence we can use functions that are defined in later modules. It also does not need to be fast so use inline mappings. For each requested variant we check that there are no characters besides `NnpcofVvx`, in particular that there are no spaces. Then we just call the internal function.

```

3041 \cs_new_protected:Npn \exp_args_generate:n #1
3042   {
3043     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
3044     {
3045       \str_map_inline:nn {##1}
3046       {
3047         \str_if_in:nnF { NnpcofeVvx } {####1}
3048         {
3049           \msg_error:nnnn { kernel } { invalid-exp-args }
3050             {####1} {##1}
3051           \str_map_break:n { \use_none:nn }
3052         }
3053       }
3054     }
3055     \__cs_generate_internal_variant:n {##1}
3056   }

```

(End of definition for `\exp_args_generate:n`. This function is documented on page 34.)

43.7 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions actually take no arguments themselves.

`\exp_args:Nnc`
`\exp_args:Nno`
`\exp_args:NnV`
`\exp_args:Nnv`
`\exp_args:Nne`
`\exp_args:Nnf`
`\exp_args:Noc`
`\exp_args:Noo`
`\exp_args:Nof`
`\exp_args:NVo`
`\exp_args:Nfo`
`\exp_args:Nff`
`\exp_args:Nee`
`\exp_args:NNx`
`\exp_args:Ncx`
`\exp_args:Nnx`
`\exp_args:Nox`
`\exp_args:Nxo`
`\exp_args:Nxx`

Here are the actual function definitions, using the helper functions above. The group is used because `__cs_generate_internal_variant:n` redefines `__cs_tmp:w` locally.

```

3057 \cs_set_protected:Npn \__cs_tmp:w #1
3058   {
3059     \group_begin:
3060     \exp_args:No \__cs_generate_internal_variant:n
3061       { \tl_to_str:n {#1} }
3062     \group_end:
3063   }
3064 \__cs_tmp:w { nc }
3065 \__cs_tmp:w { no }
3066 \__cs_tmp:w { nV }
3067 \__cs_tmp:w { nv }

```

```

3068 \__cs_tmp:w { ne }
3069 \__cs_tmp:w { nf }
3070 \__cs_tmp:w { oc }
3071 \__cs_tmp:w { oo }
3072 \__cs_tmp:w { of }
3073 \__cs_tmp:w { Vo }
3074 \__cs_tmp:w { fo }
3075 \__cs_tmp:w { ff }
3076 \__cs_tmp:w { ee }
3077 \__cs_tmp:w { Nx }
3078 \__cs_tmp:w { cx }
3079 \__cs_tmp:w { nx }
3080 \__cs_tmp:w { ox }
3081 \__cs_tmp:w { xo }
3082 \__cs_tmp:w { xx }

```

(End of definition for \exp_args:Nnc and others. These functions are documented on page 37.)

```

\exp_args:NNcf
\exp_args:NNno
\exp_args:NNnV
\exp_args:NNoo
\exp_args:NNVV
\exp_args:Ncno
\exp_args:NcnV
\exp_args:Ncoo
\exp_args:NcVV
\exp_args:Nnnc
\exp_args:Nnno
\exp_args:Nnnf
\exp_args:Nnff
\exp_args:Nooo
\exp_args:Noof
\exp_args:Nffo
\exp_args:Neee
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox

```

```

3083 \__cs_tmp:w { Ncf }
3084 \__cs_tmp:w { Nno }
3085 \__cs_tmp:w { NnV }
3086 \__cs_tmp:w { Noo }
3087 \__cs_tmp:w { NVV }
3088 \__cs_tmp:w { cno }
3089 \__cs_tmp:w { cnV }
3090 \__cs_tmp:w { coo }
3091 \__cs_tmp:w { cVV }
3092 \__cs_tmp:w { nnc }
3093 \__cs_tmp:w { nno }
3094 \__cs_tmp:w { nnf }
3095 \__cs_tmp:w { nff }
3096 \__cs_tmp:w { ooo }
3097 \__cs_tmp:w { oof }
3098 \__cs_tmp:w { ffo }
3099 \__cs_tmp:w { eee }
3100 \__cs_tmp:w { NNx }
3101 \__cs_tmp:w { Nnx }
3102 \__cs_tmp:w { Nox }
3103 \__cs_tmp:w { nnx }
3104 \__cs_tmp:w { nox }
3105 \__cs_tmp:w { ccx }
3106 \__cs_tmp:w { cnx }
3107 \__cs_tmp:w { oox }

```

(End of definition for \exp_args:NNcf and others. These functions are documented on page 38.)

43.8 Held-over variant generation

```

\cs_generate_from_arg_count:NNno
\cs_replacement_spec:c

```

A couple of variants that are from early functions.

```

3108 \cs_generate_variant:Nn \cs_generate_from_arg_count:NNnn { NNno }
3109 \cs_generate_variant:Nn \cs_replacement_spec:N { c }

```

(End of definition for `\cs_generate_from_arg_count:NNnn` and `\cs_replacement_spec:N`. These functions are documented on page 19.)

3110 `\endpackage`

Chapter 44

l3sort implementation

```
3111 (*package)
```

```
3112 (@@=sort)
```

44.1 Variables

`\g__sort_internal_seq` Sorting happens in a group; the result is stored in those global variables before being copied outside the group to the proper places. For `seq` and `tl` this is more efficient than using `\use:e` (or some `\exp_args:NMNe`) to smuggle the definition outside the group since TeX does not need to re-read tokens. For `clist` we don't gain anything since the result is converted from `seq` to `clist` anyways.

```
3113 \seq_new:N \g__sort_internal_seq
```

```
3114 \tl_new:N \g__sort_internal_tl
```

(End of definition for `\g__sort_internal_seq` and `\g__sort_internal_tl`.)

`\l__sort_length_int` The sequence has `\l__sort_length_int` items and is stored from `\l__sort_min_int` to `\l__sort_top_int - 1`. While reading the sequence in memory, we check that `\l__sort_top_int` remains at most `\l__sort_max_int`, precomputed by `__sort_compute_range:`. That bound is such that the merge sort only uses `\toks` registers less than `\l__sort_true_max_int`, namely those that have not been allocated for use in other code: the user's comparison code could alter these.

```
3115 \int_new:N \l__sort_length_int
```

```
3116 \int_new:N \l__sort_min_int
```

```
3117 \int_new:N \l__sort_top_int
```

```
3118 \int_new:N \l__sort_max_int
```

```
3119 \int_new:N \l__sort_true_max_int
```

(End of definition for `\l__sort_length_int` and others.)

`\l__sort_block_int` Merge sort is done in several passes. In each pass, blocks of size `\l__sort_block_int` are merged in pairs. The block size starts at 1, and, for a length in the range $[2^k + 1, 2^{k+1}]$, reaches 2^k in the last pass.

```
3120 \int_new:N \l__sort_block_int
```

(End of definition for `\l__sort_block_int`.)

`\l__sort_begin_int` When merging two blocks, `\l__sort_begin_int` marks the lowest index in the two blocks, and `\l__sort_end_int` marks the highest index, plus 1.

```
3121 \int_new:N \l__sort_begin_int
3122 \int_new:N \l__sort_end_int
```

(End of definition for `\l__sort_begin_int` and `\l__sort_end_int`.)

`\l__sort_A_int` When merging two blocks (whose end-points are `beg` and `end`), `A` starts from the high end of the low block, and decreases until reaching `beg`. The index `B` starts from the top of the range and marks the register in which a sorted item should be put. Finally, `C` points to the copy of the high block in the interval of registers starting at `\l__sort_length_int`, upwards. `C` starts from the upper limit of that range.

```
3123 \int_new:N \l__sort_A_int
3124 \int_new:N \l__sort_B_int
3125 \int_new:N \l__sort_C_int
```

(End of definition for `\l__sort_A_int`, `\l__sort_B_int`, and `\l__sort_C_int`.)

`\s__sort_mark` Internal scan marks.

```
\s__sort_stop 3126 \scan_new:N \s__sort_mark
3127 \scan_new:N \s__sort_stop
```

(End of definition for `\s__sort_mark` and `\s__sort_stop`.)

44.2 Finding available `\toks` registers

`__sort_shrink_range:` After `__sort_compute_range:` (defined below) determines that `\toks` registers between `\l__sort_min_int` (included) and `\l__sort_true_max_int` (excluded) have not yet been assigned, `__sort_shrink_range:` computes `\l__sort_max_int` to reflect the need for a buffer when merging blocks in the merge sort. Given $2^n \leq A \leq 2^n + 2^{n-1}$ registers we can sort $\lfloor A/2 \rfloor + 2^{n-2}$ items while if we have $2^n + 2^{n-1} \leq A \leq 2^{n+1}$ registers we can sort $A - 2^{n-1}$ items. We first find out a power 2^n such that $2^n \leq A \leq 2^{n+1}$ by repeatedly halving `\l__sort_block_int`, starting at 2^{15} or 2^{14} namely half the total number of registers, then we use the formulas and set `\l__sort_max_int`.

```
3128 \cs_new_protected:Npn \__sort_shrink_range:
3129 {
3130   \int_set:Nn \l__sort_A_int
3131     { \l__sort_true_max_int - \l__sort_min_int + 1 }
3132   \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
3133   \__sort_shrink_range_loop:
3134   \int_set:Nn \l__sort_max_int
3135     {
3136     \int_compare:nNnTF
3137       { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
3138       {
3139         \l__sort_min_int
3140         + ( \l__sort_A_int - 1 ) / 2
3141         + \l__sort_block_int / 4
3142         - 1
3143       }
3144       { \l__sort_true_max_int - \l__sort_block_int / 2 }
3145     }
```

```

3146 }
3147 \cs_new_protected:Npn \__sort_shrink_range_loop:
3148 {
3149   \if_int_compare:w \l__sort_A_int < \l__sort_block_int
3150     \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
3151     \exp_after:wN \__sort_shrink_range_loop:
3152   \fi:
3153 }

```

(End of definition for `__sort_shrink_range:` and `__sort_shrink_range_loop:.`)

`__sort_compute_range:` First find out what `\toks` have not yet been assigned. There are many cases. In L^AT_EX 2_ε with no package, available `\toks` range from `\count15 + 1` to `\c_max_register_int` included (this was not altered despite the 2015 changes). When `\loctoks` is defined, namely in plain (e)T_EX, or when the package `etex` is loaded in L^AT_EX 2_ε, redefine `__sort_compute_range:` to use the range `\count265` to `\count275 - 1`. The `elocalloc` package also defines `\loctoks` but uses yet another number for the upper bound, namely `\e@alloc@top` (minus one). We must check for `\loctoks` every time a sorting function is called, as `etex` or `elocalloc` could be loaded.

In ConT_EXt MkIV the range is from `\c_syst_last_allocated_toks+1` to `\c_max_register_int`, and in MkII it is from `\lastallocatedtoks+1` to `\c_max_register_int`. In all these cases, call `__sort_shrink_range:.`

```

3154 \cs_new_protected:Npn \__sort_compute_range:
3155 {
3156   \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
3157   \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
3158   \__sort_shrink_range:
3159   \if_meaning:w \loctoks \tex_undefined:D \else:
3160     \if_meaning:w \loctoks \scan_stop: \else:
3161       \__sort_redefine_compute_range:
3162       \__sort_compute_range:
3163     \fi:
3164   \fi:
3165 }
3166 \cs_new_protected:Npn \__sort_redefine_compute_range:
3167 {
3168   \cs_if_exist:cTF { ver@elocalloc.sty }
3169   {
3170     \cs_gset_protected:Npn \__sort_compute_range:
3171     {
3172       \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
3173       \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
3174       \__sort_shrink_range:
3175     }
3176   }
3177   {
3178     \cs_gset_protected:Npn \__sort_compute_range:
3179     {
3180       \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
3181       \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
3182       \__sort_shrink_range:
3183     }
3184   }

```

```

3185 }
3186 \cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
3187 \tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
3188 {
3189   \cs_if_exist:NT #1
3190   {
3191     \cs_gset_protected:Npn \__sort_compute_range:
3192     {
3193       \int_set:Nn \l__sort_min_int { #1 + 1 }
3194       \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
3195       \__sort_shrink_range:
3196     }
3197   }
3198 }

```

(End of definition for `__sort_compute_range:`, `__sort_redefine_compute_range:`, and `\c__sort_max_length_int`.)

44.3 Protected user commands

`__sort_main:NNNn` Sorting happens in three steps. First store items in `\toks` registers ranging from `\l__sort_min_int` to `\l__sort_top_int - 1`, while checking that the list is not too long. If we reach the maximum length, that's an error; exit the group. Secondly, sort the array of `\toks` registers, using the user-defined sorting function: `__sort_level:` calls `__sort_compare:nn` as needed. Finally, unpack the `\toks` registers (now sorted) into the target `tl`, or into `\g__sort_internal_seq` for `seq` and `clist`. This is done by `__sort_seq:NNNNn` and `__sort_tl:NNn`.

```

3199 \cs_new_protected:Npn \__sort_main:NNNn #1#2#3#4
3200 {
3201   \__sort_disable_toksdef:
3202   \__sort_compute_range:
3203   \int_set_eq:NN \l__sort_top_int \l__sort_min_int
3204   #1 #3
3205   {
3206     \if_int_compare:w \l__sort_top_int = \l__sort_max_int
3207     \__sort_too_long_error:NNw #2 #3
3208     \fi:
3209     \tex_toks:D \l__sort_top_int {##1}
3210     \int_incr:N \l__sort_top_int
3211   }
3212   \int_set:Nn \l__sort_length_int
3213   { \l__sort_top_int - \l__sort_min_int }
3214   \cs_set:Npn \__sort_compare:nn ##1 ##2 {#4}
3215   \int_set:Nn \l__sort_block_int { 1 }
3216   \__sort_level:
3217 }

```

(End of definition for `__sort_main:NNNn`.)

`\tl_sort:Nn` Call the main sorting function then unpack `\toks` registers outside the group into the target token list. The unpacking is done by `__sort_tl_toks:w`; registers are numbered from `\l__sort_min_int` to `\l__sort_top_int - 1`. For expansion behaviour we need

```

\tl_gsort:Nn
\tl_gsort:cn
__sort_tl:NNn
__sort_tl_toks:w

```


a couple of primitives. The `\tl_gclear:N` reduces memory usage. The `\prg_break_point:` is used by `__sort_main:NNNn` when the list is too long.

```

3218 \cs_new_protected:Npn \tl_sort:Nn { \__sort_tl:NNn \tl_set_eq:NN }
3219 \cs_generate_variant:Nn \tl_sort:Nn { c }
3220 \cs_new_protected:Npn \tl_gsort:Nn { \__sort_tl:NNn \tl_gset_eq:NN }
3221 \cs_generate_variant:Nn \tl_gsort:Nn { c }
3222 \cs_new_protected:Npn \__sort_tl:NNn #1#2#3
3223 {
3224   \group_begin:
3225     \__sort_main:NNNn \tl_map_inline:Nn \tl_map_break:n #2 {#3}
3226     \__kernel_tl_gset:Nx \g__sort_internal_tl
3227     { \__sort_tl_toks:w \l__sort_min_int ; }
3228   \group_end:
3229   #1 #2 \g__sort_internal_tl
3230   \tl_gclear:N \g__sort_internal_tl
3231   \prg_break_point:
3232 }
3233 \cs_new:Npn \__sort_tl_toks:w #1 ;
3234 {
3235   \if_int_compare:w #1 < \l__sort_top_int
3236     { \tex_the:D \tex_toks:D #1 }
3237     \exp_after:wN \__sort_tl_toks:w
3238     \int_value:w \int_eval:n { #1 + 1 } \exp_after:wN ;
3239   \fi:
3240 }

```

(End of definition for `\tl_sort:Nn` and others. These functions are documented on page 123.)

```

\seq_sort:Nn Use the same general framework for seq and clist. Apply the general sorting code, then
\seq_sort:cn unpack \toks into \g__sort_internal_seq. Outside the group copy or convert (for
\seq_gsort:Nn \clist) the data to the target variable. The \seq_gclear:N reduces memory usage. The
\seq_gsort:cn \prg_break_point: is used by \__sort_main:NNNn when the list is too long.
\clist_sort:Nn
\clist_sort:cn
\clist_gsort:Nn
\clist_gsort:cn
\__sort_seq:NNNNn
3241 \cs_new_protected:Npn \seq_sort:Nn
3242 { \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_set_eq:NN }
\clist_sort:Nn
3243 \cs_generate_variant:Nn \seq_sort:Nn { c }
\clist_gsort:Nn
3244 \cs_new_protected:Npn \seq_gsort:Nn
3245 { \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_gset_eq:NN }
\__sort_seq:NNNNn
3246 \cs_generate_variant:Nn \seq_gsort:Nn { c }
3247 \cs_new_protected:Npn \clist_sort:Nn
3248 {
3249   \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
3250   \clist_set_from_seq:NN
3251 }
3252 \cs_generate_variant:Nn \clist_sort:Nn { c }
3253 \cs_new_protected:Npn \clist_gsort:Nn
3254 {
3255   \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
3256   \clist_gset_from_seq:NN
3257 }
3258 \cs_generate_variant:Nn \clist_gsort:Nn { c }
3259 \cs_new_protected:Npn \__sort_seq:NNNNn #1#2#3#4#5
3260 {
3261   \group_begin:
3262     \__sort_main:NNNn #1 #2 #4 {#5}

```

```

3263     \seq_gclear:N \g__sort_internal_seq
3264     \int_step_inline:nnn
3265       \l__sort_min_int { \l__sort_top_int - 1 }
3266       {
3267         \seq_gput_right:Ne \g__sort_internal_seq
3268         { \tex_the:D \tex_toks:D ##1 }
3269       }
3270     \group_end:
3271     #3 #4 \g__sort_internal_seq
3272     \seq_gclear:N \g__sort_internal_seq
3273     \prg_break_point:
3274   }

```

(End of definition for `\seq_sort:Nn` and others. These functions are documented on page 157.)

44.4 Merge sort

`__sort_level:` This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```

3275 \cs_new_protected:Npn \__sort_level:
3276 {
3277   \if_int_compare:w \l__sort_block_int < \l__sort_length_int
3278     \l__sort_end_int \l__sort_min_int
3279     \__sort_merge_blocks:
3280     \tex_advance:D \l__sort_block_int \l__sort_block_int
3281     \exp_after:wN \__sort_level:
3282   \fi:
3283 }

```

(End of definition for `__sort_level:.`)

`__sort_merge_blocks:` This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in *A*, which indexes the first block starting from the top end. Then locate the end-point (maximum) of the second block: shift `end` upwards by one more block, but keeping it \leq `top`. Copy this upper block of `\toks` registers in registers above `length`, indexed by *C*: this is covered by `__sort_copy_block:.` Once this is done we are ready to do the actual merger using `__sort_merge_blocks_aux:.`, after shifting *A*, *B* and *C* so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```

3284 \cs_new_protected:Npn \__sort_merge_blocks:
3285 {
3286   \l__sort_begin_int \l__sort_end_int
3287   \tex_advance:D \l__sort_end_int \l__sort_block_int
3288   \if_int_compare:w \l__sort_end_int < \l__sort_top_int
3289     \l__sort_A_int \l__sort_end_int
3290     \tex_advance:D \l__sort_end_int \l__sort_block_int
3291     \if_int_compare:w \l__sort_end_int > \l__sort_top_int

```

```

3292     \l__sort_end_int \l__sort_top_int
3293     \fi:
3294     \l__sort_B_int \l__sort_A_int
3295     \l__sort_C_int \l__sort_top_int
3296     \__sort_copy_block:
3297     \int_decr:N \l__sort_A_int
3298     \int_decr:N \l__sort_B_int
3299     \int_decr:N \l__sort_C_int
3300     \exp_after:wN \__sort_merge_blocks_aux:
3301     \exp_after:wN \__sort_merge_blocks:
3302     \fi:
3303 }

```

(End of definition for __sort_merge_blocks:.)

`__sort_copy_block:` We wish to store a copy of the “upper” block of `\toks` registers, ranging between the initial value of `\l__sort_B_int` (included) and `\l__sort_end_int` (excluded) into a new range starting at the initial value of `\l__sort_C_int`, namely `\l__sort_top_int`.

```

3304 \cs_new_protected:Npn \__sort_copy_block:
3305 {
3306     \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
3307     \int_incr:N \l__sort_C_int
3308     \int_incr:N \l__sort_B_int
3309     \if_int_compare:w \l__sort_B_int = \l__sort_end_int
3310     \use_i:nn
3311     \fi:
3312     \__sort_copy_block:
3313 }

```

(End of definition for __sort_copy_block:.)

`__sort_merge_blocks_aux:` At this stage, the first block starts at `\l__sort_begin_int`, and ends at `\l__sort_A_int`, and the second block starts at `\l__sort_top_int` and ends at `\l__sort_C_int`. The result of the merger is stored at positions indexed by `\l__sort_B_int`, which starts at `\l__sort_end_int - 1` and decreases down to `\l__sort_begin_int`, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either `swapped` or `same`. Of course, this means the arguments need to be given in the order they appear originally in the list.

```

3314 \cs_new_protected:Npn \__sort_merge_blocks_aux:
3315 {
3316     \exp_after:wN \__sort_compare:nn \exp_after:wN
3317     { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
3318     \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
3319     \prg_do_nothing:
3320     \__sort_return_mark:w
3321     \__sort_return_mark:w
3322     \s__sort_mark
3323     \__sort_return_none_error:
3324 }

```

(End of definition for __sort_merge_blocks_aux:.)

`\sort_return_same:` Each comparison should call `\sort_return_same:` or `\sort_return_swapped:` exactly once. If neither is called, `__sort_return_none_error:` is called, since the `return_mark` removes tokens until `\s__sort_mark`. If one is called, the `return_mark` auxiliary removes everything except `__sort_return_same:w` (or its `swapped` analogue) followed by `__sort_return_none_error:`. Finally if two or more are called, `__sort_return_two_error:` ends up before any `__sort_return_mark:w`, so that it produces an error.

```

3325 \cs_new_protected:Npn \sort_return_same:
3326     #1 \__sort_return_mark:w #2 \s__sort_mark
3327     {
3328         #1
3329         #2
3330         \__sort_return_two_error:
3331         \__sort_return_mark:w
3332         \s__sort_mark
3333         \__sort_return_same:w
3334     }
3335 \cs_new_protected:Npn \sort_return_swapped:
3336     #1 \__sort_return_mark:w #2 \s__sort_mark
3337     {
3338         #1
3339         #2
3340         \__sort_return_two_error:
3341         \__sort_return_mark:w
3342         \s__sort_mark
3343         \__sort_return_swapped:w
3344     }
3345 \cs_new_protected:Npn \__sort_return_mark:w #1 \s__sort_mark { }
3346 \cs_new_protected:Npn \__sort_return_none_error:
3347     {
3348         \msg_error:nnee { sort } { return-none }
3349         { \tex_the:D \tex_toks:D \l__sort_A_int }
3350         { \tex_the:D \tex_toks:D \l__sort_C_int }
3351         \__sort_return_same:w \__sort_return_none_error:
3352     }
3353 \cs_new_protected:Npn \__sort_return_two_error:
3354     {
3355         \msg_error:nnee { sort } { return-two }
3356         { \tex_the:D \tex_toks:D \l__sort_A_int }
3357         { \tex_the:D \tex_toks:D \l__sort_C_int }
3358     }

```

(End of definition for `\sort_return_same:` and others. These functions are documented on page 45.)

`__sort_return_same:w` If the comparison function returns `same`, then the second argument fed to `__sort_compare:nn` should remain to the right of the other one. Since we build the merger starting from the right, we copy that `\toks` register into the allotted range, then shift the pointers `B` and `C`, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

```

3359 \cs_new_protected:Npn \__sort_return_same:w #1 \__sort_return_none_error:
3360     {
3361         \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
3362         \int_decr:N \l__sort_B_int

```

```

3363     \int_decr:N \l__sort_C_int
3364     \if_int_compare:w \l__sort_C_int < \l__sort_top_int
3365         \use_i:nn
3366     \fi:
3367     \__sort_merge_blocks_aux:
3368 }

```

(End of definition for `__sort_return_same:w`.)

`__sort_return_swapped:w` If the comparison function returns `swapped`, then the next item to add to the merger is the first argument, contents of the `\toks` register *A*. Then shift the pointers *A* and *B* to the left, and go for one more step for the merger, unless the left block was exhausted (*A* goes below the threshold). In that case, all remaining `\toks` registers in the second block, indexed by *C*, are copied to the merger by `__sort_merge_blocks_end:`.

```

3369 \cs_new_protected:Npn \__sort_return_swapped:w #1 \__sort_return_none_error:
3370 {
3371     \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
3372     \int_decr:N \l__sort_B_int
3373     \int_decr:N \l__sort_A_int
3374     \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
3375         \__sort_merge_blocks_end: \use_i:nn
3376     \fi:
3377     \__sort_merge_blocks_aux:
3378 }

```

(End of definition for `__sort_return_swapped:w`.)

`__sort_merge_blocks_end:` This function's task is to copy the `\toks` registers in the block indexed by *C* to the merger indexed by *B*. The end can equally be detected by checking when *B* reaches the threshold `begin`, or when *C* reaches `top`.

```

3379 \cs_new_protected:Npn \__sort_merge_blocks_end:
3380 {
3381     \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
3382     \int_decr:N \l__sort_B_int
3383     \int_decr:N \l__sort_C_int
3384     \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
3385         \use_i:nn
3386     \fi:
3387     \__sort_merge_blocks_end:
3388 }

```

(End of definition for `__sort_merge_blocks_end:`.)

44.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best $O(n^2 \ln n)$) than non-expandable sorting functions ($O(n \ln n)$).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument `#4` of `__sort:nnNnn`). The arguments of `__sort:nnNnn` are 1. items less than `#4`, 2. items greater or equal to `#4`, 3. comparison, 4. pivot, 5. next item to test. If `#5` is the tail of

the list, call `\tl_sort:nN` on #1 and on #2, placing #4 in between; `\use:ff` expands the parts to make `\tl_sort:nN` f-expandable. Otherwise, compare #4 and #5 using #3. If they are ordered, place #5 amongst the “greater” items, otherwise amongst the “lesser” items, and continue partitioning.

```
\cs_new:Npn \tl_sort:nN #1#2
{
  \tl_if_blank:nF {#1}
  {
    \__sort:nnNnn { } { } #2
    #1 \q__sort_recursion_tail \q__sort_recursion_stop
  }
}
\cs_new:Npn \__sort:nnNnn #1#2#3#4#5
{
  \quark_if_recursion_tail_stop_do:nn {#5}
  { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
  #3 {#4} {#5}
  { \__sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
  { \__sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
}
\cs_generate_variant:Nn \use:nn { ff }
```

There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, `__sort:nnNnn` is called $O(n \ln n)$ times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to `__sort:nnNnn`. For this, the list is prepared by changing each *⟨item⟩* of the original token list into *⟨command⟩ {⟨item⟩}*, just like sequences are stored. We arrange things such that the *⟨command⟩* is the *⟨conditional⟩* provided by the user: the loop over the *⟨prepared tokens⟩* then looks like

```
\cs_new:Npn \__sort_loop:wNn ... #6#7
{
  #6 {⟨pivot⟩} {#7} ⟨loop big⟩ ⟨loop small⟩
  ⟨extra arguments⟩
}
\__sort_loop:wNn ... ⟨prepared tokens⟩
⟨end-loop⟩ {} \s__sort_stop
```

In this example, which matches the structure of `__sort_quick_split_i:NnnnnNn` and a few other functions below, the `__sort_loop:wNn` auxiliary normally receives the user’s *⟨conditional⟩* as #6 and an *⟨item⟩* as #7. This is compared to the *⟨pivot⟩* (the argument #5, not shown here), and the *⟨conditional⟩* leaves the *⟨loop big⟩* or *⟨loop small⟩* auxiliary, which both have the same form as `__sort_loop:wNn`, receiving the next pair *⟨conditional⟩ {⟨item⟩}* as #6 and #7. At the end, #6 is the *⟨end-loop⟩* function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the `true` and `false` branches of the conditional. For this, we introduce two versions of `__sort:nnNnn`,

which receive the new item as #1 and place it either into the list #2 of items less than the pivot #4 or into the list #3 of items greater or equal to the pivot.

```
\cs_new:Npn \__sort_i:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} { #2 {#1} } {#3} {#4}
}
\cs_new:Npn \__sort_ii:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} {#2} { #3 {#1} } {#4}
}
}
```

Note that the two functions have the form of `__sort_loop:wNn` above, receiving as #5 the conditional or a function to end the loop. In fact, the lists #2 and #3 must be made of pairs $\langle conditional \rangle \{ \langle item \rangle \}$, so we have to replace {#6} above by { #5 {#6} }, and {#1} by #1. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `__sort_quick_split:NnNn` expects a list followed by `\s__sort_mark { \code }`, and expands to $\langle code \rangle \langle sorted\ list \rangle$. Sorting the two parts of the list around the pivot is done with

```
\__sort_quick_split:NnNn #2 ... \s__sort_mark
{
  \__sort_quick_split:NnNn #1 ... \s__sort_mark { \code }
  { \pivot }
}
```

Items which are larger than the $\langle pivot \rangle$ are sorted, then placed after code that sorts the smaller items, and after the (braced) $\langle pivot \rangle$.

The fourth speed up is avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `__sort_i:nnnnNn` of the last example, but aware of whether the list of $\langle conditional \rangle \{ \langle item \rangle \}$ read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the $\langle end-loop \rangle$ function (that is initially placed after the “prepared” list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the $\langle end-loop \rangle$ function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when \TeX encounters

```
\use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first `\use:n` is not completely read by the second `\use:n`, hence must remain in memory; then the argument of the second `\use:n` is not completely read when grabbing the argument of the third `\use:n`, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested `\use:n`. In

practice, this means that we must read everything until a trailing `\s__sort_stop` once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical T_EX's memory.

`\tl_sort:nN`

`__sort_quick_prepare:Nnnn`
`__sort_quick_prepare_end:NNNnw`
`__sort_quick_cleanup:w`

The code within the `\exp_not:f` sorts the list, leaving in most cases a leading `\exp_not:f`, which stops the expansion, letting the result be return within `\exp_not:n`. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list #1 by inserting the conditional #2 before each item. The `prepare` auxiliary receives the conditional as #1, the prepared token list so far as #2, the next prepared item as #3, and the item after that as #4. The loop ends when #4 contains `\prg_break_point:`, then the `prepare_end` auxiliary finds the prepared token list as #4. The scene is then set up for `__sort_quick_split:NnNn`, which sorts the prepared list and perform the post action placed after `\s__sort_mark`, namely removing the trailing `\s__sort_stop` and `\s__sort_stop` and leaving `\exp_stop_f:` to stop f-expansion.

```

3389 \cs_new:Npn \tl_sort:nN #1#2
3390   {
3391     \exp_not:f
3392     {
3393       \tl_if_blank:nF {#1}
3394       {
3395         \__sort_quick_prepare:Nnnn #2 { } { }
3396         #1
3397         { \prg_break_point: \__sort_quick_prepare_end:NNNnw }
3398         \s__sort_stop
3399       }
3400     }
3401   }
3402 \cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
3403   {
3404     \prg_break: #4 \prg_break_point:
3405     \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
3406   }
3407 \cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \s__sort_stop
3408   {
3409     \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
3410     \s__sort_mark { \__sort_quick_cleanup:w \exp_stop_f: }
3411     \s__sort_mark \s__sort_stop
3412   }
3413 \cs_new:Npn \__sort_quick_cleanup:w #1 \s__sort_mark \s__sort_stop {#1}

```

(End of definition for `\tl_sort:nN` and others. This function is documented on page 123.)

`__sort_quick_split:NnNn`

`__sort_quick_only_i:NnnnnNn`
`__sort_quick_only_ii:NnnnnNn`
`__sort_quick_split_i:NnnnnNn`
`__sort_quick_split_ii:NnnnnNn`

The `only_i`, `only_ii`, `split_i` and `split_ii` auxiliaries receive a useless first argument, the new item #2 (that they append to either one of the next two arguments), the list #3 of items less than the pivot, bigger items #4, the pivot #5, a *function* #6, and an item #7. The *function* is the user's *conditional* except at the end of the list where it is `__sort_quick_end:nnTFNn`. The comparison is applied to the *pivot* and the *item*, and calls the `only_i` or `split_i` auxiliaries if the *item* is smaller, and the `only_ii` or `split_ii` auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument #2 left for the next call has the form *conditional* {*item*}, so that the lists #3 and #4 keep the right form to be fed to the next sorting function. The `split` auxiliary differs from these in that it is missing three of the arguments, which would be

empty, and its first argument is always the user's *conditional* rather than an ending function.

```

3414 \cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
3415 {
3416   #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn
3417   \__sort_quick_only_i:NnnnnNn
3418   \__sort_quick_single_end:nnnwnw
3419   { #3 {#4} } { } { } {#2}
3420 }
3421 \cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
3422 {
3423   #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3424   \__sort_quick_only_i:NnnnnNn
3425   \__sort_quick_only_i_end:nnnwnw
3426   { #6 {#7} } { #3 #2 } { } {#5}
3427 }
3428 \cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
3429 {
3430   #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn
3431   \__sort_quick_split_i:NnnnnNn
3432   \__sort_quick_only_ii_end:nnnwnw
3433   { #6 {#7} } { } { #4 #2 } {#5}
3434 }
3435 \cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
3436 {
3437   #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3438   \__sort_quick_split_i:NnnnnNn
3439   \__sort_quick_split_end:nnnwnw
3440   { #6 {#7} } { #3 #2 } {#4} {#5}
3441 }
3442 \cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
3443 {
3444   #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3445   \__sort_quick_split_i:NnnnnNn
3446   \__sort_quick_split_end:nnnwnw
3447   { #6 {#7} } {#3} { #4 #2 } {#5}
3448 }

```

(End of definition for __sort_quick_split:NnNn and others.)

```

\__sort_quick_end:nnTFNn
  \__sort_quick_single_end:nnnwnw
  \__sort_quick_only_i_end:nnnwnw
  \__sort_quick_only_ii_end:nnnwnw
  \__sort_quick_split_end:nnnwnw

```

The `__sort_quick_end:nnTFNn` appears instead of the user's conditional, and receives as its arguments the pivot #1, a fake item #2, a true and a false branches #3 and #4, followed by an ending function #5 (one of the four auxiliaries here) and another copy #6 of the fake item. All those are discarded except the function #5. This function receives lists #1 and #2 of items less than or greater than the pivot #3, then a continuation code #5 just after `\s__sort_mark`. To avoid a memory problem described earlier, all of the ending functions read #6 until `\s__sort_stop` and place #6 back into the input stream. When the lists #1 and #2 are empty, the `single` auxiliary simply places the continuation #5 before the pivot {#3}. When #2 is empty, #1 is sorted and placed before the pivot {#3}, taking care to feed the continuation #5 as a continuation for the function sorting #1. When #1 is empty, #2 is sorted, and the continuation argument is used to place the continuation #5 and the pivot {#3} before the sorted result. Finally, when both

lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```

3449 \cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
3450 \cs_new:Npn \__sort_quick_single_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3451 { #5 {#3} #6 \s__sort_stop }
3452 \cs_new:Npn \__sort_quick_only_i_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3453 {
3454   \__sort_quick_split:NnNn #1
3455   \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
3456   {#3}
3457   #6 \s__sort_stop
3458 }
3459 \cs_new:Npn \__sort_quick_only_ii_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3460 {
3461   \__sort_quick_split:NnNn #2
3462   \__sort_quick_end:nnTFNn { } \s__sort_mark { #5 {#3} }
3463   #6 \s__sort_stop
3464 }
3465 \cs_new:Npn \__sort_quick_split_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3466 {
3467   \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \s__sort_mark
3468   {
3469     \__sort_quick_split:NnNn #1
3470     \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
3471     {#3}
3472   }
3473   #6 \s__sort_stop
3474 }

```

(End of definition for __sort_quick_end:nnTFNn and others.)

44.6 Messages

`__sort_error:` Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many `l3sort` commands to be trivial, with `__sort_level:` jumping to the break point. This error recovery won't work in a group.

```

3475 \cs_new_protected:Npn \__sort_error:
3476 {
3477   \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
3478   \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
3479   \cs_set_protected:Npn \__sort_level: { \group_end: \prg_break: }
3480 }

```

(End of definition for __sort_error:.)

`__sort_disable_toksdef:` While sorting, `\toksdef` is locally disabled to prevent users from using `\newtoks` or similar commands in their comparison code: the `\toks` registers that would be assigned are in use by `l3sort`. In format mode, none of this is needed since there is no `\toks` allocator.

```

3481 \cs_new_protected:Npn \__sort_disable_toksdef:
3482 { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
3483 \cs_new_protected:Npn \__sort_disabled_toksdef:n #1
3484 {

```

```

3485 \msg_error:nne { sort } { toksdef }
3486   { \token_to_str:N #1 }
3487 \__sort_error:
3488 \tex_toksdef:D #1
3489 }
3490 \msg_new:nmmm { sort } { toksdef }
3491 { Allocation~of~\iow_char:N\ toks~registers~impossible~while~sorting. }
3492 {
3493   The~comparison~code~used~for~sorting~a~list~has~attempted~to~
3494   define~#1~as~a~new~\iow_char:N\ toks~register~using~
3495   \iow_char:N\ newtoks~
3496   or~a~similar~command.~The~list~will~not~be~sorted.
3497 }

```

(End of definition for `__sort_disable_toksdef:` and `__sort_disabled_toksdef:n`.)

`__sort_too_long_error:NNw` When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function #1.

```

3498 \cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
3499 {
3500   \fi:
3501   \msg_error:nneee { sort } { too-large }
3502     { \token_to_str:N #2 }
3503     { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
3504     { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
3505   #1 \__sort_error:
3506 }
3507 \msg_new:nmmm { sort } { too-large }
3508 { The~list~#1~is~too~long~to~be~sorted~by~TeX. }
3509 {
3510   TeX~has~#2~toks~registers~still~available:~
3511   this~only~allows~to~sort~with~up~to~#3~
3512   items.~The~list~will~not~be~sorted.
3513 }

```

(End of definition for `__sort_too_long_error:NNw`.)

```

3514 \msg_new:nmmm { sort } { return-none }
3515 { The~comparison~code~did~not~return. }
3516 {
3517   When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~
3518   did~not~call~
3519   \iow_char:N\ sort_return_same: ~nor~
3520   \iow_char:N\ sort_return_swapped: .~
3521   Exactly~one~of~these~should~be~called.
3522 }
3523 \msg_new:nmmm { sort } { return-two }
3524 { The~comparison~code~returned~multiple~times. }
3525 {
3526   When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~called~
3527   \iow_char:N\ sort_return_same: ~or~
3528   \iow_char:N\ sort_return_swapped: ~multiple~times.~
3529   Exactly~one~of~these~should~be~called.
3530 }
3531 \prop_gput:Nnn \g_msg_module_name_prop { sort } { LaTeX }
3532 \prop_gput:Nnn \g_msg_module_type_prop { sort } { }

```

3533 </package>

Chapter 45

l3tl-analysis implementation

3534 $\langle @@=tl \rangle$

45.1 Internal functions

$\backslash s_tl$ The format used to store token lists internally uses the scan mark $\backslash s_tl$ as a delimiter.

(End of definition for $\backslash s_tl$.)

45.2 Internal format

The task of the `l3tl-analysis` module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code, character code), as well as reconstruct the token list quickly. This internal format is used in `l3regex` where we need to support arbitrary tokens, and it is used in conversion functions in `l3str-convert`, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any $\langle token \rangle$ (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find $\langle tokens \rangle$ which both `o`-expand and `e/x`-expand to the given $\langle token \rangle$. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

$$\langle tokens \rangle \backslash s_tl \langle catcode \rangle \langle char\ code \rangle \backslash s_tl$$

The $\langle tokens \rangle$ `o`- and `e/x`-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for `l3str-convert`). The $\langle catcode \rangle$ is given as a single hexadecimal digit, 0 for control sequences. The $\langle char\ code \rangle$ is given as a decimal number, -1 for control sequences.

Using delimited arguments lets us build the $\langle tokens \rangle$ progressively when doing an encoding conversion in `l3str-convert`. On the other hand, the delimiter $\backslash s_tl$ may not appear unbraced in $\langle tokens \rangle$. This is not a problem because we are careful to wrap control sequences in braces (as an argument to `\exp_not:n`) when converting from a general token list to the internal format.

The current rule for converting a $\langle token \rangle$ to a balanced set of $\langle tokens \rangle$ which both `o`-expands and `e/x`-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__tl 0 -1 \s__tl`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s__tl 1 <char code> \s__tl`.
- An end-group character `}` becomes `\if_false: { \fi: } \s__tl 2 <char code> \s__tl`.
- A character with any other category code becomes `\exp_not:n {<character>} \s__tl <hex catcode> <char code> \s__tl`.

In contrast, for `\peek_analysis_map_inline:n` we must allow for an input stream containing `\outer` macros, so that wrapping all control sequences in `\exp_not:n` is unsafe. Instead, we write the more elaborate `__kernel_exp_not:w \exp_after:wN { \exp_not:N \cs }`. (On the other hand we make a better effort by avoiding `\exp_not:n` for characters other than active and macro parameters.)

3535 `(*package)`

45.3 Variables and helper functions

`\s__tl` The scan mark `\s__tl` is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare `\int_value:w '#1 \s__tl` with `\int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark` to extract a character code followed by the delimiter in an e-expansion.

3536 `\scan_new:N \s__tl`

(End of definition for `\s__tl`.)

`\l__tl_analysis_token`
`\l__tl_analysis_char_token` The tokens in the token list are probed with the TeX primitive `\futurelet`. We use `\l__tl_analysis_token` in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is `\l__tl_analysis_char_token`.

3537 `\cs_new_eq:NN \l__tl_analysis_token ?`

3538 `\cs_new_eq:NN \l__tl_analysis_char_token ?`

(End of definition for `\l__tl_analysis_token` and `\l__tl_analysis_char_token`.)

`\l__tl_peek_code_tl` Holds some code to be run once the next token has been fully analysed in `\peek_analysis_map_inline:n`.

3539 `\tl_new:N \l__tl_peek_code_tl`

(End of definition for `\l__tl_peek_code_tl`.)

`\c__tl_peek_catcodes_tl` A token list containing the character number 32 (space) with all possible category codes except 1 and 2 (begin-group and end-group). Why 32? Because some LuaTeX versions only allow creation of catcode 10 (space) tokens with this character code, so that we decided to make `\char_generate:nn` refuse to create such weird spaces as well. We do not include the macro parameter case (catcode 6) because it cannot be used as a macro delimiter.

3540 `\group_begin:`

3541 `\char_set_active_eq:NN \ \scan_stop:`

3542 `\tl_const:Ne \c__tl_peek_catcodes_tl`

```

3543 {
3544   \char_generate:nn { 32 } { 3 } 3
3545   \char_generate:nn { 32 } { 4 } 4
3546   \char_generate:nn { 32 } { 7 } 7
3547   \char_generate:nn { 32 } { 8 } 8
3548   \c_space_tl \token_to_str:N A
3549   \char_generate:nn { 32 } { 11 } \token_to_str:N B
3550   \char_generate:nn { 32 } { 12 } \token_to_str:N C
3551   \char_generate:nn { 32 } { 13 } \token_to_str:N D
3552 }
3553 \group_end:

```

(End of definition for \c__tl_peek_catcodes_tl.)

`\l__tl_analysis_normal_int` The number of normal (N-type argument) tokens since the last special token.

```
3554 \int_new:N \l__tl_analysis_normal_int
```

(End of definition for \l__tl_analysis_normal_int.)

`\l__tl_analysis_index_int` During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

```
3555 \int_new:N \l__tl_analysis_index_int
```

(End of definition for \l__tl_analysis_index_int.)

`\l__tl_analysis_nesting_int` Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

```
3556 \int_new:N \l__tl_analysis_nesting_int
```

(End of definition for \l__tl_analysis_nesting_int.)

`\l__tl_analysis_type_int` When encountering special characters, we record their “type” in this integer.

```
3557 \int_new:N \l__tl_analysis_type_int
```

(End of definition for \l__tl_analysis_type_int.)

`\g__tl_analysis_result_tl` The result of the conversion is stored in this token list, with a succession of items of the form

```
⟨tokens⟩ \s__tl ⟨catcode⟩ ⟨char code⟩ \s__tl
```

```
3558 \tl_new:N \g__tl_analysis_result_tl
```

(End of definition for \g__tl_analysis_result_tl.)

`_tl_analysis_extract_charcode:`
`_tl_analysis_extract_charcode_aux:w` Extracting the character code from the meaning of `\l__tl_analysis_token`. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form ‘⟨char⟩’.

```
3559 \cs_new:Npn \_tl_analysis_extract_charcode:
```

```
3560 {
```

```
3561   \exp_after:wN \_tl_analysis_extract_charcode_aux:w
```

```
3562     \token_to_meaning:N \l__tl_analysis_token
```

```
3563 }
```

```
3564 \cs_new:Npn \_tl_analysis_extract_charcode_aux:w #1 ~ #2 ~ { ‘ }
```

(End of definition for `_tl_analysis_extract_charcode:` and `_tl_analysis_extract_charcode_aux:w.`)

`_tl_analysis_cs_space_count:NN`
`_tl_analysis_cs_space_count:w`
`_tl_analysis_cs_space_count_end:w`

Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as semicolon-delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.

```

3565 \cs_new:Npn \_tl\_analysis\_cs\_space\_count:NN #1 #2
3566   {
3567     \exp_after:wN #1
3568     \int_value:w \int_eval:w 0
3569     \exp_after:wN \_tl\_analysis\_cs\_space\_count:w
3570     \token_to_str:N #2
3571     \fi: \_tl\_analysis\_cs\_space\_count\_end:w ; ~ !
3572   }
3573 \cs_new:Npn \_tl\_analysis\_cs\_space\_count:w #1 ~
3574   {
3575     \if_false: #1 #1 \fi:
3576     + 1
3577     \_tl\_analysis\_cs\_space\_count:w
3578   }
3579 \cs_new:Npn \_tl\_analysis\_cs\_space\_count\_end:w ; #1 \fi: #2 !
3580   { \exp_after:wN ; \int_value:w \str_count_ignore_spaces:n {#1} ; }

```

(End of definition for `_tl_analysis_cs_space_count:NN`, `_tl_analysis_cs_space_count:w`, and `_tl_analysis_cs_space_count_end:w.`)

45.4 Plan of attack

Our goal is to produce a token list of the form roughly

```

⟨token 1⟩ \s__tl ⟨catcode 1⟩ ⟨char code 1⟩ \s__tl
⟨token 2⟩ \s__tl ⟨catcode 2⟩ ⟨char code 2⟩ \s__tl
... ⟨token N⟩ \s__tl ⟨catcode N⟩ ⟨char code N⟩ \s__tl

```

Most but not all tokens can be grabbed as an undelimited (N-type) argument by `TEX`. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various `\toks` registers. In the second pass, which is done within an e-expanding assignment, normal tokens are taken in as N-type arguments, and special tokens are retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its `\meaning`, and what it looks like for `TEX` when it is in scanning mode (e.g., when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);

- anything else (then the token is always an N-type argument).

The token itself can “look like” one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it “true” character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

`__tl_analysis:n` Everything is done within a group, and all definitions are local. We use `\group_align_safe_begin/end:` to avoid problems in case `__tl_analysis:n` is used within an alignment and its argument contains alignment tab tokens.

```

3581 \cs_new_protected:Npn \__tl_analysis:n #1
3582   {
3583     \group_begin:
3584     \group_align_safe_begin:
3585       \__tl_analysis_a:n {#1}
3586       \__tl_analysis_b:n {#1}
3587     \group_align_safe_end:
3588     \group_end:
3589   }

```

(End of definition for `__tl_analysis:n`.)

45.5 Disabling active characters

`__tl_analysis_disable:n` Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to `undefined`. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For `pTeX` and `upTeX` we skip characters beyond `[0, 255]` because `\lccode` only allows those values.

```

3590 \group_begin:
3591   \char_set_catcode_active:N ^^@
3592   \cs_new_protected:Npn \__tl_analysis_disable:n #1
3593     {
3594       \tex_lccode:D 0 = #1 \exp_stop_f:
3595       \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
3596     }
3597   \bool_lazy_or:nnT
3598     { \sys_if_engine_ptex_p: }
3599     { \sys_if_engine_uptex_p: }
3600     {
3601       \cs_gset_protected:Npn \__tl_analysis_disable:n #1
3602         {
3603           \if_int_compare:w 256 > #1 \exp_stop_f:
3604           \tex_lccode:D 0 = #1 \exp_stop_f:

```

```

3605         \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
3606         \fi:
3607     }
3608 }
3609 \group_end:

```

(End of definition for `_tl_analysis_disable:n`.)

`_tl_analysis_disable_char:N` Similar to `_tl_analysis_disable:n`, but it receives a normal character token, tests if that token is active (by turning it into a space: the active space has been undefined at this point), and if so, disables it. Even if the character is active and set equal to a primitive conditional, nothing blows up. Again, in `pTeX` and `upTeX` we skip characters beyond `[0,255]`, which cannot be active anyways.

```

3610 \group_begin:
3611   \char_set_catcode_active:N ^^@
3612   \cs_new_protected:Npn \_tl_analysis_disable_char:N #1
3613     {
3614       \tex_lccode:D '#1 = 32 \exp_stop_f:
3615       \tex_lowercase:D { \if_meaning:w #1 } \tex_undefined:D
3616       \tex_let:D #1 \tex_undefined:D
3617       \fi:
3618     }
3619   \bool_lazy_or:nnT
3620     { \sys_if_engine_ptex_p: }
3621     { \sys_if_engine_uptex_p: }
3622     {
3623       \cs_gset_protected:Npn \_tl_analysis_disable_char:N #1
3624         {
3625           \if_int_compare:w 256 > '#1 \exp_stop_f:
3626           \tex_lccode:D '#1 = 32 \exp_stop_f:
3627           \tex_lowercase:D { \if_meaning:w #1 } \tex_undefined:D
3628           \tex_let:D #1 \tex_undefined:D
3629           \fi:
3630         }
3631     }
3632 }
3633 \group_end:

```

(End of definition for `_tl_analysis_disable_char:N`.)

45.6 First pass

The goal of this pass is to detect special (non-N-type) tokens, and count how many N-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a `\toks` register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;
3. a true non-space end-group character;
4. a true space end-group character;

5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);
11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence’s string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid N-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {<token>}` is non-empty, because the escape character is printable.

`__tl_analysis_a:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l__tl_analysis_nesting_int`. This reaches -1 when we read the closing brace.

```

3634 \cs_new_protected:Npn \__tl_analysis_a:n #1
3635   {
3636     \__tl_analysis_disable:n { 32 }
3637     \int_set:Nn \tex_escapechar:D { 92 }
3638     \int_zero:N \l__tl_analysis_normal_int
3639     \int_zero:N \l__tl_analysis_index_int
3640     \int_zero:N \l__tl_analysis_nesting_int
3641     \if_false: { \fi: \__tl_analysis_a_loop:w #1 }
3642     \int_decr:N \l__tl_analysis_index_int
3643   }

```

(End of definition for `__tl_analysis_a:n`.)

`__tl_analysis_a_loop:w` Read one character and check its type.

```

3644 \cs_new_protected:Npn \__tl_analysis_a_loop:w
3645   { \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_type:w }

```

(End of definition for `__tl_analysis_a_loop:w`.)

`__tl_analysis_a_type:w` At this point, `\l__tl_analysis_token` holds the meaning of the following token. We store in `\l__tl_analysis_type_int` information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;
- -1 end-group token;
- 2 other.

The values 0, 1, -1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over `\l__tl_analysis_token` if it matches with one of the character tokens (hence is not a primitive conditional).

```

3646 \cs_new_protected:Npn \__tl_analysis_a_type:w
3647 {
3648   \l__tl_analysis_type_int =
3649   \if_meaning:w \l__tl_analysis_token \c_space_token
3650     0
3651   \else:
3652     \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
3653       1
3654     \else:
3655       \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
3656         - 1
3657     \else:
3658       2
3659   \fi:
3660   \fi:
3661   \exp_stop_f:
3662   \if_case:w \l__tl_analysis_type_int
3663     \exp_after:wN \__tl_analysis_a_space:w
3664   \or: \exp_after:wN \__tl_analysis_a_bgroup:w
3665   \or: \exp_after:wN \__tl_analysis_a_safe:N
3666   \else: \exp_after:wN \__tl_analysis_a_egroup:w
3667   \fi:
3668   \fi:
3669 }

```

(End of definition for `__tl_analysis_a_type:w`.)

```

\__tl_analysis_a_space:w
  \__tl_analysis_a_space_test:w

```

In this branch, the following token's meaning is a blank space. Apply `\string` to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as `\l__tl_analysis_char_token` the first character of the string representation then test it in `__tl_analysis_a_space_test:w`. Also, since `__tl_analysis_a_store:` expects the special token to be stored in the relevant `\toks` register, we do that. The extra `\exp_not:n` is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```

3670 \cs_new_protected:Npn \__tl_analysis_a_space:w
3671 {
3672   \tex_afterassignment:D \__tl_analysis_a_space_test:w
3673   \exp_after:wN \cs_set_eq:NN
3674   \exp_after:wN \l__tl_analysis_char_token
3675   \token_to_str:N
3676 }
3677 \cs_new_protected:Npn \__tl_analysis_a_space_test:w

```

```

3678 {
3679 \if_meaning:w \l__tl_analysis_char_token \c_space_token
3680 \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
3681 \__tl_analysis_a_store:
3682 \else:
3683 \int_incr:N \l__tl_analysis_normal_int
3684 \fi:
3685 \__tl_analysis_a_loop:w
3686 }

```

(End of definition for `__tl_analysis_a_space:w` and `__tl_analysis_a_space_test:w`.)

`__tl_analysis_a_bgroup:w` The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a `toks` register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need `\l__tl_analysis_char_token` to be a separate control sequence from `\l__tl_analysis_token`, to compare them.

```

3687 \group_begin:
3688 \char_set_catcode_group_begin:N \^^@ % {
3689 \cs_new_protected:Npn \__tl_analysis_a_bgroup:w
3690 { \__tl_analysis_a_group:nw { \exp_after:wN \^^@ \if_false: } \fi: } }
3691 \char_set_catcode_group_end:N \^^@
3692 \cs_new_protected:Npn \__tl_analysis_a_egroup:w
3693 { \__tl_analysis_a_group:nw { \if_false: { \fi: \^^@ } } % }
3694 \group_end:
3695 \cs_new_protected:Npn \__tl_analysis_a_group:nw #1
3696 {
3697 \tex_lccode:D 0 = \__tl_analysis_extract_charcode: \scan_stop:
3698 \tex_lowercase:D { \tex_toks:D \l__tl_analysis_index_int {#1} }
3699 \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
3700 \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
3701 \fi:
3702 \__tl_analysis_disable:n { \tex_lccode:D 0 }
3703 \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_group_aux:w
3704 }
3705 \cs_new_protected:Npn \__tl_analysis_a_group_aux:w
3706 {
3707 \if_meaning:w \l__tl_analysis_token \tex_undefined:D
3708 \exp_after:wN \__tl_analysis_a_safe:N
3709 \else:
3710 \exp_after:wN \__tl_analysis_a_group_auxii:w
3711 \fi:
3712 }
3713 \cs_new_protected:Npn \__tl_analysis_a_group_auxii:w
3714 {
3715 \tex_afterassignment:D \__tl_analysis_a_group_test:w
3716 \exp_after:wN \cs_set_eq:NN
3717 \exp_after:wN \l__tl_analysis_char_token
3718 \token_to_str:N

```

```

3719 }
3720 \cs_new_protected:Npn \__tl_analysis_a_group_test:w
3721 {
3722   \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
3723     \__tl_analysis_a_store:
3724   \else:
3725     \int_incr:N \l__tl_analysis_normal_int
3726   \fi:
3727   \__tl_analysis_a_loop:w
3728 }

```

(End of definition for `__tl_analysis_a_bgroup:w` and others.)

`__tl_analysis_a_store:` This function is called each time we meet a special token; at this point, the `\toks` register `\l__tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l__tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l__tl_analysis_type_int`. The number of normal tokens until here and the type of special token are packed into a `\skip` register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```

3729 \cs_new_protected:Npn \__tl_analysis_a_store:
3730 {
3731   \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
3732   \if_int_compare:w \tex_lccode:D 0 = '\ \exp_stop_f:
3733     \tex_advance:D \l__tl_analysis_type_int \l__tl_analysis_type_int
3734   \fi:
3735   \tex_skip:D \l__tl_analysis_index_int
3736     = \l__tl_analysis_normal_int sp
3737     plus \l__tl_analysis_type_int sp \scan_stop:
3738   \int_incr:N \l__tl_analysis_index_int
3739   \int_zero:N \l__tl_analysis_normal_int
3740   \if_int_compare:w \l__tl_analysis_nesting_int = - \c_one_int
3741     \cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
3742   \fi:
3743 }

```

(End of definition for `_tl_analysis_a_store:`)

`_tl_analysis_a_safe:N`
`_tl_analysis_a_cs:ww`

This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. If the token is a single character (including space), the `\if_charcode:w` test yields true; we disable a potentially active character (that could otherwise masquerade as the true character in the next pass) and we count one “normal” token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead of slowly looping through the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing `\l_tl_analysis_index_int` (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of “normal” tokens.

```
3744 \cs_new_protected:Npn \_tl\_analysis\_a\_safe:N #1
3745 {
3746   \if_charcode:w
3747     \scan_stop:
3748     \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
3749     \scan_stop:
3750     \exp_after:wN \use_i:nn
3751   \else:
3752     \exp_after:wN \use_ii:nn
3753   \fi:
3754   {
3755     \_tl\_analysis\_disable\_char:N #1
3756     \int_incr:N \l\_tl\_analysis\_normal\_int
3757   }
3758   { \_tl\_analysis\_cs\_space\_count:NN \_tl\_analysis\_a\_cs:ww #1 }
3759   \_tl\_analysis\_a\_loop:w
3760 }
3761 \cs_new_protected:Npn \_tl\_analysis\_a\_cs:ww #1; #2;
3762 {
3763   \if_int_compare:w #1 > \c_zero_int
3764     \tex_skip:D \l\_tl\_analysis\_index\_int
3765     = \int_eval:n { \l\_tl\_analysis\_normal\_int + 1 } sp \exp_stop_f:
3766     \tex_advance:D \l\_tl\_analysis\_index\_int #1 \exp_stop_f:
3767   \else:
3768     \tex_advance:D
3769   \fi:
3770   \l\_tl\_analysis\_normal\_int #2 \exp_stop_f:
3771 }
```

(End of definition for `_tl_analysis_a_safe:N` and `_tl_analysis_a_cs:ww`.)

45.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

`_tl_analysis_b:n`
`_tl_analysis_b_loop:w`

Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```

3772 \cs_new_protected:Npn \__tl_analysis_b:n #1
3773 {
3774   \__kernel_tl_gset:Nx \g__tl_analysis_result_tl
3775   {
3776     \__tl_analysis_b_loop:w 0; #1
3777     \prg_break_point:
3778   }
3779 }
3780 \cs_new:Npn \__tl_analysis_b_loop:w #1;
3781 {
3782   \exp_after:wN \__tl_analysis_b_normals:ww
3783   \int_value:w \tex_skip:D #1 ; #1 ;
3784 }

```

(End of definition for __tl_analysis_b:n and __tl_analysis_b_loop:w.)

__tl_analysis_b_normals:ww
 __tl_analysis_b_normal:wwN

The first argument is the number of normal tokens which remain to be read, and the second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave `\exp_not:n` $\langle token \rangle$ `\s__tl` in the input stream (after e-expansion). Here, `\exp_not:n` is used rather than `\exp_not:N` because #3 could be a macro parameter character or could be `\s__tl` (which must be hidden behind braces in the result).

```

3785 \cs_new:Npn \__tl_analysis_b_normals:ww #1;
3786 {
3787   \if_int_compare:w #1 = \c_zero_int
3788   \__tl_analysis_b_special:w
3789   \fi:
3790   \__tl_analysis_b_normal:wwN #1;
3791 }
3792 \cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
3793 {
3794   \exp_not:n { \exp_not:n { #3 } } \s__tl
3795   \if_charcode:w
3796     \scan_stop:
3797     \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
3798     \scan_stop:
3799     \exp_after:wN \__tl_analysis_b_char:Nn
3800     \exp_after:wN \__tl_analysis_b_char_aux:nww
3801   \else:
3802     \exp_after:wN \__tl_analysis_b_cs:Nww
3803   \fi:
3804   #3 #1; #2;
3805 }

```

(End of definition for __tl_analysis_b_normals:ww and __tl_analysis_b_normal:wwN.)

__tl_analysis_b_char:Nn
 __tl_analysis_b_char_aux:nww

This function is called here with arguments `__tl_analysis_b_char_aux:nww` and a normal character, while in the peek analysis code it is called with `\use_none:n` and possibly a space character, which is why the function has signature `Nn`. If the normal token we grab is a character, leave $\langle catcode \rangle$ $\langle charcode \rangle$ followed by `\s__tl` in the input stream, and call `__tl_analysis_b_normals:ww` with its first argument decremented.

```

3806 \cs_new:Npe \__tl_analysis_b_char:Nn #1#2
3807 {

```



```

3808 \exp_not:N \if_meaning:w #2 \exp_not:N \tex_undefined:D
3809 \token_to_str:N D \exp_not:N \else:
3810 \exp_not:N \if_catcode:w #2 \c_catcode_other_token
3811 \token_to_str:N C \exp_not:N \else:
3812 \exp_not:N \if_catcode:w #2 \c_catcode_letter_token
3813 \token_to_str:N B \exp_not:N \else:
3814 \exp_not:N \if_catcode:w #2 \c_math_toggle_token 3
3815 \exp_not:N \else:
3816 \exp_not:N \if_catcode:w #2 \c_alignment_token 4
3817 \exp_not:N \else:
3818 \exp_not:N \if_catcode:w #2 \c_math_superscript_token 7
3819 \exp_not:N \else:
3820 \exp_not:N \if_catcode:w #2 \c_math_subscript_token 8
3821 \exp_not:N \else:
3822 \exp_not:N \if_catcode:w #2 \c_space_token
3823 \token_to_str:N A \exp_not:N \else:
3824 6
3825 \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: }
3826 #1 {#2}
3827 }
3828 \cs_new:Npn \__tl_analysis_b_char_aux:nww #1
3829 {
3830 \int_value:w ‘#1 \s__tl
3831 \exp_after:wN \__tl_analysis_b_normals:ww
3832 \int_value:w \int_eval:w - 1 +
3833 }

```

(End of definition for `__tl_analysis_b_char:Nn` and `__tl_analysis_b_char_aux:nww`.)

`__tl_analysis_b_cs:Nww` `__tl_analysis_b_cs_test:ww` If the token we grab is a control sequence, leave 0 -1 (as category code and character code) in the input stream, followed by `\s__tl`, and call `__tl_analysis_b_normals:ww` with updated arguments.

```

3834 \cs_new:Npn \__tl_analysis_b_cs:Nww #1
3835 {
3836 0 -1 \s__tl
3837 \__tl_analysis_cs_space_count:NN \__tl_analysis_b_cs_test:ww #1
3838 }
3839 \cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
3840 {
3841 \exp_after:wN \__tl_analysis_b_normals:ww
3842 \int_value:w \int_eval:w
3843 \if_int_compare:w #1 = \c_zero_int
3844 #3
3845 \else:
3846 \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
3847 \fi:
3848 - #2
3849 \exp_after:wN ;
3850 \int_value:w \int_eval:n { #4 + #1 } ;
3851 }

```

(End of definition for `__tl_analysis_b_cs:Nww` and `__tl_analysis_b_cs_test:ww`.)

`__tl_analysis_b_special:w` `__tl_analysis_b_special_char:wN` `__tl_analysis_b_special_space:w` Here, #1 is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the

end of the token list in the first pass). Unpack the `\toks` register: when `e/x`-expanding again, we will get the special token. Then leave the category code in the input stream, followed by the character code, and call `__tl_analysis_b_loop:w` with the next index.

```

3852 \group_begin:
3853   \char_set_catcode_other:N A
3854   \cs_new:Npn \__tl_analysis_b_special:w
3855     \fi: \__tl_analysis_b_normal:wwN 0 ; #1 ;
3856     {
3857       \fi:
3858       \if_int_compare:w #1 = \l__tl_analysis_index_int
3859         \exp_after:wN \prg_break:
3860       \fi:
3861       \tex_the:D \tex_toks:D #1 \s__tl
3862       \if_case:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
3863         \token_to_str:N A
3864       \or: 1
3865       \or: 1
3866       \else: 2
3867       \fi:
3868       \if_int_odd:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
3869         \exp_after:wN \__tl_analysis_b_special_char:wN \int_value:w
3870       \else:
3871         \exp_after:wN \__tl_analysis_b_special_space:w \int_value:w
3872       \fi:
3873       \int_eval:n { 1 + #1 } \exp_after:wN ;
3874       \token_to_str:N
3875     }
3876 \group_end:
3877 \cs_new:Npn \__tl_analysis_b_special_char:wN #1 ; #2
3878   {
3879     \int_value:w '#2 \s__tl
3880     \__tl_analysis_b_loop:w #1 ;
3881   }
3882 \cs_new:Npn \__tl_analysis_b_special_space:w #1 ; ~
3883   {
3884     32 \s__tl
3885     \__tl_analysis_b_loop:w #1 ;
3886   }

```

(End of definition for `__tl_analysis_b_special:w`, `__tl_analysis_b_special_char:wN`, and `__tl_analysis_b_special_space:w`.)

45.8 Mapping through the analysis

`\tl_analysis_map_inline:Nn` First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__kernel_prg_map_int` (shared between all modules), then define the payload macro, which runs the user code and has a name specific to that nesting depth. The looping macro grabs the `\tokens`, `\catcode` and `\char code`; it checks for the end of the loop with `\use_none:n ##2`, normally empty, but which becomes `\tl_map_break:` at the end; it then calls the payload macro with the arguments in the correct order (this is the reason why we cannot directly use the

same macro for looping and payload), and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```

3887 \cs_new_protected:Npn \tl_analysis_map_inline:Nn #1
3888   { \exp_args:No \tl_analysis_map_inline:nn #1 }
3889 \cs_new_protected:Npn \tl_analysis_map_inline:nn #1
3890   {
3891     \__tl_analysis:n {#1}
3892     \int_gincr:N \g__kernel_prg_map_int
3893     \exp_args:Nc \__tl_analysis_map:Nn
3894       { \__tl_analysis_map_inline_ \int_use:N \g__kernel_prg_map_int :wNw }
3895   }
3896 \cs_new_protected:Npn \__tl_analysis_map:Nn #1#2
3897   {
3898     \cs_gset_protected:Npn #1 ##1##2##3 {#2}
3899     \exp_after:wN \__tl_analysis_map:NwNw \exp_after:wN #1
3900     \g__tl_analysis_result_tl
3901     \s__tl { ? \tl_map_break: } \s__tl
3902     \prg_break_point:Nn \tl_map_break:
3903     { \int_gdecr:N \g__kernel_prg_map_int }
3904   }
3905 \cs_new_protected:Npn \__tl_analysis_map:NwNw #1 #2 \s__tl #3 #4 \s__tl
3906   {
3907     \use_none:n #3
3908     #1 {#2} {#4} {#3}
3909     \__tl_analysis_map:NwNw #1
3910   }

```

(End of definition for `\tl_analysis_map_inline:Nn` and others. These functions are documented on page 46.)

45.9 Showing the results

`\tl_analysis_show:N` Add to `__tl_analysis:n` a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as `\tl_show:N` by simply calling that function.

`\tl_analysis_log:N`
`__tl_analysis_show:NNN`

```

3911 \cs_new_protected:Npn \tl_analysis_show:N
3912   { \__tl_analysis_show:NNN \msg_show:nneeee \tl_show:N }
3913 \cs_new_protected:Npn \tl_analysis_log:N
3914   { \__tl_analysis_show:NNN \msg_log:nneeee \tl_log:N }
3915 \cs_new_protected:Npn \__tl_analysis_show:NNN #1#2#3
3916   {
3917     \tl_if_exist:NTF #3
3918     {
3919       \exp_args:No \__tl_analysis:n {#3}
3920       #1 { tl } { show-analysis }
3921       { \token_to_str:N #3 } { \__tl_analysis_show: } { } { }
3922     }
3923     { #2 #3 }
3924   }

```

(End of definition for `\tl_analysis_show:N`, `\tl_analysis_log:N`, and `__tl_analysis_show:NNN`. These functions are documented on page 46.)

```

\tl_analysis_show:n No existence test needed here.
\tl_analysis_log:n
\__tl_analysis_show:Nn
3925 \cs_new_protected:Npn \tl_analysis_show:n
3926 { \__tl_analysis_show:Nn \msg_show:nneeee }
3927 \cs_new_protected:Npn \tl_analysis_log:n
3928 { \__tl_analysis_show:Nn \msg_log:nneeee }
3929 \cs_new_protected:Npn \__tl_analysis_show:Nn #1#2
3930 {
3931   \__tl_analysis:n {#2}
3932   #1 { tl } { show-analysis } { } { \__tl_analysis_show: } { } { }
3933 }

```

(End of definition for `\tl_analysis_show:n`, `\tl_analysis_log:n`, and `__tl_analysis_show:Nn`. These functions are documented on page 46.)

`__tl_analysis_show:` Here, #1 o- and e/x-expands to the token; #2 is the category code (one uppercase hexadecimal digit), 0 for control sequences; #3 is the character code, which we ignore. In the cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```

\__tl_analysis_show_loop:wNw
3934 \cs_new:Npn \__tl_analysis_show:
3935 {
3936   \exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
3937   \s__tl { ? \prg_break: } \s__tl
3938   \prg_break_point:
3939 }
3940 \cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
3941 {
3942   \use_none:n #2
3943   \iow_newline: > \use:nn { ~ } { ~ }
3944   \if_int_compare:w "#2 = \c_zero_int
3945     \exp_after:wN \__tl_analysis_show_cs:n
3946   \else:
3947     \if_int_compare:w "#2 = 13 \exp_stop_f:
3948       \exp_after:wN \exp_after:wN
3949       \exp_after:wN \__tl_analysis_show_active:n
3950     \else:
3951       \exp_after:wN \exp_after:wN
3952       \exp_after:wN \__tl_analysis_show_normal:n
3953     \fi:
3954   \fi:
3955   {#1}
3956   \__tl_analysis_show_loop:wNw
3957 }

```

(End of definition for `__tl_analysis_show:` and `__tl_analysis_show_loop:wNw`.)

`__tl_analysis_show_normal:n` Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up TeX's alignment status.

```

3958 \cs_new:Npn \__tl_analysis_show_normal:n #1
3959 {
3960   \exp_after:wN \token_to_str:N #1 ~
3961   ( \exp_after:wN \token_to_meaning:N #1 )
3962 }

```

(End of definition for `_tl_analysis_show_normal:n`.)

```
\_tl\_analysis\_show\_value:N This expands to the value of #1 if it has any.
3963 \cs_new:Npn \_tl\_analysis\_show\_value:N #1
3964   {
3965     \token_if_expandable:NF #1
3966     {
3967       \token_if_chardef:NTF #1 \prg_break: { }
3968       \token_if_mathchardef:NTF #1 \prg_break: { }
3969       \token_if_dim_register:NTF #1 \prg_break: { }
3970       \token_if_int_register:NTF #1 \prg_break: { }
3971       \token_if_skip_register:NTF #1 \prg_break: { }
3972       \token_if_toks_register:NTF #1 \prg_break: { }
3973       \use_none:nmn
3974       \prg_break_point:
3975       \use:n { \exp_after:wN = \tex_the:D #1 }
3976     }
3977   }
```

(End of definition for `_tl_analysis_show_value:N`.)

`_tl_analysis_show_cs:n` Control sequences and active characters are printed in the same way, making sure not to go beyond the `\l_iow_line_count_int`. In case of an overflow, we replace the last characters by `\c_tl_analysis_show_etc_str`.

```
\_tl\_analysis\_show\_active:n
\_tl\_analysis\_show\_long:nn
\_tl\_analysis\_show\_long\_aux:nmnn
3978 \cs_new:Npn \_tl\_analysis\_show\_cs:n #1
3979   { \exp_args:No \_tl\_analysis\_show\_long:nn {#1} { control~sequence= } }
3980 \cs_new:Npn \_tl\_analysis\_show\_active:n #1
3981   { \exp_args:No \_tl\_analysis\_show\_long:nn {#1} { active~character= } }
3982 \cs_new:Npn \_tl\_analysis\_show\_long:nn #1
3983   {
3984     \_tl\_analysis\_show\_long\_aux:oofn
3985     { \token_to_str:N #1 }
3986     { \token_to_meaning:N #1 }
3987     { \_tl\_analysis\_show\_value:N #1 }
3988   }
3989 \cs_new:Npn \_tl\_analysis\_show\_long\_aux:nmnn #1#2#3#4
3990   {
3991     \int_compare:nNnTF
3992     { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
3993     > { \l\_iow\_line\_count\_int - 3 }
3994     {
3995       \str_range:nmn { #1 ~ ( #4 #2 #3 ) } { 1 }
3996       {
3997         \l\_iow\_line\_count\_int - 3
3998         - \str_count:N \c\_tl\_analysis\_show\_etc\_str
3999       }
4000       \c\_tl\_analysis\_show\_etc\_str
4001     }
4002     { #1 ~ ( #4 #2 #3 ) }
4003   }
4004 \cs_generate_variant:Nn \_tl\_analysis\_show\_long\_aux:nmnn { oof }
```

(End of definition for `_tl_analysis_show_cs:n` and others.)

45.10 Peeking ahead

The break statements use the general `\prg_map_break:Nn`.

`\peek_analysis_map_break:`
`\peek_analysis_map_break:n`

```
4005 \cs_new:Npn \peek_analysis_map_break:
4006   { \prg_map_break:Nn \peek_analysis_map_break: { } }
4007 \cs_new:Npn \peek_analysis_map_break:n
4008   { \prg_map_break:Nn \peek_analysis_map_break: }
```

(End of definition for `\peek_analysis_map_break:` and `\peek_analysis_map_break:n`. These functions are documented on page 208.)

`\l__tl_peek_charcode_int`

```
4009 \int_new:N \l__tl_peek_charcode_int
```

(End of definition for `\l__tl_peek_charcode_int`.)

`__tl_analysis_char_arg:Nw`
`__tl_analysis_char_arg_aux:Nw`

After a call to `\futurelet \l__tl_analysis_token` followed by a stringified character token (either explicit space or catcode other character), grab the argument and pass it to `#1`. We only need to do anything in the case of a space.

```
4010 \cs_new:Npn \__tl_analysis_char_arg:Nw
4011   {
4012     \if_meaning:w \l__tl_analysis_token \c_space_token
4013     \exp_after:wN \__tl_analysis_char_arg_aux:Nw
4014     \fi:
4015   }
4016 \cs_new:Npn \__tl_analysis_char_arg_aux:Nw #1 ~ { #1 { ~ } }
```

(End of definition for `__tl_analysis_char_arg:Nw` and `__tl_analysis_char_arg_aux:Nw`.)

`\peek_analysis_map_inline:n`

Save the user's code in a control sequence that is suitable for nested maps. We may wish to pass to this function an `\outer` control sequence or active character; for this we will undefine any expandable token (testing if it is `\outer` is much slower) within a group, closed immediately after the function reads its arguments to avoid affecting the user's code or even our peek code (there is no risk of undefining `\group_end:` itself since that is not expandable). This user's code function also calls the loop auxiliary, and includes the trailing `\prg_break_point:Nn` for when the user wants to stop the loop. The loop auxiliary must remove that break point because it must look at the input stream.

`__tl_peek_analysis_loop:NNn`
`__tl_peek_analysis_test:`
`__tl_peek_analysis_exp:N`
`__tl_peek_analysis_exp_aux:N`
`__tl_peek_analysis_nonexp:N`
`__tl_peek_analysis_cs:N`
`__tl_peek_analysis_char:N`
`__tl_peek_analysis_char:w`
`__tl_peek_analysis_special:`
`__tl_peek_analysis_retest:`
`__tl_peek_analysis_str:`
`__tl_peek_analysis_str:w`
`__tl_peek_analysis_str:n`
`__tl_peek_analysis_active_str:n`
`__tl_peek_analysis_explicit:n`
`__tl_peek_analysis_escape:`
`__tl_peek_analysis_collect:w`
`__tl_peek_analysis_collect:n`
`__tl_peek_analysis_collect_loop:`
`__tl_peek_analysis_collect_test:`
`__tl_peek_analysis_collect_end:NNNN`

```
4017 \cs_new_protected:Npn \peek_analysis_map_inline:n #1
4018   {
4019     \group_align_safe_begin:
4020     \int_gincr:N \g__kernel_prg_map_int
4021     \cs_set_protected:cpn
4022       { __tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
4023     ##1##2##3
4024     {
4025       \group_end:
4026       #1
4027       \__tl_peek_analysis_loop:NNn
4028       \prg_break_point:Nn \peek_analysis_map_break:
4029         {
4030           \int_gdecr:N \g__kernel_prg_map_int
4031           \group_align_safe_end:
4032         }
4033     }
```

```

4034   \_tl_peek_analysis_loop:NNn ? ? ?
4035 }

```

The loop starts a group (closed by the user-code function defined above) with a normalized escape character, and checks if the next token is special or N-type (distinguishing expandable from non-expandable tokens). The test for nonexpandable tokens in `_tl_peek_analysis_test`: must be done after the tests for begin-group, end-group, and space tokens, in case `\l_peek_token` is either `\outer` or is a primitive \TeX conditional, as such tokens cannot be skipped over correctly by conditional code.

```

4036 \cs_new_protected:Npn \_tl_peek_analysis_loop:NNn #1#2#3
4037 {
4038   \group_begin:
4039   \tl_set:Nc \l__tl_peek_code_tl
4040     {
4041     \exp_not:c
4042       { \_tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
4043     }
4044   \int_set:Nn \tex_escapechar:D { '\ }
4045   \peek_after:Nw \_tl_peek_analysis_test:
4046 }
4047 \cs_new_protected:Npn \_tl_peek_analysis_test:
4048 {
4049   \if_case:w
4050     \if_catcode:w \exp_not:N \l_peek_token { \c_max_int \fi:
4051     \if_catcode:w \exp_not:N \l_peek_token } \c_max_int \fi:
4052     \if_meaning:w \l_peek_token \c_space_token \c_max_int \fi:
4053     \exp_after:wN \if_meaning:w \exp_not:N \l_peek_token \l_peek_token
4054     \c_one_int
4055     \fi:
4056     \c_zero_int
4057     \exp_after:wN \exp_after:wN
4058     \exp_after:wN \_tl_peek_analysis_exp:N
4059     \exp_after:wN \exp_not:N
4060   \or:
4061     \exp_after:wN \_tl_peek_analysis_nonexp:N
4062   \else:
4063     \exp_after:wN \_tl_peek_analysis_special:
4064   \fi:
4065 }

```

Expandable tokens (which are automatically N-type) can be `\outer` macros, hence the need for `\exp_after:wN` and `\exp_not:N` in the code above, which allows the next function to safely grab the token as an argument. We run some code that is expanded using the primitive `\cs_set_nopar:Npc` rather than `\tl_set:Nc` to avoid grabbing it as an argument as `#1` may be `\outer`. To allow `#1` as an argument of the user's function (stored in `\l__tl_peek_code_tl`), we set it equal to `\scan_stop:`, but we do it at the last minute because `#1` may be some pretty important function such as `\exp_after:wN`. Then we put the user's function and the elaborate first argument `_kernel_exp_not:w \exp_after:wN { \exp_not:N #1 }`: indeed we cannot use `\exp_not:n {#1}` as this breaks for an `\outer` macro and we cannot use `\exp_not:N #1`, as o-expanding this yields a “notexpanded” token equal to (a weird) `\relax`, which would have the wrong value for primitive \TeX conditionals such as `\if_meaning:w`.

Then we must add `{-1}0` if the token is a control sequence and `{\charcode}D` otherwise. Distinguishing the two cases is easy: since we have made the escape character

printable, `\token_to_str:N` gives at least two characters for a control sequence versus a single one for an active character (possibly being a space). Importantly, once we apply `\token_to_str:N` we no longer need to worry about `\outer` tokens.

```

4066 \cs_new_protected:Npn \__tl_peek_analysis_exp:N #1
4067 {
4068   \cs_set_nopar:Npe \l__tl_peek_code_tl
4069   {
4070     \tex_let:D \exp_not:N #1 \scan_stop:
4071     \exp_not:o \l__tl_peek_code_tl
4072     {
4073       \exp_not:n { \__kernel_exp_not:w \exp_after:wN }
4074       { \exp_not:N \exp_not:N \exp_not:N #1 }
4075     }
4076     \exp_after:wN \__tl_peek_analysis_exp_aux:Nw
4077     \token_to_str:N #1 \prg_do_nothing: \s__tl
4078   }
4079   \l__tl_peek_code_tl
4080 }
4081 \cs_new:Npe \__tl_peek_analysis_exp_aux:Nw #1#2 \s__tl
4082 {
4083   \exp_not:N \if:w \scan_stop: #2 \scan_stop:
4084   { \exp_not:N \int_value:w '#1 } \token_to_str:N D
4085   \exp_not:N \else:
4086   { -1 } 0
4087   \exp_not:N \fi:
4088 }

```

For normal non-expandable tokens we must distinguish characters (including active ones and macro parameter characters) from control sequences (whose string representation is more than one character because we made the escape character printable). For a control sequence call the user code with suitable arguments, wrapping `#1` within `\exp_not:n` just in case it happens to be equal to a macro parameter character. We do not skip `\exp_not:n` when unnecessary, because this auxiliary is also called in `__tl_peek_analysis_retest:` where we have changed some control sequences or active characters to `\scan_stop:` temporarily.

```

4089 \cs_new_protected:Npn \__tl_peek_analysis_nonexp:N #1
4090 {
4091   \if_charcode:w
4092     \scan_stop:
4093     \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
4094     \scan_stop:
4095     \exp_after:wN \__tl_peek_analysis_char:N
4096   \else:
4097     \exp_after:wN \__tl_peek_analysis_cs:N
4098   \fi:
4099   #1
4100 }
4101 \cs_new_protected:Npn \__tl_peek_analysis_cs:N #1
4102 { \l__tl_peek_code_tl { \exp_not:n {#1} } { -1 } 0 }

```

For normal characters we must determine their catcode. The main difficulty is that the character may be an active character masquerading as (i.e., set equal to) itself with a different catcode. Two approaches based on `\lowercase` can detect this. One could make an active character with the same catcode as `#1` and change its definition before testing the

catcode of #1, but in some Unicode engine this fills up the hash table uselessly. Instead, we lowercase #1 itself, changing its character code to 32, namely space (because LuaTeX cannot turn catcode 10 characters to anything else than character code 32), then we apply `__tl_analysis_b_char:Nn`, which detects active characters by comparing them to `\tex_undefined:D`, and we must have undefined the active space (locally) for this test to work. To define `__tl_peek_analysis_char:N` itself we use an e-expanding assignment to get the active space in the right place after making it (just for this definition) unexpandable. Finally `__tl_peek_analysis_char:w` receives the `<charcode>`, `<user function>`, `<catcode>`, and `<token>`, and places the arguments in the correct order. It keeps `\exp_not:n` for macro parameter characters and active characters (the latter could be macro parameter characters, and it seems more uniform to always put `\exp_not:n`), and otherwise eliminates it by expanding once with `\exp_args:NNNo`.

```

4103 \group_begin:
4104 \char_set_active_eq:NN \ \scan_stop:
4105 \cs_new_protected:Npe \__tl_peek_analysis_char:N #1
4106   {
4107     \cs_set_eq:NN
4108       \char_generate:nn { 32 } { 13 }
4109       \exp_not:N \tex_undefined:D
4110     \tex_lccode:D '#1 = 32 \exp_stop_f:
4111     \tex_lowercase:D
4112     {
4113       \tl_put_right:Ne \exp_not:N \l__tl_peek_code_tl
4114         { \exp_not:n { \__tl_analysis_b_char:Nn \use_none:n } {#1} }
4115     }
4116     \exp_not:n
4117     {
4118       \exp_after:wN \__tl_peek_analysis_char:w
4119       \int_value:w
4120     }
4121     '#1
4122     \exp_not:n { \exp_after:wN \s__tl \l__tl_peek_code_tl }
4123     #1
4124   }
4125 \group_end:
4126 \cs_new_protected:Npn \__tl_peek_analysis_char:w #1 \s__tl #2#3#4
4127   {
4128     \if_charcode:w 6 #3
4129     \else:
4130       \if_charcode:w D #3
4131       \else:
4132         \exp_args:NNNo
4133         \fi:
4134       \fi:
4135       #2 { \exp_not:n {#4} } {#1} #3
4136   }

```

For special characters the idea is to eventually act with `\token_to_str:N`, then pick up one by one the characters of this string representation until hitting the token that follows. First determine the character code of (the meaning of) the `<token>` (which we know is a special token), make sure the escape character is different from it, normalize the meanings of two active characters and the empty control sequence, and filter out these cases in `__tl_peek_analysis_retest:`.

```

4137 \cs_new_protected:Npn \__tl_peek_analysis_special:
4138 {
4139   \tex_let:D \l__tl_analysis_token = ~ \l_peek_token
4140   \int_set:Nn \l__tl_peek_charcode_int
4141     { \__tl_analysis_extract_charcode: }
4142   \if_int_compare:w \l__tl_peek_charcode_int = \tex_escapechar:D
4143     \int_set:Nn \tex_escapechar:D { '\ / }
4144   \fi:
4145   \char_set_active_eq:nN { \l__tl_peek_charcode_int } \scan_stop:
4146   \char_set_active_eq:nN { \tex_escapechar:D } \scan_stop:
4147   \cs_set_eq:cN { } \scan_stop:
4148   \tex_futurelet:D \l__tl_analysis_token
4149   \__tl_peek_analysis_retest:
4150 }
4151 \cs_new_protected:Npn \__tl_peek_analysis_retest:
4152 {
4153   \if_meaning:w \l__tl_analysis_token \scan_stop:
4154     \exp_after:wN \__tl_peek_analysis_nonexp:N
4155   \else:
4156     \exp_after:wN \__tl_peek_analysis_str:
4157   \fi:
4158 }

```

At this point we know the meaning of the $\langle token \rangle$ in the input stream is \l_peek_token , either a space (32, 10) or a begin-group or end-group token (catcode 1 or 2), and we excluded a few cases that would be difficult later (empty control sequence, active character with the same character code as its meaning or as the escape character). The idea is to apply $\l_token_to_str:N$ to the $\langle token \rangle$ then grab characters (of category code 12 except for spaces that have category code 10) to reconstruct it. In earlier versions of the code we would peek at the $\langle next\ token \rangle$ that lies after $\langle token \rangle$ in the input stream, which would help us be more accurate in reconstructing the $\langle token \rangle$ case in edge cases (mentioned below), but this had the side-effect of tokenizing the input stream (turning characters into tokens) farther ahead than needed.

We hit the $\langle token \rangle$ with $\l_token_to_str:N$ and start grabbing characters. More precisely, by looking at the first character in the string representation of the $\langle token \rangle$ we distinguish three cases: a stringified control sequence starts with the escape character; for an explicit character we find that same character; for an active character we find anything else (we made sure to exclude the case of an active character whose string representation coincides with the other two cases).

```

4159 \cs_new_protected:Npn \__tl_peek_analysis_str:
4160 {
4161   \exp_after:wN \tex_futurelet:D
4162   \exp_after:wN \l__tl_analysis_token
4163   \exp_after:wN \__tl_peek_analysis_str:w
4164   \l_token_to_str:N
4165 }
4166 \cs_new_protected:Npn \__tl_peek_analysis_str:w
4167 { \__tl_analysis_char_arg:Nw \__tl_peek_analysis_str:n }
4168 \cs_new_protected:Npn \__tl_peek_analysis_str:n #1
4169 {
4170   \int_case:mnF { '#1 }
4171   {
4172     { \l__tl_peek_charcode_int }

```

```

4173     { \__tl_peek_analysis_explicit:n {#1} }
4174     { \tex_escapechar:D } { \__tl_peek_analysis_escape: }
4175   }
4176   { \__tl_peek_analysis_active_str:n {#1} }
4177 }

```

When #1 is a stringified active character we pass appropriate arguments to the user's code; thankfully `\char_generate:nn` can make active characters.

```

4178 \cs_new_protected:Npn \__tl_peek_analysis_active_str:n #1
4179 {
4180   \tl_put_right:Ne \l__tl_peek_code_tl
4181   {
4182     { \char_generate:nn { '#1 } { 13 } }
4183     { \int_value:w '#1 }
4184     \token_to_str:N D
4185   }
4186   \l__tl_peek_code_tl
4187 }

```

When #1 matches the character we had extracted from the meaning of `\l_peek_token`, the token was an explicit character, which can be a standard space, or a begin-group or end-group character with some character code. In the latter two cases we call `\char_generate:nn` with suitable arguments and put suitable `\if_false: \fi:` constructions to make the result balanced and such that o-expanding or e/x-expanding gives back a single (unbalanced) begin-group or end-group character.

```

4188 \cs_new_protected:Npn \__tl_peek_analysis_explicit:n #1
4189 {
4190   \tl_put_right:Ne \l__tl_peek_code_tl
4191   {
4192     \if_meaning:w \l_peek_token \c_space_token
4193     { ~ } { 32 } \token_to_str:N A
4194     \else:
4195       \if_catcode:w \l_peek_token \c_group_begin_token
4196       {
4197         \exp_not:N \exp_after:wN
4198         \char_generate:nn { '#1 } { 1 }
4199         \exp_not:N \if_false:
4200         \if_false: { \fi: }
4201         \exp_not:N \fi:
4202       }
4203       { \int_value:w '#1 }
4204       1
4205     \else:
4206     {
4207       \exp_not:N \if_false:
4208       { \if_false: } \fi:
4209       \exp_not:N \fi:
4210       \char_generate:nn { '#1 } { 2 }
4211     }
4212     { \int_value:w '#1 }
4213     2
4214     \fi:
4215   \fi:
4216 }

```

```

4217 \l__tl_peek_code_tl
4218 }

```

Finally there is the case of a special token whose string representation starts with an escape character, namely the token was a control sequence. In that case we could have grabbed the token directly as an N-type argument, but of course we couldn't know that until we had run all the various tests including stringifying the token. We are thus left with the hard work of picking up one by one the characters in the csname (being careful about spaces), until the constructed csname has the expected meaning. This fails if someone defines a token like `\bgroup@my` whose string representation starts the same as another token with the same meaning being an implicit character token of category code 1, 2, or 10.

```

4219 \cs_new_protected:Npn \__tl_peek_analysis_escape:
4220 {
4221   \tl_clear:N \l__tl_internal_a_tl
4222   \tex_futurelet:D \l__tl_analysis_token
4223   \__tl_peek_analysis_collect:w
4224 }
4225 \cs_new_protected:Npn \__tl_peek_analysis_collect:w
4226 { \__tl_analysis_char_arg:Nw \__tl_peek_analysis_collect:n }
4227 \cs_new_protected:Npn \__tl_peek_analysis_collect:n #1
4228 {
4229   \tl_put_right:Nn \l__tl_internal_a_tl {#1}
4230   \__tl_peek_analysis_collect_loop:
4231 }
4232 \cs_new_protected:Npn \__tl_peek_analysis_collect_loop:
4233 {
4234   \exp_after:wN \if_meaning:w
4235   \cs:w
4236   \if_cs_exist:w \l__tl_internal_a_tl \cs_end:
4237   \l__tl_internal_a_tl
4238   \else:
4239     c_one % anything short
4240   \fi:
4241   \cs_end:
4242   \l_peek_token
4243   \__tl_peek_analysis_collect_end:NNNN
4244   \fi:
4245   \tex_futurelet:D \l__tl_analysis_token
4246   \__tl_peek_analysis_collect:w
4247 }

```

As in all other cases, end by calling the user code with suitable arguments (here #1 is `\fi:`).

```

4248 \cs_new_protected:Npn \__tl_peek_analysis_collect_end:NNNN #1#2#3#4
4249 {
4250   #1
4251   \tl_put_right:Ne \l__tl_peek_code_tl
4252   {
4253     { \exp_not:N \exp_not:n { \exp_not:c { \l__tl_internal_a_tl } } }
4254     { -1 }
4255     0
4256   }
4257   \l__tl_peek_code_tl

```

```
4258 }
```

(End of definition for `\peek_analysis_map_inline:n` and others. This function is documented on page 208.)

45.11 Messages

`\c__tl_analysis_show_etc_str` When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```
4259 \tl_const:Ne \c__tl_analysis_show_etc_str % (  
4260 { \token_to_str:N \ETC.) }
```

(End of definition for `\c__tl_analysis_show_etc_str`.)

```
4261 \msg_new:nnn { tl } { show-analysis }  
4262 {  
4263   The-token-list~ \tl_if_empty:nF {#1} { #1 ~ }  
4264   \tl_if_empty:nTF {#2}  
4265     { is-empty }  
4266     { contains-the-tokens: #2 }  
4267 }  
4268 </package>
```

Chapter 46

l3regex implementation

4269 `<*package>`

4270 `<@@=regex>`

46.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since T_EX is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of n characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with $O(n)$ states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one “position”) at a time, exploring in parallel every possible path (“active thread”) through the NFA, considering active threads in an order determined by the quantifiers’ greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group, -1 for non-capturing groups.
- *Position*: each token in the query is labelled by an integer `<position>`, with $\text{min_pos} - 1 \leq \text{<position>} \leq \text{max_pos}$. The lowest and highest positions $\text{min_pos} - 1$ and max_pos correspond to imaginary begin and end markers (with non-existent category code and character code). max_pos is only set quite late in the processing.
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer `<state>` with $\text{min_state} \leq \text{<state>} < \text{max_state}$.
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.

- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches. The integer `\l__regex_step_int` is a unique id for all the steps of the matching algorithm.

We use `\l3intarray` to manipulate arrays of integers. We also abuse \TeX 's `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When building, `\toks<state>` holds the tests and actions to perform in the `<state>` of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last `<step>` in which each `<state>` was active.
- `\g__regex_thread_info_intarray` consists of blocks for each `<thread>` (with $\text{min_thread} \leq \langle \text{thread} \rangle < \text{max_thread}$). Each block has $1+2\backslash\l__regex_capturing_group_int$ entries: the `<state>` in which the `<thread>` currently is, followed by the beginnings of all submatches, and then the ends of all submatches. The `<threads>` are ordered starting from the best to the least preferred.
- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

When actually building the result,

- `\toks<position>` holds `<tokens>` which `o`- and `e`-expand to the `<position>`-th token in the query.
- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

46.2 Helpers

```

\__regex_int_eval:w Access the primitive: performance is key here, so we do not use the slower route via
\int_eval:n
4271 \cs_new_eq:NN \__regex_int_eval:w \tex_numexpr:D
(End of definition for \__regex_int_eval:w)

\__regex_standard_escapechar: Make the \escapechar into the standard backslash.
4272 \cs_new_protected:Npn \__regex_standard_escapechar:
4273 { \int_set:Nn \tex_escapechar:D { '\ } }

```

(End of definition for `__regex_standard_escapechar:`.)

`__regex_toks_use:w` Unpack a `\toks` given its number.
4274 `\cs_new:Npn __regex_toks_use:w { \tex_the:D \tex_toks:D }`

(End of definition for `__regex_toks_use:w`.)

`__regex_toks_clear:N` Empty a `\toks` or set it to a value, given its number.
`__regex_toks_set:Nn` 4275 `\cs_new_protected:Npn __regex_toks_clear:N #1`
`__regex_toks_set:No` 4276 `{ \tex_toks:D #1 = { } }`
4277 `\cs_new_eq:NN __regex_toks_set:Nn \tex_toks:D`
4278 `\cs_new_protected:Npn __regex_toks_set:No #1`
4279 `{ \tex_toks:D #1 = \exp_after:wN }`

(End of definition for `__regex_toks_clear:N` and `__regex_toks_set:Nn`.)

`__regex_toks_memcpy:NNn` Copy #3 `\toks` registers from #2 onwards to #1 onwards, like C's `memcpy`.
4280 `\cs_new_protected:Npn __regex_toks_memcpy:NNn #1#2#3`
4281 `{`
4282 `\prg_replicate:nn {#3}`
4283 `{`
4284 `\tex_toks:D #1 = \tex_toks:D #2`
4285 `\int_incr:N #1`
4286 `\int_incr:N #2`
4287 `}`
4288 `}`

(End of definition for `__regex_toks_memcpy:NNn`.)

`__regex_toks_put_left:Ne` During the building phase we wish to add e-expanded material to `\toks`, either to the left
`__regex_toks_put_right:Ne` or to the right. The expansion is done “by hand” for optimization (these operations are
`__regex_toks_put_right:Nn` used quite a lot). The `Nn` version of `__regex_toks_put_right:Ne` is provided because
it is more efficient than e-expanding with `\exp_not:n`.

4289 `\cs_if_exist:NTF \tex_etokspre:D`
4290 `{ \cs_new_eq:NN __regex_toks_put_left:Ne \tex_etokspre:D }`
4291 `{`
4292 `\cs_new_protected:Npn __regex_toks_put_left:Ne #1#2`
4293 `{ \tex_toks:D #1 = \tex_expanded:D { { #2 \tex_the:D \tex_toks:D #1 } } }`
4294 `}`
4295 `\cs_if_exist:NTF \tex_etoksapp:D`
4296 `{ \cs_new_eq:NN __regex_toks_put_right:Ne \tex_etoksapp:D }`
4297 `{`
4298 `\cs_new_protected:Npn __regex_toks_put_right:Ne #1#2`
4299 `{ \tex_toks:D #1 = \tex_expanded:D { { \tex_the:D \tex_toks:D #1 #2 } } }`
4300 `}`
4301 `\cs_if_exist:NTF \tex_toksapp:D`
4302 `{ \cs_new_eq:NN __regex_toks_put_right:Nn \tex_toksapp:D }`
4303 `{`
4304 `\cs_new_protected:Npn __regex_toks_put_right:Nn #1#2`
4305 `{ \tex_toks:D #1 = \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }`
4306 `}`

(End of definition for `__regex_toks_put_left:Ne` and `__regex_toks_put_right:Ne`.)

`__regex_curr_cs_to_str:` Expands to the string representation of the token (known to be a control sequence) at the current position `\l__regex_curr_pos_int`. It should only be used in e/x-expansion to avoid losing a leading space.

```

4307 \cs_new:Npn \__regex_curr_cs_to_str:
4308   {
4309     \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
4310     \l__regex_curr_token_tl
4311   }

```

(End of definition for __regex_curr_cs_to_str:.)

`__regex_intarray_item:NnF` Item of intarray, with a default value.

```

\__regex_intarray_item_aux:nNF
4312 \cs_new:Npn \__regex_intarray_item:NnF #1#2
4313   { \exp_args:No \__regex_intarray_item_aux:nNF { \tex_the:D \__regex_int_eval:w #2 } #1 }
4314 \cs_new:Npn \__regex_intarray_item_aux:nNF #1#2
4315   {
4316     \if_int_compare:w #1 > \c_zero_int
4317       \exp_after:wN \use_ii:nnn
4318     \fi:
4319     \use_ii:nn { \__kernel_intarray_item:Nn #2 {#1} }
4320   }

```

(End of definition for __regex_intarray_item:NnF and __regex_intarray_item_aux:nNF.)

`__regex_maplike_break:` Analogous to `\tl_map_break:`, this correctly exits `\tl_map_inline:nn` and similar constructions and jumps to the matching `\prg_break_point:Nn __regex_maplike_break: { }`.

```

4321 \cs_new:Npn \__regex_maplike_break:
4322   { \prg_map_break:Nn \__regex_maplike_break: { } }

```

(End of definition for __regex_maplike_break:.)

`__regex_tl_odd_items:n` Map through a token list one pair at a time, leaving the odd-numbered or even-numbered items (the first item is numbered 1).

```

\__regex_tl_even_items:n
\__regex_tl_even_items_loop:nn
4323 \cs_new:Npn \__regex_tl_odd_items:n #1 { \__regex_tl_even_items:n { ? #1 } }
4324 \cs_new:Npn \__regex_tl_even_items:n #1
4325   {
4326     \__regex_tl_even_items_loop:nn #1 \q__regex_nil \q__regex_nil
4327     \prg_break_point:
4328   }
4329 \cs_new:Npn \__regex_tl_even_items_loop:nn #1#2
4330   {
4331     \__regex_use_none_delimit_by_q_nil:w #2 \prg_break: \q__regex_nil
4332     { \exp_not:n {#2} }
4333     \__regex_tl_even_items_loop:nn
4334   }

```

(End of definition for __regex_tl_odd_items:n, __regex_tl_even_items:n, and __regex_tl_even_items_loop:nn.)

46.2.1 Constants and variables

`__regex_tmp:w` Temporary function used for various short-term purposes.

```
4335 \cs_new:Npn \__regex_tmp:w { }
```

(End of definition for `__regex_tmp:w`.)

`\l__regex_internal_a_tl` Temporary variables used for various purposes.

```
\l__regex_internal_b_tl 4336 \tl_new:N \l__regex_internal_a_tl
\l__regex_internal_a_int 4337 \tl_new:N \l__regex_internal_b_tl
\l__regex_internal_b_int 4338 \int_new:N \l__regex_internal_a_int
\l__regex_internal_c_int 4339 \int_new:N \l__regex_internal_b_int
\l__regex_internal_bool 4340 \int_new:N \l__regex_internal_c_int
\l__regex_internal_seq 4341 \bool_new:N \l__regex_internal_bool
\g__regex_internal_tl 4342 \seq_new:N \l__regex_internal_seq
4343 \tl_new:N \g__regex_internal_tl
```

(End of definition for `\l__regex_internal_a_tl` and others.)

`\l__regex_build_tl` This temporary variable is specifically for use with the `tl_build` machinery.

```
4344 \tl_new:N \l__regex_build_tl
```

(End of definition for `\l__regex_build_tl`.)

`\c__regex_no_match_regex` This regular expression matches nothing, but is still a valid regular expression. We could use a failing assertion, but I went for an empty class. It is used as the initial value for regular expressions declared using `\regex_new:N`.

```
4345 \tl_const:Nn \c__regex_no_match_regex
4346 {
4347   \__regex_branch:n
4348   { \__regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
4349 }
```

(End of definition for `\c__regex_no_match_regex`.)

`\l__regex_balance_int` During this phase, `\l__regex_balance_int` counts the balance of begin-group and end-group character tokens which appear before a given point in the token list. This variable is also used to keep track of the balance in the replacement text.

```
4350 \int_new:N \l__regex_balance_int
```

(End of definition for `\l__regex_balance_int`.)

46.2.2 Testing characters

```
\c__regex_ascii_min_int
\c__regex_ascii_max_control_int 4351 \int_const:Nn \c__regex_ascii_min_int { 0 }
\c__regex_ascii_max_int 4352 \int_const:Nn \c__regex_ascii_max_control_int { 31 }
4353 \int_const:Nn \c__regex_ascii_max_int { 127 }
```

(End of definition for `\c__regex_ascii_min_int`, `\c__regex_ascii_max_control_int`, and `\c__regex_ascii_max_int`.)

```
\c__regex_ascii_lower_int
4354 \int_const:Nn \c__regex_ascii_lower_int { 'a - 'A }
```

(End of definition for `\c__regex_ascii_lower_int`.)

46.2.3 Internal auxiliaries

`\q__regex_recursion_stop` Internal recursion quarks.
 4355 `\quark_new:N \q__regex_recursion_stop`
(End of definition for \q__regex_recursion_stop.)

`\q__regex_nil` Internal quarks.
 4356 `\quark_new:N \q__regex_nil`
(End of definition for \q__regex_nil.)

Functions to gobble up to a quark.
 4357 `\cs_new:Npn __regex_use_none_delimit_by_q_recursion_stop:w`
 4358 `#1 \q__regex_recursion_stop { }`
 4359 `\cs_new:Npn __regex_use_i_delimit_by_q_recursion_stop:nw`
 4360 `#1 #2 \q__regex_recursion_stop {#1}`
 4361 `\cs_new:Npn __regex_use_none_delimit_by_q_nil:w #1 \q__regex_nil { }`
(End of definition for __regex_use_none_delimit_by_q_recursion_stop:w, __regex_use_i_delimit_by_q_recursion_stop:nw, and __regex_use_none_delimit_by_q_nil:w.)

`__regex_quark_if_nil_p:n` Branching quark conditional.
 4362 `__kernel_quark_new_conditional:Nn __regex_quark_if_nil:N { F }`
(End of definition for __regex_quark_if_nil:nTF.)

`__regex_break_point:TF` When testing whether a character of the query token list matches a given character class
`__regex_break_true:w` in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like

```

  <test1> ... <test_n>
  \__regex_break_point:TF {<true code>} {<false code>}

```

If any of the tests succeeds, it calls `__regex_break_true:w`, which cleans up and leaves `<true code>` in the input stream. Otherwise, `__regex_break_point:TF` leaves the `<false code>` in the input stream.

```

  4363 \cs_new_protected:Npn \__regex_break_true:w
  4364    #1 \__regex_break_point:TF #2 #3 {#2}
  4365 \cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }

```

(End of definition for __regex_break_point:TF and __regex_break_true:w.)

`__regex_item_reverse:n` This function makes showing regular expressions easier, and lets us define `\D` in terms of `\d` for instance. There is a subtlety: the end of the query is marked by `-2`, and thus matches `\D` and other negated properties; this case is caught by another part of the code.

```

  4366 \cs_new_protected:Npn \__regex_item_reverse:n #1
  4367    {
  4368      #1
  4369      \__regex_break_point:TF { } \__regex_break_true:w
  4370    }

```

(End of definition for __regex_item_reverse:n.)

`_regex_item_caseful_equal:n` Simple comparisons triggering `_regex_break_true:w` when true.

```
\_regex_item_caseful_range:nn
4371 \cs_new_protected:Npn \_regex_item_caseful_equal:n #1
4372 {
4373   \if_int_compare:w #1 = \l__regex_curr_char_int
4374   \exp_after:wN \_regex_break_true:w
4375   \fi:
4376 }
4377 \cs_new_protected:Npn \_regex_item_caseful_range:nn #1 #2
4378 {
4379   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
4380   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
4381   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
4382   \fi:
4383   \fi:
4384 }
```

(End of definition for `_regex_item_caseful_equal:n` and `_regex_item_caseful_range:nn`.)

`_regex_item_caseless_equal:n` For caseless matching, we perform the test both on the `curr_char` and on the `case_`
`_regex_item_caseless_range:nn` `changed_char`. Before doing the second set of tests, we make sure that `case_changed_`
`char` has been computed.

```
4385 \cs_new_protected:Npn \_regex_item_caseless_equal:n #1
4386 {
4387   \if_int_compare:w #1 = \l__regex_curr_char_int
4388   \exp_after:wN \_regex_break_true:w
4389   \fi:
4390   \_regex_maybe_compute_ccc:
4391   \if_int_compare:w #1 = \l__regex_case_changed_char_int
4392   \exp_after:wN \_regex_break_true:w
4393   \fi:
4394 }
4395 \cs_new_protected:Npn \_regex_item_caseless_range:nn #1 #2
4396 {
4397   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
4398   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
4399   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
4400   \fi:
4401   \fi:
4402   \_regex_maybe_compute_ccc:
4403   \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
4404   \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
4405   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
4406   \fi:
4407   \fi:
4408 }
```

(End of definition for `_regex_item_caseless_equal:n` and `_regex_item_caseless_range:nn`.)

`_regex_compute_case_changed_char:` This function is called when `\l__regex_case_changed_char_int` has not yet been computed. If the current character code is in the range [65, 90] (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range [97, 122], subtract 32.

```
4409 \cs_new_protected:Npn \_regex_compute_case_changed_char:
4410 {
4411   \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
```

```

4412 \if_int_compare:w \l__regex_curr_char_int > 'Z \exp_stop_f:
4413 \if_int_compare:w \l__regex_curr_char_int > 'z \exp_stop_f: \else:
4414 \if_int_compare:w \l__regex_curr_char_int < 'a \exp_stop_f: \else:
4415 \int_sub:Nn \l__regex_case_changed_char_int \c__regex_ascii_lower_int
4416 \fi:
4417 \fi:
4418 \else:
4419 \if_int_compare:w \l__regex_curr_char_int < 'A \exp_stop_f: \else:
4420 \int_add:Nn \l__regex_case_changed_char_int \c__regex_ascii_lower_int
4421 \fi:
4422 \fi:
4423 \cs_set_eq:NN \__regex_maybe_compute_ccc: \prg_do_nothing:
4424 }
4425 \cs_new_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:

```

(End of definition for __regex_compute_case_changed_char:.)

`__regex_item_equal:n` Those must always be defined to expand to a `caseful` (default) or `caseless` version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```

4426 \cs_new_eq:NN \__regex_item_equal:n ?
4427 \cs_new_eq:NN \__regex_item_range:nn ?

```

(End of definition for __regex_item_equal:n and __regex_item_range:nn.)

`__regex_item_catcode:nT` The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```

4428 \cs_new_protected:Npn \__regex_item_catcode:
4429 {
4430 "
4431 \if_case:w \l__regex_curr_catcode_int
4432 1 \or: 4 \or: 10 \or: 40
4433 \or: 100 \or: \or: 1000 \or: 4000
4434 \or: 10000 \or: \or: 100000 \or: 400000
4435 \or: 1000000 \or: 4000000 \else: 1*0
4436 \fi:
4437 }
4438 \prg_new_protected_conditional:Npnn \__regex_item_catcode:n #1 { T }
4439 {
4440 \if_int_odd:w \__regex_int_eval:w #1 / \__regex_item_catcode: \scan_stop:
4441 \prg_return_true:
4442 \else:
4443 \prg_return_false:
4444 \fi:
4445 }
4446 \cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
4447 { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }

```

(End of definition for __regex_item_catcode:nT, __regex_item_catcode_reverse:nT, and __-regex_item_catcode:.)

`__regex_item_exact:nn` This matches an exact *(category)-(character code)* pair, or an exact control sequence, more precisely one of several possible control sequences, separated by `\scan_stop:`.

```

4448 \cs_new_protected:Npn \__regex_item_exact:nn #1#2
4449 {
4450   \if_int_compare:w #1 = \l__regex_curr_catcode_int
4451     \if_int_compare:w #2 = \l__regex_curr_char_int
4452       \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
4453     \fi:
4454   \fi:
4455 }
4456 \cs_new_protected:Npn \__regex_item_exact_cs:n #1
4457 {
4458   \int_compare:nNnTF \l__regex_curr_catcode_int = \c_zero_int
4459     {
4460       \__kernel_tl_set:Nx \l__regex_internal_a_tl
4461       { \scan_stop: \__regex_curr_cs_to_str: \scan_stop: }
4462       \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }
4463         \l__regex_internal_a_tl
4464         { \__regex_break_true:w } { }
4465     }
4466     { }
4467 }

```

(End of definition for `__regex_item_exact:nn` and `__regex_item_exact_cs:n`.)

`__regex_item_cs:n` Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches.

```

4468 \cs_new_protected:Npn \__regex_item_cs:n #1
4469 {
4470   \int_compare:nNnT \l__regex_curr_catcode_int = \c_zero_int
4471     {
4472       \group_begin:
4473       \__regex_single_match:
4474       \__regex_disable_submatches:
4475       \__regex_build_for_cs:n {#1}
4476       \bool_set_eq:NN \l__regex_saved_success_bool
4477       \g__regex_success_bool
4478       \exp_args:Ne \__regex_match_cs:n { \__regex_curr_cs_to_str: }
4479       \if_meaning:w \c_true_bool \g__regex_success_bool
4480       \group_insert_after:N \__regex_break_true:w
4481       \fi:
4482       \bool_gset_eq:NN \g__regex_success_bool
4483       \l__regex_saved_success_bool
4484       \group_end:
4485     }
4486 }

```

(End of definition for `__regex_item_cs:n`.)

46.2.4 Character property tests

`__regex_prop_d:` Character property tests for `\d`, `\W`, *etc.* These character properties are not affected by the `(?i)` option. The characters recognized by each one are as follows: `\d=[0-9]`,
`__regex_prop_h:`
`__regex_prop_s:`
`__regex_prop_v:`
`__regex_prop_w:`
`__regex_prop_N:`

`\w=[0-9A-Z_a-z]`, `\s=[_\^I\^J\^L\^M]`, `\h=[_\^I]`, `\v=[\^J-\^M]`, and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

```

4487 \cs_new_protected:Npn \__regex_prop_d:
4488   { \__regex_item_caseful_range:nn { '0 } { '9 } }
4489 \cs_new_protected:Npn \__regex_prop_h:
4490   {
4491     \__regex_item_caseful_equal:n { '\ }
4492     \__regex_item_caseful_equal:n { '\^I }
4493   }
4494 \cs_new_protected:Npn \__regex_prop_s:
4495   {
4496     \__regex_item_caseful_equal:n { '\ }
4497     \__regex_item_caseful_equal:n { '\^I }
4498     \__regex_item_caseful_equal:n { '\^J }
4499     \__regex_item_caseful_equal:n { '\^L }
4500     \__regex_item_caseful_equal:n { '\^M }
4501   }
4502 \cs_new_protected:Npn \__regex_prop_v:
4503   { \__regex_item_caseful_range:nn { '\^J } { '\^M } } % lf, vtab, ff, cr
4504 \cs_new_protected:Npn \__regex_prop_w:
4505   {
4506     \__regex_item_caseful_range:nn { 'a } { 'z }
4507     \__regex_item_caseful_range:nn { 'A } { 'Z }
4508     \__regex_item_caseful_range:nn { '0 } { '9 }
4509     \__regex_item_caseful_equal:n { '_' }
4510   }
4511 \cs_new_protected:Npn \__regex_prop_N:
4512   {
4513     \__regex_item_reverse:n
4514     { \__regex_item_caseful_equal:n { '\^J } }
4515   }

```

(End of definition for `__regex_prop_d:` and others.)

```

\__regex_posix_alnum: POSIX properties. No surprise.
\__regex_posix_alpha: 4516 \cs_new_protected:Npn \__regex_posix_alnum:
\__regex_posix_ascii: 4517   { \__regex_posix_alpha: \__regex_posix_digit: }
\__regex_posix_blank: 4518 \cs_new_protected:Npn \__regex_posix_alpha:
\__regex_posix_cntrl: 4519   { \__regex_posix_lower: \__regex_posix_upper: }
\__regex_posix_digit: 4520 \cs_new_protected:Npn \__regex_posix_ascii:
\__regex_posix_graph: 4521   {
\__regex_posix_lower: 4522     \__regex_item_caseful_range:nn
\__regex_posix_print: 4523     \c__regex_ascii_min_int
\__regex_posix_punct: 4524     \c__regex_ascii_max_int
\__regex_posix_space: 4525   }
\__regex_posix_upper: 4526 \cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\__regex_posix_word: 4527 \cs_new_protected:Npn \__regex_posix_cntrl:
\__regex_posix_xdigit: 4528   {
4529     \__regex_item_caseful_range:nn
4530     \c__regex_ascii_min_int
4531     \c__regex_ascii_max_control_int
4532     \__regex_item_caseful_equal:n \c__regex_ascii_max_int
4533   }

```

```

4534 \cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
4535 \cs_new_protected:Npn \__regex_posix_graph:
4536   { \__regex_item_caseful_range:nn { '!' } { '\~ } }
4537 \cs_new_protected:Npn \__regex_posix_lower:
4538   { \__regex_item_caseful_range:nn { 'a' } { 'z' } }
4539 \cs_new_protected:Npn \__regex_posix_print:
4540   { \__regex_item_caseful_range:nn { '\ ' } { '\~ } }
4541 \cs_new_protected:Npn \__regex_posix_punct:
4542   {
4543     \__regex_item_caseful_range:nn { '!' } { '/' }
4544     \__regex_item_caseful_range:nn { ':' } { '@' }
4545     \__regex_item_caseful_range:nn { '[' } { '[' }
4546     \__regex_item_caseful_range:nn { '\{ } { '\~ }
4547   }
4548 \cs_new_protected:Npn \__regex_posix_space:
4549   {
4550     \__regex_item_caseful_equal:n { '\ ' }
4551     \__regex_item_caseful_range:nn { '\^I } { '\^M }
4552   }
4553 \cs_new_protected:Npn \__regex_posix_upper:
4554   { \__regex_item_caseful_range:nn { 'A' } { 'Z' } }
4555 \cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
4556 \cs_new_protected:Npn \__regex_posix_xdigit:
4557   {
4558     \__regex_posix_digit:
4559     \__regex_item_caseful_range:nn { 'A' } { 'F' }
4560     \__regex_item_caseful_range:nn { 'a' } { 'f' }
4561   }

```

(End of definition for `__regex_posix_alnum:` and others.)

46.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special character (`*`, `?`, `{`, *etc.*) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `__regex_escape_use:nnnn` *<inline 1>* *<inline 2>* *<inline 3>* *{<token list>}* The *<token list>* is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function *<inline 1>*, and escaped characters are fed to the function *<inline 2>* within an e-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to *<inline 3>*. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an e-expanding assignment.

`__regex_escape_use:nnnn` The result is built in `\l__regex_internal_a_tl`, which is then left in the input stream. Tracing code is added as appropriate inside this token list. Go through #4 once, applying

#1, #2, or #3 as relevant to each character (after de-escaping it).

```
4562 \cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
4563 {
4564   \group_begin:
4565     \tl_clear:N \l__regex_internal_a_tl
4566     \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
4567     \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
4568     \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
4569     \__regex_standard_escapechar:
4570     \__kernel_tl_gset:Nx \g__regex_internal_tl
4571     { \__kernel_str_to_other_fast:n {#4} }
4572     \tl_put_right:Ne \l__regex_internal_a_tl
4573     {
4574       \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
4575       \scan_stop: \prg_break_point:
4576     }
4577     \exp_after:wN
4578     \group_end:
4579     \l__regex_internal_a_tl
4580 }
```

(End of definition for `__regex_escape_use:nnnn`.)

`__regex_escape_loop:N` `__regex_escape_loop:N` reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are “escaped”.

```
4581 \cs_new:Npn \__regex_escape_loop:N #1
4582 {
4583   \cs_if_exist_use:cF { __regex_escape_\token_to_str:N #1:w }
4584   { \__regex_escape_unescaped:N #1 }
4585   \__regex_escape_loop:N
4586 }
4587 \cs_new:cpn { __regex_escape_ \c_backslash_str :w }
4588   \__regex_escape_loop:N #1
4589 {
4590   \cs_if_exist_use:cF { __regex_escape_/\token_to_str:N #1:w }
4591   { \__regex_escape_escaped:N #1 }
4592   \__regex_escape_loop:N
4593 }
```

(End of definition for `__regex_escape_loop:N` and `__regex_escape_\:w`.)

`__regex_escape_unescaped:N` `__regex_escape_escaped:N` `__regex_escape_raw:N` Those functions are never called before being given a new meaning, so their definitions here don’t matter.

```
4594 \cs_new_eq:NN \__regex_escape_unescaped:N ?
4595 \cs_new_eq:NN \__regex_escape_escaped:N ?
4596 \cs_new_eq:NN \__regex_escape_raw:N ?
```

(End of definition for `__regex_escape_unescaped:N`, `__regex_escape_escaped:N`, and `__regex_escape_raw:N`.)

`__regex_escape_\scan_stop:w` `__regex_escape_/scan_stop:w` `__regex_escape_/a:w` `__regex_escape_/e:w` `__regex_escape_/f:w` `__regex_escape_/n:w` `__regex_escape_/r:w` `__regex_escape_/t:w` `__regex_escape_\:w` The loop is ended upon seeing the end-marker “break”, with an error if the string ended in a backslash. Spaces are ignored, and `\a`, `\e`, `\f`, `\n`, `\r`, `\t` take their meaning here.

```
4597 \cs_new_eq:cN { __regex_escape_ \iow_char:N\scan_stop: :w } \prg_break:
```

```

4598 \cs_new:cpn { __regex_escape_/ \iow_char:N\scan_stop: :w }
4599 {
4600   \msg_expandable_error:nn { regex } { trailing-backslash }
4601   \prg_break:
4602 }
4603 \cs_new:cpn { __regex_escape_~:w } { }
4604 \cs_new:cpe { __regex_escape_/a:w }
4605   { \exp_not:N __regex_escape_raw:N \iow_char:N \^^G }
4606 \cs_new:cpe { __regex_escape_/t:w }
4607   { \exp_not:N __regex_escape_raw:N \iow_char:N \^^I }
4608 \cs_new:cpe { __regex_escape_/n:w }
4609   { \exp_not:N __regex_escape_raw:N \iow_char:N \^^J }
4610 \cs_new:cpe { __regex_escape_/f:w }
4611   { \exp_not:N __regex_escape_raw:N \iow_char:N \^^L }
4612 \cs_new:cpe { __regex_escape_/r:w }
4613   { \exp_not:N __regex_escape_raw:N \iow_char:N \^^M }
4614 \cs_new:cpe { __regex_escape_/e:w }
4615   { \exp_not:N __regex_escape_raw:N \iow_char:N \^^[ }

```

(End of definition for `__regex_escape_/scan_stop: :w` and others.)

```

__regex_escape_/x:w
__regex_escape_x_end:w
__regex_escape_x_large:n

```

When `\x` is encountered, `__regex_escape_x_test:N` is responsible for grabbing some hexadecimal digits, and feeding the result to `__regex_escape_x_end:w`. If the number is too big interrupt the assignment and produce an error, otherwise call `__regex_escape_raw:N` on the corresponding character token.

```

4616 \cs_new:cpn { __regex_escape_/x:w } __regex_escape_loop:N
4617 {
4618   \exp_after:wN __regex_escape_x_end:w
4619   \int_value:w "0 __regex_escape_x_test:N
4620 }
4621 \cs_new:Npn __regex_escape_x_end:w #1 ;
4622 {
4623   \int_compare:nNnTF {#1} > \c_max_char_int
4624   {
4625     \msg_expandable_error:nfff { regex } { x-overflow }
4626     {#1} { \int_to_Hex:n {#1} }
4627   }
4628   {
4629     \exp_last_unbraced:Nf __regex_escape_raw:N
4630     { \char_generate:nn {#1} { 12 } }
4631   }
4632 }

```

(End of definition for `__regex_escape_/x:w`, `__regex_escape_x_end:w`, and `__regex_escape_x_large:n`.)

```

__regex_escape_x_test:N
__regex_escape_x_testii:N

```

Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either `__regex_escape_x_loop:N` or `__regex_escape_x:N`.

```

4633 \cs_new:Npn __regex_escape_x_test:N #1
4634 {
4635   \if_meaning:w \scan_stop: #1
4636   \exp_after:wN \use_i:nnn \exp_after:wN ;

```

```

4637 \fi:
4638 \use:n
4639 {
4640   \if_charcode:w \c_space_token #1
4641   \exp_after:wN \_regex_escape_x_test:N
4642   \else:
4643     \exp_after:wN \_regex_escape_x_testii:N
4644     \exp_after:wN #1
4645   \fi:
4646 }
4647 }
4648 \cs_new:Npn \_regex_escape_x_testii:N #1
4649 {
4650   \if_charcode:w \c_left_brace_str #1
4651   \exp_after:wN \_regex_escape_x_loop:N
4652   \else:
4653     \_regex_hexadecimal_use:NTF #1
4654     { \exp_after:wN \_regex_escape_x:N }
4655     { ; \exp_after:wN \_regex_escape_loop:N \exp_after:wN #1 }
4656   \fi:
4657 }

```

(End of definition for _regex_escape_x_test:N and _regex_escape_x_testii:N.)

_regex_escape_x:N This looks for the second digit in the unbraced case.

```

4658 \cs_new:Npn \_regex_escape_x:N #1
4659 {
4660   \if_meaning:w \scan_stop: #1
4661   \exp_after:wN \use_i:nnn \exp_after:wN ;
4662   \fi:
4663   \use:n
4664   {
4665     \_regex_hexadecimal_use:NTF #1
4666     { ; \_regex_escape_loop:N }
4667     { ; \_regex_escape_loop:N #1 }
4668   }
4669 }

```

(End of definition for _regex_escape_x:N.)

_regex_escape_x_loop:N Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace,
 _regex_escape_x_loop_error: otherwise raise an error outside the assignment.

```

4670 \cs_new:Npn \_regex_escape_x_loop:N #1
4671 {
4672   \if_meaning:w \scan_stop: #1
4673   \exp_after:wN \use_ii:nnn
4674   \fi:
4675   \use_ii:nn
4676   { ; \_regex_escape_x_loop_error:n { } {#1} }
4677   {
4678     \_regex_hexadecimal_use:NTF #1
4679     { \_regex_escape_x_loop:N }
4680     {
4681       \token_if_eq_charcode:NNTF \c_space_token #1

```

```

4682         { \_regex_escape_x_loop:N }
4683         {
4684             ;
4685             \exp_after:wN
4686             \token_if_eq_charcode:NNTF \c_right_brace_str #1
4687             { \_regex_escape_loop:N }
4688             { \_regex_escape_x_loop_error:n {#1} }
4689         }
4690     }
4691 }
4692 }
4693 \cs_new:Npn \_regex_escape_x_loop_error:n #1
4694 {
4695     \msg_expandable_error:nnn { regex } { x-missing-rbrace } {#1}
4696     \_regex_escape_loop:N #1
4697 }

```

(End of definition for _regex_escape_x_loop:N and _regex_escape_x_loop_error:.)

`_regex_hexadecimal_use:NTF` **T**_E**X** detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

4698 \cs_new:Npn \_regex_hexadecimal_use:NTF #1
4699 {
4700     \if_int_compare:w \c_one_int < "1 \token_to_str:N #1 \exp_stop_f:
4701     #1
4702     \else:
4703         \if_case:w
4704             \_regex_int_eval:w \exp_after:wN ‘ \token_to_str:N #1 - ‘a \scan_stop:
4705             A
4706             \or: B
4707             \or: C
4708             \or: D
4709             \or: E
4710             \or: F
4711             \else:
4712                 \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
4713             \fi:
4714         \fi:
4715         \use_i:nn
4716     }

```

(End of definition for _regex_hexadecimal_use:NTF.)

`_regex_char_if_alphanumeric:NTF` and `_regex_char_if_special:NTF` These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as “raw” characters. Namely,

- alphanumeric characters are “raw” if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ASCII characters are “raw” if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ASCII are always “raw”.

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means 0–9, A–Z, a–z; “special” character means non-alphanumeric but printable ascii, from space (hex 20) to del (hex 7E).

```

4717 \prg_new_conditional:Npnn \_regex_char_if_special:N #1 { TF }
4718 {
4719   \if:w
4720     T
4721     \if_int_compare:w '#1 > 'Z \exp_stop_f:
4722     \if_int_compare:w '#1 > 'z \exp_stop_f:
4723     \if_int_compare:w '#1 < \c__regex_ascii_max_int
4724     \else: F \fi:
4725   \else:
4726     \if_int_compare:w '#1 < 'a \exp_stop_f:
4727     \else: F \fi:
4728   \fi:
4729 \else:
4730 \if_int_compare:w '#1 > '9 \exp_stop_f:
4731 \if_int_compare:w '#1 < 'A \exp_stop_f:
4732 \else: F \fi:
4733 \else:
4734 \if_int_compare:w '#1 < '0 \exp_stop_f:
4735 \if_int_compare:w '#1 < '\' \exp_stop_f:
4736 F \fi:
4737 \else: F \fi:
4738 \fi:
4739 \fi:
4740 T
4741 \prg_return_true:
4742 \else:
4743 \prg_return_false:
4744 \fi:
4745 }
4746 \prg_new_conditional:Npnn \_regex_char_if_alphanumeric:N #1 { TF }
4747 {
4748   \if:w
4749     T
4750     \if_int_compare:w '#1 > 'Z \exp_stop_f:
4751     \if_int_compare:w '#1 > 'z \exp_stop_f:
4752     F
4753     \else:
4754     \if_int_compare:w '#1 < 'a \exp_stop_f:
4755     F \fi:
4756   \fi:
4757 \else:
4758 \if_int_compare:w '#1 > '9 \exp_stop_f:
4759 \if_int_compare:w '#1 < 'A \exp_stop_f:
4760 F \fi:
4761 \else:
4762 \if_int_compare:w '#1 < '0 \exp_stop_f:
4763 F \fi:
4764 \fi:
4765 \fi:
4766 T

```

```

4767     \prg_return_true:
4768     \else:
4769     \prg_return_false:
4770     \fi:
4771 }

```

(End of definition for `_regex_char_if_alphanumeric:NTF` and `_regex_char_if_special:NTF`.)

46.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a “compiled” regular expression. This compiled expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- `_regex_class:NnnnN` \langle boolean \rangle $\{\langle$ tests $\rangle\}$ $\{\langle$ min $\rangle\}$ $\{\langle$ more $\rangle\}$ \langle laziness \rangle
- `_regex_group:nnnN` $\{\langle$ branches $\rangle\}$ $\{\langle$ min $\rangle\}$ $\{\langle$ more $\rangle\}$ \langle laziness \rangle , also `_regex_group_no_capture:nnnN` and `_regex_group_resetting:nnnN` with the same syntax.
- `_regex_branch:n` $\{\langle$ contents $\rangle\}$
- `_regex_command_K`:
- `_regex_assertion:Nn` \langle boolean \rangle $\{\langle$ assertion test $\rangle\}$, where the \langle assertion test \rangle is `_regex_b_test:` or `_regex_Z_test:` or `_regex_A_test:` or `_regex_G_test:`

Tests can be the following:

- `_regex_item_caseful_equal:n` $\{\langle$ char code $\rangle\}$
- `_regex_item_caseless_equal:n` $\{\langle$ char code $\rangle\}$
- `_regex_item_caseful_range:nn` $\{\langle$ min $\rangle\}$ $\{\langle$ max $\rangle\}$
- `_regex_item_caseless_range:nn` $\{\langle$ min $\rangle\}$ $\{\langle$ max $\rangle\}$
- `_regex_item_catcode:nT` $\{\langle$ catcode bitmap $\rangle\}$ $\{\langle$ tests $\rangle\}$
- `_regex_item_catcode_reverse:nT` $\{\langle$ catcode bitmap $\rangle\}$ $\{\langle$ tests $\rangle\}$
- `_regex_item_reverse:n` $\{\langle$ tests $\rangle\}$
- `_regex_item_exact:nn` $\{\langle$ catcode $\rangle\}$ $\{\langle$ char code $\rangle\}$
- `_regex_item_exact_cs:n` $\{\langle$ csnames $\rangle\}$, more precisely given as \langle csname \rangle `\scan_stop:` \langle csname \rangle `\scan_stop:` \langle csname \rangle and so on in a brace group.
- `_regex_item_cs:n` $\{\langle$ compiled regex $\rangle\}$

46.3.1 Variables used when compiling

`\l__regex_group_level_int` We make sure to open the same number of groups as we close.

```
4772 \int_new:N \l__regex_group_level_int
```

(End of definition for \l__regex_group_level_int.)

`\l__regex_mode_int` While compiling, ten modes are recognized, labelled -63 , -23 , -6 , -2 , 0 , 2 , 3 , 6 , 23 , 63 .
`\c__regex_cs_in_class_mode_int` See section 46.3.3. We only define some of these as constants.

```
4773 \int_new:N \l__regex_mode_int
4774 \int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
4775 \int_const:Nn \c__regex_cs_mode_int { -2 }
4776 \int_const:Nn \c__regex_outer_mode_int { 0 }
4777 \int_const:Nn \c__regex_catcode_mode_int { 2 }
4778 \int_const:Nn \c__regex_class_mode_int { 3 }
4779 \int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }
```

(End of definition for \l__regex_mode_int and others.)

`\l__regex_catcodes_int` We wish to allow constructions such as `\c[^BE](. .\cL[a-z] . .)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l__regex_catcodes_int` and `\l__regex_default_catcodes_int` are bitmaps, sums of 4^c , for all allowed catcodes c . The latter is local to each capturing group, and we reset `\l__regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
4780 \int_new:N \l__regex_catcodes_int
4781 \int_new:N \l__regex_default_catcodes_int
4782 \bool_new:N \l__regex_catcodes_bool
```

(End of definition for \l__regex_catcodes_int, \l__regex_default_catcodes_int, and \l__regex_catcodes_bool.)

`\c__regex_catcode_C_int` Constants: 4^c for each category, and the sum of all powers of 4.

```
4783 \int_const:Nn \c__regex_catcode_C_int { "1 }
4784 \int_const:Nn \c__regex_catcode_B_int { "4 }
4785 \int_const:Nn \c__regex_catcode_E_int { "10 }
4786 \int_const:Nn \c__regex_catcode_M_int { "40 }
4787 \int_const:Nn \c__regex_catcode_T_int { "100 }
4788 \int_const:Nn \c__regex_catcode_P_int { "1000 }
4789 \int_const:Nn \c__regex_catcode_U_int { "4000 }
4790 \int_const:Nn \c__regex_catcode_D_int { "10000 }
4791 \int_const:Nn \c__regex_catcode_S_int { "100000 }
4792 \int_const:Nn \c__regex_catcode_L_int { "400000 }
4793 \int_const:Nn \c__regex_catcode_O_int { "1000000 }
4794 \int_const:Nn \c__regex_catcode_A_int { "4000000 }
4795 \int_const:Nn \c__regex_all_catcodes_int { "5515155 }
```

(End of definition for \c__regex_catcode_C_int and others.)

`\l__regex_internal_regex` The compilation step stores its result in this variable.

```
4796 \cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex
```

(End of definition for \l__regex_internal_regex.)

`\l__regex_show_prefix_seq` This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
4797 \seq_new:N \l__regex_show_prefix_seq
(End of definition for \l__regex_show_prefix_seq.)
```

`\l__regex_show_lines_int` A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
4798 \int_new:N \l__regex_show_lines_int
(End of definition for \l__regex_show_lines_int.)
```

46.3.2 Generic helpers used when compiling

`__regex_two_if_eq:NNNNTF` Used to compare pairs of things like `__regex_compile_special:N ?` together. It's often inconvenient to get the catcodes of the character to match so we just compare the character code. Besides, the expanding behaviour of `\if:w` is very useful as that means we can use `\c_left_brace_str` and the like.

```
4799 \cs_new:Npn \__regex_two_if_eq:NNNNTF #1#2#3#4
4800 {
4801   \if_meaning:w #1 #3
4802   \if:w #2 #4
4803   \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nnn
4804   \fi:
4805   \fi:
4806   \use_ii:nn
4807 }
```

(End of definition for `__regex_two_if_eq:NNNNTF`.)

`__regex_get_digits:NTFw` If followed by some raw digits, collect them one by one in the integer variable #1, and
`__regex_get_digits_loop:w` take the true branch. Otherwise, take the false branch.

```
4808 \cs_new_protected:Npn \__regex_get_digits:NTFw #1#2#3#4#5
4809 {
4810   \__regex_if_raw_digit:NNTF #4 #5
4811   { #1 = #5 \__regex_get_digits_loop:nw {#2} }
4812   { #3 #4 #5 }
4813 }
4814 \cs_new:Npn \__regex_get_digits_loop:nw #1#2#3
4815 {
4816   \__regex_if_raw_digit:NNTF #2 #3
4817   { #3 \__regex_get_digits_loop:nw {#1} }
4818   { \scan_stop: #1 #2 #3 }
4819 }
```

(End of definition for `__regex_get_digits:NTFw` and `__regex_get_digits_loop:w`.)

`__regex_if_raw_digit:NNTF` Test used when grabbing digits for the `{m,n}` quantifier. It only accepts non-escaped digits.

```
4820 \cs_new:Npn \__regex_if_raw_digit:NNTF #1#2
4821 {
4822   \if_meaning:w \__regex_compile_raw:N #1
4823   \if_int_compare:w \c_one_int < 1 #2 \exp_stop_f:
```



```

4824         \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nnn
4825         \fi:
4826     \fi:
4827     \use_ii:nn
4828 }

```

(End of definition for `_regex_if_raw_digit:NNTF`.)

46.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

- 6 `[\c{...}]` control sequence in a class,
- 2 `\c{...}` control sequence,
- 0 ... outer,
- 2 `\c...` catcode test,
- 6 `[\c...]` catcode test in a class,
- 63 `[\c{[...]}]` class inside mode -6,
- 23 `\c{[...]}` class inside mode -2,
- 3 `[...]` class inside mode 0,
- 23 `\c[...]` class inside mode 2,
- 63 `[\c[...]]` class inside mode 6.

This list is exhaustive, because `\c` escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as $m \rightarrow (m - 15)/13$, truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from m to $(m - 15)/13$, truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the `\d` character class) can appear in any mode.

`_regex_if_in_class:TF` Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```

4829 \prg_new_conditional:Npnn \_regex_if_in_class: { TF }
4830   {
4831     \if_int_odd:w \l__regex_mode_int
4832     \prg_return_true:
4833   \else:
4834     \prg_return_false:
4835   \fi:
4836   }

```

(End of definition for _regex_if_in_class:TF.)

`_regex_if_in_cs:TF` Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```

4837 \cs_new:Npn \_regex_if_in_cs:TF
4838   {
4839     \if_int_odd:w \l__regex_mode_int
4840   \else:
4841     \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
4842     \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nnn
4843   \fi:
4844   \fi:
4845   \use_ii:nn
4846   }

```

(End of definition for _regex_if_in_cs:TF.)

`_regex_if_in_class_or_catcode:TF` Assertions are only allowed in modes 0, -2, and -6, *i.e.*, even, non-positive modes.

```

4847 \cs_new:Npn \_regex_if_in_class_or_catcode:TF
4848   {
4849     \if_int_odd:w \l__regex_mode_int
4850   \else:
4851     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
4852   \else:
4853     \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
4854   \fi:
4855   \fi:
4856   \use_i:nn
4857   }

```

(End of definition for _regex_if_in_class_or_catcode:TF.)

`_regex_if_within_catcode:TF` This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```

4858 \prg_new_conditional:Npnn \_regex_if_within_catcode: { TF }
4859   {
4860     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
4861     \prg_return_true:
4862   \else:
4863     \prg_return_false:
4864   \fi:
4865   }

```

(End of definition for `__regex_if_within_catcode:TF`.)

`__regex_chk_c_allowed:T` The `\c` escape sequence is only allowed in modes 0 and 3, *i.e.*, not within any other `\c` escape sequence.

```
4866 \cs_new_protected:Npn \__regex_chk_c_allowed:T
4867   {
4868     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
4869     \else:
4870       \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
4871       \else:
4872         \msg_error:nn { regex } { c-bad-mode }
4873         \exp_after:wN \use_i:nnn
4874       \fi:
4875     \fi:
4876     \use:n
4877   }
```

(End of definition for `__regex_chk_c_allowed:T`.)

`__regex_mode_quit_c:` This function changes the mode as it is needed just after a catcode test.

```
4878 \cs_new_protected:Npn \__regex_mode_quit_c:
4879   {
4880     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
4881     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
4882     \else:
4883       \if_int_compare:w \l__regex_mode_int =
4884       \c__regex_catcode_in_class_mode_int
4885       \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
4886     \fi:
4887     \fi:
4888   }
```

(End of definition for `__regex_mode_quit_c:.`)

46.3.4 Framework

`__regex_compile:w` Used when compiling a user regex or a regex for the `\c{...}` escape sequence within another regex. Start building a token list within a group (with `e`-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building `\l__regex_internal_regex`.

```
4889 \cs_new_protected:Npn \__regex_compile:w
4890   {
4891     \group_begin:
4892     \tl_build_begin:N \l__regex_build_tl
4893     \int_zero:N \l__regex_group_level_int
4894     \int_set_eq:NN \l__regex_default_catcodes_int
4895     \c__regex_all_catcodes_int
4896     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
4897     \cs_set:Npn \__regex_item_equal:n { \__regex_item_caseful_equal:n }
4898     \cs_set:Npn \__regex_item_range:mn { \__regex_item_caseful_range:mn }
4899     \tl_build_put_right:Nn \l__regex_build_tl
4900     { \__regex_branch:n { \if_false: } \fi: }
4901   }
```

```

4902 \cs_new_protected:Npn \__regex_compile_end:
4903 {
4904   \__regex_if_in_class:TF
4905   {
4906     \msg_error:nn { regex } { missing-rbrack }
4907     \use:c { __regex_compile_]: }
4908     \prg_do_nothing: \prg_do_nothing:
4909   }
4910   { }
4911   \if_int_compare:w \l__regex_group_level_int > \c_zero_int
4912   \msg_error:nne { regex } { missing-rparen }
4913   { \int_use:N \l__regex_group_level_int }
4914   \prg_replicate:nn
4915     \l__regex_group_level_int
4916     {
4917       \tl_build_put_right:Nn \l__regex_build_tl
4918       {
4919         \if_false: { \fi: }
4920         \if_false: { \fi: } { 1 } { 0 } \c_true_bool
4921       }
4922       \tl_build_end:N \l__regex_build_tl
4923       \exp_args:NNNo
4924       \group_end:
4925       \tl_build_put_right:Nn \l__regex_build_tl
4926       { \l__regex_build_tl }
4927     }
4928   \fi:
4929   \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
4930   \tl_build_end:N \l__regex_build_tl
4931   \exp_args:NNNe
4932   \group_end:
4933   \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
4934 }

```

(End of definition for __regex_compile:w and __regex_compile_end:.)

`__regex_compile:n` The compilation is done between `__regex_compile:w` and `__regex_compile_end:`, starting in mode 0. Then `__regex_escape_use:n` distinguishes special characters, escaped alphanumerics, and raw characters, interpreting `\a`, `\x` and other sequences. The 4 trailing `\prg_do_nothing:` are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any `\c{...}` is properly closed. No need to check that brackets are closed properly since `__regex_compile_end:` does that. However, catch the case of a trailing `\cL` construction.

```

4935 \cs_new_protected:Npn \__regex_compile:n #1
4936 {
4937   \__regex_compile:w
4938   \__regex_standard_escapechar:
4939   \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
4940   \__regex_escape_use:n
4941   {
4942     \__regex_char_if_special:NTF ##1
4943     \__regex_compile_special:N \__regex_compile_raw:N ##1
4944   }
4945   {

```

```

4946     \__regex_char_if_alphanumeric:N\TF ##1
4947     \__regex_compile_escaped:N \__regex_compile_raw:N ##1
4948   }
4949   { \__regex_compile_raw:N ##1 }
4950   { #1 }
4951   \prg_do_nothing: \prg_do_nothing:
4952   \prg_do_nothing: \prg_do_nothing:
4953   \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
4954   { \msg_error:nn { regex } { c-trailing } }
4955   \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
4956   {
4957     \msg_error:nn { regex } { c-missing-rbrace }
4958     \__regex_compile_end_cs:
4959     \prg_do_nothing: \prg_do_nothing:
4960     \prg_do_nothing: \prg_do_nothing:
4961   }
4962   \__regex_compile_end:
4963 }

```

(End of definition for __regex_compile:n.)

`__regex_compile_use:n` Use a regex, regardless of whether it is given as a string (in which case we need to compile) or as a regex variable. This is used for `\regex_match_case:nn` and related functions to allow a mixture of explicit regex and regex variables.

```

4964 \cs_new_protected:Npn \__regex_compile_use:n #1
4965 {
4966   \tl_if_single_token:nT {#1}
4967   {
4968     \exp_after:wN \__regex_compile_use_aux:w
4969     \token_to_meaning:N #1 ~ \q__regex_nil
4970   }
4971   \__regex_compile:n {#1} \l__regex_internal_regex
4972 }
4973 \cs_new_protected:Npn \__regex_compile_use_aux:w #1 ~ #2 \q__regex_nil
4974 {
4975   \str_if_eq:nnT { #1 ~ } { macro:->\__regex_branch:n }
4976   { \use_ii:nnn }
4977 }

```

(End of definition for __regex_compile_use:n.)

`__regex_compile_escaped:N` If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

`__regex_compile_special:N`

```

4978 \cs_new_protected:Npn \__regex_compile_special:N #1
4979 {
4980   \cs_if_exist_use:cF { __regex_compile_#1: }
4981   { \__regex_compile_raw:N #1 }
4982 }
4983 \cs_new_protected:Npn \__regex_compile_escaped:N #1
4984 {
4985   \cs_if_exist_use:cF { __regex_compile_/#1: }
4986   { \__regex_compile_raw:N #1 }
4987 }

```

(End of definition for `__regex_compile_escaped:N` and `__regex_compile_special:N`)

`__regex_compile_one:n` This is used after finding one “test”, such as `\d`, or a raw character. If that followed a catcode test (e.g., `\cL`), then restore the mode. If we are not in a class, then the test is “standalone”, and we need to add `__regex_class:NnnnN` and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```

4988 \cs_new_protected:Npn \__regex_compile_one:n #1
4989   {
4990     \__regex_mode_quit_c:
4991     \__regex_if_in_class:TF { }
4992     {
4993       \tl_build_put_right:Nn \l__regex_build_tl
4994       { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
4995     }
4996     \tl_build_put_right:Ne \l__regex_build_tl
4997     {
4998       \if_int_compare:w \l__regex_catcodes_int <
4999       \c__regex_all_catcodes_int
5000       \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
5001       { \exp_not:N \exp_not:n {#1} }
5002       \else:
5003       \exp_not:N \exp_not:n {#1}
5004       \fi:
5005     }
5006     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5007     \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
5008   }

```

(End of definition for `__regex_compile_one:n`)

`__regex_compile_abort_tokens:n` This function places the collected tokens back in the input stream, each as a raw character.
`__regex_compile_abort_tokens:e` Spaces are not preserved.

```

5009 \cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
5010   {
5011     \use:e
5012     {
5013       \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
5014       \__regex_compile_raw:N
5015     }
5016   }
5017 \cs_generate_variant:Nn \__regex_compile_abort_tokens:n { e }

```

(End of definition for `__regex_compile_abort_tokens:n`)

46.3.5 Quantifiers

`__regex_compile_if_quantifier:TFw` This looks ahead and checks whether there are any quantifier (special character equal to either of `?+*{}`). This is useful for the `\u` and `\ur` escape sequences.

```

5018 \cs_new_protected:Npn \__regex_compile_if_quantifier:TFw #1#2#3#4
5019   {
5020     \token_if_eq_meaning:NNTF #3 \__regex_compile_special:N
5021     { \cs_if_exist:cTF { __regex_compile_quantifier_#4:w } }
5022     { \use_ii:nn }
5023     {#1} {#2} #3 #4

```

```
5024 }
```

(End of definition for `__regex_compile_if_quantifier:TFw`.)

`__regex_compile_quantifier:w` This looks ahead and finds any quantifier (special character equal to either of `?+*{`).

```
5025 \cs_new_protected:Npn \__regex_compile_quantifier:w #1#2
5026 {
5027   \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
5028   {
5029     \cs_if_exist_use:cF { \__regex_compile_quantifier_#2:w }
5030     { \__regex_compile_quantifier_none: #1 #2 }
5031   }
5032   { \__regex_compile_quantifier_none: #1 #2 }
5033 }
```

(End of definition for `__regex_compile_quantifier:w`.)

`__regex_compile_quantifier_none:` Those functions are called whenever there is no quantifier, or a braced construction is invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).
`__regex_compile_quantifier_abort:eNN`

```
5034 \cs_new_protected:Npn \__regex_compile_quantifier_none:
5035 {
5036   \tl_build_put_right:Nn \l__regex_build_tl
5037   { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
5038 }
5039 \cs_new_protected:Npn \__regex_compile_quantifier_abort:eNN #1#2#3
5040 {
5041   \__regex_compile_quantifier_none:
5042   \msg_warning:nnee { regex } { invalid-quantifier } {#1} {#3}
5043   \__regex_compile_abort_tokens:e {#1}
5044   #2 #3
5045 }
```

(End of definition for `__regex_compile_quantifier_none:` and `__regex_compile_quantifier_abort:eNN`.)

`__regex_compile_quantifier_laziness:nnNN` Once the “main” quantifier (`?`, `*`, `+` or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending `__regex_class:NnnnN` and friends), the start-point of the range, its end-point, and a boolean, `true` for lazy and `false` for greedy operators.

```
5046 \cs_new_protected:Npn \__regex_compile_quantifier_laziness:nnNN #1#2#3#4
5047 {
5048   \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ?
5049   {
5050     \tl_build_put_right:Nn \l__regex_build_tl
5051     { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
5052   }
5053   {
5054     \tl_build_put_right:Nn \l__regex_build_tl
5055     { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
5056     #3 #4
5057   }
5058 }
```

(End of definition for `__regex_compile_quantifier_laziness:nnNN`.)

`_regex_compile_quantifier_?:w` For each “basic” quantifier, `?`, `*`, `+`, feed the correct arguments to `_regex_compile_`
`_regex_compile_quantifier_*:w` `quantifier_laziness:nnNN`, `-1` means that there is no upper bound on the number of
`_regex_compile_quantifier_+:w` repetitions.

```
5059 \cs_new_protected:cpn { \_regex_compile_quantifier_?:w }
5060   { \_regex_compile_quantifier_laziness:nnNN { 0 } { 1 } }
5061 \cs_new_protected:cpn { \_regex_compile_quantifier_*:w }
5062   { \_regex_compile_quantifier_laziness:nnNN { 0 } { -1 } }
5063 \cs_new_protected:cpn { \_regex_compile_quantifier_+:w }
5064   { \_regex_compile_quantifier_laziness:nnNN { 1 } { -1 } }
```

(End of definition for `_regex_compile_quantifier_?:w`, `_regex_compile_quantifier_*:w`, and `_regex_compile_quantifier_+:w`.)

`_regex_compile_quantifier_{:w` Three possible syntaxes: `{⟨int⟩}`, `{⟨int⟩,}`, or `{⟨int⟩,⟨int⟩}`. Any other syntax causes
`_regex_compile_quantifier_braced_auxi:w` us to abort and put whatever we collected back in the input stream, as raw characters,
`_regex_compile_quantifier_braced_auxii:w` including the opening brace. Grab a number into `\l__regex_internal_a_int`. If the
`_regex_compile_quantifier_braced_auxiii:w` number is followed by a right brace, the range is $[a, a]$. If followed by a comma, grab one
more number, and call the `_ii` or `_iii` auxiliary. Those auxiliaries check for a closing
brace, leading to the range $[a, \infty]$ or $[a, b]$, encoded as `{a}{-1}` and `{a}{b-a}`.

```
5065 \cs_new_protected:cpn { \_regex_compile_quantifier_ \c_left_brace_str :w }
5066   {
5067     \_regex_get_digits:NTFw \l__regex_internal_a_int
5068     { \_regex_compile_quantifier_braced_auxi:w }
5069     { \_regex_compile_quantifier_abort:eNN { \c_left_brace_str } }
5070   }
5071 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxi:w #1#2
5072   {
5073     \str_case_e:nnF { #1 #2 }
5074     {
5075       { \_regex_compile_special:N \c_right_brace_str }
5076       {
5077         \exp_args:No \_regex_compile_quantifier_laziness:nnNN
5078         { \int_use:N \l__regex_internal_a_int } 0
5079       }
5080     } { \_regex_compile_special:N , }
5081     {
5082       \_regex_get_digits:NTFw \l__regex_internal_b_int
5083       { \_regex_compile_quantifier_braced_auxiii:w }
5084       { \_regex_compile_quantifier_braced_auxii:w }
5085     }
5086   }
5087   {
5088     \_regex_compile_quantifier_abort:eNN
5089     { \c_left_brace_str \int_use:N \l__regex_internal_a_int }
5090     #1 #2
5091   }
5092 }
5093 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxii:w #1#2
5094   {
5095     \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N \c_right_brace_str
5096     {
5097       \exp_args:No \_regex_compile_quantifier_laziness:nnNN
5098       { \int_use:N \l__regex_internal_a_int } { -1 }
5099     }
5100   }
```



```

5100     {
5101         \_regex_compile_quantifier_abort:eNN
5102         { \c_left_brace_str \int_use:N \l__regex_internal_a_int , }
5103         #1 #2
5104     }
5105 }
5106 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxiii:w #1#2
5107 {
5108     \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N \c_right_brace_str
5109     {
5110         \if_int_compare:w \l__regex_internal_a_int >
5111             \l__regex_internal_b_int
5112             \msg_error:nnee { regex } { backwards-quantifier }
5113             { \int_use:N \l__regex_internal_a_int }
5114             { \int_use:N \l__regex_internal_b_int }
5115             \int_zero:N \l__regex_internal_b_int
5116         \else:
5117             \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
5118         \fi:
5119         \exp_args:Noo \_regex_compile_quantifier_laziness:nnNN
5120             { \int_use:N \l__regex_internal_a_int }
5121             { \int_use:N \l__regex_internal_b_int }
5122     }
5123     {
5124         \_regex_compile_quantifier_abort:eNN
5125         {
5126             \c_left_brace_str
5127             \int_use:N \l__regex_internal_a_int ,
5128             \int_use:N \l__regex_internal_b_int
5129         }
5130         #1 #2
5131     }
5132 }

```

(End of definition for _regex_compile_quantifier_{:w and others.})

46.3.6 Raw characters

_regex_compile_raw_error:N Within character classes, and following catcode tests, some escaped alphanumeric sequences such as \b do not have any meaning. They are replaced by a raw character, after spitting out an error.

```

5133 \cs_new_protected:Npn \_regex_compile_raw_error:N #1
5134 {
5135     \msg_error:nne { regex } { bad-escape } {#1}
5136     \_regex_compile_raw:N #1
5137 }

```

(End of definition for _regex_compile_raw_error:N.)

_regex_compile_raw:N If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character #1 matches itself.

```

5138 \cs_new_protected:Npn \_regex_compile_raw:N #1#2#3
5139 {
5140     \_regex_if_in_class:TF

```

```

5141     {
5142     \_regex_two_if_eq:NNNTF #2 #3 \_regex_compile_special:N -
5143     { \_regex_compile_range:Nw #1 }
5144     {
5145     \_regex_compile_one:n
5146     { \_regex_item_equal:n { \int_value:w '#1 } }
5147     #2 #3
5148     }
5149     }
5150     {
5151     \_regex_compile_one:n
5152     { \_regex_item_equal:n { \int_value:w '#1 } }
5153     #2 #3
5154     }
5155     }

```

(End of definition for _regex_compile_raw:N.)

_regex_compile_range:Nw
_regex_if_end_range:NNTF

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```

5156 \cs_new_protected:Npn \_regex_if_end_range:NNTF #1#2
5157 {
5158   \if_meaning:w \_regex_compile_raw:N #1
5159   \else:
5160     \if_meaning:w \_regex_compile_special:N #1
5161     \if_charcode:w ] #2
5162     \use_i:nn
5163     \fi:
5164   \else:
5165     \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
5166     \fi:
5167   \fi:
5168   \use_i:nn
5169 }
5170 \cs_new_protected:Npn \_regex_compile_range:Nw #1#2#3
5171 {
5172   \_regex_if_end_range:NNTF #2 #3
5173   {
5174     \if_int_compare:w '#1 > '#3 \exp_stop_f:
5175     \msg_error:nnee { regex } { range-backwards } {#1} {#3}
5176   \else:
5177     \tl_build_put_right:Ne \l__regex_build_tl
5178     {
5179       \if_int_compare:w '#1 = '#3 \exp_stop_f:
5180       \_regex_item_equal:n
5181       \else:
5182         \_regex_item_range:nn { \int_value:w '#1 }
5183       \fi:
5184       { \int_value:w '#3 }
5185     }
5186   \fi:
5187 }
5188 {

```

```

5189     \msg_warning:nnee { regex } { range-missing-end }
5190     {#1} { \c_backslash_str #3 }
5191     \tl_build_put_right:Ne \l__regex_build_tl
5192     {
5193         \__regex_item_equal:n { \int_value:w '#1 \exp_stop_f: }
5194         \__regex_item_equal:n { \int_value:w '- \exp_stop_f: }
5195     }
5196     #2#3
5197 }
5198 }

```

(End of definition for `__regex_compile_range:Nw` and `__regex_if_end_range:NNTF`.)

46.3.7 Character properties

`__regex_compile_.`: In a class, the dot has no special meaning. Outside, insert `__regex_prop_.`, which matches any character or control sequence, and refuses `-2` (end-marker).

```

5199 \cs_new_protected:cpe { __regex_compile_.: }
5200 {
5201     \exp_not:N \__regex_if_in_class:TF
5202     { \__regex_compile_raw:N . }
5203     { \__regex_compile_one:n \exp_not:c { __regex_prop_.: } }
5204 }
5205 \cs_new_protected:cpn { __regex_prop_.: }
5206 {
5207     \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
5208     \exp_after:wN \__regex_break_true:w
5209     \fi:
5210 }

```

(End of definition for `__regex_compile_.` and `__regex_prop_.`.)

`__regex_compile_/d:` The constants `__regex_prop_d:`, etc. hold a list of tests which match the corresponding character class, and jump to the `__regex_break_point:TF` marker. As for a normal character, we check for quantifiers.

```

5211 \cs_set_protected:Npn \__regex_tmp:w #1#2
5212 {
5213     \cs_new_protected:cpe { __regex_compile_/#1: }
5214     { \__regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
5215     \cs_new_protected:cpe { __regex_compile_/#2: }
5216     {
5217         \__regex_compile_one:n
5218         { \__regex_item_reverse:n { \exp_not:c { __regex_prop_#1: } } }
5219     }
5220 }
5221 \__regex_tmp:w d D
5222 \__regex_tmp:w h H
5223 \__regex_tmp:w s S
5224 \__regex_tmp:w v V
5225 \__regex_tmp:w w W
5226 \cs_new_protected:cpn { __regex_compile_/N: }
5227 { \__regex_compile_one:n \__regex_prop_N: }

```

(End of definition for `__regex_compile_/d:` and others.)

46.3.8 Anchoring and simple assertions

```

\__regex_compile_anchor_letter:NNN
\__regex_compile_/A:
\__regex_compile_/G:
\__regex_compile_/Z:
\__regex_compile_/z:
\__regex_compile_/b:
\__regex_compile_/B:
\__regex_compile/^:
\__regex_compile/$:

```

In modes where assertions are forbidden, anchors such as \A produce an error (\A is invalid in classes); otherwise they add an __regex_assertion:Nn test as appropriate (the only negative assertion is \B). The test functions are defined later. The implementation for \$ and ^ is only different from \A etc because these are valid in a class.

```

5228 \cs_new_protected:Npn \__regex_compile_anchor_letter:NNN #1#2#3
5229 {
5230   \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw_error:N #1 }
5231   {
5232     \tl_build_put_right:Nn \l__regex_build_tl
5233     { \__regex_assertion:Nn #2 {#3} }
5234   }
5235 }
5236 \cs_new_protected:cpn { __regex_compile_/A: }
5237 { \__regex_compile_anchor_letter:NNN A \c_true_bool \__regex_A_test: }
5238 \cs_new_protected:cpn { __regex_compile_/G: }
5239 { \__regex_compile_anchor_letter:NNN G \c_true_bool \__regex_G_test: }
5240 \cs_new_protected:cpn { __regex_compile_/Z: }
5241 { \__regex_compile_anchor_letter:NNN Z \c_true_bool \__regex_Z_test: }
5242 \cs_new_protected:cpn { __regex_compile_/z: }
5243 { \__regex_compile_anchor_letter:NNN z \c_true_bool \__regex_Z_test: }
5244 \cs_new_protected:cpn { __regex_compile_/b: }
5245 { \__regex_compile_anchor_letter:NNN b \c_true_bool \__regex_b_test: }
5246 \cs_new_protected:cpn { __regex_compile_/B: }
5247 { \__regex_compile_anchor_letter:NNN B \c_false_bool \__regex_b_test: }
5248 \cs_set_protected:Npn \__regex_tmp:w #1#2
5249 {
5250   \cs_new_protected:cpn { __regex_compile_#1: }
5251   {
5252     \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw:N #1 }
5253     {
5254       \tl_build_put_right:Nn \l__regex_build_tl
5255       { \__regex_assertion:Nn \c_true_bool {#2} }
5256     }
5257   }
5258 }
5259 \exp_args:Ne \__regex_tmp:w { \iow_char:N \^ } { \__regex_A_test: }
5260 \exp_args:Ne \__regex_tmp:w { \iow_char:N \$ } { \__regex_Z_test: }

```

(End of definition for __regex_compile_anchor_letter:NNN and others.)

46.3.9 Character classes

__regex_compile_]: Outside a class, right brackets have no meaning. In a class, change the mode ($m \rightarrow (m - 15)/13$, truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of [...\cL[...]]). quantifiers.

```

5261 \cs_new_protected:cpn { __regex_compile_]: }
5262 {
5263   \__regex_if_in_class:TF
5264   {
5265     \if_int_compare:w \l__regex_mode_int >
5266     \c__regex_catcode_in_class_mode_int
5267     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }

```

```

5268     \fi:
5269     \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
5270     \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
5271     \if_int_odd:w \l__regex_mode_int \else:
5272         \exp_after:wN \__regex_compile_quantifier:w
5273     \fi:
5274 }
5275 { \__regex_compile_raw:N ] }
5276 }

```

(End of definition for `__regex_compile[:]`.)

`__regex_compile[:]` In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following `\c<category>`, we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, -2 and -6) just parse the class. The mode is updated later.

```

5277 \cs_new_protected:cpn { \__regex_compile[: ]
5278 {
5279     \__regex_if_in_class:TF
5280     { \__regex_compile_class_posix_test:w }
5281     {
5282         \__regex_if_within_catcode:TF
5283         {
5284             \exp_after:wN \__regex_compile_class_catcode:w
5285             \int_use:N \l__regex_catcodes_int ;
5286         }
5287         { \__regex_compile_class_normal:w }
5288     }
5289 }

```

(End of definition for `__regex_compile[:]`.)

`__regex_compile_class_normal:w` In the “normal” case, we insert `__regex_class:NnnnN <boolean>` in the compiled code. The `<boolean>` is true for positive classes, and false for negative classes, characterized by a leading `^`. The auxiliary `__regex_compile_class:TFNN` also checks for a leading `] which has a special meaning.`

```

5290 \cs_new_protected:Npn \__regex_compile_class_normal:w
5291 {
5292     \__regex_compile_class:TFNN
5293     { \__regex_class:NnnnN \c_true_bool }
5294     { \__regex_class:NnnnN \c_false_bool }
5295 }

```

(End of definition for `__regex_compile_class_normal:w`.)

`__regex_compile_class_catcode:w` This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting `__regex_item_catcode:nT` or the reverse variant as appropriate, each with the current catcodes bitmap #1 as an argument, and reset the catcodes.

```

5296 \cs_new_protected:Npn \__regex_compile_class_catcode:w #1;
5297 {
5298     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
5299     \tl_build_put_right:Nn \l__regex_build_tl

```

```

5300     { \_regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
5301   \fi:
5302   \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5303   \_regex_compile_class:TFNN
5304     { \_regex_item_catcode:nT {#1} }
5305     { \_regex_item_catcode_reverse:nT {#1} }
5306   }

```

(End of definition for _regex_compile_class_catcode:w.)

_regex_compile_class:TFNN If the first character is ^, then the class is negative (use #2), otherwise it is positive (use
_regex_compile_class:NN #1). If the next character is a right bracket, then it should be changed to a raw one.

```

5307   \cs_new_protected:Npn \_regex_compile_class:TFNN #1#2#3#4
5308     {
5309     \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
5310     \_regex_two_if_eq:NNNTF #3 #4 \_regex_compile_special:N ^
5311     {
5312     \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
5313     \_regex_compile_class:NN
5314     }
5315     {
5316     \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
5317     \_regex_compile_class:NN #3 #4
5318     }
5319   }
5320   \cs_new_protected:Npn \_regex_compile_class:NN #1#2
5321     {
5322     \token_if_eq_charcode:NNTF #2 ]
5323     { \_regex_compile_raw:N #2 }
5324     { #1 #2 }
5325   }

```

(End of definition for _regex_compile_class:TFNN and _regex_compile_class:NN.)

_regex_compile_class_posix_test:w Here we check for a syntax such as [:alpha:]. We also detect [= and [. which have a
_regex_compile_class_posix:NNNNw meaning in POSIX regular expressions, but are not implemented in l3regex. In case we
_regex_compile_class_posix_loop:w see [:, grab raw characters until hopefully reaching :]. If that's missing, or the POSIX
_regex_compile_class_posix_end:w class is unknown, abort. If all is right, add the test to the current class, with an extra
_regex_item_reverse:n for negative classes (we make sure to wrap its argument in
braces otherwise \regex_show:N would not recognize the regex as valid).

```

5326   \cs_new_protected:Npn \_regex_compile_class_posix_test:w #1#2
5327     {
5328     \token_if_eq_meaning:NNT \_regex_compile_special:N #1
5329     {
5330     \str_case:nn { #2 }
5331     {
5332     : { \_regex_compile_class_posix:NNNNw }
5333     = {
5334     \msg_warning:nne { regex }
5335     { posix-unsupported } { = }
5336     }
5337     . {
5338     \msg_warning:nne { regex }
5339     { posix-unsupported } { . }

```

```

5340     }
5341   }
5342 }
5343   \__regex_compile_raw:N [ #1 #2
5344 }
5345 \cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
5346 {
5347   \__regex_two_if_eq:NNNNTF #5 #6 \__regex_compile_special:N ^
5348   {
5349     \bool_set_false:N \l__regex_internal_bool
5350     \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
5351     \__regex_compile_class_posix_loop:w
5352   }
5353   {
5354     \bool_set_true:N \l__regex_internal_bool
5355     \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
5356     \__regex_compile_class_posix_loop:w #5 #6
5357   }
5358 }
5359 \cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
5360 {
5361   \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
5362   { #2 \__regex_compile_class_posix_loop:w }
5363   { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
5364 }
5365 \cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
5366 {
5367   \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N :
5368   { \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ] }
5369   { \use_ii:nn }
5370   {
5371     \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
5372     {
5373       \__regex_compile_one:n
5374       {
5375         \bool_if:NTF \l__regex_internal_bool \use:n \__regex_item_reverse:n
5376         { \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : } }
5377       }
5378     }
5379     {
5380       \msg_warning:nne { regex } { posix-unknown }
5381       { \l__regex_internal_a_tl }
5382       \__regex_compile_abort_tokens:e
5383       {
5384         [: \bool_if:NF \l__regex_internal_bool { ^ }
5385         \l__regex_internal_a_tl :]
5386       }
5387     }
5388   }
5389   {
5390     \msg_error:nnee { regex } { posix-missing-close }
5391     { [: \l__regex_internal_a_tl ] { #2 #4 }
5392     \__regex_compile_abort_tokens:e { [: \l__regex_internal_a_tl ]
5393     #1 #2 #3 #4

```

```

5394     }
5395 }

```

(End of definition for `__regex_compile_class_posix_test:w` and others.)

46.3.10 Groups and alternations

```

\__regex_compile_group_begin:N
\__regex_compile_group_end:

```

The contents of a regex group are turned into compiled code in `\l__regex_build_tl`, which ends up with items of the form `__regex_branch:n {⟨concatenation⟩}`. This construction is done using `\tl_build_...` functions within a T_EX group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument #1 is `__regex_group:nnnN` or a variant thereof. A small subtlety to support `\cL(abc)` as a shorthand for `(\cLa\cLb\cLc)`: exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```

5396 \cs_new_protected:Npn \__regex_compile_group_begin:N #1
5397 {
5398   \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
5399   \__regex_mode_quit_c:
5400   \group_begin:
5401     \tl_build_begin:N \l__regex_build_tl
5402     \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
5403     \int_incr:N \l__regex_group_level_int
5404     \tl_build_put_right:Nn \l__regex_build_tl
5405       { \__regex_branch:n { \if_false: } \fi: }
5406   }
5407 \cs_new_protected:Npn \__regex_compile_group_end:
5408 {
5409   \if_int_compare:w \l__regex_group_level_int > \c_zero_int
5410     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
5411     \tl_build_end:N \l__regex_build_tl
5412     \exp_args:NNNe
5413     \group_end:
5414     \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
5415     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5416     \exp_after:wN \__regex_compile_quantifier:w
5417   \else:
5418     \msg_warning:nn { regex } { extra-rparen }
5419     \exp_after:wN \__regex_compile_raw:N \exp_after:wN )
5420   \fi:
5421 }

```

(End of definition for `__regex_compile_group_begin:N` and `__regex_compile_group_end:.`)

```

\__regex_compile_(:

```

In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch `[a\cL(bcd)e]`. Otherwise check for a `?`, denoting special groups, and run the code for the corresponding special group.

```

5422 \cs_new_protected:cpn { __regex_compile_(: }
5423 {
5424   \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
5425   {
5426     \if_int_compare:w \l__regex_mode_int =

```



```

5427         \c__regex_catcode_in_class_mode_int
5428         \msg_error:nn { regex } { c-lparen-in-class }
5429         \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
5430         \else:
5431         \exp_after:wN \__regex_compile_lparen:w
5432         \fi:
5433     }
5434 }
5435 \cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
5436 {
5437     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ?
5438     {
5439         \cs_if_exist_use:cF
5440         { \__regex_compile_special_group\_token_to_str:N #4 :w }
5441         {
5442             \msg_warning:nne { regex } { special-group-unknown }
5443             { (? #4 }
5444             \__regex_compile_group_begin:N \__regex_group:nnnN
5445             \__regex_compile_raw:N ? #3 #4
5446         }
5447     }
5448     {
5449         \__regex_compile_group_begin:N \__regex_group:nnnN
5450         #1 #2 #3 #4
5451     }
5452 }

```

(End of definition for __regex_compile_(:))

`__regex_compile_|`: In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```

5453 \cs_new_protected:cpn { \__regex_compile_| }
5454 {
5455     \__regex_if_in_class:TF { \__regex_compile_raw:N | }
5456     {
5457         \tl_build_put_right:Nn \l__regex_build_tl
5458         { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
5459     }
5460 }

```

(End of definition for __regex_compile_|:))

`__regex_compile_)`: Within a class, parentheses are not special. Outside, close a group.

```

5461 \cs_new_protected:cpn { \__regex_compile_): }
5462 {
5463     \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
5464     { \__regex_compile_group_end: }
5465 }

```

(End of definition for __regex_compile_):))

`_regex_compile_special_group::w` and `_regex_compile_special_group_|:w`: Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```

5466 \cs_new_protected:cpn { \__regex_compile_special_group::w }
5467 { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }

```

```

5468 \cs_new_protected:cpn { __regex_compile_special_group_|:w }
5469 { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }

```

(End of definition for `__regex_compile_special_group_:w` and `__regex_compile_special_group_|:w`.)

```

\__regex_compile_special_group_i:w
\__regex_compile_special_group_-:w

```

The match can be made case-insensitive by setting the option with `(?i)`; the original behaviour is restored by `(?-i)`. This is the only supported option.

```

5470 \cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
5471 {
5472   \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N )
5473   {
5474     \cs_set:Npn \__regex_item_equal:n
5475       { \__regex_item_caseless_equal:n }
5476     \cs_set:Npn \__regex_item_range:nn
5477       { \__regex_item_caseless_range:nn }
5478   }
5479   {
5480     \msg_warning:nne { regex } { unknown-option } { (?i #2 }
5481     \__regex_compile_raw:N (
5482     \__regex_compile_raw:N ?
5483     \__regex_compile_raw:N i
5484     #1 #2
5485   }
5486 }
5487 \cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
5488 {
5489   \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_raw:N i
5490   { \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ) }
5491   { \use_ii:nn }
5492   {
5493     \cs_set:Npn \__regex_item_equal:n
5494       { \__regex_item_caseful_equal:n }
5495     \cs_set:Npn \__regex_item_range:nn
5496       { \__regex_item_caseful_range:nn }
5497   }
5498   {
5499     \msg_warning:nne { regex } { unknown-option } { (?-#2#4 }
5500     \__regex_compile_raw:N (
5501     \__regex_compile_raw:N ?
5502     \__regex_compile_raw:N -
5503     #1 #2 #3 #4
5504   }
5505 }

```

(End of definition for `__regex_compile_special_group_i:w` and `__regex_compile_special_group_-:w`.)

46.3.11 Catcodes and csnames

```

\__regex_compile_/c:
\__regex_compile_c_test:NN

```

The `\c` escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

```

5506 \cs_new_protected:cpn { __regex_compile_/c: }

```

```

5507 { \_regex_chk_c_allowed:T { \_regex_compile_c_test:NN } }
5508 \cs_new_protected:Npn \_regex_compile_c_test:NN #1#2
5509 {
5510   \token_if_eq_meaning:NNTF #1 \_regex_compile_raw:N
5511   {
5512     \int_if_exist:cTF { c__regex_catcode_#2_int }
5513     {
5514       \int_set_eq:Nc \l__regex_catcodes_int
5515       { c__regex_catcode_#2_int }
5516       \l__regex_mode_int
5517       = \if_case:w \l__regex_mode_int
5518         \c__regex_catcode_mode_int
5519         \else:
5520           \c__regex_catcode_in_class_mode_int
5521         \fi:
5522       \token_if_eq_charcode:NNT C #2 { \_regex_compile_c_C:NN }
5523     }
5524   }
5525   { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
5526   {
5527     \msg_error:nne { regex } { c-missing-category } {#2}
5528     #1 #2
5529   }
5530 }

```

(End of definition for _regex_compile_/c: and _regex_compile_c_test:NN.)

_regex_compile_c_C:NN If \cC is not followed by . or (...) then complain because that construction cannot match anything, except in cases like \cC[\c{...}], where it has no effect.

```

5531 \cs_new_protected:Npn \_regex_compile_c_C:NN #1#2
5532 {
5533   \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
5534   {
5535     \token_if_eq_charcode:NNTF #2 .
5536     { \use_none:n }
5537     { \token_if_eq_charcode:NNTF #2 ( ) % )
5538   }
5539   { \use:n }
5540   { \msg_error:nnn { regex } { c-C-invalid } {#2} }
5541   #1 #2
5542 }

```

(End of definition for _regex_compile_c_C:NN.)

_regex_compile_c[:w] When encountering \c[, the task is to collect uppercase letters representing character categories. First check for ^ which negates the list of category codes.

```

\_regex_compile_c_lbrack_loop:NN
\_regex_compile_c_lbrack_add:N
\_regex_compile_c_lbrack_end:
5543 \cs_new_protected:cpn { __regex_compile_c[:w] } #1#2
5544 {
5545   \l__regex_mode_int
5546   = \if_case:w \l__regex_mode_int
5547     \c__regex_catcode_mode_int
5548     \else:
5549       \c__regex_catcode_in_class_mode_int
5550     \fi:

```

```

5551 \int_zero:N \l__regex_catcodes_int
5552 \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ^
5553 {
5554   \bool_set_false:N \l__regex_catcodes_bool
5555   \__regex_compile_c_lbrack_loop:NN
5556 }
5557 {
5558   \bool_set_true:N \l__regex_catcodes_bool
5559   \__regex_compile_c_lbrack_loop:NN
5560   #1 #2
5561 }
5562 }
5563 \cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
5564 {
5565   \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
5566   {
5567     \int_if_exist:cTF { c__regex_catcode_#2_int }
5568     {
5569       \exp_args:Nc \__regex_compile_c_lbrack_add:N
5570         { c__regex_catcode_#2_int }
5571       \__regex_compile_c_lbrack_loop:NN
5572     }
5573   }
5574   {
5575     \token_if_eq_charcode:NNTF #2 ]
5576     { \__regex_compile_c_lbrack_end: }
5577   }
5578   {
5579     \msg_error:nne { regex } { c-missing-rbrack } {#2}
5580     \__regex_compile_c_lbrack_end:
5581     #1 #2
5582   }
5583 }
5584 \cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
5585 {
5586   \if_int_odd:w \__regex_int_eval:w \l__regex_catcodes_int / #1 \scan_stop:
5587   \else:
5588     \int_add:Nn \l__regex_catcodes_int {#1}
5589   \fi:
5590 }
5591 \cs_new_protected:Npn \__regex_compile_c_lbrack_end:
5592 {
5593   \if_meaning:w \c_false_bool \l__regex_catcodes_bool
5594   \int_set:Nn \l__regex_catcodes_int
5595   { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
5596   \fi:
5597 }

```

(End of definition for __regex_compile_c[:w and others.)

__regex_compile_c_{: The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting \c. Additionally, disable submatch tracking since groups don't escape the scope of \c{...}.

```

5598 \cs_new_protected:cpn { __regex_compile_c \c_left_brace_str :w }

```

```

5599 {
5600   \__regex_compile:w
5601   \__regex_disable_submatches:
5602   \l__regex_mode_int
5603   = \if_case:w \l__regex_mode_int
5604     \c__regex_cs_mode_int
5605   \else:
5606     \c__regex_cs_in_class_mode_int
5607   \fi:
5608 }

```

(End of definition for __regex_compile_c_{:})

__regex_compile_{: We forbid unescaped left braces inside a \c{...} escape because they otherwise lead to the confusing question of whether the first right brace in \c{{}x} should end \c or whether one should match braces.

```

5609 \cs_new_protected:cpn { __regex_compile_ \c_left_brace_str : }
5610 {
5611   \__regex_if_in_cs:TF
5612   { \msg_error:nnn { regex } { cu-lbrace } { c } }
5613   { \exp_after:wN \__regex_compile_raw:N \c_left_brace_str }
5614 }

```

(End of definition for __regex_compile_{:})

\l__regex_cs_flag Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: \c{[{}]} matches the control sequences \{ and \}. So, end compiling the inner regex (this closes any dangling class or group). __regex_compile_end_cs: Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use __regex_item_exact_cs:n with an argument consisting of all possibilities separated by \scan_stop:.

```

5615 \flag_new:N \l__regex_cs_flag
5616 \cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
5617 {
5618   \__regex_if_in_cs:TF
5619   { \__regex_compile_end_cs: }
5620   { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
5621 }
5622 \cs_new_protected:Npn \__regex_compile_end_cs:
5623 {
5624   \__regex_compile_end:
5625   \flag_clear:N \l__regex_cs_flag
5626   \__kernel_tl_set:Nx \l__regex_internal_a_tl
5627   {
5628     \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
5629     \q__regex_nil \q__regex_nil \q__regex_recursion_stop
5630   }
5631   \exp_args:Ne \__regex_compile_one:n
5632   {
5633     \flag_if_raised:NTF \l__regex_cs_flag
5634     { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
5635     {
5636       \__regex_item_exact_cs:n

```

```

5637         { \tl_tail:N \l__regex_internal_a_tl }
5638     }
5639 }
5640 }
5641 \cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
5642 {
5643     \cs_if_eq:NNTF #1 \__regex_branch:n
5644     {
5645         \scan_stop:
5646         \__regex_compile_cs_aux:NNnnN #2
5647         \q__regex_nil \q__regex_nil \q__regex_nil
5648         \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
5649         \__regex_compile_cs_aux:Nn
5650     }
5651     {
5652         \__regex_quark_if_nil:NF #1 { \flag_ensure_raised:N \l__regex_cs_flag }
5653         \__regex_use_none_delimit_by_q_recursion_stop:w
5654     }
5655 }
5656 \cs_new:Npn \__regex_compile_cs_aux:NNnnN #1#2#3#4#5#6
5657 {
5658     \bool_lazy_all:nTF
5659     {
5660         { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
5661         {#2}
5662         { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
5663         { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
5664         { \int_compare_p:nNn {#5} = \c_zero_int }
5665     }
5666     {
5667         \prg_replicate:nn {#4}
5668         { \char_generate:nn { \use_ii:nn #3 } {12} }
5669         \__regex_compile_cs_aux:NNnnN
5670     }
5671     {
5672         \__regex_quark_if_nil:NF #1
5673         {
5674             \flag_ensure_raised:N \l__regex_cs_flag
5675             \__regex_use_i_delimit_by_q_recursion_stop:nw
5676         }
5677         \__regex_use_none_delimit_by_q_recursion_stop:w
5678     }
5679 }

```

(End of definition for `\l__regex_cs_flag` and others.)

46.3.12 Raw token lists with `\u`

`__regex_compile_/u:` The `\u` escape is invalid in classes and directly following a catcode test. Otherwise test for a following `r` (for `\ur`), and call an auxiliary responsible for finding the variable name.

```

5680 \cs_new_protected:cpn { __regex_compile_/u: } #1#2
5681 {
5682     \__regex_if_in_class_or_catcode:TF
5683     { \__regex_compile_raw_error:N u #1 #2 }

```

```

5684     {
5685     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_raw:N r
5686     { \__regex_compile_u_brace:NNN \__regex_compile_ur_end: }
5687     { \__regex_compile_u_brace:NNN \__regex_compile_u_end: #1 #2 }
5688     }
5689 }

```

(End of definition for __regex_compile_/u.)

__regex_compile_u_brace:NNN This enforces the presence of a left brace, then starts a loop to find the variable name.

```

5690 \cs_new:Npn \__regex_compile_u_brace:NNN #1#2#3
5691 {
5692 \__regex_two_if_eq:NNNTF #2 #3 \__regex_compile_special:N \c_left_brace_str
5693 {
5694 \tl_set:Nn \l__regex_internal_b_tl {#1}
5695 \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
5696 \__regex_compile_u_loop:NN
5697 }
5698 {
5699 \msg_error:nn { regex } { u-missing-lbrace }
5700 \token_if_eq_meaning:NNTF #1 \__regex_compile_ur_end:
5701 { \__regex_compile_raw:N u \__regex_compile_raw:N r }
5702 { \__regex_compile_raw:N u }
5703 #2 #3
5704 }
5705 }

```

(End of definition for __regex_compile_u_brace:NNN.)

__regex_compile_u_loop:NN We collect the characters for the argument of \u within an e-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```

5706 \cs_new:Npn \__regex_compile_u_loop:NN #1#2
5707 {
5708 \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
5709 { #2 \__regex_compile_u_loop:NN }
5710 {
5711 \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
5712 {
5713 \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
5714 { \if_false: { \fi: } \l__regex_internal_b_tl }
5715 {
5716 \if_charcode:w \c_left_brace_str #2
5717 \msg_expandable_error:nnn { regex } { cu-lbrace } { u }
5718 \else:
5719 #2
5720 \fi:
5721 \__regex_compile_u_loop:NN
5722 }
5723 }
5724 {
5725 \if_false: { \fi: }

```

```

5726         \msg_error:nne { regex } { u-missing-rbrace } {#2}
5727         \l__regex_internal_b_tl
5728         #1 #2
5729     }
5730 }
5731 }

```

(End of definition for `__regex_compile_u_loop:NN`.)

`__regex_compile_ur_end:` For the `\ur{...}` construction, once we have extracted the variable's name, we replace
`__regex_compile_ur:n` all groups by non-capturing groups in the compiled regex (passed as the argument of
`__regex_compile_ur_aux:w` `__regex_compile_ur:n`). If that has a single branch (namely `\tl_if_empty:oTF` is
false) and there is no quantifier, then simply insert the contents of this branch (obtained
by `\use_ii:nn`, which is expanded later). In all other cases, insert a non-capturing group
and look for quantifiers to determine the number of repetition etc.

```

5732 \cs_new_protected:Npn \__regex_compile_ur_end:
5733 {
5734     \group_begin:
5735     \cs_set:Npn \__regex_group:nnnN { \__regex_group_no_capture:nnnN }
5736     \cs_set:Npn \__regex_group_resetting:nnnN { \__regex_group_no_capture:nnnN }
5737     \exp_args:NNe
5738     \group_end:
5739     \__regex_compile_ur:n { \use:c { \l__regex_internal_a_tl } }
5740 }
5741 \cs_new_protected:Npn \__regex_compile_ur:n #1
5742 {
5743     \tl_if_empty:oTF { \__regex_compile_ur_aux:w #1 {} ? ? \q__regex_nil }
5744     { \__regex_compile_if_quantifier:TFw }
5745     { \use_i:nn }
5746     {
5747         \tl_build_put_right:Nn \l__regex_build_tl
5748         { \__regex_group_no_capture:nnnN { \if_false: } \fi: #1 }
5749         \__regex_compile_quantifier:w
5750     }
5751     { \tl_build_put_right:Nn \l__regex_build_tl { \use_ii:nn #1 } }
5752 }
5753 \cs_new:Npn \__regex_compile_ur_aux:w \__regex_branch:n #1#2#3 \q__regex_nil {#2}

```

(End of definition for `__regex_compile_ur_end:`, `__regex_compile_ur:n`, and `__regex_compile_ur_aux:w`.)

`__regex_compile_u_end:` Once we have extracted the variable's name, we check for quantifiers, in which case we
`__regex_compile_u_payload:` set up a non-capturing group with a single branch. Inside this branch (we omit it and
the group if there is no quantifier), `__regex_compile_u_payload:` puts the right tests
corresponding to the contents of the variable, which we store in `\l__regex_internal_a_tl`. The behaviour of `\u` then depends on whether we are within a `\c{...}` escape (in
this case, the variable is turned to a string), or not.

```

5754 \cs_new_protected:Npn \__regex_compile_u_end:
5755 {
5756     \__regex_compile_if_quantifier:TFw
5757     {
5758         \tl_build_put_right:Nn \l__regex_build_tl
5759         {
5760             \__regex_group_no_capture:nnnN { \if_false: } \fi:

```



```

5761         \__regex_branch:n { \if_false: } \fi:
5762     }
5763     \__regex_compile_u_payload:
5764     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
5765     \__regex_compile_quantifier:w
5766 }
5767 { \__regex_compile_u_payload: }
5768 }
5769 \cs_new_protected:Npn \__regex_compile_u_payload:
5770 {
5771     \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
5772     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
5773     \__regex_compile_u_not_cs:
5774     \else:
5775     \__regex_compile_u_in_cs:
5776     \fi:
5777 }

```

(End of definition for __regex_compile_u_end: and __regex_compile_u_payload:.)

__regex_compile_u_in_cs: When \u appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```

5778 \cs_new_protected:Npn \__regex_compile_u_in_cs:
5779 {
5780     \__kernel_tl_gset:Nx \g__regex_internal_tl
5781     {
5782         \exp_args:No \__kernel_str_to_other_fast:n
5783         { \l__regex_internal_a_tl }
5784     }
5785     \tl_build_put_right:Ne \l__regex_build_tl
5786     {
5787         \tl_map_function:NN \g__regex_internal_tl
5788         \__regex_compile_u_in_cs_aux:n
5789     }
5790 }
5791 \cs_new:Npn \__regex_compile_u_in_cs_aux:n #1
5792 {
5793     \__regex_class:NnnnN \c_true_bool
5794     { \__regex_item_caseful_equal:n { \int_value:w '#1 } }
5795     { 1 } { 0 } \c_false_bool
5796 }

```

(End of definition for __regex_compile_u_in_cs:.)

__regex_compile_u_not_cs: In mode 0, the \u escape adds one state to the NFA for each token in \l__regex_internal_a_tl. If a given <token> is a control sequence, then insert a string comparison test, otherwise, __regex_item_exact:nn which compares catcode and character code.

```

5797 \cs_new_protected:Npn \__regex_compile_u_not_cs:
5798 {
5799     \tl_analysis_map_inline:Nn \l__regex_internal_a_tl
5800     {
5801         \tl_build_put_right:Ne \l__regex_build_tl
5802         {

```

```

5803     \__regex_class:NnnnN \c_true_bool
5804     {
5805         \if_int_compare:w "##3 = \c_zero_int
5806         \__regex_item_exact_cs:n
5807         { \exp_after:wN \cs_to_str:N ##1 }
5808         \else:
5809         \__regex_item_exact:nn { \int_value:w "##3 } { ##2 }
5810         \fi:
5811     }
5812     { 1 } { 0 } \c_false_bool
5813 }
5814 }
5815 }

```

(End of definition for __regex_compile_u_not_cs:.)

46.3.13 Other

__regex_compile_/K: The \K control sequence is currently the only “command”, which performs some action, rather than matching something. It is allowed in the same contexts as \b. At the compilation stage, we leave it as a single control sequence, defined later.

```

5816 \cs_new_protected:cpn { __regex_compile_/K: }
5817 {
5818     \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
5819     { \tl_build_put_right:Nn \l__regex_build_tl { \__regex_command_K: } }
5820     { \__regex_compile_raw_error:N K }
5821 }

```

(End of definition for __regex_compile_/K:.)

46.3.14 Showing regexes

Before showing a regex we check that it is “clean” in the sense that it has the correct internal structure. We do this (in the implementation of \regex_show:N and \regex_log:N) by comparing it with a cleaned-up version of the same regex. Along the way we also need similar functions for other types: all __regex_clean_(type):n functions produce valid <type> tokens (bool, explicit integer, etc.) from arbitrary input, and the output coincides with the input if that was valid.

```

5822 \cs_new:Npn \__regex_clean_bool:n #1
5823 {
5824     \tl_if_single:nTF {#1}
5825     { \bool_if:NTF #1 \c_true_bool \c_false_bool }
5826     { \c_true_bool }
5827 }
5828 \cs_new:Npn \__regex_clean_int:n #1
5829 {
5830     \tl_if_head_eq_meaning:nNTF {#1} -
5831     { - \exp_args:No \__regex_clean_int:n { \use_none:n #1 } }
5832     { \int_eval:n { 0 \str_map_function:nN {#1} \__regex_clean_int_aux:N } }
5833 }
5834 \cs_new:Npn \__regex_clean_int_aux:N #1
5835 {
5836     \if_int_compare:w \c_one_int < 1 #1 ~

```

```

5837     #1
5838     \else:
5839         \str_map_break:n
5840     \fi:
5841 }
5842 \cs_new:Npn \__regex_clean_regex:n #1
5843 {
5844     \__regex_clean_regex_loop:w #1
5845     \__regex_branch:n { \q_recursion_tail } \q_recursion_stop
5846 }
5847 \cs_new:Npn \__regex_clean_regex_loop:w #1 \__regex_branch:n #2
5848 {
5849     \quark_if_recursion_tail_stop:n {#2}
5850     \__regex_branch:n { \__regex_clean_branch:n {#2} }
5851     \__regex_clean_regex_loop:w
5852 }
5853 \cs_new:Npn \__regex_clean_branch:n #1
5854 {
5855     \__regex_clean_branch_loop:n #1
5856     ? ? ? ? ? \prg_break_point:
5857 }
5858 \cs_new:Npn \__regex_clean_branch_loop:n #1
5859 {
5860     \tl_if_single:nF {#1} \prg_break:
5861     \token_case_meaning:NnF #1
5862     {
5863         \__regex_command_K: { #1 \__regex_clean_branch_loop:n }
5864         \__regex_assertion:Nn { #1 \__regex_clean_assertion:Nn }
5865         \__regex_class:NnnnN { #1 \__regex_clean_class:NnnnN }
5866         \__regex_group:nnnN { #1 \__regex_clean_group:nnnN }
5867         \__regex_group_no_capture:nnnN { #1 \__regex_clean_group:nnnN }
5868         \__regex_group_resetting:nnnN { #1 \__regex_clean_group:nnnN }
5869     }
5870     \prg_break:
5871 }
5872 \cs_new:Npn \__regex_clean_assertion:Nn #1#2
5873 {
5874     \__regex_clean_bool:n {#1}
5875     \tl_if_single:nF {#2} { { \__regex_A_test: } \prg_break: }
5876     \token_case_meaning:NnTF #2
5877     {
5878         \__regex_A_test: { }
5879         \__regex_G_test: { }
5880         \__regex_Z_test: { }
5881         \__regex_b_test: { }
5882     }
5883     { {#2} }
5884     { { \__regex_A_test: } \prg_break: }
5885     \__regex_clean_branch_loop:n
5886 }
5887 \cs_new:Npn \__regex_clean_class:NnnnN #1#2#3#4#5
5888 {
5889     \__regex_clean_bool:n {#1}
5890     { \__regex_clean_class:n {#2} }

```

```

5891     { \int_max:nn \c_zero_int { \__regex_clean_int:n {#3} } }
5892     { \int_max:nn { -\c_one_int } { \__regex_clean_int:n {#4} } }
5893     \__regex_clean_bool:n {#5}
5894     \__regex_clean_branch_loop:n
5895   }
5896 \cs_new:Npn \__regex_clean_group:nnnN #1#2#3#4
5897   {
5898     { \__regex_clean_regex:n {#1} }
5899     { \int_max:nn \c_zero_int { \__regex_clean_int:n {#2} } }
5900     { \int_max:nn { -\c_one_int } { \__regex_clean_int:n {#3} } }
5901     \__regex_clean_bool:n {#4}
5902     \__regex_clean_branch_loop:n
5903   }
5904 \cs_new:Npn \__regex_clean_class:n #1
5905   { \__regex_clean_class_loop:nnn #1 ????? \prg_break_point: }

```

When cleaning a class there are many cases, including a dozen or so like `__regex_prop_d:` or `__regex_posix_alpha:`. To avoid listing all of them we allow any command that starts with the 13 characters `__regex_prop_` or `__regex_posix` (handily these have the same length, except for the trailing underscore).

```

5906 \cs_new:Npn \__regex_clean_class_loop:nnn #1#2#3
5907   {
5908     \tl_if_single:nF {#1} \prg_break:
5909     \token_case_meaning:NnTF #1
5910     {
5911       \__regex_item_cs:n { #1 { \__regex_clean_regex:n {#2} } }
5912       \__regex_item_exact_cs:n { #1 { \__regex_clean_exact_cs:n {#2} } }
5913       \__regex_item_caseful_equal:n { #1 { \__regex_clean_int:n {#2} } }
5914       \__regex_item_caseless_equal:n { #1 { \__regex_clean_int:n {#2} } }
5915       \__regex_item_reverse:n { #1 { \__regex_clean_class:n {#2} } }
5916     }
5917     { \__regex_clean_class_loop:nnn {#3} }
5918     {
5919       \token_case_meaning:NnTF #1
5920       {
5921         \__regex_item_caseful_range:nn { }
5922         \__regex_item_caseless_range:nn { }
5923         \__regex_item_exact:nn { }
5924       }
5925       {
5926         #1 { \__regex_clean_int:n {#2} } { \__regex_clean_int:n {#3} }
5927         \__regex_clean_class_loop:nnn
5928       }
5929       {
5930         \token_case_meaning:NnTF #1
5931         {
5932           \__regex_item_catcode:nT { }
5933           \__regex_item_catcode_reverse:nT { }
5934         }
5935         {
5936           #1 { \__regex_clean_int:n {#2} } { \__regex_clean_class:n {#3} }
5937           \__regex_clean_class_loop:nnn
5938         }
5939       }

```

```

5940         \exp_args:Ne \str_case:nnTF
5941         {
5942             \exp_args:Ne \str_range:nnn
5943             { \cs_to_str:N #1 } \c_one_int { 13 }
5944         }
5945         {
5946             { __regex_prop_ } { }
5947             { __regex_posix } { }
5948         }
5949         {
5950             #1
5951             \__regex_clean_class_loop:nnn {#2} {#3}
5952         }
5953         \prg_break:
5954     }
5955 }
5956 }
5957 }
5958 \cs_new:Npn \__regex_clean_exact_cs:n #1
5959 {
5960     \exp_last_unbraced:Nf \use_none:n
5961     {
5962         \__regex_clean_exact_cs:w #1
5963         \scan_stop: \q_recursion_tail \scan_stop:
5964         \q_recursion_stop
5965     }
5966 }
5967 \cs_new:Npn \__regex_clean_exact_cs:w #1 \scan_stop:
5968 {
5969     \quark_if_recursion_tail_stop:n {#1}
5970     \scan_stop: \tl_to_str:n {#1}
5971     \__regex_clean_exact_cs:w
5972 }

```

(End of definition for `__regex_clean_bool:n` and others.)

`__regex_show:N` Within a group and within `\tl_build_begin:N ... \tl_build_end:N` we redefine all the function that can appear in a compiled regex, then run the regex. The result stored in `\l__regex_internal_a_tl` is then meant to be shown.

```

5973 \cs_new_protected:Npn \__regex_show:N #1
5974 {
5975     \group_begin:
5976     \tl_build_begin:N \l__regex_build_tl
5977     \cs_set_protected:Npn \__regex_branch:n
5978     {
5979         \seq_pop_right:NN \l__regex_show_prefix_seq
5980         \l__regex_internal_a_tl
5981         \__regex_show_one:n { +-branch }
5982         \seq_put_right:No \l__regex_show_prefix_seq
5983         \l__regex_internal_a_tl
5984         \use:n
5985     }
5986     \cs_set_protected:Npn \__regex_group:nnnN
5987     { \__regex_show_group_aux:nnnnN { } }

```

```

5988 \cs_set_protected:Npn \__regex_group_no_capture:nnnN
5989 { \__regex_show_group_aux:nnnnN { ~(no-capture) } }
5990 \cs_set_protected:Npn \__regex_group_resetting:nnnN
5991 { \__regex_show_group_aux:nnnnN { ~(resetting) } }
5992 \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
5993 \cs_set_protected:Npn \__regex_command_K:
5994 { \__regex_show_one:n { reset~match~start~(\iow_char:N\K) } }
5995 \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
5996 {
5997   \__regex_show_one:n
5998   { \bool_if:NF ##1 { negative~ } assertion:~##2 }
5999 }
6000 \cs_set:Npn \__regex_b_test: { word-boundary }
6001 \cs_set:Npn \__regex_Z_test: { anchor~at~end~(\iow_char:N\Z) }
6002 \cs_set:Npn \__regex_A_test: { anchor~at~start~(\iow_char:N\A) }
6003 \cs_set:Npn \__regex_G_test: { anchor~at~start~of~match~(\iow_char:N\G) }
6004 \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
6005 { \__regex_show_one:n { char~code~\__regex_show_char:n{##1} } }
6006 \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
6007 {
6008   \__regex_show_one:n
6009   { range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}] }
6010 }
6011 \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
6012 { \__regex_show_one:n { char~code~\__regex_show_char:n{##1}~(caseless) } }
6013 \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
6014 {
6015   \__regex_show_one:n
6016   { Range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}]~(caseless) }
6017 }
6018 \cs_set_protected:Npn \__regex_item_catcode:nT
6019 { \__regex_show_item_catcode:NnT \c_true_bool }
6020 \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
6021 { \__regex_show_item_catcode:NnT \c_false_bool }
6022 \cs_set_protected:Npn \__regex_item_reverse:n
6023 { \__regex_show_scope:nn { Reversed~match } }
6024 \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
6025 { \__regex_show_one:n { char~\__regex_show_char:n{##2},~catcode~##1 } }
6026 \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
6027 \cs_set_protected:Npn \__regex_item_cs:n
6028 { \__regex_show_scope:nn { control~sequence } }
6029 \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any-token } }
6030 \seq_clear:N \l__regex_show_prefix_seq
6031 \__regex_show_push:n { ~ }
6032 \cs_if_exist_use:N #1
6033 \tl_build_end:N \l__regex_build_tl
6034 \exp_args:NNNo
6035 \group_end:
6036 \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
6037 }

```

(End of definition for __regex_show:N.)

__regex_show_char:n Show a single character, together with its ascii representation if available. This could be

extended beyond ascii. It is not ideal for parentheses themselves.

```

6038 \cs_new:Npn \__regex_show_char:n #1
6039 {
6040   \int_eval:n {#1}
6041   \int_compare:nT { 32 <= #1 <= 126 }
6042   { ~ ( \char_generate:nn {#1} {12} ) }
6043 }

```

(End of definition for __regex_show_char:n.)

`__regex_show_one:n` Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```

6044 \cs_new_protected:Npn \__regex_show_one:n #1
6045 {
6046   \int_incr:N \l__regex_show_lines_int
6047   \tl_build_put_right:Ne \l__regex_build_tl
6048   {
6049     \exp_not:N \iow_newline:
6050     \seq_map_function:NN \l__regex_show_prefix_seq \use:n
6051     #1
6052   }
6053 }

```

(End of definition for __regex_show_one:n.)

`__regex_show_push:n` Enter and exit levels of nesting. The scope function prints its first argument as an “introduction”, then performs its second argument in a deeper level of nesting.
`__regex_show_pop:`
`__regex_show_scope:nn`

```

6054 \cs_new_protected:Npn \__regex_show_push:n #1
6055 { \seq_put_right:Ne \l__regex_show_prefix_seq { #1 ~ } }
6056 \cs_new_protected:Npn \__regex_show_pop:
6057 { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
6058 \cs_new_protected:Npn \__regex_show_scope:nn #1#2
6059 {
6060   \__regex_show_one:n {#1}
6061   \__regex_show_push:n { ~ }
6062   #2
6063   \__regex_show_pop:
6064 }

```

(End of definition for __regex_show_push:n, __regex_show_pop:, and __regex_show_scope:nn.)

`__regex_show_group_aux:nnnnN` We display all groups in the same way, simply adding a message, (no capture) or (resetting), to special groups. The odd `\use_ii:nn` avoids printing a spurious `+-`branch for the first branch.

```

6065 \cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
6066 {
6067   \__regex_show_one:n { , -group~begin #1 }
6068   \__regex_show_push:n { | }
6069   \use_ii:nn #2
6070   \__regex_show_pop:
6071   \__regex_show_one:n
6072   { ‘-group~end \__regex_msg_repeated:nnN {#3} {#4} #5 }
6073 }

```

(End of definition for __regex_show_group_aux:nnnnN.)

`__regex_show_class:NnnnN` I'm entirely unhappy about this function: I couldn't find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write `Match` or `Don't match` on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That's clunky, but not too expensive, since it's only one test.

```

6074 \cs_new:Npn \__regex_show_class:NnnnN #1#2#3#4#5
6075 {
6076   \group_begin:
6077   \tl_build_begin:N \l__regex_build_tl
6078   \int_zero:N \l__regex_show_lines_int
6079   \__regex_show_push:n {~}
6080   #2
6081   \int_compare:nTF { \l__regex_show_lines_int = \c_zero_int }
6082   {
6083     \group_end:
6084     \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
6085   }
6086   {
6087     \bool_if:nTF
6088     { #1 && \int_compare_p:n { \l__regex_show_lines_int = \c_one_int } }
6089     {
6090       \group_end:
6091       #2
6092       \tl_build_put_right:Nn \l__regex_build_tl
6093       { \__regex_msg_repeated:nnN {#3} {#4} #5 }
6094     }
6095     {
6096       \tl_build_end:N \l__regex_build_tl
6097       \exp_args:NNNo
6098       \group_end:
6099       \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
6100       \__regex_show_one:n
6101       {
6102         \bool_if:NTF #1 { Match } { Don't-match }
6103         \__regex_msg_repeated:nnN {#3} {#4} #5
6104       }
6105       \tl_build_put_right:Ne \l__regex_build_tl
6106       { \exp_not:o \l__regex_internal_a_tl }
6107     }
6108   }
6109 }

```

(End of definition for `__regex_show_class:NnnnN`.)

`__regex_show_item_catcode:NnT` Produce a sequence of categories which the catcode bitmap #2 contains, and show it, indenting the tests on which this catcode constraint applies.

```

6110 \cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
6111 {
6112   \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
6113   \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
6114   { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
6115   \__regex_show_scope:nn
6116   {

```



```

6117     categories~
6118     \seq_map_function:NN \l__regex_internal_seq \use:n
6119     , ~
6120     \bool_if:NF #1 { negative~ } class
6121   }
6122 }

```

(End of definition for __regex_show_item_catcode:NnT.)

`__regex_show_item_exact_cs:n`

```

6123 \cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
6124   {
6125   \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
6126   \seq_set_map_e:NNn \l__regex_internal_seq
6127     \l__regex_internal_seq { \iow_char:N\##1 }
6128   \__regex_show_one:n
6129     { control~sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
6130   }

```

(End of definition for __regex_show_item_exact_cs:n.)

46.4 Building

46.4.1 Variables used while building

`\l__regex_min_state_int` The last state that was allocated is `\l__regex_max_state_int - 1`, so that `\l__regex_max_state_int` always points to a free state. The `min_state` variable is 1 to begin with, but gets shifted in nested calls to the matching code, namely in `\c{...}` constructions.

```

6131 \int_new:N \l__regex_min_state_int
6132 \int_set:Nn \l__regex_min_state_int { 1 }
6133 \int_new:N \l__regex_max_state_int

```

(End of definition for \l__regex_min_state_int and \l__regex_max_state_int.)

`\l__regex_left_state_int` Alternatives are implemented by branching from a `left` state into the various choices, then merging those into a `right` state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

```

6134 \int_new:N \l__regex_left_state_int
6135 \int_new:N \l__regex_right_state_int
6136 \seq_new:N \l__regex_left_state_seq
6137 \seq_new:N \l__regex_right_state_seq

```

(End of definition for \l__regex_left_state_int and others.)

`\l__regex_capturing_group_int` `\l__regex_capturing_group_int` is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering `resetting` groups.

```

6138 \int_new:N \l__regex_capturing_group_int

```

(End of definition for \l__regex_capturing_group_int.)

46.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a `\toks`. The operations which can appear in the `\toks` are

- `__regex_action_start_wildcard:N` $\langle boolean \rangle$ inserted at the start of the regular expression, where a `true` $\langle boolean \rangle$ makes it unanchored.
- `__regex_action_success:` marks the exit state of the NFA.
- `__regex_action_cost:n` $\{\langle shift \rangle\}$ is a transition from the current $\langle state \rangle$ to $\langle state \rangle + \langle shift \rangle$, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- `__regex_action_free:n` $\{\langle shift \rangle\}$, and `__regex_action_free_group:n` $\{\langle shift \rangle\}$ are free transitions, which immediately perform the actions for the state $\langle state \rangle + \langle shift \rangle$ of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the `group` variant must be used for transitions back to the start of a group.
- `__regex_action_submatch:nN` $\{\langle group \rangle\}$ $\langle key \rangle$ where the $\langle key \rangle$ is `<` or `>` for the beginning or end of group numbered $\langle group \rangle$. This causes the current position in the query to be stored as the $\langle key \rangle$ submatch boundary.
- One of these actions, within a conditional.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group - 1`, and if a group opened now it would be labelled `capturing_group`.
- The last allocated state is `max_state - 1`, so `max_state` is a free state.
- The `left_state` points to a state to the left of the current group or of the last class.
- The `right_state` points to a newly created, empty state, with some transitions leading to it.
- The `left/right` sequences hold a list of the corresponding end-points of nested groups.

```
\__regex_build:n The n-type function first compiles its argument. Reset some variables. Allocate two
\__regex_build_aux:Nn states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build
\__regex_build:N the regex within a (capturing) group numbered 0 (current value of capturing_group).
\__regex_build_aux:NN Finally, if the match reaches the last state, it is successful. A false boolean for argument
#1 for the auxiliaries will suppress the wildcard and make the match anchored: used for
\peek_regex:nTF and similar.
```

```
6139 \cs_new_protected:Npn \__regex_build:n
6140   { \__regex_build_aux:Nn \c_true_bool }
6141 \cs_new_protected:Npn \__regex_build:N
6142   { \__regex_build_aux:NN \c_true_bool }
6143 \cs_new_protected:Npn \__regex_build_aux:Nn #1#2
6144   {
6145     \__regex_compile:n {#2}
```

```

6146     \__regex_build_aux:NN #1 \l__regex_internal_regex
6147   }
6148 \cs_new_protected:Npn \__regex_build_aux:NN #1#2
6149   {
6150     \__regex_standard_escapechar:
6151     \int_zero:N \l__regex_capturing_group_int
6152     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
6153     \__regex_build_new_state:
6154     \__regex_build_new_state:
6155     \__regex_toks_put_right:Nn \l__regex_left_state_int
6156     { \__regex_action_start_wildcard:N #1 }
6157     \__regex_group:nnnN {#2} { 1 } { 0 } \c_false_bool
6158     \__regex_toks_put_right:Nn \l__regex_right_state_int
6159     { \__regex_action_success: }
6160   }

```

(End of definition for __regex_build:n and others.)

`\g__regex_case_int` Case number that was successfully matched in `\regex_match_case:nn` and related functions.

```

6161 \int_new:N \g__regex_case_int

```

(End of definition for `\g__regex_case_int`.)

`\l__regex_case_max_group_int` The largest group number appearing in any of the `<regex>` in the argument of `\regex_match_case:nn` and related functions.

```

6162 \int_new:N \l__regex_case_max_group_int

```

(End of definition for `\l__regex_case_max_group_int`.)

`__regex_case_build:n` See `__regex_build:n`, but with a loop.

```

\__regex_case_build:e
\__regex_case_build_aux:Nn
\__regex_case_build_loop:n
6163 \cs_new_protected:Npn \__regex_case_build:n #1
6164   {
6165     \__regex_case_build_aux:Nn \c_true_bool {#1}
6166     \int_gzero:N \g__regex_case_int
6167   }
6168 \cs_generate_variant:Nn \__regex_case_build:n { e }
6169 \cs_new_protected:Npn \__regex_case_build_aux:Nn #1#2
6170   {
6171     \__regex_standard_escapechar:
6172     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
6173     \__regex_build_new_state:
6174     \__regex_build_new_state:
6175     \__regex_toks_put_right:Nn \l__regex_left_state_int
6176     { \__regex_action_start_wildcard:N #1 }
6177     %
6178     \__regex_build_new_state:
6179     \__regex_toks_put_left:Ne \l__regex_left_state_int
6180     { \__regex_action_submatch:nN \c_zero_int < }
6181     \__regex_push_lr_states:
6182     \int_zero:N \l__regex_case_max_group_int
6183     \int_gzero:N \g__regex_case_int
6184     \tl_map_inline:nn {#2}
6185     {
6186       \int_gincr:N \g__regex_case_int

```

```

6187     \_regex_case_build_loop:n {##1}
6188   }
6189   \int_set_eq:NN \l__regex_capturing_group_int \l__regex_case_max_group_int
6190   \_regex_pop_lr_states:
6191 }
6192 \cs_new_protected:Npn \_regex_case_build_loop:n #1
6193 {
6194   \int_set_eq:NN \l__regex_capturing_group_int \c_one_int
6195   \_regex_compile_use:n {#1}
6196   \int_set:Nn \l__regex_case_max_group_int
6197     { \int_max:nn \l__regex_case_max_group_int \l__regex_capturing_group_int }
6198   \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
6199   \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
6200   \_regex_toks_put_left:Ne \l__regex_right_state_int
6201   {
6202     \_regex_action_submatch:nN \c_zero_int >
6203     \int_gset:Nn \g__regex_case_int
6204       { \int_use:N \g__regex_case_int }
6205     \_regex_action_success:
6206   }
6207   \_regex_toks_clear:N \l__regex_max_state_int
6208   \seq_push:No \l__regex_right_state_seq
6209     { \int_use:N \l__regex_max_state_int }
6210   \int_incr:N \l__regex_max_state_int
6211 }

```

(End of definition for `_regex_case_build:n`, `_regex_case_build_aux:Nn`, and `_regex_case_build_loop:n`.)

`_regex_build_for_cs:n` The matching code relies on some global intarray variables, but only uses a range of their entries. Specifically,

- `\g__regex_state_active_intarray` from `\l__regex_min_state_int` to `\l__regex_max_state_int`;

Here, in this nested call to the matching code, we need the new versions of this range to involve completely new entries of the intarray variables, so we begin by setting (the new) `\l__regex_min_state_int` to (the old) `\l__regex_max_state_int` to use higher entries.

When using a regex to match a cs, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate `left` and `right` states in their sequence.

```

6212 \cs_new_protected:Npn \_regex_build_for_cs:n #1
6213 {
6214   \int_set_eq:NN \l__regex_min_state_int \l__regex_max_state_int
6215   \_regex_build_new_state:
6216   \_regex_build_new_state:
6217   \_regex_push_lr_states:
6218   #1
6219   \_regex_pop_lr_states:
6220   \_regex_toks_put_right:Nn \l__regex_right_state_int
6221   {
6222     \if_int_compare:w -2 = \l__regex_curr_char_int

```

```

6223         \exp_after:wN \_regex_action_success:
6224         \fi:
6225     }
6226 }

```

(End of definition for `_regex_build_for_cs:n`.)

46.4.3 Helpers for building an nfa

`_regex_push_lr_states:` When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from T_EX's grouping.

`_regex_pop_lr_states:`

```

6227 \cs_new_protected:Npn \_regex_push_lr_states:
6228 {
6229     \seq_push:No \l_regex_left_state_seq
6230     { \int_use:N \l_regex_left_state_int }
6231     \seq_push:No \l_regex_right_state_seq
6232     { \int_use:N \l_regex_right_state_int }
6233 }
6234 \cs_new_protected:Npn \_regex_pop_lr_states:
6235 {
6236     \seq_pop:NN \l_regex_left_state_seq \l_regex_internal_a_tl
6237     \int_set:Nn \l_regex_left_state_int \l_regex_internal_a_tl
6238     \seq_pop:NN \l_regex_right_state_seq \l_regex_internal_a_tl
6239     \int_set:Nn \l_regex_right_state_int \l_regex_internal_a_tl
6240 }

```

(End of definition for `_regex_push_lr_states:` and `_regex_pop_lr_states:.`)

`_regex_build_transition_left:NNN`
`_regex_build_transition_right:nNn`

Add a transition from #2 to #3 using the function #1. The `left` function is used for higher priority transitions, and the `right` function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.

```

6241 \cs_new_protected:Npn \_regex_build_transition_left:NNN #1#2#3
6242 { \_regex_toks_put_left:Ne #2 { #1 { \tex_the:D \_regex_int_eval:w #3 - #2 } } }
6243 \cs_new_protected:Npn \_regex_build_transition_right:nNn #1#2#3
6244 { \_regex_toks_put_right:Ne #2 { #1 { \tex_the:D \_regex_int_eval:w #3 - #2 } } }

```

(End of definition for `_regex_build_transition_left:NNN` and `_regex_build_transition_right:nNn`.)

`_regex_build_new_state:`

Add a new empty state to the NFA. Then update the `left`, `right`, and `max` states, so that the `right` state is the new empty state, and the `left` state points to the previously “current” state.

```

6245 \cs_new_protected:Npn \_regex_build_new_state:
6246 {
6247     \_regex_toks_clear:N \l_regex_max_state_int
6248     \int_set_eq:NN \l_regex_left_state_int \l_regex_right_state_int
6249     \int_set_eq:NN \l_regex_right_state_int \l_regex_max_state_int
6250     \int_incr:N \l_regex_max_state_int
6251 }

```

(End of definition for `_regex_build_new_state:.`)

`__regex_build_transitions_laziness:NNNN`

This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by #1, true for lazy quantifiers, and false for greedy quantifiers.

```

6252 \cs_new_protected:Npn \__regex_build_transitions_laziness:NNNN #1#2#3#4#5
6253 {
6254   \__regex_build_new_state:
6255   \__regex_toks_put_right:Ne \l__regex_left_state_int
6256   {
6257     \if_meaning:w \c_true_bool #1
6258       #2 { \tex_the:D \__regex_int_eval:w #3 - \l__regex_left_state_int }
6259       #4 { \tex_the:D \__regex_int_eval:w #5 - \l__regex_left_state_int }
6260     \else:
6261       #4 { \tex_the:D \__regex_int_eval:w #5 - \l__regex_left_state_int }
6262       #2 { \tex_the:D \__regex_int_eval:w #3 - \l__regex_left_state_int }
6263     \fi:
6264   }
6265 }

```

(End of definition for `__regex_build_transitions_laziness:NNNN`.)

46.4.4 Building classes

`__regex_class:NnnnN`

`__regex_tests_action_cost:n`

The arguments are: `<boolean>` `{<tests>}` `{<min>}` `{<more>}` `<laziness>`. First store the tests with a trailing `__regex_action_cost:n`, in the true branch of `__regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer `<more>` is 0 for fixed repetitions, `-1` for unbounded repetitions, and `<max> - <min>` for a range of repetitions.

```

6266 \cs_new_protected:Npn \__regex_class:NnnnN #1#2#3#4#5
6267 {
6268   \cs_set:Npe \__regex_tests_action_cost:n ##1
6269   {
6270     \exp_not:n { \exp_not:n {#2} }
6271     \bool_if:NTF #1
6272     { \__regex_break_point:TF { \__regex_action_cost:n {##1} } { } }
6273     { \__regex_break_point:TF { } { \__regex_action_cost:n {##1} } }
6274   }
6275   \if_case:w - #4 \exp_stop_f:
6276     \__regex_class_repeat:n {#3}
6277   \or: \__regex_class_repeat:nN {#3} #5
6278   \else: \__regex_class_repeat:nnN {#3} {#4} #5
6279   \fi:
6280 }
6281 \cs_new:Npn \__regex_tests_action_cost:n { \__regex_action_cost:n }

```

(End of definition for `__regex_class:NnnnN` and `__regex_tests_action_cost:n`.)

`__regex_class_repeat:n`

This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for #1 = 0 repetitions: nothing is built.

```

6282 \cs_new_protected:Npn \__regex_class_repeat:n #1
6283 {
6284   \prg_replicate:nn {#1}
6285   {

```

```

6286     \__regex_build_new_state:
6287     \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
6288     \l__regex_left_state_int \l__regex_right_state_int
6289   }
6290 }

```

(End of definition for __regex_class_repeat:n.)

__regex_class_repeat:nN This implements unbounded repetitions of a single class (e.g. the * and + quantifiers). If the minimum number #1 of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call __regex_class_repeat:n for the code to match #1 repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the laziness boolean #2.

```

6291 \cs_new_protected:Npn \__regex_class_repeat:nN #1#2
6292 {
6293   \if_int_compare:w #1 = \c_zero_int
6294     \__regex_build_transitions_laziness:NNNNN #2
6295     \__regex_action_free:n \l__regex_right_state_int
6296     \__regex_tests_action_cost:n \l__regex_left_state_int
6297   \else:
6298     \__regex_class_repeat:n {#1}
6299     \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
6300     \__regex_build_transitions_laziness:NNNNN #2
6301     \__regex_action_free:n \l__regex_right_state_int
6302     \__regex_action_free:n \l__regex_internal_a_int
6303   \fi:
6304 }

```

(End of definition for __regex_class_repeat:nN.)

__regex_class_repeat:nnN We want to build the code to match from #1 to #1+#2 repetitions. Match #1 repetitions (can be 0). Compute the final state of the next construction as a. Build #2 > 0 states, each with a transition to the next state governed by the tests, and a transition to the final state a. The computation of a is safe because states are allocated in order, starting from max_state.

```

6305 \cs_new_protected:Npn \__regex_class_repeat:nnN #1#2#3
6306 {
6307   \__regex_class_repeat:n {#1}
6308   \int_set:Nn \l__regex_internal_a_int
6309   { \l__regex_max_state_int + #2 - \c_one_int }
6310   \prg_replicate:nn { #2 }
6311   {
6312     \__regex_build_transitions_laziness:NNNNN #3
6313     \__regex_action_free:n \l__regex_internal_a_int
6314     \__regex_tests_action_cost:n \l__regex_right_state_int
6315   }
6316 }

```

(End of definition for __regex_class_repeat:nnN.)

46.4.5 Building groups

`_regex_group_aux:nnnnN` Arguments: $\langle label \rangle$ $\langle contents \rangle$ $\langle min \rangle$ $\langle more \rangle$ $\langle laziness \rangle$. If $\langle min \rangle$ is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the `left_state` is the left end of the group, from which all branches stem, and the `right_state` is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents $\#2$ of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly $\#3$ times, or $\#3$ or more times, or between $\#3$ and $\#3 + \#4$ times, with laziness $\#5$. The $\langle label \rangle$ $\#1$ is used for submatch tracking. Each of the three auxiliaries expects `left_state` and `right_state` to be set properly.

```

6317 \cs_new_protected:Npn \_regex_group_aux:nnnnN #1#2#3#4#5
6318   {
6319     \if_int_compare:w #3 = \c_zero_int
6320       \_regex_build_new_state:
6321       \_regex_build_transition_right:nNn \_regex_action_free_group:n
6322       \l__regex_left_state_int \l__regex_right_state_int
6323     \fi:
6324     \_regex_build_new_state:
6325     \_regex_push_lr_states:
6326     #2
6327     \_regex_pop_lr_states:
6328     \if_case:w - #4 \exp_stop_f:
6329       \_regex_group_repeat:nn {#1} {#3}
6330     \or: \_regex_group_repeat:nnN {#1} {#3} #5
6331     \else: \_regex_group_repeat:nnnN {#1} {#3} {#4} #5
6332     \fi:
6333   }

```

(End of definition for `_regex_group_aux:nnnnN`.)

`_regex_group:nnnN` Hand to `_regex_group_aux:nnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

`_regex_group_no_capture:nnnN`

```

6334 \cs_new_protected:Npn \_regex_group:nnnN #1
6335   {
6336     \exp_args:No \_regex_group_aux:nnnnN
6337     { \int_use:N \l__regex_capturing_group_int }
6338     {
6339       \int_incr:N \l__regex_capturing_group_int
6340       #1
6341     }
6342   }
6343 \cs_new_protected:Npn \_regex_group_no_capture:nnnN
6344   { \_regex_group_aux:nnnnN { -1 } }

```

(End of definition for `_regex_group:nnnN` and `_regex_group_no_capture:nnnN`.)

`_regex_group_resetting:nnnN`
`_regex_group_resetting_loop:nnnN`

Again, hand the label -1 to `_regex_group_aux:nnnnN`, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form `_regex_branch:n` $\langle branch \rangle$.

```

6345 \cs_new_protected:Npn \_regex_group_resetting:nnnN #1

```



```

6346 {
6347   \__regex_group_aux:nnnnN { -1 }
6348   {
6349     \exp_args:Noo \__regex_group_resetting_loop:nnNn
6350     { \int_use:N \l__regex_capturing_group_int }
6351     { \int_use:N \l__regex_capturing_group_int }
6352     #1
6353     { ?? \prg_break:n } { }
6354     \prg_break_point:
6355   }
6356 }
6357 \cs_new_protected:Npn \__regex_group_resetting_loop:nnNn #1#2#3#4
6358 {
6359   \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
6360   \int_set:Nn \l__regex_capturing_group_int {#2}
6361   #3 {#4}
6362   \exp_args:Ne \__regex_group_resetting_loop:nnNn
6363   { \int_max:nn {#1} \l__regex_capturing_group_int }
6364   {#2}
6365 }

```

(End of definition for `__regex_group_resetting:nnnN` and `__regex_group_resetting_loop:nnNn`.)

`__regex_branch:n` Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```

6366 \cs_new_protected:Npn \__regex_branch:n #1
6367 {
6368   \__regex_build_new_state:
6369   \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
6370   \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
6371   \__regex_build_transition_right:nNn \__regex_action_free:n
6372   \l__regex_left_state_int \l__regex_right_state_int
6373   #1
6374   \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
6375   \__regex_build_transition_right:nNn \__regex_action_free:n
6376   \l__regex_right_state_int \l__regex_internal_a_tl
6377 }

```

(End of definition for `__regex_branch:n`.)

`__regex_group_repeat:nn` This function is called to repeat a group a fixed number of times `#2`; if this is 0 we remove the group altogether (but don't reset the `capturing_group` label). Otherwise, the auxiliary `__regex_group_repeat_aux:n` copies `#2` times the `\toks` for the group, and leaves `internal_a` pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```

6378 \cs_new_protected:Npn \__regex_group_repeat:nn #1#2
6379 {
6380   \if_int_compare:w #2 = \c_zero_int
6381     \int_set:Nn \l__regex_max_state_int
6382     { \l__regex_left_state_int - \c_one_int }
6383     \__regex_build_new_state:

```

```

6384     \else:
6385         \__regex_group_repeat_aux:n {#2}
6386         \__regex_group_submatches:nnN {#1}
6387         \l__regex_internal_a_int \l__regex_right_state_int
6388         \__regex_build_new_state:
6389     \fi:
6390 }

```

(End of definition for `__regex_group_repeat:nn`.)

`__regex_group_submatches:nnN` This inserts in states #2 and #3 the code for tracking submatches of the group #1, unless inhibited by a label of -1.

```

6391 \cs_new_protected:Npn \__regex_group_submatches:nnN #1#2#3
6392 {
6393     \if_int_compare:w #1 > - \c_one_int
6394         \__regex_toks_put_left:Ne #2 { \__regex_action_submatch:n {#1} < }
6395         \__regex_toks_put_left:Ne #3 { \__regex_action_submatch:n {#1} > }
6396     \fi:
6397 }

```

(End of definition for `__regex_group_submatches:nnN`.)

`__regex_group_repeat_aux:n` Here we repeat `\toks` ranging from `left_state` to `max_state`, #1 > 0 times. First add a transition so that the copies “chain” properly. Compute the shift `c` between the original copy and the last copy we want. Shift the `right_state` and `max_state` to their final values. We then want to perform `c` copy operations. At the end, `b` is equal to the `max_state`, and `a` points to the left of the last copy of the group.

```

6398 \cs_new_protected:Npn \__regex_group_repeat_aux:n #1
6399 {
6400     \__regex_build_transition_right:nNn \__regex_action_free:n
6401     \l__regex_right_state_int \l__regex_max_state_int
6402     \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
6403     \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
6404     \if_int_compare:w \__regex_int_eval:w #1 > \c_one_int
6405         \int_set:Nn \l__regex_internal_c_int
6406         {
6407             ( #1 - \c_one_int )
6408             * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
6409         }
6410     \int_add:Nn \l__regex_right_state_int \l__regex_internal_c_int
6411     \int_add:Nn \l__regex_max_state_int \l__regex_internal_c_int
6412     \__regex_toks_memcpy:NNn
6413     \l__regex_internal_b_int
6414     \l__regex_internal_a_int
6415     \l__regex_internal_c_int
6416     \fi:
6417 }

```

(End of definition for `__regex_group_repeat_aux:n`.)

`__regex_group_repeat:nnN` This function is called to repeat a group at least `n` times; the case `n = 0` is very different from `n > 0`. Assume first that `n = 0`. Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the “true” left state `a`

(remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from `a` to a new state.

Now consider the case $n > 0$. Repeat the group n times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from `__regex_group_repeat_aux:n`.

```

6418 \cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
6419 {
6420   \if_int_compare:w #2 = \c_zero_int
6421     \__regex_group_submatches:nnN {#1}
6422     \l__regex_left_state_int \l__regex_right_state_int
6423     \int_set:Nn \l__regex_internal_a_int
6424     { \l__regex_left_state_int - \c_one_int }
6425     \__regex_build_transition_right:nNn \__regex_action_free:n
6426     \l__regex_right_state_int \l__regex_internal_a_int
6427     \__regex_build_new_state:
6428     \if_meaning:w \c_true_bool #3
6429       \__regex_build_transition_left:NNN \__regex_action_free:n
6430       \l__regex_internal_a_int \l__regex_right_state_int
6431     \else:
6432       \__regex_build_transition_right:nNn \__regex_action_free:n
6433       \l__regex_internal_a_int \l__regex_right_state_int
6434     \fi:
6435   \else:
6436     \__regex_group_repeat_aux:n {#2}
6437     \__regex_group_submatches:nnN {#1}
6438     \l__regex_internal_a_int \l__regex_right_state_int
6439     \if_meaning:w \c_true_bool #3
6440       \__regex_build_transition_right:nNn \__regex_action_free_group:n
6441       \l__regex_right_state_int \l__regex_internal_a_int
6442     \else:
6443       \__regex_build_transition_left:NNN \__regex_action_free_group:n
6444       \l__regex_right_state_int \l__regex_internal_a_int
6445     \fi:
6446     \__regex_build_new_state:
6447   \fi:
6448 }

```

(End of definition for `__regex_group_repeat:nnN`.)

`__regex_group_repeat:nnnN`

We wish to repeat the group between `#2` and `#2 + #3` times, with a laziness controlled by `#4`. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first `#2` copies of the group, but that forces us to treat specially the case `#2 = 0`. Repeat that group with submatch tracking `#2 + #3` times (the maximum number of repetitions). Then our goal is to add `#3` transitions from the end of the `#2`-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with `#2 = 0`, the transition which skips over all

copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it “by hand” earlier.

```

6449 \cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
6450 {
6451   \__regex_group_submatches:nNN {#1}
6452   \l__regex_left_state_int \l__regex_right_state_int
6453   \__regex_group_repeat_aux:n { #2 + #3 }
6454   \if_meaning:w \c_true_bool #4
6455     \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
6456     \prg_replicate:nn { #3 }
6457     {
6458       \int_sub:Nn \l__regex_left_state_int
6459       { \l__regex_internal_b_int - \l__regex_internal_a_int }
6460       \__regex_build_transition_left:NNN \__regex_action_free:n
6461       \l__regex_left_state_int \l__regex_max_state_int
6462     }
6463   \else:
6464     \prg_replicate:nn { #3 - \c_one_int }
6465     {
6466       \int_sub:Nn \l__regex_right_state_int
6467       { \l__regex_internal_b_int - \l__regex_internal_a_int }
6468       \__regex_build_transition_right:nNn \__regex_action_free:n
6469       \l__regex_right_state_int \l__regex_max_state_int
6470     }
6471     \if_int_compare:w #2 = \c_zero_int
6472       \int_set:Nn \l__regex_right_state_int
6473       { \l__regex_left_state_int - \c_one_int }
6474     \else:
6475       \int_sub:Nn \l__regex_right_state_int
6476       { \l__regex_internal_b_int - \l__regex_internal_a_int }
6477     \fi:
6478     \__regex_build_transition_right:nNn \__regex_action_free:n
6479     \l__regex_right_state_int \l__regex_max_state_int
6480   \fi:
6481   \__regex_build_new_state:
6482 }

```

(End of definition for __regex_group_repeat:nnnN.)

46.4.6 Others

__regex_assertion:Nn Usage: __regex_assertion:Nn <boolean> {<test>}, where the <test> is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The __regex_b_test: test is used by the \b and \B escape: check if the last character was a word character or not, and do the same to the current character. The __regex_A_test: boundary-markers of the string are non-word characters for this purpose. The __regex_G_test: test is used by the \G escape: check if the last character was a group character or not, and do the same to the current character. The __regex_Z_test: test is used by the \Z escape: check if the last character was a zero-width space character or not, and do the same to the current character.

```

6483 \cs_new_protected:Npn \__regex_assertion:Nn #1#2
6484 {
6485   \__regex_build_new_state:
6486   \__regex_toks_put_right:Ne \l__regex_left_state_int
6487   {
6488     \exp_not:n {#2}
6489     \__regex_break_point:TF

```

```

6490     \bool_if:NF #1 { { } }
6491     {
6492       \__regex_action_free:n
6493       {
6494         \tex_the:D \__regex_int_eval:w
6495         \l__regex_right_state_int - \l__regex_left_state_int
6496       }
6497     }
6498     \bool_if:NT #1 { { } }
6499   }
6500 }
6501 \cs_new_protected:Npn \__regex_b_test:
6502 {
6503   \group_begin:
6504   \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
6505   \__regex_prop_w:
6506   \__regex_break_point:TF
6507   { \group_end: \__regex_item_reverse:n { \__regex_prop_w: } }
6508   { \group_end: \__regex_prop_w: }
6509 }
6510 \cs_new_protected:Npn \__regex_Z_test:
6511 {
6512   \if_int_compare:w -2 = \l__regex_curr_char_int
6513   \exp_after:wN \__regex_break_true:w
6514   \fi:
6515 }
6516 \cs_new_protected:Npn \__regex_A_test:
6517 {
6518   \if_int_compare:w -2 = \l__regex_last_char_int
6519   \exp_after:wN \__regex_break_true:w
6520   \fi:
6521 }
6522 \cs_new_protected:Npn \__regex_G_test:
6523 {
6524   \if_int_compare:w \l__regex_curr_pos_int = \l__regex_start_pos_int
6525   \exp_after:wN \__regex_break_true:w
6526   \fi:
6527 }

```

(End of definition for `__regex_assertion:Nn` and others.)

`__regex_command_K:` Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```

6528 \cs_new_protected:Npn \__regex_command_K:
6529 {
6530   \__regex_build_new_state:
6531   \__regex_toks_put_right:Ne \l__regex_left_state_int
6532   {
6533     \__regex_action_submatch:nN \c_zero_int <
6534     \bool_set_true:N \l__regex_fresh_thread_bool
6535     \__regex_action_free:n
6536     {
6537       \tex_the:D \__regex_int_eval:w
6538       \l__regex_right_state_int - \l__regex_left_state_int

```

```

6539     }
6540     \bool_set_false:N \l__regex_fresh_thread_bool
6541   }
6542 }

```

(End of definition for `__regex_command_K:`.)

46.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains “free” transitions to other states, and transitions which “consume” the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of “active states”, stored in `\g__regex_thread_info_intarray` (together with submatch information): this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA “collide” in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching `(a?)*` against `a` is broken, isn’t it? No. The group first matches `a`, as it should, then repeats; it attempts to match `a` again but fails; it skips `a`, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) `a`. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing “normal” free transitions `__regex_action_free:n` from transitions `__regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a “completed” thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

46.5.1 Variables used when matching

```

\l__regex_min_pos_int
\l__regex_max_pos_int
\l__regex_curr_pos_int
\l__regex_start_pos_int
\l__regex_success_pos_int

```

The tokens in the query are indexed from `min_pos` for the first to `max_pos - 1` for the last, and their information is stored in several arrays and `\toks` registers with those numbers. We match without backtracking, keeping all threads in lockstep at the `curr_pos` in the query. The starting point of the current match attempt is `start_pos`, and `success_pos`, updated whenever a thread succeeds, is used as the next starting position.

```

6543 \int_new:N \l__regex_min_pos_int
6544 \int_new:N \l__regex_max_pos_int
6545 \int_new:N \l__regex_curr_pos_int
6546 \int_new:N \l__regex_start_pos_int
6547 \int_new:N \l__regex_success_pos_int

```

(End of definition for `\l__regex_min_pos_int` and others.)

`\l__regex_curr_char_int` The character and category codes of the token at the current position and a token list
`\l__regex_curr_catcode_int` expanding to that token; the character code of the token at the previous position; the
`\l__regex_curr_token_tl` character code of the token just before a successful match; and the character code of the
`\l__regex_last_char_int` result of changing the case of the current token (A-Z↔a-z). This last integer is only
`\l__regex_last_char_success_int` computed when necessary, and is otherwise `\c_max_int`. The `curr_char` variable is also
`\l__regex_case_changed_char_int` used in various other phases to hold a character code.

```
6548 \int_new:N \l__regex_curr_char_int
6549 \int_new:N \l__regex_curr_catcode_int
6550 \tl_new:N \l__regex_curr_token_tl
6551 \int_new:N \l__regex_last_char_int
6552 \int_new:N \l__regex_last_char_success_int
6553 \int_new:N \l__regex_case_changed_char_int
```

(End of definition for `\l__regex_curr_char_int` and others.)

`\l__regex_curr_state_int` For every character in the token list, each of the active states is considered in turn.
The variable `\l__regex_curr_state_int` holds the state of the NFA which is currently
considered: transitions are then given as shifts relative to the current state.

```
6554 \int_new:N \l__regex_curr_state_int
```

(End of definition for `\l__regex_curr_state_int`.)

`\l__regex_curr_submatches_tl` The submatches for the thread which is currently active are stored in the `curr_-`
`\l__regex_success_submatches_tl` `submatches` list, which is almost a comma list, but ends with a comma. This list is stored
by `__regex_store_state:n` into an intarray variable, to be retrieved when matching at
the next position. When a thread succeeds, this list is copied to `\l__regex_success_-`
`submatches_tl`: only the last successful thread remains there.

```
6555 \tl_new:N \l__regex_curr_submatches_tl
6556 \tl_new:N \l__regex_success_submatches_tl
```

(End of definition for `\l__regex_curr_submatches_tl` and `\l__regex_success_submatches_tl`.)

`\l__regex_step_int` This integer, always even, is increased every time a character in the query is read, and not
reset when doing multiple matches. We store in `\g__regex_state_active_intarray` the
last step in which each `<state>` in the NFA was encountered. This lets us break infinite
loops by not visiting the same state twice in the same step. In fact, the step we store
is equal to `step` when we have started performing the operations of `\toks<state>`, but
not finished yet. However, once we finish, we store `step + 1` in `\g__regex_state_-`
`active_intarray`. This is needed to track submatches properly (see building phase).
The `step` is also used to attach each set of submatch information to a given iteration
(and automatically discard it when it corresponds to a past step).

```
6557 \int_new:N \l__regex_step_int
```

(End of definition for `\l__regex_step_int`.)

`\l__regex_min_thread_int` All the currently active threads are kept in order of precedence in `\g__regex_thread_-`
`\l__regex_max_thread_int` `info_intarray` together with the corresponding submatch information. Data in this
intarray is organized as blocks from `min_thread` (included) to `max_thread` (excluded).
At the start of every step, the whole array is unpacked, so that the space can immediately
be reused, and `max_thread` is reset to `min_thread`, effectively clearing the array.

```
6558 \int_new:N \l__regex_min_thread_int
6559 \int_new:N \l__regex_max_thread_int
```

(End of definition for `\l__regex_min_thread_int` and `\l__regex_max_thread_int`.)

`\g__regex_state_active_intarray` `\g__regex_thread_info_intarray` stores the last *⟨step⟩* in which each *⟨state⟩* was active. `\g__regex_thread_info_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
6560 \intarray_new:Nn \g__regex_state_active_intarray { 65536 }
6561 \intarray_new:Nn \g__regex_thread_info_intarray { 65536 }
```

(End of definition for `\g__regex_state_active_intarray` and `\g__regex_thread_info_intarray`.)

`\l__regex_matched_analysis_tl` `\l__regex_curr_analysis_tl` The list `\l__regex_curr_analysis_tl` consists of a brace group containing three brace groups corresponding to the current token, with the same syntax as `\tl_analysis_map_inline:nn`. The list `\l__regex_matched_analysis_tl` (constructed under the `tl_build` machinery) has one item for each token that has already been treated so far in a given match attempt: each item consists of three brace groups with the same syntax as `\tl_analysis_map_inline:nn`.

```
6562 \tl_new:N \l__regex_matched_analysis_tl
6563 \tl_new:N \l__regex_curr_analysis_tl
```

(End of definition for `\l__regex_matched_analysis_tl` and `\l__regex_curr_analysis_tl`.)

`\l__regex_every_match_tl` Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See `__regex_single_match:` and `__regex_multi_match:n`.

```
6564 \tl_new:N \l__regex_every_match_tl
```

(End of definition for `\l__regex_every_match_tl`.)

`\l__regex_fresh_thread_bool` `\l__regex_empty_success_bool` `__regex_if_two_empty_matches:F` When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting `\l__regex_fresh_thread_bool` to `true` for threads which directly come from the start of the regex or from the `\K` command, and testing that boolean whenever a thread succeeds. The function `__regex_if_two_empty_matches:F` is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is `\use:n`.

```
6565 \bool_new:N \l__regex_fresh_thread_bool
6566 \bool_new:N \l__regex_empty_success_bool
6567 \cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(End of definition for `\l__regex_fresh_thread_bool`, `\l__regex_empty_success_bool`, and `__regex_if_two_empty_matches:F`.)

`\g__regex_success_bool` `\l__regex_saved_success_bool` `\l__regex_match_success_bool` The boolean `\l__regex_match_success_bool` is true if the current match attempt was successful, and `\g__regex_success_bool` is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with `\c{...}`. This is done by saving the global variable into `\l__regex_saved_success_bool`, which is local, hence not affected by the changes due to inner regex functions.

```
6568 \bool_new:N \g__regex_success_bool
6569 \bool_new:N \l__regex_saved_success_bool
6570 \bool_new:N \l__regex_match_success_bool
```


(End of definition for `\g__regex_success_bool`, `\l__regex_saved_success_bool`, and `\l__regex-match_success_bool`.)

46.5.2 Matching: framework

`__regex_match:n` Initialize the variables that should be set once for each user function (even for multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (`\g__regex_state_active_intarray`), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

```

6571 \cs_new_protected:Npn \__regex_match:n #1
6572   {
6573     \__regex_match_init:
6574     \__regex_match_once_init:
6575     \tl_analysis_map_inline:nn {#1}
6576     { \__regex_match_one_token:nnN {##1} {##2} ##3 }
6577     \__regex_match_one_token:nnN { } { -2 } F
6578     \prg_break_point:Nn \__regex_maplike_break: { }
6579   }
6580 \cs_new_protected:Npn \__regex_match_cs:n #1
6581   {
6582     \int_set_eq:NN \l__regex_min_thread_int \l__regex_max_thread_int
6583     \__regex_match_init:
6584     \__regex_match_once_init:
6585     \str_map_inline:nn {#1}
6586     {
6587       \tl_if_blank:nTF {##1}
6588       { \__regex_match_one_token:nnN {##1} {'##1} A }
6589       { \__regex_match_one_token:nnN {##1} {'##1} C }
6590     }
6591     \__regex_match_one_token:nnN { } { -2 } F
6592     \prg_break_point:Nn \__regex_maplike_break: { }
6593   }
6594 \cs_new_protected:Npn \__regex_match_init:
6595   {
6596     \bool_gset_false:N \g__regex_success_bool
6597     \int_step_inline:nnn
6598     \l__regex_min_state_int { \l__regex_max_state_int - \c_one_int }
6599     {
6600       \__kernel_intarray_gset:Nnn
6601       \g__regex_state_active_intarray {##1} \c_one_int
6602     }
6603     \int_zero:N \l__regex_step_int
6604     \int_set:Nn \l__regex_min_pos_int { 2 }
6605     \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
6606     \int_set:Nn \l__regex_last_char_success_int { -2 }
6607     \tl_build_begin:N \l__regex_matched_analysis_tl
6608     \tl_clear:N \l__regex_curr_analysis_tl
6609     \int_set_eq:NN \l__regex_min_submatch_int \c_one_int
6610     \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
6611     \bool_set_false:N \l__regex_empty_success_bool
6612   }

```

(End of definition for `__regex_match:n`, `__regex_match_cs:n`, and `__regex_match_init:.`)

`__regex_match_once_init:` This function resets various variables used when finding one match. It is called before the loop through characters, and every time we find a match, before searching for another match (this is controlled by the `every_match` token list).

First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start because `__regex_match_one_token:nnN` increments `\l__regex_curr_pos_int` and saves `\l__regex_curr_char_int` as the `last_char` so that word boundaries can be correctly identified.

```
6613 \cs_new_protected:Npn \__regex_match_once_init:
6614   {
6615     \if_meaning:w \c_true_bool \l__regex_empty_success_bool
6616       \cs_set:Npn \__regex_if_two_empty_matches:F
6617         {
6618           \int_compare:nNnF
6619             \l__regex_start_pos_int = \l__regex_curr_pos_int
6620         }
6621     \else:
6622       \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
6623     \fi:
6624     \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
6625     \bool_set_false:N \l__regex_match_success_bool
6626     \tl_set:Ne \l__regex_curr_submatches_tl
6627       { \prg_replicate:nn { 2 * \l__regex_capturing_group_int } { 0 , } }
6628     \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
6629     \__regex_store_state:n { \l__regex_min_state_int }
6630     \int_set:Nn \l__regex_curr_pos_int { \l__regex_start_pos_int - \c_one_int }
6631     \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_success_int
6632     \tl_build_get_intermediate:NN \l__regex_matched_analysis_tl \l__regex_internal_a_tl
6633     \exp_args:NNf \__regex_match_once_init_aux:
6634     \tl_map_inline:nn
6635       { \exp_after:wN \l__regex_internal_a_tl \l__regex_curr_analysis_tl }
6636       { \__regex_match_one_token:nnN ##1 }
6637     \prg_break_point:Nn \__regex_maplike_break: { }
6638   }
6639 \cs_new_protected:Npn \__regex_match_once_init_aux:
6640   {
6641     \tl_build_begin:N \l__regex_matched_analysis_tl
6642     \tl_clear:N \l__regex_curr_analysis_tl
6643   }
```

(End of definition for `__regex_match_once_init:.`)

`__regex_single_match:` For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action #1, then find the next match.

```
6644 \cs_new_protected:Npn \__regex_single_match:
6645   {
6646     \tl_set:Nn \l__regex_every_match_tl
6647     {
```

```

6648     \bool_gset_eq:NN
6649     \g__regex_success_bool
6650     \l__regex_match_success_bool
6651     \__regex_maplike_break:
6652   }
6653 }
6654 \cs_new_protected:Npn \__regex_multi_match:n #1
6655 {
6656   \tl_set:Nn \l__regex_every_match_tl
6657   {
6658     \if_meaning:w \c_false_bool \l__regex_match_success_bool
6659     \exp_after:wN \__regex_maplike_break:
6660     \fi:
6661     \bool_gset_true:N \g__regex_success_bool
6662     #1
6663     \__regex_match_once_init:
6664   }
6665 }

```

(End of definition for `__regex_single_match:` and `__regex_multi_match:n`.)

`__regex_match_one_token:nnN` `__regex_match_one_active:n` At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (`max_thread`). This results in a sequence of `__regex_use_state_and_submatches:w` (`state`), (`submatch-list`); and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is the match. We explain the `fresh_thread` business when describing `__regex_action_wildcard:`.

```

6666 \cs_new_protected:Npn \__regex_match_one_token:nnN #1#2#3
6667 {
6668   \int_add:Nn \l__regex_step_int { 2 }
6669   \int_incr:N \l__regex_curr_pos_int
6670   \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
6671   \cs_set_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
6672   \tl_set:Nn \l__regex_curr_token_tl {#1}
6673   \int_set:Nn \l__regex_curr_char_int {#2}
6674   \int_set:Nn \l__regex_curr_catcode_int { "#3 }
6675   \tl_build_put_right:Ne \l__regex_matched_analysis_tl
6676   { \exp_not:o \l__regex_curr_analysis_tl }
6677   \tl_set:Nn \l__regex_curr_analysis_tl { { {#1} {#2} #3 } }
6678   \use:e
6679   {
6680     \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
6681     \int_step_function:nnN
6682     \l__regex_min_thread_int
6683     { \l__regex_max_thread_int - \c_one_int }
6684     \__regex_match_one_active:n
6685   }
6686   \prg_break_point:
6687   \bool_set_false:N \l__regex_fresh_thread_bool
6688   \if_int_compare:w \l__regex_max_thread_int > \l__regex_min_thread_int
6689   \if_int_compare:w -2 < \l__regex_curr_char_int
6690   \exp_after:wN \use_i:nn

```

```

6691     \fi:
6692     \fi:
6693     \l__regex_every_match_tl
6694   }
6695 \cs_new:Npn \__regex_match_one_active:n #1
6696   {
6697     \__regex_use_state_and_submatches:w
6698     \__kernel_intarray_range_to_clist:Nnn
6699     \g__regex_thread_info_intarray
6700     { \c_one_int + #1 * (\l__regex_capturing_group_int * 2 + \c_one_int) }
6701     { (\c_one_int + #1) * (\l__regex_capturing_group_int * 2 + \c_one_int) }
6702   } ;
6703 }

```

(End of definition for __regex_match_one_token:nnN and __regex_match_one_active:n.)

46.5.3 Using states of the nfa

__regex_use_state: Use the current NFA instruction. The state is initially marked as belonging to the current **step**: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as **step + 1**: any thread hitting it at that point will be terminated.

```

6704 \cs_new_protected:Npn \__regex_use_state:
6705   {
6706     \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
6707     \l__regex_curr_state_int \l__regex_step_int
6708     \__regex_toks_use:w \l__regex_curr_state_int
6709     \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
6710     \l__regex_curr_state_int
6711     { \__regex_int_eval:w \l__regex_step_int + \c_one_int \scan_stop: }
6712   }

```

(End of definition for __regex_use_state:.)

__regex_use_state_and_submatches:w This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the `curr_state` and `curr_submatches` and use the state if it has not yet been encountered at this step.

```

6713 \cs_new_protected:Npn \__regex_use_state_and_submatches:w #1 , #2 ;
6714   {
6715     \int_set:Nn \l__regex_curr_state_int {#1}
6716     \if_int_compare:w
6717       \__kernel_intarray_item:Nn \g__regex_state_active_intarray
6718       \l__regex_curr_state_int
6719       < \l__regex_step_int
6720     \tl_set:Nn \l__regex_curr_submatches_tl { #2 , }
6721     \exp_after:wN \__regex_use_state:
6722     \fi:
6723     \scan_stop:
6724   }

```

(End of definition for __regex_use_state_and_submatches:w.)

46.5.4 Actions when matching

`__regex_action_start_wildcard:N` For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting `\l__regex_fresh_thread_bool` may be skipped by a successful thread, hence we had to add it to `__regex_match_one_token:nnN` too.

```

6725 \cs_new_protected:Npn \__regex_action_start_wildcard:N #1
6726   {
6727     \bool_set_true:N \l__regex_fresh_thread_bool
6728     \__regex_action_free:n {1}
6729     \bool_set_false:N \l__regex_fresh_thread_bool
6730     \bool_if:NT #1 { \__regex_action_cost:n {0} }
6731   }

```

(End of definition for `__regex_action_start_wildcard:N`.)

`__regex_action_free:n`
`__regex_action_free_group:n`
`__regex_action_free_aux:nn` These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of `\l__regex_curr_state_int` and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the `group` version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the “normal” version will revisit a state even within the thread itself.

```

6732 \cs_new_protected:Npn \__regex_action_free:n
6733   { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
6734 \cs_new_protected:Npn \__regex_action_free_group:n
6735   { \__regex_action_free_aux:nn { < \l__regex_step_int } }
6736 \cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
6737   {
6738     \use:e
6739     {
6740       \int_add:Nn \l__regex_curr_state_int {#2}
6741       \exp_not:n
6742       {
6743         \if_int_compare:w
6744           \__kernel_intarray_item:Nn \g__regex_state_active_intarray
6745             \l__regex_curr_state_int
6746             #1
6747           \exp_after:wN \__regex_use_state:
6748           \fi:
6749       }
6750       \int_set:Nn \l__regex_curr_state_int
6751       { \int_use:N \l__regex_curr_state_int }
6752       \tl_set:Nn \exp_not:N \l__regex_curr_submatches_tl
6753       { \exp_not:o \l__regex_curr_submatches_tl }
6754     }
6755   }

```

(End of definition for `__regex_action_free:n`, `__regex_action_free_group:n`, and `__regex_action_free_aux:nn`.)

`__regex_action_cost:n` A transition which consumes the current character and shifts the state by #1. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

```

6756 \cs_new_protected:Npn \__regex_action_cost:n #1
6757 {
6758   \exp_args:No \__regex_store_state:n
6759   { \tex_the:D \__regex_int_eval:w \l__regex_curr_state_int + #1 }
6760 }

```

(End of definition for __regex_action_cost:n.)

__regex_store_state:n Put the given state and current submatch information in \g__regex_thread_info_intarray, and increment the length of the array.

```

6761 \cs_new_protected:Npn \__regex_store_state:n #1
6762 {
6763   \exp_args:No \__regex_store_submatches:nn
6764   \l__regex_curr_submatches_tl {#1}
6765   \int_incr:N \l__regex_max_thread_int
6766 }
6767 \cs_new_protected:Npn \__regex_store_submatches:nn #1#2
6768 {
6769   \__kernel_intarray_gset_range_from_clist:Nnn
6770   \g__regex_thread_info_intarray
6771   {
6772     \__regex_int_eval:w
6773     \c_one_int + \l__regex_max_thread_int *
6774     (\l__regex_capturing_group_int * 2 + \c_one_int)
6775   }
6776   { #2 , #1 }
6777 }

```

(End of definition for __regex_store_state:n and __regex_store_submatches:.)

__regex_disable_submatches: Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```

6778 \cs_new_protected:Npn \__regex_disable_submatches:
6779 {
6780   \cs_set_protected:Npn \__regex_store_submatches:n ##1 { }
6781   \cs_set_protected:Npn \__regex_action_submatch:nN ##1##2 { }
6782 }

```

(End of definition for __regex_disable_submatches:.)

__regex_action_submatch:nN Update the current submatches with the information from the current position. Maybe a bottleneck.

```

\__regex_action_submatch_aux:w
\__regex_action_submatch_auxii:w
\__regex_action_submatch_auxiii:w
\__regex_action_submatch_auxiv:w
6783 \cs_new_protected:Npn \__regex_action_submatch:nN #1#2
6784 {
6785   \exp_after:wN \__regex_action_submatch_aux:w
6786   \l__regex_curr_submatches_tl ; {#1} #2
6787 }
6788 \cs_new_protected:Npn \__regex_action_submatch_aux:w #1 ; #2#3
6789 {
6790   \tl_set:Ne \l__regex_curr_submatches_tl
6791   {
6792     \prg_replicate:nn
6793     { #2 \if_meaning:w > #3 + \l__regex_capturing_group_int \fi: }

```

```

6794         { \_regex_action_submatch_auxii:w }
6795         \_regex_action_submatch_auxiii:w
6796         #1
6797     }
6798 }
6799 \cs_new:Npn \_regex_action_submatch_auxii:w
6800     #1 \_regex_action_submatch_auxiii:w #2 ,
6801     { #2 , #1 \_regex_action_submatch_auxiii:w }
6802 \cs_new:Npn \_regex_action_submatch_auxiii:w #1 ,
6803     { \int_use:N \l__regex_curr_pos_int , }

```

(End of definition for `_regex_action_submatch:nN` and others.)

`_regex_action_success:` There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is “fresh”; and we store the current position and submatches. The current step is then interrupted with `\prg_break:`, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```

6804 \cs_new_protected:Npn \_regex_action_success:
6805     {
6806     \_regex_if_two_empty_matches:F
6807     {
6808         \bool_set_true:N \l__regex_match_success_bool
6809         \bool_set_eq:NN \l__regex_empty_success_bool
6810         \l__regex_fresh_thread_bool
6811         \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
6812         \int_set_eq:NN \l__regex_last_char_success_int \l__regex_last_char_int
6813         \tl_build_begin:N \l__regex_matched_analysis_tl
6814         \tl_set_eq:NN \l__regex_success_submatches_tl
6815         \l__regex_curr_submatches_tl
6816         \prg_break:
6817     }
6818 }

```

(End of definition for `_regex_action_success:`.)

46.6 Replacement

46.6.1 Variables and helpers used in replacement

`\l__regex_replacement_csnames_int` The behaviour of closing braces inside a replacement text depends on whether a sequences `\c{` or `\u{` has been encountered. The number of “open” such sequences that should be closed by `}` is stored in `\l__regex_replacement_csnames_int`, and decreased by 1 by each `}`.

```

6819 \int_new:N \l__regex_replacement_csnames_int

```

(End of definition for `\l__regex_replacement_csnames_int`.)

`\l__regex_replacement_category_tl`
`\l__regex_replacement_category_seq` This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD()d)`.

```

6820 \tl_new:N \l__regex_replacement_category_tl
6821 \seq_new:N \l__regex_replacement_category_seq

```

(End of definition for \l__regex_replacement_category_tl and \l__regex_replacement_category_seq.)

`\g__regex_balance_tl` This token list holds the replacement text for `__regex_replacement_balance_one_match:n` while it is being built incrementally.

```
6822 \tl_new:N \g__regex_balance_tl
```

(End of definition for \g__regex_balance_tl.)

`__regex_replacement_balance_one_match:n` This expects as an argument the first index of a set of entries in `\g__regex_submatch_begin_intarray` (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

```
6823 \cs_new:Npn \__regex_replacement_balance_one_match:n #1
6824 { - \__regex_submatch_balance:n {#1} }
```

(End of definition for __regex_replacement_balance_one_match:n.)

`__regex_replacement_do_one_match:n` The input is the same as `__regex_replacement_balance_one_match:n`. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
6825 \cs_new:Npn \__regex_replacement_do_one_match:n #1
6826 {
6827   \__regex_query_range:nn
6828   { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
6829   { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
6830 }
```

(End of definition for __regex_replacement_do_one_match:n.)

`__regex_replacement_exp_not:N` This function lets us navigate around the fact that the primitive `\exp_not:n` requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as `\c_parameter_token`. Indeed, within an e/x-expanding assignment, `\exp_not:N #` behaves as a single #, whereas `\exp_not:n {#}` behaves as a doubled ##.

```
6831 \cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(End of definition for __regex_replacement_exp_not:N.)

`__regex_replacement_exp_not:V` This is used for the implementation of `\u`, and it gets redefined for `\peek_regex_replace_once:nnTF`.

```
6832 \cs_new_eq:NN \__regex_replacement_exp_not:V \exp_not:V
```

(End of definition for __regex_replacement_exp_not:V.)

46.6.2 Query and brace balance

`__regex_query_range:nn`
`__regex_query_range_loop:ww`

When it is time to extract submatches from the token list, the various tokens are stored in `\toks` registers numbered from `\l__regex_min_pos_int` inclusive to `\l__regex_max_pos_int` exclusive. The function `__regex_query_range:nn {<min>} {<max>}` unpacks registers from the position `<min>` to the position `<max>-1` included. Once this is expanded, a second e-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```

6833 \cs_new:Npn __regex_query_range:nn #1#2
6834   {
6835     \exp_after:wN __regex_query_range_loop:ww
6836     \int_value:w __regex_int_eval:w #1 \exp_after:wN ;
6837     \int_value:w __regex_int_eval:w #2 ;
6838     \prg_break_point:
6839   }
6840 \cs_new:Npn __regex_query_range_loop:ww #1 ; #2 ;
6841   {
6842     \if_int_compare:w #1 < #2 \exp_stop_f:
6843     \else:
6844       \prg_break:n
6845     \fi:
6846     __regex_toks_use:w #1 \exp_stop_f:
6847     \exp_after:wN __regex_query_range_loop:ww
6848     \int_value:w __regex_int_eval:w #1 + \c_one_int ; #2 ;
6849   }

```

(End of definition for `__regex_query_range:nn` and `__regex_query_range_loop:ww`.)

`__regex_query_submatch:n`

Find the start and end positions for a given submatch (of a given match).

```

6850 \cs_new:Npn __regex_query_submatch:n #1
6851   {
6852     __regex_query_range:nn
6853     { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
6854     { \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
6855   }

```

(End of definition for `__regex_query_submatch:n`.)

`__regex_submatch_balance:n`

Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the `<max pos>` and `<min pos>`. These two positions are found in the corresponding “submatch” arrays.

```

6856 \cs_new_protected:Npn __regex_submatch_balance:n #1
6857   {
6858     \tex_the:D __regex_int_eval:w
6859     __regex_intarray_item:NnF \g__regex_balance_intarray
6860     {
6861       __kernel_intarray_item:Nn
6862       \g__regex_submatch_end_intarray {#1}
6863     }
6864     \c_zero_int
6865   -

```

```

6866     \_regex_intarray_item:NnF \g__regex_balance_intarray
6867     {
6868         \_kernel_intarray_item:Nn
6869         \g__regex_submatch_begin_intarray {#1}
6870     }
6871     \c_zero_int
6872 \scan_stop:
6873 }

```

(End of definition for _regex_submatch_balance:n.)

46.6.3 Framework

_regex_replacement:n The replacement text is built incrementally. We keep track in \l__regex_balance_int of the balance of explicit begin- and end-group tokens and we store in \g__regex_balance_tl some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing \prg_do_nothing: because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the balance_one_match and do_one_match functions.

```

6874 \cs_new_protected:Npn \_regex_replacement:n
6875 { \_regex_replacement_apply:Nn \_regex_replacement_set:n }
6876 \cs_new_protected:Npn \_regex_replacement_apply:Nn #1#2
6877 {
6878     \group_begin:
6879     \tl_build_begin:N \l__regex_build_tl
6880     \int_zero:N \l__regex_balance_int
6881     \tl_gclear:N \g__regex_balance_tl
6882     \_regex_escape_use:nmmn
6883     {
6884         \if_charcode:w \c_right_brace_str ##1
6885         \_regex_replacement_rbrace:N
6886     \else:
6887         \if_charcode:w \c_left_brace_str ##1
6888         \_regex_replacement_lbrace:N
6889     \else:
6890         \_regex_replacement_normal:n
6891     \fi:
6892     \fi:
6893     ##1
6894     }
6895     { \_regex_replacement_escaped:N ##1 }
6896     { \_regex_replacement_normal:n ##1 }
6897     {#2}
6898     \prg_do_nothing: \prg_do_nothing:
6899     \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
6900     \msg_error:nne { regex } { replacement-missing-rbrace }
6901     { \int_use:N \l__regex_replacement_csnames_int }
6902     \tl_build_put_right:Ne \l__regex_build_tl
6903     { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
6904     \fi:
6905     \seq_if_empty:NF \l__regex_replacement_category_seq
6906     {

```

```

6907         \msg_error:nne { regex } { replacement-missing-rparen }
6908         { \seq_count:N \l__regex_replacement_category_seq }
6909         \seq_clear:N \l__regex_replacement_category_seq
6910     }
6911     \tl_gput_right:Ne \g__regex_balance_tl
6912     { + \int_use:N \l__regex_balance_int }
6913     \tl_build_end:N \l__regex_build_tl
6914     \exp_args:NNo
6915     \group_end:
6916     #1 \l__regex_build_tl
6917 }
6918 \cs_generate_variant:Nn \__regex_replacement:n { e }
6919 \cs_new_protected:Npn \__regex_replacement_set:n #1
6920 {
6921     \cs_set:Npn \__regex_replacement_do_one_match:n ##1
6922     {
6923         \__regex_query_range:nn
6924         {
6925             \__kernel_intarray_item:Nn
6926             \g__regex_submatch_prev_intarray {##1}
6927         }
6928         {
6929             \__kernel_intarray_item:Nn
6930             \g__regex_submatch_begin_intarray {##1}
6931         }
6932         #1
6933     }
6934     \exp_args:Nno \use:n
6935     { \cs_gset:Npn \__regex_replacement_balance_one_match:n ##1 }
6936     {
6937         \g__regex_balance_tl
6938         - \__regex_submatch_balance:n {##1}
6939     }
6940 }

```

(End of definition for `__regex_replacement:n`, `__regex_replacement_apply:Nn`, and `__regex_replacement_set:n`.)

```

\__regex_case_replacement:n
\__regex_case_replacement:e
6941 \tl_new:N \g__regex_case_replacement_tl
6942 \tl_new:N \g__regex_case_balance_tl
6943 \cs_new_protected:Npn \__regex_case_replacement:n #1
6944 {
6945     \tl_gset:Nn \g__regex_case_balance_tl
6946     {
6947         \if_case:w
6948         \__kernel_intarray_item:Nn
6949         \g__regex_submatch_case_intarray {##1}
6950     }
6951     \tl_gset_eq:NN \g__regex_case_replacement_tl \g__regex_case_balance_tl
6952     \tl_map_tokens:nn {#1}
6953     { \__regex_replacement_apply:Nn \__regex_case_replacement_aux:n }
6954     \tl_gset:No \g__regex_balance_tl
6955     { \g__regex_case_balance_tl \fi: }

```

```

6956 \exp_args:No \__regex_replacement_set:n
6957 { \g__regex_case_replacement_tl \fi: }
6958 }
6959 \cs_generate_variant:Nn \__regex_case_replacement:n { e }
6960 \cs_new_protected:Npn \__regex_case_replacement_aux:n #1
6961 {
6962   \tl_gput_right:Nn \g__regex_case_replacement_tl { \or: #1 }
6963   \tl_gput_right:No \g__regex_case_balance_tl
6964   { \exp_after:wN \or: \g__regex_balance_tl }
6965 }

```

(End of definition for __regex_case_replacement:n.)

__regex_replacement_put:n This gets redefined for \peek_regex_replace_once:nnTF.

```

6966 \cs_new_protected:Npn \__regex_replacement_put:n
6967 { \tl_build_put_right:Nn \l__regex_build_tl }

```

(End of definition for __regex_replacement_put:n.)

__regex_replacement_normal:n
 __regex_replacement_normal_aux:N

Most characters are simply sent to the output by \tl_build_put_right:Nn, unless a particular category code has been requested: then __regex_replacement_c_A:w or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of \l__regex_replacement_category_tl. The argument #1 is a single character (including the case of a catcode-other space). In case no specific catcode is requested, we taked into account the current catcode regime (at the time the replacement is performed) as much as reasonable, with all impossible catcodes (escape, newline, etc.) being mapped to “other”.

```

6968 \cs_new_protected:Npn \__regex_replacement_normal:n #1
6969 {
6970   \int_compare:nNnTF \l__regex_replacement_csnames_int > \c_zero_int
6971   { \exp_args:No \__regex_replacement_put:n { \token_to_str:N #1 } }
6972   {
6973     \tl_if_empty:NTF \l__regex_replacement_category_tl
6974     { \__regex_replacement_normal_aux:N #1 }
6975     { % (
6976       \token_if_eq_charcode:NNTF #1 )
6977       {
6978         \seq_pop:NN \l__regex_replacement_category_seq
6979         \l__regex_replacement_category_tl
6980       }
6981       {
6982         \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
6983         ? #1
6984       }
6985     }
6986   }
6987 }
6988 \cs_new_protected:Npn \__regex_replacement_normal_aux:N #1
6989 {
6990   \token_if_eq_charcode:NNTF #1 \c_space_token
6991   { \__regex_replacement_c_S:w }
6992   {
6993     \exp_after:wN \exp_after:wN

```

```

6994     \if_case:w \tex_catcode:D '#1 \exp_stop_f:
6995         \__regex_replacement_c_0:w
6996     \or: \__regex_replacement_c_B:w
6997     \or: \__regex_replacement_c_E:w
6998     \or: \__regex_replacement_c_M:w
6999     \or: \__regex_replacement_c_T:w
7000     \or: \__regex_replacement_c_0:w
7001     \or: \__regex_replacement_c_P:w
7002     \or: \__regex_replacement_c_U:w
7003     \or: \__regex_replacement_c_D:w
7004     \or: \__regex_replacement_c_0:w
7005     \or: \__regex_replacement_c_S:w
7006     \or: \__regex_replacement_c_L:w
7007     \or: \__regex_replacement_c_0:w
7008     \or: \__regex_replacement_c_A:w
7009     \else: \__regex_replacement_c_0:w
7010     \fi:
7011 }
7012 ? #1
7013 }

```

(End of definition for `__regex_replacement_normal:n` and `__regex_replacement_normal_aux:N`.)

`__regex_replacement_escaped:N`

As in parsing a regular expression, we use an auxiliary built from #1 if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character.

```

7014 \cs_new_protected:Npn \__regex_replacement_escaped:N #1
7015 {
7016     \cs_if_exist_use:cF { __regex_replacement_#1:w }
7017     {
7018         \if_int_compare:w \c_one_int < 1#1 \exp_stop_f:
7019         \__regex_replacement_put_submatch:n {#1}
7020     \else:
7021         \__regex_replacement_normal:n {#1}
7022     \fi:
7023     }
7024 }

```

(End of definition for `__regex_replacement_escaped:N`.)

46.6.4 Submatches

`__regex_replacement_put_submatch:n`
`__regex_replacement_put_submatch_aux:n`

Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match.

```

7025 \cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
7026 {
7027     \if_int_compare:w #1 < \l__regex_capturing_group_int
7028     \__regex_replacement_put_submatch_aux:n {#1}
7029     \else:
7030     \msg_expandable_error:nnff { regex } { submatch-too-big }
7031     {#1} { \int_eval:n { \l__regex_capturing_group_int - \c_one_int } }
7032     \fi:

```

```

7033 }
7034 \cs_new_protected:Npn \__regex_replacement_put_submatch_aux:n #1
7035 {
7036   \tl_build_put_right:Nn \l__regex_build_tl
7037   { \__regex_query_submatch:n { \__regex_int_eval:w #1 + ##1 \scan_stop: } }
7038   \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7039   \tl_gput_right:Nn \g__regex_balance_tl
7040   { + \__regex_submatch_balance:n { \__regex_int_eval:w #1 + ##1 \scan_stop: } }
7041   \fi:
7042 }

```

(End of definition for `__regex_replacement_put_submatch:n` and `__regex_replacement_put_submatch_aux:n`.)

`__regex_replacement_g:w` Grab digits for the `\g` escape sequence in a primitive assignment to the integer `\l__regex_internal_a_int`. At the end of the run of digits, check that it ends with a right brace.

```

7043 \cs_new_protected:Npn \__regex_replacement_g:w #1#2
7044 {
7045   \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
7046   { \l__regex_internal_a_int = \__regex_replacement_g_digits:NN }
7047   { \__regex_replacement_error:NNN g #1 #2 }
7048 }
7049 \cs_new:Npn \__regex_replacement_g_digits:NN #1#2
7050 {
7051   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
7052   {
7053     \if_int_compare:w \c_one_int < 1#2 \exp_stop_f:
7054     #2
7055     \exp_after:wN \use_i:nnn
7056     \exp_after:wN \__regex_replacement_g_digits:NN
7057   \else:
7058     \exp_stop_f:
7059     \exp_after:wN \__regex_replacement_error:NNN
7060     \exp_after:wN g
7061   \fi:
7062 }
7063 {
7064   \exp_stop_f:
7065   \if_meaning:w \__regex_replacement_rbrace:N #1
7066   \exp_args:No \__regex_replacement_put_submatch:n
7067   { \int_use:N \l__regex_internal_a_int }
7068   \exp_after:wN \use_none:nn
7069   \else:
7070     \exp_after:wN \__regex_replacement_error:NNN
7071     \exp_after:wN g
7072   \fi:
7073 }
7074 #1 #2
7075 }

```

(End of definition for `__regex_replacement_g:w` and `__regex_replacement_g_digits:NN`.)

46.6.5 Csnames in replacement

`__regex_replacement_c:w` `\c` may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with `\u`. Otherwise test whether the category is known; if it is not, complain.

```
7076 \cs_new_protected:Npn \__regex_replacement_c:w #1#2
7077 {
7078   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
7079   {
7080     \cs_if_exist:cTF { __regex_replacement_c:#2:w }
7081     { \__regex_replacement_cat:NNN #2 }
7082     { \__regex_replacement_error:NNN c #1#2 }
7083   }
7084   {
7085     \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
7086     { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:N }
7087     { \__regex_replacement_error:NNN c #1#2 }
7088   }
7089 }
```

(End of definition for `__regex_replacement_c:w`.)

`__regex_replacement_cu_aux:Nw` Start a control sequence with `\cs:w`, protected from expansion by #1 (either `__regex_replacement_exp_not:N` or `\exp_not:V`), or turned to a string by `\tl_to_str:V` if inside another csname construction `\c` or `\u`. We use `\tl_to_str:V` rather than `\tl_to_str:N` to deal with integers and other registers.

```
7090 \cs_new_protected:Npn \__regex_replacement_cu_aux:Nw #1
7091 {
7092   \if_case:w \l__regex_replacement_csnames_int
7093   \tl_build_put_right:Nn \l__regex_build_tl
7094   { \exp_not:n { \exp_after:wN #1 \cs:w } }
7095   \else:
7096   \tl_build_put_right:Nn \l__regex_build_tl
7097   { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
7098   \fi:
7099   \int_incr:N \l__regex_replacement_csnames_int
7100 }
```

(End of definition for `__regex_replacement_cu_aux:Nw`.)

`__regex_replacement_u:w` Check that `\u` is followed by a left brace. If so, start a control sequence with `\cs:w`, which is then unpacked either with `\exp_not:V` or `\tl_to_str:V` depending on the current context.

```
7101 \cs_new_protected:Npn \__regex_replacement_u:w #1#2
7102 {
7103   \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
7104   { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:V }
7105   { \__regex_replacement_error:NNN u #1#2 }
7106 }
```

(End of definition for `__regex_replacement_u:w`.)

`_regex_replacement_rbrace:N` Within a `\c{...}` or `\u{...}` construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```

7107 \cs_new_protected:Npn \_regex_replacement_rbrace:N #1
7108   {
7109     \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
7110       \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
7111       \int_decr:N \l__regex_replacement_csnames_int
7112     \else:
7113       \_regex_replacement_normal:n {#1}
7114     \fi:
7115   }

```

(End of definition for _regex_replacement_rbrace:N.)

`_regex_replacement_lbrace:N` Within a `\c{...}` or `\u{...}` construction, this is forbidden. Otherwise, this is a raw left brace.

```

7116 \cs_new_protected:Npn \_regex_replacement_lbrace:N #1
7117   {
7118     \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
7119       \msg_error:nnn { regex } { cu-lbrace } { u }
7120     \else:
7121       \_regex_replacement_normal:n {#1}
7122     \fi:
7123   }

```

(End of definition for _regex_replacement_lbrace:N.)

46.6.6 Characters in replacement

`_regex_replacement_cat:NNN` Here, `#1` is a letter among BEMTPUDSLOA and `#2#3` denote the next character. Complain if we reach the end of the replacement or if the construction appears inside `\c{...}` or `\u{...}`, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```

7124 \cs_new_protected:Npn \_regex_replacement_cat:NNN #1#2#3
7125   {
7126     \token_if_eq_meaning:NNTF \prg_do_nothing: #3
7127     { \msg_error:nn { regex } { replacement-catcode-end } }
7128     {
7129       \int_compare:nNnTF \l__regex_replacement_csnames_int > \c_zero_int
7130         {
7131           \msg_error:nnnn
7132             { regex } { replacement-catcode-in-cs } {#1} {#3}
7133           #2 #3
7134         }
7135         {
7136           \_regex_two_if_eq:NNNNTF #2 #3 \_regex_replacement_normal:n (
7137             {
7138               \seq_push:NV \l__regex_replacement_category_seq
7139               \l__regex_replacement_category_tl
7140               \tl_set:Nn \l__regex_replacement_category_tl {#1}
7141             }
7142             {
7143               \token_if_eq_meaning:NNT #2 \_regex_replacement_escaped:N

```



```

7144         {
7145             \__regex_char_if_alphanumeric:NTF #3
7146             {
7147                 \msg_error:nnnn
7148                 { regex } { replacement-catcode-escaped }
7149                 {#1} {#3}
7150             }
7151             { }
7152         }
7153     \use:c { __regex_replacement_c_#1:w } #2 #3
7154 }
7155 }
7156 }
7157 }

```

(End of definition for `__regex_replacement_cat:NNN`.)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```
7158 \group_begin:
```

```
\__regex_replacement_char:nNN
```

The only way to produce an arbitrary character–catcode pair is to use the `\lowercase` or `\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: `#3` is the character whose character code to reproduce. We could use `\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```

7159     \cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
7160     {
7161         \tex_lccode:D \c_zero_int = '#3 \scan_stop:
7162         \tex_lowercase:D { \__regex_replacement_put:n {#1} }
7163     }

```

(End of definition for `__regex_replacement_char:nNN`.)

```
\__regex_replacement_c_A:w
```

For an active character, expansion must be avoided, twice because we later do two e-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```

7164     \char_set_catcode_active:N \^^@
7165     \cs_new_protected:Npn \__regex_replacement_c_A:w
7166     { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:N \^^@ } } }

```

(End of definition for `__regex_replacement_c_A:w`.)

```
\__regex_replacement_c_B:w
```

An explicit begin-group token increases the balance, unless within a `\c{...}` or `\u{...}` construction. Add the desired begin-group character, using the standard `\if_false:` trick. We eventually e-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The `\exp_after:wN` is not strictly needed, but is more consistent with `l3tl-analysis`.

```

7167     \char_set_catcode_group_begin:N \^^@
7168     \cs_new_protected:Npn \__regex_replacement_c_B:w
7169     {

```

```

7170     \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7171     \int_incr:N \l__regex_balance_int
7172     \fi:
7173     \__regex_replacement_char:nNN
7174     { \exp_not:n { \exp_after:wN ^^@ \if_false: } \fi: } }
7175 }

```

(End of definition for `__regex_replacement_c_B:w`.)

`__regex_replacement_c_C:w` This is not quite catcode-related: when the user requests a character with category “control sequence”, the one-character control symbol is returned. As for the active character, we prepare for two e-expansions.

```

7176     \cs_new_protected:Npn \__regex_replacement_c_C:w #1#2
7177     {
7178         \tl_build_put_right:Nn \l__regex_build_tl
7179         { \exp_not:N \__regex_replacement_exp_not:N \exp_not:c {#2} }
7180     }

```

(End of definition for `__regex_replacement_c_C:w`.)

`__regex_replacement_c_D:w` Subscripts fit the mould: `\lowercase` the null byte with the correct category.

```

7181     \char_set_catcode_math_subscript:N ^^@
7182     \cs_new_protected:Npn \__regex_replacement_c_D:w
7183     { \__regex_replacement_char:nNN { ^^@ } }

```

(End of definition for `__regex_replacement_c_D:w`.)

`__regex_replacement_c_E:w` Similar to the begin-group case, the second e-expansion produces the bare end-group token.

```

7184     \char_set_catcode_group_end:N ^^@
7185     \cs_new_protected:Npn \__regex_replacement_c_E:w
7186     {
7187         \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7188         \int_decr:N \l__regex_balance_int
7189         \fi:
7190         \__regex_replacement_char:nNN
7191         { \exp_not:n { \if_false: { \fi: ^^@ } } }
7192     }

```

(End of definition for `__regex_replacement_c_E:w`.)

`__regex_replacement_c_L:w` Simply `\lowercase` a letter null byte to produce an arbitrary letter.

```

7193     \char_set_catcode_letter:N ^^@
7194     \cs_new_protected:Npn \__regex_replacement_c_L:w
7195     { \__regex_replacement_char:nNN { ^^@ } }

```

(End of definition for `__regex_replacement_c_L:w`.)

`__regex_replacement_c_M:w` No surprise here, we lowercase the null math toggle.

```

7196     \char_set_catcode_math_toggle:N ^^@
7197     \cs_new_protected:Npn \__regex_replacement_c_M:w
7198     { \__regex_replacement_char:nNN { ^^@ } }

```

(End of definition for `__regex_replacement_c_M:w`.)

`__regex_replacement_c_0:w` Lowercase an other null byte.

```
7199 \char_set_catcode_other:N \^^@
7200 \cs_new_protected:Npn \__regex_replacement_c_0:w
7201 { \__regex_replacement_char:nNN { ^^@ } }
```

(End of definition for __regex_replacement_c_0:w.)

`__regex_replacement_c_P:w` For macro parameters, expansion is a tricky issue. We need to prepare for two e-expansions and passing through various macro definitions. Note that we cannot replace one `\exp_not:n` by doubling the macro parameter characters because this would misbehave if a mischievous user asks for `\c{\cP\#}`, since that macro parameter character would be doubled.

```
7202 \char_set_catcode_parameter:N \^^@
7203 \cs_new_protected:Npn \__regex_replacement_c_P:w
7204 {
7205   \__regex_replacement_char:nNN
7206   { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
7207 }
```

(End of definition for __regex_replacement_c_P:w.)

`__regex_replacement_c_S:w` Spaces are normalized on input by T_EX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```
7208 \cs_new_protected:Npn \__regex_replacement_c_S:w #1#2
7209 {
7210   \if_int_compare:w '#2 = \c_zero_int
7211     \msg_error:nn { regex } { replacement-null-space }
7212   \fi:
7213   \tex_lccode:D '\ = '#2 \scan_stop:
7214   \tex_lowercase:D { \__regex_replacement_put:n {~} }
7215 }
```

(End of definition for __regex_replacement_c_S:w.)

`__regex_replacement_c_T:w` No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```
7216 \char_set_catcode_alignment:N \^^@
7217 \cs_new_protected:Npn \__regex_replacement_c_T:w
7218 { \__regex_replacement_char:nNN { ^^@ } }
```

(End of definition for __regex_replacement_c_T:w.)

`__regex_replacement_c_U:w` Simple call to `__regex_replacement_char:nNN` which lowercases the math superscript `^^@`.

```
7219 \char_set_catcode_math_superscript:N \^^@
7220 \cs_new_protected:Npn \__regex_replacement_c_U:w
7221 { \__regex_replacement_char:nNN { ^^@ } }
```

(End of definition for __regex_replacement_c_U:w.)

Restore the catcode of the null byte.

```
7222 \group_end:
```

46.6.7 An error

`_regex_replacement_error:NNN` Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
7223 \cs_new_protected:Npn \_regex_replacement_error:NNN #1#2#3
7224 {
7225     \msg_error:nne { regex } { replacement-#1 } {#3}
7226     #2 #3
7227 }
```

(End of definition for _regex_replacement_error:NNN.)

46.7 User functions

`\regex_new:N` Before being assigned a sensible value, a regex variable matches nothing.

```
7228 \cs_new_protected:Npn \regex_new:N #1
7229 { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(End of definition for \regex_new:N. This function is documented on page 55.)

`\l_tmpa_regex` The usual scratch space.

```
\l_tmpb_regex 7230 \regex_new:N \l_tmpa_regex
\g_tmpa_regex 7231 \regex_new:N \l_tmpb_regex
\g_tmpb_regex 7232 \regex_new:N \g_tmpa_regex
7233 \regex_new:N \g_tmpb_regex
```

(End of definition for \l_tmpa_regex and others. These variables are documented on page 60.)

`\regex_set:Nn` Compile, then store the result in the user variable with the appropriate assignment function.
`\regex_gset:Nn`
`\regex_const:Nn`

```
7234 \cs_new_protected:Npn \regex_set:Nn #1#2
7235 {
7236     \__regex_compile:n {#2}
7237     \tl_set_eq:NN #1 \l__regex_internal_regex
7238 }
7239 \cs_new_protected:Npn \regex_gset:Nn #1#2
7240 {
7241     \__regex_compile:n {#2}
7242     \tl_gset_eq:NN #1 \l__regex_internal_regex
7243 }
7244 \cs_new_protected:Npn \regex_const:Nn #1#2
7245 {
7246     \__regex_compile:n {#2}
7247     \tl_const:Ne #1 { \exp_not:o \l__regex_internal_regex }
7248 }
```

(End of definition for \regex_set:Nn, \regex_gset:Nn, and \regex_const:Nn. These functions are documented on page 55.)

`\regex_show:n` User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary `__regex_show:N` is defined in a different section.

```
\__regex_show:Nn 7249 \cs_new_protected:Npn \regex_show:n { \__regex_show:Nn \msg_show:nneeee }
\regex_show:N 7250 \cs_new_protected:Npn \regex_log:n { \__regex_show:Nn \msg_log:nneeee }
\regex_log:N 7251 \cs_new_protected:Npn \__regex_show:Nn #1#2
\__regex_show:NN
```

```

7252 {
7253   \__regex_compile:n {#2}
7254   \__regex_show:N \l__regex_internal_regex
7255   #1 { regex } { show }
7256   { \tl_to_str:n {#2} } { }
7257   { \l__regex_internal_a_tl } { }
7258 }
7259 \cs_new_protected:Npn \regex_show:N { \__regex_show:NN \msg_show:nneeee }
7260 \cs_new_protected:Npn \regex_log:N { \__regex_show:NN \msg_log:nneeee }
7261 \cs_new_protected:Npn \__regex_show:NN #1#2
7262 {
7263   \__kernel_chk_tl_type:NnnT #2 { regex }
7264   { \exp_args:No \__regex_clean_regex:n {#2} }
7265   {
7266     \__regex_show:N #2
7267     #1 { regex } { show }
7268     { } { \token_to_str:N #2 }
7269     { \l__regex_internal_a_tl } { }
7270   }
7271 }

```

(End of definition for `\regex_show:n` and others. These functions are documented on page 55.)

`\regex_match:nnTF` Those conditionals are based on a common auxiliary defined later. Its first argument builds the NFA corresponding to the regex, and the second argument is the query token list. Once we have performed the match, convert the resulting boolean to `\prg_return_true:` or false.

```

7272 \prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
7273 {
7274   \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
7275   \__regex_return:
7276 }
7277 \prg_generate_conditional_variant:Nnn \regex_match:nn { nV } { T , F , TF }
7278 \prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
7279 {
7280   \__regex_if_match:nn { \__regex_build:N #1 } {#2}
7281   \__regex_return:
7282 }
7283 \prg_generate_conditional_variant:Nnn \regex_match:Nn { NV } { T , F , TF }

```

(End of definition for `\regex_match:nnTF` and `\regex_match:NnTF`. These functions are documented on page 56.)

`\regex_count:nnN` Again, use an auxiliary whose first argument builds the NFA.

```

7284 \cs_new_protected:Npn \regex_count:nnN #1
7285 { \__regex_count:nnN { \__regex_build:n {#1} } }
7286 \cs_new_protected:Npn \regex_count:NnN #1
7287 { \__regex_count:nnN { \__regex_build:N #1 } }
7288 \cs_generate_variant:Nn \regex_count:nnN { nV }
7289 \cs_generate_variant:Nn \regex_count:NnN { NV }

```

(End of definition for `\regex_count:nnN` and `\regex_count:NnN`. These functions are documented on page 56.)

`\regex_match_case:nn`
`\regex_match_case:nnTF`

The auxiliary errors if #1 has an odd number of items, and otherwise it sets `\g__regex_case_int` according to which case was found (zero if not found). The true branch leaves the corresponding code in the input stream.

```
7290 \cs_new_protected:Npn \regex_match_case:nnTF #1#2#3
7291 {
7292   \__regex_match_case:nnTF {#1} {#2}
7293   {
7294     \tl_item:nn {#1} { 2 * \g__regex_case_int }
7295     #3
7296   }
7297 }
7298 \cs_new_protected:Npn \regex_match_case:nn #1#2
7299 { \regex_match_case:nnTF {#1} {#2} { } { } }
7300 \cs_new_protected:Npn \regex_match_case:nnT #1#2#3
7301 { \regex_match_case:nnTF {#1} {#2} {#3} { } }
7302 \cs_new_protected:Npn \regex_match_case:nnF #1#2
7303 { \regex_match_case:nnTF {#1} {#2} { } }
```

(End of definition for `\regex_match_case:nnTF`. This function is documented on page 56.)

`\regex_extract_once:nnN`
`\regex_extract_once:nVN`
`\regex_extract_once:nnNTF`
`\regex_extract_once:nVNTF`
`\regex_extract_once:NnN`
`\regex_extract_once:NVN`
`\regex_extract_once:NnNTF`
`\regex_extract_once:NVNTF`
`\regex_extract_all:nnN`
`\regex_extract_all:nVN`
`\regex_extract_all:nnNTF`
`\regex_extract_all:nVNTF`
`\regex_extract_all:NnN`
`\regex_extract_all:NVN`
`\regex_extract_all:NnNTF`
`\regex_extract_all:NVNTF`
`\regex_replace_once:nnN`
`\regex_replace_once:nVN`
`\regex_replace_once:nnNTF`
`\regex_replace_once:nVNTF`
`\regex_replace_once:NnN`
`\regex_replace_once:NVN`
`\regex_replace_once:NnNTF`
`\regex_replace_once:NVNTF`
`\regex_replace_all:nnN`
`\regex_replace_all:nVN`
`\regex_replace_all:nnNTF`
`\regex_replace_all:nVNTF`

We define here 40 user functions, following a common pattern in terms of `:nnN` auxiliaries, defined in the coming subsections. The auxiliary is handed `__regex_build:n` or `__regex_build:N` with the appropriate regex argument, then all other necessary arguments (replacement text, token list, *etc.* The conditionals call `__regex_return:` to return either true or false once matching has been performed.

```
7304 \cs_set_protected:Npn \__regex_tmp:w #1#2#3
7305 {
7306   \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
7307   \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N ##1 } }
7308   \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
7309   { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
7310   \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
7311   { #1 { \__regex_build:N ##1 } {##2} ##3 \__regex_return: }
7312   \cs_generate_variant:Nn #2 { nV }
7313   \prg_generate_conditional_variant:Nnn #2 { nV } { T , F , TF }
7314   \cs_generate_variant:Nn #3 { NV }
7315   \prg_generate_conditional_variant:Nnn #3 { NV } { T , F , TF }
7316 }
7317 \__regex_tmp:w \__regex_extract_once:nnN
7318 \regex_extract_once:nnN \regex_extract_once:NnN
7319 \__regex_tmp:w \__regex_extract_all:nnN
7320 \regex_extract_all:nnN \regex_extract_all:NnN
7321 \__regex_tmp:w \__regex_replace_once:nnN
7322 \regex_replace_once:nnN \regex_replace_once:NnN
7323 \__regex_tmp:w \__regex_replace_all:nnN
7324 \regex_replace_all:nnN \regex_replace_all:NnN
7325 \__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN
```

(End of definition for `\regex_extract_once:nnNTF` and others. These functions are documented on page 57.)

`\regex_replace_all:NnN`
`\regex_replace_case_once:nnN`
`\regex_replace_all:NVN`
`\regex_replace_case_once:nnNTF`
`\regex_replace_all:NnNTF`
`\regex_replace_all:NVNTF`
`\regex_split:NnN`
`\regex_split:NVN`
`\regex_split:NnNTF`
`\regex_split:NVNTF`
`\regex_split:nnN`
`\regex_split:nVN`
`\regex_split:nnNTF`
`\regex_split:nVNTF`

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_once:nnN`, but with more complicated code to

build the automaton, and to find what replacement text to use. The `\tl_item:nn` is only expanded once we know the value of `\g__regex_case_int`, namely which case matched.

```

7326 \cs_new_protected:Npn \regex_replace_case_once:nNTF #1#2
7327 {
7328   \int_if_odd:nTF { \tl_count:n {#1} }
7329   {
7330     \msg_error:nneeee { regex } { case-odd }
7331     { \token_to_str:N \regex_replace_case_once:nN(TF) } { code }
7332     { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7333     \use_ii:nn
7334   }
7335   {
7336     \__regex_replace_once_aux:nnN
7337     { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
7338     { \__regex_replacement:e { \tl_item:nn {#1} { 2 * \g__regex_case_int } } }
7339     #2
7340     \bool_if:NTF \g__regex_success_bool
7341   }
7342 }
7343 \cs_new_protected:Npn \regex_replace_case_once:nN #1#2
7344 { \regex_replace_case_once:nNTF {#1} {#2} { } { } }
7345 \cs_new_protected:Npn \regex_replace_case_once:nNT #1#2#3
7346 { \regex_replace_case_once:nNTF {#1} {#2} {#3} { } }
7347 \cs_new_protected:Npn \regex_replace_case_once:nNF #1#2
7348 { \regex_replace_case_once:nNTF {#1} {#2} { } }

```

(End of definition for `\regex_replace_case_once:nNTF`. This function is documented on page 59.)

`\regex_replace_case_all:nN`
`\regex_replace_case_all:nNTF`

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_all:nnN`, but with more complicated code to build the automaton, and to find what replacement text to use.

```

7349 \cs_new_protected:Npn \regex_replace_case_all:nNTF #1#2
7350 {
7351   \int_if_odd:nTF { \tl_count:n {#1} }
7352   {
7353     \msg_error:nneeee { regex } { case-odd }
7354     { \token_to_str:N \regex_replace_case_all:nN(TF) } { code }
7355     { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7356     \use_ii:nn
7357   }
7358   {
7359     \__regex_replace_all_aux:nnN
7360     { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
7361     { \__regex_case_replacement:e { \__regex_tl_even_items:n {#1} } }
7362     #2
7363     \bool_if:NTF \g__regex_success_bool
7364   }
7365 }
7366 \cs_new_protected:Npn \regex_replace_case_all:nN #1#2
7367 { \regex_replace_case_all:nNTF {#1} {#2} { } { } }
7368 \cs_new_protected:Npn \regex_replace_case_all:nNT #1#2#3
7369 { \regex_replace_case_all:nNTF {#1} {#2} {#3} { } }
7370 \cs_new_protected:Npn \regex_replace_case_all:nNF #1#2
7371 { \regex_replace_case_all:nNTF {#1} {#2} { } }

```

(End of definition for `\regex_replace_case_all:nTF`. This function is documented on page 59.)

46.7.1 Variables and helpers for user functions

`\l__regex_match_count_int` The number of matches found so far is stored in `\l__regex_match_count_int`. This is only used in the `\regex_count:nnN` functions.

```
7372 \int_new:N \l__regex_match_count_int
```

(End of definition for `\l__regex_match_count_int`.)

`\l__regex_begin_flag` `\l__regex_end_flag` Those flags are raised to indicate begin-group or end-group tokens that had to be added when extracting submatches.

```
7373 \flag_new:N \l__regex_begin_flag
```

```
7374 \flag_new:N \l__regex_end_flag
```

(End of definition for `\l__regex_begin_flag` and `\l__regex_end_flag`.)

`\l__regex_min_submatch_int` `\l__regex_submatch_int` `\l__regex_zeroth_submatch_int` The end-points of each submatch are stored in two arrays whose index `<submatch>` ranges from `\l__regex_min_submatch_int` (inclusive) to `\l__regex_submatch_int` (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at `zeroth_submatch`. The entry `\l__regex_zeroth_submatch_int` in `\g__regex_submatch_prev_intarray` holds the position at which that match attempt started: this is used for splitting and replacements.

```
7375 \int_new:N \l__regex_min_submatch_int
```

```
7376 \int_new:N \l__regex_submatch_int
```

```
7377 \int_new:N \l__regex_zeroth_submatch_int
```

(End of definition for `\l__regex_min_submatch_int`, `\l__regex_submatch_int`, and `\l__regex_zeroth_submatch_int`.)

`\g__regex_submatch_prev_intarray` `\g__regex_submatch_begin_intarray` `\g__regex_submatch_end_intarray` `\g__regex_submatch_case_intarray` Hold the place where the match attempt begun, the end-points of each submatch, and which regex case the match corresponds to, respectively.

```
7378 \intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
```

```
7379 \intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
```

```
7380 \intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
```

```
7381 \intarray_new:Nn \g__regex_submatch_case_intarray { 65536 }
```

(End of definition for `\g__regex_submatch_prev_intarray` and others.)

`\g__regex_balance_intarray` The first thing we do when matching is to store the balance of begin-group/end-group characters into `\g__regex_balance_intarray`.

```
7382 \intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(End of definition for `\g__regex_balance_intarray`.)

`\l__regex_added_begin_int` `\l__regex_added_end_int` Keep track of the number of left/right braces to add when performing a regex operation such as a replacement.

```
7383 \int_new:N \l__regex_added_begin_int
```

```
7384 \int_new:N \l__regex_added_end_int
```

(End of definition for `\l__regex_added_begin_int` and `\l__regex_added_end_int`.)

`__regex_return:` This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not. It is used by all user conditionals.

```

7385 \cs_new_protected:Npn \__regex_return:
7386 {
7387   \if_meaning:w \c_true_bool \g__regex_success_bool
7388     \prg_return_true:
7389   \else:
7390     \prg_return_false:
7391   \fi:
7392 }

```

(End of definition for __regex_return:.)

`__regex_query_set:n` To easily extract subsets of the input once we found the positions at which to cut, store the input tokens one by one into successive `\toks` registers. Also store the brace balance (used to check for overall brace balance) in an array.

```

7393 \cs_new_protected:Npn \__regex_query_set:n #1
7394 {
7395   \int_zero:N \l__regex_balance_int
7396   \int_zero:N \l__regex_curr_pos_int
7397   \__regex_query_set_aux:nN { } F
7398   \tl_analysis_map_inline:nn {#1}
7399   { \__regex_query_set_aux:nN {##1} ##3 }
7400   \__regex_query_set_aux:nN { } F
7401   \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
7402 }
7403 \cs_new_protected:Npn \__regex_query_set_aux:nN #1#2
7404 {
7405   \int_incr:N \l__regex_curr_pos_int
7406   \__regex_toks_set:Nn \l__regex_curr_pos_int {#1}
7407   \__kernel_intarray_gset:Nnn \g__regex_balance_intarray
7408     \l__regex_curr_pos_int \l__regex_balance_int
7409   \if_case:w "#2 \exp_stop_f:
7410   \or: \int_incr:N \l__regex_balance_int
7411   \or: \int_decr:N \l__regex_balance_int
7412   \fi:
7413 }

```

(End of definition for __regex_query_set:n and __regex_query_set_aux:nN.)

46.7.2 Matching

`__regex_if_match:nn` We don't track submatches, and stop after a single match. Build the NFA with `#1`, and perform the match on the query `#2`.

```

7414 \cs_new_protected:Npn \__regex_if_match:nn #1#2
7415 {
7416   \group_begin:
7417     \__regex_disable_submatches:
7418     \__regex_single_match:
7419     #1
7420     \__regex_match:n {#2}
7421   \group_end:
7422 }

```

(End of definition for `__regex_if_match:nn`.)

`__regex_match_case:nnTF`
`__regex_match_case_aux:nn` The code would get badly messed up if the number of items in #1 were not even, so we catch this case, then follow the same code as `\regex_match:nnTF` but using `__regex_case_build:n` and without returning a result.

```
7423 \cs_new_protected:Npn \__regex_match_case:nnTF #1#2
7424   {
7425     \int_if_odd:nTF { \tl_count:n {#1} }
7426     {
7427       \msg_error:nneeee { regex } { case-odd }
7428       { \token_to_str:N \regex_match_case:nn(TF) } { code }
7429       { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7430       \use_ii:nn
7431     }
7432     {
7433       \__regex_if_match:nn
7434       { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
7435       {#2}
7436       \bool_if:NTF \g__regex_success_bool
7437     }
7438   }
7439 \cs_new:Npn \__regex_match_case_aux:nn #1#2 { \exp_not:n { {#1} } }
```

(End of definition for `__regex_match_case:nnTF` and `__regex_match_case_aux:nn`.)

`__regex_count:nnN` Again, we don't care about submatches. Instead of aborting after the first “longest match” is found, we search for multiple matches, incrementing `\l__regex_match_count_int` every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```
7440 \cs_new_protected:Npn \__regex_count:nnN #1#2#3
7441   {
7442     \group_begin:
7443     \__regex_disable_submatches:
7444     \int_zero:N \l__regex_match_count_int
7445     \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
7446     #1
7447     \__regex_match:n {#2}
7448     \exp_args:NNNo
7449     \group_end:
7450     \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
7451   }
```

(End of definition for `__regex_count:nnN`.)

46.7.3 Extracting submatches

`__regex_extract_once:nnN`
`__regex_extract_all:nnN` Match once or multiple times. After each match (or after the only match), extract the submatches using `__regex_extract:.`. At the end, store the sequence containing all the submatches into the user variable #3 after closing the group.

```
7452 \cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
7453   {
7454     \group_begin:
7455     \__regex_single_match:
```

```

7456     #1
7457     \__regex_match:n {#2}
7458     \__regex_extract:
7459     \__regex_query_set:n {#2}
7460     \__regex_group_end_extract_seq:N #3
7461   }
7462 \cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
7463 {
7464   \group_begin:
7465     \__regex_multi_match:n { \__regex_extract: }
7466     #1
7467     \__regex_match:n {#2}
7468     \__regex_query_set:n {#2}
7469     \__regex_group_end_extract_seq:N #3
7470   }

```

(End of definition for `__regex_extract_once:nnN` and `__regex_extract_all:nnN`.)

`__regex_split:nnN` Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement `\l__regex_submatch_int`, which controls which matches will be used.

```

7471 \cs_new_protected:Npn \__regex_split:nnN #1#2#3
7472 {
7473   \group_begin:
7474     \__regex_multi_match:n
7475     {
7476       \if_int_compare:w
7477         \l__regex_start_pos_int < \l__regex_success_pos_int
7478         \__regex_extract:
7479         \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7480         \l__regex_zeroth_submatch_int \c_zero_int
7481         \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7482         \l__regex_zeroth_submatch_int
7483         {
7484           \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
7485           \l__regex_zeroth_submatch_int
7486         }
7487         \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7488         \l__regex_zeroth_submatch_int
7489         \l__regex_start_pos_int
7490       \fi:
7491     }
7492     #1
7493     \__regex_match:n {#2}
7494     \__regex_query_set:n {#2}
7495     \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7496     \l__regex_submatch_int \c_zero_int
7497     \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7498     \l__regex_submatch_int
7499     \l__regex_max_pos_int

```

```

7500     \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7501     \l__regex_submatch_int
7502     \l__regex_start_pos_int
7503     \int_incr:N \l__regex_submatch_int
7504     \if_meaning:w \c_true_bool \l__regex_empty_success_bool
7505     \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
7506     \int_decr:N \l__regex_submatch_int
7507     \fi:
7508     \fi:
7509     \__regex_group_end_extract_seq:N #3
7510 }

```

(End of definition for __regex_split:nnN.)

__regex_group_end_extract_seq:N The end-points of submatches are stored as entries of two arrays from \l__regex_min_submatch_int to \l__regex_submatch_int (exclusive). Extract the relevant ranges into \g__regex_internal_tl, separated by __regex_tmp:w {}. We keep track in the two flags __regex_begin and __regex_end of the number of begin-group or end-group tokens added to make each of these items overall balanced. At this step, }{ is counted as being balanced (same number of begin-group and end-group tokens). This problem is caught by __regex_extract_check:w, explained later. After complaining about any begin-group or end-group tokens we had to add, we are ready to construct the user's sequence outside the group.

```

7511 \cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
7512 {
7513     \flag_clear:N \l__regex_begin_flag
7514     \flag_clear:N \l__regex_end_flag
7515     \cs_set_eq:NN \__regex_tmp:w \scan_stop:
7516     \__kernel_tl_gset:Nx \g__regex_internal_tl
7517     {
7518         \int_step_function:nnN \l__regex_min_submatch_int
7519         { \l__regex_submatch_int - \c_one_int } \__regex_extract_seq_aux:n
7520         \__regex_tmp:w
7521     }
7522     \int_set:Nn \l__regex_added_begin_int
7523     { \flag_height:N \l__regex_begin_flag }
7524     \int_set:Nn \l__regex_added_end_int
7525     { \flag_height:N \l__regex_end_flag }
7526     \tex_afterassignment:D \__regex_extract_check:w
7527     \__kernel_tl_gset:Nx \g__regex_internal_tl
7528     { \g__regex_internal_tl \if_false: { \fi: } }
7529     \int_compare:nNnT
7530     { \l__regex_added_begin_int + \l__regex_added_end_int } > \c_zero_int
7531     {
7532         \msg_error:nneee { regex } { result-unbalanced }
7533         { splitting-or-extracting-submatches }
7534         { \int_use:N \l__regex_added_begin_int }
7535         { \int_use:N \l__regex_added_end_int }
7536     }
7537     \group_end:
7538     \__regex_extract_seq:N #1
7539 }
7540 \cs_gset_protected:Npn \__regex_extract_seq:N #1
7541 {

```

```

7542 \seq_clear:N #1
7543 \cs_set_eq:NN \__regex_tmp:w \__regex_extract_seq_loop:Nw
7544 \exp_after:wN \__regex_extract_seq:NNn
7545 \exp_after:wN #1
7546 \g__regex_internal_tl \use_none:nnn
7547 }
7548 \cs_new_protected:Npn \__regex_extract_seq:NNn #1#2#3
7549 { #3 #2 #1 \prg_do_nothing: }
7550 \cs_new_protected:Npn \__regex_extract_seq_loop:Nw #1#2 \__regex_tmp:w #3
7551 {
7552 \seq_put_right:No #1 {#2}
7553 #3 \__regex_extract_seq_loop:Nw #1 \prg_do_nothing:
7554 }

```

(End of definition for __regex_group_end_extract_seq:N and others.)

__regex_extract_seq_aux:n The :n auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```

7555 \cs_new:Npn \__regex_extract_seq_aux:n #1
7556 {
7557 \__regex_tmp:w { }
7558 \exp_after:wN \__regex_extract_seq_aux:ww
7559 \int_value:w \__regex_submatch_balance:n {#1} ; #1;
7560 }
7561 \cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
7562 {
7563 \if_int_compare:w #1 < \c_zero_int
7564 \prg_replicate:nn {-#1}
7565 {
7566 \flag_raise:N \l__regex_begin_flag
7567 \exp_not:n { { \if_false: } \fi: }
7568 }
7569 \fi:
7570 \__regex_query_submatch:n {#2}
7571 \if_int_compare:w #1 > \c_zero_int
7572 \prg_replicate:nn {#1}
7573 {
7574 \flag_raise:N \l__regex_end_flag
7575 \exp_not:n { \if_false: { \fi: } }
7576 }
7577 \fi:
7578 }

```

(End of definition for __regex_extract_seq_aux:n and __regex_extract_seq_aux:ww.)

__regex_extract_check:w In __regex_group_end_extract_seq:N we had to expand \g__regex_internal_tl to turn \if_false: constructions into actual begin-group and end-group tokens. This is done with a __kernel_tl_gset:Nx assignment, and __regex_extract_check:w is run immediately after this assignment ends, thanks to the \afterassignment primitive. If all of the items were properly balanced (enough begin-group tokens before end-group tokens, so }{ is not) then __regex_extract_check:w is called just before the closing brace of the __kernel_tl_gset:Nx (thanks to our sneaky \if_false: { \fi: } construction), and finds that there is nothing left to expand. If any of the items is unbalanced, the

assignment gets ended early by an extra end-group token, and our check finds more tokens needing to be expanded in a new `__kernel_tl_gset:Nx` assignment. We need to add a begin-group and an end-group tokens to the unbalanced item, namely to the last item found so far, which we reach through a loop.

```

7579 \cs_new_protected:Npn \__regex_extract_check:w
7580 {
7581   \exp_after:wN \__regex_extract_check:n
7582   \exp_after:wN { \if_false: } \fi:
7583 }
7584 \cs_new_protected:Npn \__regex_extract_check:n #1
7585 {
7586   \tl_if_empty:nF {#1}
7587   {
7588     \int_incr:N \l__regex_added_begin_int
7589     \int_incr:N \l__regex_added_end_int
7590     \tex_afterassignment:D \__regex_extract_check:w
7591     \__kernel_tl_gset:Nx \g__regex_internal_tl
7592     {
7593       \exp_after:wN \__regex_extract_check_loop:w
7594       \g__regex_internal_tl
7595       \__regex_tmp:w \__regex_extract_check_end:w
7596       #1
7597     }
7598   }
7599 }
7600 \cs_new:Npn \__regex_extract_check_loop:w #1 \__regex_tmp:w #2
7601 {
7602   #2
7603   \exp_not:o {#1}
7604   \__regex_tmp:w { }
7605   \__regex_extract_check_loop:w \prg_do_nothing:
7606 }

```

Arguments of `__regex_extract_check_end:w` are: #1 is the part of the item before the extra end-group token; #2 is junk; #3 is `\prg_do_nothing:` followed by the not-yet-expanded part of the item after the extra end-group token. In the replacement text, the first brace and the `\if_false: { \fi: }` construction are the added begin-group and end-group tokens (the latter being not-yet expanded, just like #3), while the closing brace after `\exp_not:o {#1}` replaces the extra end-group token that had ended the assignment early. In particular this means that the character code of that end-group token is lost.

```

7607 \cs_new:Npn \__regex_extract_check_end:w
7608   \exp_not:o #1#2 \__regex_extract_check_loop:w #3 \__regex_tmp:w
7609 {
7610   { \exp_not:o {#1} }
7611   #3
7612   \if_false: { \fi: }
7613   \__regex_tmp:w
7614 }

```

(End of definition for `__regex_extract_check:w` and others.)

`__regex_extract:` Our task here is to store the list of end-points of submatches, and store them in appropriate array entries, from `\l__regex_zeroth_submatch_int` upwards. First, we store in `\g__regex_submatch_prev_intarray` the position at which the match attempt started.

We extract the rest from the comma list `\l__regex_success_submatches_tl`, which starts with entries to be stored in `\g__regex_submatch_begin_intarray` and continues with entries for `\g__regex_submatch_end_intarray`.

```

7615 \cs_new_protected:Npn \__regex_extract:
7616 {
7617   \if_meaning:w \c_true_bool \g__regex_success_bool
7618     \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
7619     \prg_replicate:nn \l__regex_capturing_group_int
7620     {
7621       \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7622         \l__regex_submatch_int \c_zero_int
7623       \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
7624         \l__regex_submatch_int \c_zero_int
7625       \int_incr:N \l__regex_submatch_int
7626     }
7627   \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7628     \l__regex_zeroth_submatch_int \l__regex_start_pos_int
7629   \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
7630     \l__regex_zeroth_submatch_int \g__regex_case_int
7631   \int_zero:N \l__regex_internal_a_int
7632   \exp_after:wN \__regex_extract_aux:w \l__regex_success_submatches_tl
7633     \prg_break_point: \__regex_use_none_delimit_by_q_recursion_stop:w ,
7634     \q__regex_recursion_stop
7635   \fi:
7636 }
7637 \cs_new_protected:Npn \__regex_extract_aux:w #1 ,
7638 {
7639   \prg_break: #1 \prg_break_point:
7640   \if_int_compare:w \l__regex_internal_a_int < \l__regex_capturing_group_int
7641     \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7642       { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int } {#1}
7643   \else:
7644     \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7645     {
7646       \__regex_int_eval:w
7647         \l__regex_zeroth_submatch_int + \l__regex_internal_a_int
7648         - \l__regex_capturing_group_int
7649     }
7650     {#1}
7651   \fi:
7652   \int_incr:N \l__regex_internal_a_int
7653   \__regex_extract_aux:w
7654 }

```

(End of definition for `__regex_extract:` and `__regex_extract_aux:w`.)

46.7.4 Replacement

`__regex_replace_once:nnN` Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for

this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional e-expansion, and checks that braces are balanced in the final result.

```

7655 \cs_new_protected:Npn \__regex_replace_once:nnN #1#2
7656   { \__regex_replace_once_aux:nnN {#1} { \__regex_replacement:n {#2} } }
7657 \cs_new_protected:Npn \__regex_replace_once_aux:nnN #1#2#3
7658   {
7659     \group_begin:
7660     \__regex_single_match:
7661     #1
7662     \exp_args:No \__regex_match:n {#3}
7663     \bool_if:NTF \g__regex_success_bool
7664     {
7665       \__regex_extract:
7666       \exp_args:No \__regex_query_set:n {#3}
7667       #2
7668       \int_set:Nn \l__regex_balance_int
7669         { \__regex_replacement_balance_one_match:n \l__regex_zeroth_submatch_int }
7670       \__kernel_tl_set:Nx \l__regex_internal_a_tl
7671       {
7672         \__regex_replacement_do_one_match:n \l__regex_zeroth_submatch_int
7673         \__regex_query_range:nn
7674         {
7675           \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
7676             \l__regex_zeroth_submatch_int
7677         }
7678         \l__regex_max_pos_int
7679       }
7680       \__regex_group_end_replace:N #3
7681     }
7682     { \group_end: }
7683   }

```

(End of definition for `__regex_replace_once:nnN` and `__regex_replace_once_aux:nnN`.)

`__regex_replace_all:nnN` Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` hold information about submatches of every match in order; each match corresponds to `\l__regex_capturing_group_int` consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```

7684 \cs_new_protected:Npn \__regex_replace_all:nnN #1#2
7685   { \__regex_replace_all_aux:nnN {#1} { \__regex_replacement:n {#2} } }
7686 \cs_new_protected:Npn \__regex_replace_all_aux:nnN #1#2#3
7687   {
7688     \group_begin:
7689     \__regex_multi_match:n { \__regex_extract: }
7690     #1
7691     \exp_args:No \__regex_match:n {#3}
7692     \exp_args:No \__regex_query_set:n {#3}
7693     #2

```



```

7694     \int_set:Nn \l__regex_balance_int
7695     {
7696         \c_zero_int
7697         \int_step_function:nnnN
7698         \l__regex_min_submatch_int
7699         \l__regex_capturing_group_int
7700         { \l__regex_submatch_int - \c_one_int }
7701         \__regex_replacement_balance_one_match:n
7702     }
7703     \__kernel_tl_set:Nx \l__regex_internal_a_tl
7704     {
7705         \int_step_function:nnnN
7706         \l__regex_min_submatch_int
7707         \l__regex_capturing_group_int
7708         { \l__regex_submatch_int - \c_one_int }
7709         \__regex_replacement_do_one_match:n
7710         \__regex_query_range:nn
7711         \l__regex_start_pos_int \l__regex_max_pos_int
7712     }
7713     \__regex_group_end_replace:N #3
7714 }

```

(End of definition for __regex_replace_all:nnN.)

```

\__regex_group_end_replace:N
  \__regex_group_end_replace_try:
  \__regex_group_end_replace_check:w
  \__regex_group_end_replace_check:n

```

At this stage `\l__regex_internal_a_tl` (e-expands to the desired result). Guess from `\l__regex_balance_int` the number of braces to add before or after the result then try expanding. The simplest case is when `\l__regex_internal_a_tl` together with the braces we insert via `\prg_replicate:nn` give a balanced result, and the assignment ends at the `\if_false: { \fi: }` construction: then `__regex_group_end_replace_check:w` sees that there is no material left and we successfully found the result. The harder case is that expanding `\l__regex_internal_a_tl` may produce extra closing braces and end the assignment early. Then we grab the remaining code using; importantly, what follows has not yet been expanded so that `__regex_group_end_replace_check:n` grabs everything until the last brace in `__regex_group_end_replace_try:`, letting us try again with an extra surrounding pair of braces.

```

7715 \cs_new_protected:Npn \__regex_group_end_replace:N #1
7716 {
7717     \int_set:Nn \l__regex_added_begin_int
7718     { \int_max:nn { - \l__regex_balance_int } \c_zero_int }
7719     \int_set:Nn \l__regex_added_end_int
7720     { \int_max:nn \l__regex_balance_int \c_zero_int }
7721     \__regex_group_end_replace_try:
7722     \int_compare:nNnT { \l__regex_added_begin_int + \l__regex_added_end_int }
7723     > \c_zero_int
7724     {
7725         \msg_error:nneee { regex } { result-unbalanced }
7726         { replacing } { \int_use:N \l__regex_added_begin_int }
7727         { \int_use:N \l__regex_added_end_int }
7728     }
7729     \group_end:
7730     \tl_set_eq:NN #1 \g__regex_internal_tl
7731 }
7732 \cs_new_protected:Npn \__regex_group_end_replace_try:

```

```

7733 {
7734   \tex_afterassignment:D \__regex_group_end_replace_check:w
7735   \__kernel_tl_gset:Nx \g__regex_internal_tl
7736   {
7737     \prg_replicate:nn \l__regex_added_begin_int { { \if_false: } \fi: }
7738     \l__regex_internal_a_tl
7739     \prg_replicate:nn \l__regex_added_end_int { \if_false: { \fi: } }
7740     \if_false: { \fi: }
7741   }
7742 }
7743 \cs_new_protected:Npn \__regex_group_end_replace_check:w
7744 {
7745   \exp_after:wN \__regex_group_end_replace_check:n
7746   \exp_after:wN { \if_false: } \fi:
7747 }
7748 \cs_new_protected:Npn \__regex_group_end_replace_check:n #1
7749 {
7750   \tl_if_empty:nF {#1}
7751   {
7752     \int_incr:N \l__regex_added_begin_int
7753     \int_incr:N \l__regex_added_end_int
7754     \__regex_group_end_replace_try:
7755   }
7756 }

```

(End of definition for __regex_group_end_replace:N and others.)

46.7.5 Peeking ahead

\l__regex_peek_true_tl True/false code arguments of \peek_regex:nTF or similar.

```

\l__regex_peek_false_tl
7757 \tl_new:N \l__regex_peek_true_tl
7758 \tl_new:N \l__regex_peek_false_tl

```

(End of definition for \l__regex_peek_true_tl and \l__regex_peek_false_tl.)

\l__regex_replacement_tl When peeking in \peek_regex_replace_once:nnTF we need to store the replacement text.

```

7759 \tl_new:N \l__regex_replacement_tl

```

(End of definition for \l__regex_replacement_tl.)

\l__regex_input_tl Stores each token found as __regex_input_item:n {<tokens>}, where the <tokens> o-expand to the token found, as for \tl_analysis_map_inline:nn.

```

7760 \tl_new:N \l__regex_input_tl
7761 \cs_new_eq:NN \__regex_input_item:n ?

```

(End of definition for \l__regex_input_tl and __regex_input_item:n.)

\peek_regex:nTF

\peek_regex:NTF

\peek_regex_remove_once:nTF

\peek_regex_remove_once:NTF

The T and F functions just call the corresponding TF function. The four TF functions differ along two axes: whether to remove the token or not, distinguished by using __regex_peek_end: or __regex_peek_remove_end:n (the latter case needs an argument, as we will see), and whether the regex has to be compiled or is already in an N-type variable, distinguished by calling __regex_build_aux:Nn or __regex_build_aux:NN. The first argument of these functions is \c_false_bool to indicate that there should be no implicit

insertion of a wildcard at the start of the pattern: otherwise the code would keep looking further into the input stream until matching the regex.

```

7762 \cs_new_protected:Npn \peek_regex:nTF #1
7763 {
7764   \__regex_peek:nnTF
7765     { \__regex_build_aux:Nn \c_false_bool {#1} }
7766     { \__regex_peek_end: }
7767 }
7768 \cs_new_protected:Npn \peek_regex:nT #1#2
7769 { \peek_regex:nTF {#1} {#2} { } }
7770 \cs_new_protected:Npn \peek_regex:nF #1 { \peek_regex:nTF {#1} { } }
7771 \cs_new_protected:Npn \peek_regex:NTF #1
7772 {
7773   \__regex_peek:nnTF
7774     { \__regex_build_aux:NN \c_false_bool #1 }
7775     { \__regex_peek_end: }
7776 }
7777 \cs_new_protected:Npn \peek_regex:NT #1#2
7778 { \peek_regex:NTF #1 {#2} { } }
7779 \cs_new_protected:Npn \peek_regex:NF #1 { \peek_regex:NTF {#1} { } }
7780 \cs_new_protected:Npn \peek_regex_remove_once:nTF #1
7781 {
7782   \__regex_peek:nnTF
7783     { \__regex_build_aux:Nn \c_false_bool {#1} }
7784     { \__regex_peek_remove_end:n {##1} }
7785 }
7786 \cs_new_protected:Npn \peek_regex_remove_once:nT #1#2
7787 { \peek_regex_remove_once:nTF {#1} {#2} { } }
7788 \cs_new_protected:Npn \peek_regex_remove_once:nF #1
7789 { \peek_regex_remove_once:nTF {#1} { } }
7790 \cs_new_protected:Npn \peek_regex_remove_once:NTF #1
7791 {
7792   \__regex_peek:nnTF
7793     { \__regex_build_aux:NN \c_false_bool #1 }
7794     { \__regex_peek_remove_end:n {##1} }
7795 }
7796 \cs_new_protected:Npn \peek_regex_remove_once:NT #1#2
7797 { \peek_regex_remove_once:NTF #1 {#2} { } }
7798 \cs_new_protected:Npn \peek_regex_remove_once:NF #1
7799 { \peek_regex_remove_once:NTF #1 { } }

```

(End of definition for `\peek_regex:nTF` and others. These functions are documented on page 209.)

`__regex_peek:nnTF`
`__regex_peek_aux:nnTF`

Store the user's true/false codes (plus `\group_end:`) into two token lists. Then build the automaton with `#1`, without submatch tracking, and aiming for a single match. Then start matching by setting up a few variables like for any regex matching like `\regex_match:nnTF`, with the addition of `\l__regex_input_tl` that keeps track of the tokens seen, to reinsert them at the end. Instead of `\tl_analysis_map_inline:nn` on the input, we call `\peek_analysis_map_inline:n` to go through tokens in the input stream. Since `__regex_match_one_token:nnN` calls `__regex_maplike_break:` we need to catch that and break the `\peek_analysis_map_inline:n` loop instead.

```

7800 \cs_new_protected:Npn \__regex_peek:nnTF #1
7801 {

```

```

7802 \__regex_peek_aux:nnTF
7803 {
7804   \__regex_disable_submatches:
7805   #1
7806 }
7807 }
7808 \cs_new_protected:Npn \__regex_peek_aux:nnTF #1#2#3#4
7809 {
7810   \group_begin:
7811   \tl_set:Nn \l__regex_peek_true_tl { \group_end: #3 }
7812   \tl_set:Nn \l__regex_peek_false_tl { \group_end: #4 }
7813   \__regex_single_match:
7814   #1
7815   \__regex_match_init:
7816   \tl_build_begin:N \l__regex_input_tl
7817   \__regex_match_once_init:
7818   \peek_analysis_map_inline:n
7819   {
7820     \tl_build_put_right:Nn \l__regex_input_tl
7821     { \__regex_input_item:n {##1} }
7822     \__regex_match_one_token:nnN {##1} {##2} ##3
7823     \use_none:nnn
7824     \prg_break_point:Nn \__regex_maplike_break:
7825     { \peek_analysis_map_break:n {#2} }
7826   }
7827 }

```

(End of definition for __regex_peek:nnTF and __regex_peek_aux:nnTF.)

__regex_peek_end: Once the regex matches (or permanently fails to match) we call __regex_peek_end:, or __regex_peek_remove_end:n with argument the last token seen. For \peek_regex:nTF we reinsert tokens seen by calling __regex_peek_reinsert:N regardless of the result of the match. For \peek_regex_remove_once:nTF we reinsert the tokens seen only if the match failed; otherwise we just reinsert the tokens #1, with one expansion. To be more precise, #1 consists of tokens that o-expand and e-expand to the last token seen, for example it is \exp_not:N <cs> for a control sequence. This means that just doing \exp_after:wN \l__regex_peek_true_tl #1 would be unsafe because the expansion of <cs> would be suppressed.

```

7828 \cs_new_protected:Npn \__regex_peek_end:
7829 {
7830   \bool_if:NTF \g__regex_success_bool
7831   { \__regex_peek_reinsert:N \l__regex_peek_true_tl }
7832   { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
7833 }
7834 \cs_new_protected:Npn \__regex_peek_remove_end:n #1
7835 {
7836   \bool_if:NTF \g__regex_success_bool
7837   { \exp_args:NNo \use:nn \l__regex_peek_true_tl {#1} }
7838   { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
7839 }

```

(End of definition for __regex_peek_end: and __regex_peek_remove_end:n.)

`__regex_peek_reinsert:N` Insert the true/false code #1, followed by the tokens found, which were stored in `\l__-regex_input_tl`. For this, loop through that token list using `__regex_reinsert_item:n`, which expands #1 once to get a single token, and jumps over it to expand what follows, with suitable `\exp:w` and `\exp_end:`. We cannot just use `\use:e` on the whole token list because the result may be unbalanced, which would stop the primitive prematurely, or let it continue beyond where we would like.

```

7840 \cs_new_protected:Npn \__regex_peek_reinsert:N #1
7841 {
7842   \tl_build_end:N \l__regex_input_tl
7843   \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
7844   \exp_after:wN #1 \exp:w \l__regex_input_tl \exp_end:
7845 }
7846 \cs_new_protected:Npn \__regex_reinsert_item:n #1
7847 {
7848   \exp_after:wN \exp_after:wN
7849   \exp_after:wN \exp_end:
7850   \exp_after:wN \exp_after:wN
7851   #1
7852   \exp:w
7853 }

```

(End of definition for `__regex_peek_reinsert:N` and `__regex_reinsert_item:n`.)

`\peek_regex_replace_once:nm` Similar to `\peek_regex:nTF` above.

```

7854 \cs_new_protected:Npn \peek_regex_replace_once:nmTF #1
7855 { \__regex_peek_replace:nmTF { \__regex_build_aux:Nn \c_false_bool {#1} } }
7856 \cs_new_protected:Npn \peek_regex_replace_once:nmT #1#2#3
7857 { \peek_regex_replace_once:nmTF {#1} {#2} {#3} { } }
7858 \cs_new_protected:Npn \peek_regex_replace_once:nmF #1#2
7859 { \peek_regex_replace_once:nmTF {#1} {#2} { } }
7860 \cs_new_protected:Npn \peek_regex_replace_once:nm #1#2
7861 { \peek_regex_replace_once:nmTF {#1} {#2} { } { } }
7862 \cs_new_protected:Npn \peek_regex_replace_once:NmTF #1
7863 { \__regex_peek_replace:nmTF { \__regex_build_aux:NN \c_false_bool #1 } }
7864 \cs_new_protected:Npn \peek_regex_replace_once:NmT #1#2#3
7865 { \peek_regex_replace_once:NmTF #1 {#2} {#3} { } }
7866 \cs_new_protected:Npn \peek_regex_replace_once:NmF #1#2
7867 { \peek_regex_replace_once:NmTF #1 {#2} { } }
7868 \cs_new_protected:Npn \peek_regex_replace_once:Nm #1#2
7869 { \peek_regex_replace_once:NmTF #1 {#2} { } { } }

```

(End of definition for `\peek_regex_replace_once:nmTF` and `\peek_regex_replace_once:NmTF`. These functions are documented on page 210.)

`__regex_peek_replace:nmTF` Same as `__regex_peek:nmTF` (used for `\peek_regex:nTF` above), but without disabling submatches, and with a different end. The replacement text #2 is stored, to be analyzed later.

```

7870 \cs_new_protected:Npn \__regex_peek_replace:nmTF #1#2
7871 {
7872   \tl_set:Nn \l__regex_replacement_tl {#2}
7873   \__regex_peek_aux:nmTF {#1} { \__regex_peek_replace_end: }
7874 }

```

(End of definition for `__regex_peek_replace:nmTF`.)

`__regex_peek_replace_end:` If the match failed `__regex_peek_reinsert:N` reinserts the tokens found. Otherwise, finish storing the submatch information using `__regex_extract:`, and store the input into `\toks`. Redefine a few auxiliaries to change slightly their expansion behaviour as explained below. Analyse the replacement text with `__regex_replacement:n`, which as usual defines `__regex_replacement_do_one_match:n` to insert the tokens from the start of the match attempt to the beginning of the match, followed by the replacement text. The `\use:e` expands for instance the trailing `__regex_query_range:nn` down to a sequence of `__regex_reinsert_item:n` `{\tokens}` where `\tokens` o-expand to a single token that we want to insert. After e-expansion, `\use:e` does `\use:n`, so we have `\exp_after:wN \l__regex_peek_true_tl \exp:w ... \exp_end:`. This is set up such as to obtain `\l__regex_peek_true_tl` followed by the replaced tokens (possibly unbalanced) in the input stream.

```

7875 \cs_new_protected:Npn \__regex_peek_replace_end:
7876   {
7877     \bool_if:NTF \g__regex_success_bool
7878       {
7879         \__regex_extract:
7880         \__regex_query_set_from_input_tl:
7881         \cs_set_eq:NN \__regex_replacement_put:n \__regex_peek_replacement_put:n
7882         \cs_set_eq:NN \__regex_replacement_put_submatch_aux:n
7883         \__regex_peek_replacement_put_submatch_aux:n
7884         \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
7885         \cs_set_eq:NN \__regex_replacement_exp_not:N \__regex_peek_replacement_token:n
7886         \cs_set_eq:NN \__regex_replacement_exp_not:V \__regex_peek_replacement_var:N
7887         \exp_args:No \__regex_replacement:n { \l__regex_replacement_tl }
7888         \use:e
7889         {
7890           \exp_not:n { \exp_after:wN \l__regex_peek_true_tl \exp:w }
7891           \__regex_replacement_do_one_match:n \l__regex_zeroth_submatch_int
7892           \__regex_query_range:nn
7893           {
7894             \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
7895             \l__regex_zeroth_submatch_int
7896           }
7897           \l__regex_max_pos_int
7898           \exp_end:
7899         }
7900       }
7901     { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
7902   }

```

(End of definition for `__regex_peek_replace_end:`)

`__regex_query_set_from_input_tl:` The input was stored into `\l__regex_input_tl` as successive items `__regex_input_item:n` `{\tokens}`. Store that in successive `\toks`. It's not clear whether the empty entries before and after are both useful.

```

7903 \cs_new_protected:Npn \__regex_query_set_from_input_tl:
7904   {
7905     \tl_build_end:N \l__regex_input_tl
7906     \int_zero:N \l__regex_curr_pos_int
7907     \cs_set_eq:NN \__regex_input_item:n \__regex_query_set_item:n
7908     \__regex_query_set_item:n { }
7909     \l__regex_input_tl

```

```

7910     \_regex_query_set_item:n { }
7911     \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
7912   }
7913 \cs_new_protected:Npn \_regex_query_set_item:n #1
7914   {
7915     \int_incr:N \l__regex_curr_pos_int
7916     \_regex_toks_set:Nn \l__regex_curr_pos_int { \_regex_input_item:n {#1} }
7917   }

```

(End of definition for `_regex_query_set_from_input_tl:` and `_regex_query_set_item:n`.)

`_regex_peek_replacement_put:n`

While building the replacement function `_regex_replacement_do_one_match:n`, we often want to put simple material, given as `#1`, whose e-expansion o-expands to a single token. Normally we can just add the token to `\l__regex_build_tl`, but for `\peek_regex_replace_once:nnTF` we eventually want to do some strange expansion that is basically using `\exp_after:wN` to jump through numerous tokens (we cannot use e-expansion like for `\regex_replace_once:nnTF` because it is ok for the result to be unbalanced since we insert it in the input stream rather than storing it. When within a csname we don't do any such shenanigan because `\cs:w ... \cs_end:` does all the expansion we need.

```

7918 \cs_new_protected:Npn \_regex_peek_replacement_put:n #1
7919   {
7920     \if_case:w \l__regex_replacement_csnames_int
7921       \tl_build_put_right:Nn \l__regex_build_tl
7922         { \exp_not:N \_regex_reinsert_item:n {#1} }
7923     \else:
7924       \tl_build_put_right:Nn \l__regex_build_tl {#1}
7925     \fi:
7926   }

```

(End of definition for `_regex_peek_replacement_put:n`.)

`_regex_peek_replacement_token:n`

When hit with `\exp:w, _regex_peek_replacement_token:n {<token>}` stops `\exp_end:` and does `\exp_after:wN <token> \exp:w` to continue expansion after it.

```

7927 \cs_new_protected:Npn \_regex_peek_replacement_token:n #1
7928   { \exp_after:wN \exp_end: \exp_after:wN #1 \exp:w }

```

(End of definition for `_regex_peek_replacement_token:n`.)

`_regex_peek_replacement_put_submatch_aux:n`

While analyzing the replacement we also have to insert submatches found in the query. Since query items `_regex_input_item:n {<tokens>}` expand correctly only when surrounded by `\exp:w ... \exp_end:`, and since these expansion controls are not there within csnames (because `\cs:w ... \cs_end:` make them unnecessary in most cases), we have to put `\exp:w` and `\exp_end:` by hand here.

```

7929 \cs_new_protected:Npn \_regex_peek_replacement_put_submatch_aux:n #1
7930   {
7931     \if_case:w \l__regex_replacement_csnames_int
7932       \tl_build_put_right:Nn \l__regex_build_tl
7933         { \_regex_query_submatch:n { \_regex_int_eval:w #1 + ##1 \scan_stop: } }
7934     \else:
7935       \tl_build_put_right:Nn \l__regex_build_tl
7936         {
7937           \exp:w
7938           \_regex_query_submatch:n { \_regex_int_eval:w #1 + ##1 \scan_stop: }

```

```

7939         \exp_end:
7940     }
7941     \fi:
7942 }

```

(End of definition for `__regex_peek_replacement_put_submatch_aux:n`.)

`__regex_peek_replacement_var:N` This is used for `\u` outside csnames. It makes sure to continue expansion with `\exp:w` before expanding the variable #1 and stopping the `\exp:w` that precedes.

```

7943 \cs_new_protected:Npn \__regex_peek_replacement_var:N #1
7944 {
7945     \exp_after:wN \exp_last_unbraced:NV
7946     \exp_after:wN \exp_end:
7947     \exp_after:wN #1
7948     \exp:w
7949 }

```

(End of definition for `__regex_peek_replacement_var:N`.)

46.8 Messages

Messages for the preparising phase.

```

7950 \use:e
7951 {
7952     \msg_new:nnn { regex } { trailing-backslash }
7953     { Trailing~'\iow_char:N\}'~in~regex~or~replacement. }
7954     \msg_new:nnn { regex } { x-missing-rbrace }
7955     {
7956         Missing~brace~'\iow_char:N\}'~in~regex~
7957         '...\iow_char:N\{#\iow_char:N\{...\##1'.
7958     }
7959     \msg_new:nnn { regex } { x-overflow }
7960     {
7961         Character~code~##1~too~large~in~
7962         \iow_char:N\{#\iow_char:N\{##2\iow_char:N\}~regex.
7963     }
7964 }

```

Invalid quantifier.

```

7965 \msg_new:nnnn { regex } { invalid-quantifier }
7966 { Braced~quantifier~'#1'~may~not~be~followed~by~'#2'. }
7967 {
7968     The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'.~
7969     The~only~valid~quantifiers~are~'*',~'?',~'+',~'{<int>}',~
7970     '{<min>}'~and~'{<min>,<max>}',~optionally~followed~by~'?''.
7971 }

```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```

7972 \msg_new:nnnn { regex } { missing-rbrack }
7973 { Missing~right~bracket~inserted~in~regular~expression. }
7974 {
7975     LaTeX~was~given~a~regular~expression~where~a~character~class~
7976     was~started~with~'[',~but~the~matching~']'~is~missing.

```



```

7977 }
7978 \msg_new:nnnn { regex } { missing-rparen }
7979 {
7980   Missing-right~
7981   \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
7982   inserted-in-regular-expression.
7983 }
7984 {
7985   LaTeX-was-given-a-regular-expression-with-\int_eval:n {#1} ~
7986   more-left-parentheses-than-right-parentheses.
7987 }
7988 \msg_new:nnnn { regex } { extra-rparen }
7989 { Extra-right-parenthesis-ignored-in-regular-expression. }
7990 {
7991   LaTeX-came-across-a-closing-parenthesis-when-no-submatch-group~
7992   was-open.~The-parenthesis-will-be-ignored.
7993 }

```

Some escaped alphanumerics are not allowed everywhere.

```

7994 \msg_new:nnnn { regex } { bad-escape }
7995 {
7996   Invalid-escape~'\iow_char:N\|#1'~
7997   \__regex_if_in_cs:TF { within-a-control-sequence. }
7998   {
7999     \__regex_if_in_class:TF
8000     { in-a-character-class. }
8001     { following-a-category-test. }
8002   }
8003 }
8004 {
8005   The-escape-sequence~'\iow_char:N\|#1'~may-not-appear~
8006   \__regex_if_in_cs:TF
8007   {
8008     within-a-control-sequence-test-introduced-by~
8009     '\iow_char:N\|c\iow_char:N\{' .
8010   }
8011   {
8012     \__regex_if_in_class:TF
8013     { within-a-character-class~
8014     { following-a-category-test-such-as~'\iow_char:N\|cL'~ }
8015     because-it-does-not-match-exactly-one-character.
8016   }
8017 }

```

Range errors.

```

8018 \msg_new:nnnn { regex } { range-missing-end }
8019 { Invalid-end-point-for-range~'#1-#2'~in-character-class. }
8020 {
8021   The-end-point~'#2'~of~the~range~'#1-#2'~may~not~serve~as~an~
8022   end-point~for~a~range:~alphanumeric~characters~should~not~be~
8023   escaped,~and~non-alphanumeric~characters~should~be~escaped.
8024 }
8025 \msg_new:nnnn { regex } { range-backwards }
8026 { Range~'#1-#2'~out-of-order~in-character-class. }
8027 {

```

8028 In~ranges~of~characters~'[x-y]~'~appearing~in~character~classes,~
 8029 the~first~character~code~must~not~be~larger~than~the~second.~
 8030 Here,~'#1'~has~character~code~\int_eval:n~{'#1},~while~
 8031 '#2'~has~character~code~\int_eval:n~{'#2}.
 8032 }

Errors related to \c and \u.

8033 \msg_new:nnnn { regex } { c-bad-mode }
 8034 { Invalid~nested~'\iow_char:N\\c'~escape~in~regular~expression. }
 8035 {
 8036 The~'\iow_char:N\\c'~escape~cannot~be~used~within~
 8037 a~control~sequence~test~'\iow_char:N\\c{...}'~
 8038 nor~another~category~test.~
 8039 To~combine~several~category~tests,~use~'\iow_char:N\\c[...]' .
 8040 }
 8041 \msg_new:nnnn { regex } { c-C-invalid }
 8042 { '\iow_char:N\\cC'~should~be~followed~by~'.'~or~'(' ,~not~'#1'. }
 8043 {
 8044 The~'\iow_char:N\\cC'~construction~restricts~the~next~item~to~be~a~
 8045 control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
 8046 It~only~makes~sense~to~follow~it~by~'.'~or~by~a~group.
 8047 }
 8048 \msg_new:nnnn { regex } { cu-lbrace }
 8049 { Left~braces~must~be~escaped~in~'\iow_char:N\\#1{...}' . }
 8050 {
 8051 Constructions~such~as~'\iow_char:N\\#1{... \iow_char:N\\{...}'~are~
 8052 not~allowed~and~should~be~replaced~by~
 8053 '\iow_char:N\\#1{... \token_to_str:N\\{...}' .
 8054 }
 8055 \msg_new:nnnn { regex } { c-lparen-in-class }
 8056 { Catcode~test~cannot~apply~to~group~in~character~class }
 8057 {
 8058 Construction~such~as~'\iow_char:N\\cL(abc)'~are~not~allowed~inside~a~
 8059 class~'[...]'~because~classes~do~not~match~multiple~characters~at~once.
 8060 }
 8061 \msg_new:nnnn { regex } { c-missing-rbrace }
 8062 { Missing~right~brace~inserted~for~'\iow_char:N\\c'~escape. }
 8063 {
 8064 LaTeX~was~given~a~regular~expression~where~a~
 8065 '\iow_char:N\\c\iow_char:N\\{...}'~construction~was~not~ended~
 8066 with~a~closing~brace~'\iow_char:N\\}' .
 8067 }
 8068 \msg_new:nnnn { regex } { c-missing-rbrack }
 8069 { Missing~right~bracket~inserted~for~'\iow_char:N\\c'~escape. }
 8070 {
 8071 A~construction~'\iow_char:N\\c[...]'~appears~in~a~
 8072 regular~expression,~but~the~closing~'}'~is~not~present.
 8073 }
 8074 \msg_new:nnnn { regex } { c-missing-category }
 8075 { Invalid~character~'#1'~following~'\iow_char:N\\c'~escape. }
 8076 {
 8077 In~regular~expressions,~the~'\iow_char:N\\c'~escape~sequence~
 8078 may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
 8079 capital~letter~representing~a~character~category,~namely~
 8080 one~of~'ABCDELMOPTU' .

```

8081 }
8082 \msg_new:nnnn { regex } { c-trailing }
8083 { Trailing-category-code-escape~'\iow_char:N\c'... }
8084 {
8085   A~regular~expression~ends~with~'\iow_char:N\c'~followed~
8086   by~a~letter.~It~will~be~ignored.
8087 }
8088 \msg_new:nnnn { regex } { u-missing-lbrace }
8089 { Missing~left~brace~following~'\iow_char:N\{u'~escape. }
8090 {
8091   The~'\iow_char:N\{u'~escape~sequence~must~be~followed~by~
8092   a~brace~group~with~the~name~of~the~variable~to~use.
8093 }
8094 \msg_new:nnnn { regex } { u-missing-rbrace }
8095 { Missing~right~brace~inserted~for~'\iow_char:N\}u'~escape. }
8096 {
8097   LaTeX~
8098   \str_if_eq:eeTF { } {#2}
8099   { reached~the~end~of~the~string~ }
8100   { encountered~an~escaped~alphanumeric~character~'\iow_char:N\{#2'~ }
8101   when~parsing~the~argument~of~an~
8102   '\iow_char:N\{u\iow_char:N\{...\}'~escape.
8103 }

```

Errors when encountering the POSIX syntax [:...:].

```

8104 \msg_new:nnnn { regex } { posix-unsupported }
8105 { POSIX~collating~element~'#1 ~ #1'~not~supported. }
8106 {
8107   The~'.foo.'~and~'[=bar=]'~syntaxes~have~a~special~meaning~
8108   in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
8109   Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
8110 }
8111 \msg_new:nnnn { regex } { posix-unknown }
8112 { POSIX~class~'[:#1:]'~unknown. }
8113 {
8114   '[:#1:]'~is~not~among~the~known~POSIX~classes~
8115   '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
8116   '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
8117   '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
8118   '[:word:]',~and~'[:xdigit:]'.
8119 }
8120 \msg_new:nnnn { regex } { posix-missing-close }
8121 { Missing~closing~':'~for~POSIX~class. }
8122 { The~POSIX~syntax~'#1'~must~be~followed~by~':'',~not~'#2'. }

```

In various cases, the result of a `l3regex` operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```

8123 \msg_new:nnnn { regex } { result-unbalanced }
8124 { Missing~brace~inserted~when~#1. }
8125 {
8126   LaTeX~was~asked~to~do~some~regular~expression~operation,~
8127   and~the~resulting~token~list~would~not~have~the~same~number~
8128   of~begin~group~and~end~group~tokens.~Braces~were~inserted:~
8129   #2~left,~#3~right.

```

```

8130 }
      Error message for unknown options.
8131 \msg_new:nnnn { regex } { unknown-option }
8132 { Unknown-option-#1'-for-regular-expressions. }
8133 {
8134   The-only-available-option-is-'case-insensitive',~toggled-by-
8135   '(?i)''~and~'(?-i)'.
8136 }
8137 \msg_new:nnnn { regex } { special-group-unknown }
8138 { Unknown-special-group-#1~...''in-a-regular-expression. }
8139 {
8140   The-only-valid-constructions-starting-with-'(?~are-
8141   '(?:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
8142 }
      Errors in the replacement text.
8143 \msg_new:nnnn { regex } { replacement-c }
8144 { Misused-'\iow_char:N\c'~command-in-a-replacement-text. }
8145 {
8146   In-a-replacement-text,~the-'\iow_char:N\c'~escape-sequence-
8147   can-be-followed-by-one-of-the-letters-'ABCDELMPSTU'~
8148   or-a-brace-group,~not-by-#1'.
8149 }
8150 \msg_new:nnnn { regex } { replacement-u }
8151 { Misused-'\iow_char:N\u'~command-in-a-replacement-text. }
8152 {
8153   In-a-replacement-text,~the-'\iow_char:N\u'~escape-sequence-
8154   must-be~followed-by-a-brace-group-holding-the-name-of-the-
8155   variable-to-use.
8156 }
8157 \msg_new:nnnn { regex } { replacement-g }
8158 {
8159   Missing-brace-for-the-'\iow_char:N\g'~construction-
8160   in-a-replacement-text.
8161 }
8162 {
8163   In-the-replacement-text-for-a-regular-expression-search,~
8164   submatches-are-represented-either-as-'\iow_char:N \g{dd..d}',~
8165   or-'\d',~where-'d'~are-single-digits.~Here,~a-brace-is-missing.
8166 }
8167 \msg_new:nnnn { regex } { replacement-catcode-end }
8168 {
8169   Missing-character-for-the-'\iow_char:N\c<category><character>'~
8170   construction-in-a-replacement-text.
8171 }
8172 {
8173   In-a-replacement-text,~the-'\iow_char:N\c'~escape-sequence-
8174   can-be-followed-by-one-of-the-letters-'ABCDELMPSTU'~representing-
8175   the-character-category.~Then,~a-character-must-follow.~LaTeX-
8176   reached-the-end-of-the-replacement-when-looking-for-that.
8177 }
8178 \msg_new:nnnn { regex } { replacement-catcode-escaped }
8179 {
8180   Escaped-letter-or-digit-after-category-code-in-replacement-text.

```

```

8181 }
8182 {
8183   In-a-replacement-text,~the~'\iow_char:N\c'~escape-sequence~
8184   can-be-followed-by-one-of-the-letters~'ABCDELMOPSTU'~representing~
8185   the-character-category.~Then,~a-character-must-follow,~not~
8186   '\iow_char:N\#2'.
8187 }
8188 \msg_new:nnnn { regex } { replacement-catcode-in-cs }
8189 {
8190   Category-code~'\iow_char:N\c#1#3'~ignored-inside~
8191   '\iow_char:N\c\{...\}'~in-a-replacement-text.
8192 }
8193 {
8194   In-a-replacement-text,~the~category~codes~of~the~argument~of~
8195   '\iow_char:N\c\{...\}'~are-ignored-when~building~the~control~
8196   sequence-name.
8197 }
8198 \msg_new:nnnn { regex } { replacement-null-space }
8199 { TeX-cannot-build-a-space-token-with-character-code-0. }
8200 {
8201   You-asked-for-a-character-token-with-category-space,~
8202   and-character-code-0,~for-instance-through~
8203   '\iow_char:N\cS\iow_char:N\#x00'.~
8204   This-specific-case-is-impossible-and-will-be-replaced~
8205   by-a-normal-space.
8206 }
8207 \msg_new:nnnn { regex } { replacement-missing-rbrace }
8208 { Missing-right-brace-inserted-in-replacement-text. }
8209 {
8210   There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
8211   missing-right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
8212 }
8213 \msg_new:nnnn { regex } { replacement-missing-rparen }
8214 { Missing-right-parenthesis-inserted-in-replacement-text. }
8215 {
8216   There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
8217   missing-right~
8218   \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
8219 }
8220 \msg_new:nnn { regex } { submatch-too-big }
8221 { Submatch-#1-used-but-regex-only-has-#2-group(s) }
      Some escaped alphanumerics are not allowed everywhere.
8222 \msg_new:nnnn { regex } { backwards-quantifier }
8223 { Quantifer~"#{1,#2}"~is~backwards. }
8224 { The-values-given-in-a-quantifier-must-be-in-order. }
      Used in user commands, and when showing a regex.
8225 \msg_new:nnnn { regex } { case-odd }
8226 { #1-with-odd-number-of-items }
8227 {
8228   There-must-be-a-#2-part-for-each-regex:~
8229   found-odd-number-of-items~(#{3})~in\\
8230   \iow_indent:n {#4}
8231 }

```

```

8232 \msg_new:nnn { regex } { show }
8233 {
8234   >~Compiled~regex~
8235   \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
8236   #3
8237 }
8238 \prop_gput:Nnn \g_msg_module_name_prop { regex } { LaTeX }
8239 \prop_gput:Nnn \g_msg_module_type_prop { regex } { }

```

`__regex_msg_repeated:nnN` This is not technically a message, but seems related enough to go there. The arguments are: `#1` is the minimum number of repetitions; `#2` is the number of allowed extra repetitions (`-1` for infinite number), and `#3` tells us about laziness.

```

8240 \cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
8241 {
8242   \str_if_eq:eeF { #1 #2 } { 1 0 }
8243   {
8244     , ~ repeated ~
8245     \int_case:nnF {#2}
8246     {
8247       { -1 } { #1~or-more-times,~\bool_if:NTF #3 { lazy } { greedy } }
8248       { 0 } { #1~times }
8249     }
8250     {
8251       between~#1~and~\int_eval:n {#1+#2}~times,~
8252       \bool_if:NTF #3 { lazy } { greedy }
8253     }
8254   }
8255 }

```

(End of definition for `__regex_msg_repeated:nnN`.)

46.9 Code for tracing

There is a more extensive implementation of tracing in the `l3trial` package `l3trace`. Function names are a bit different but could be merged.

`__regex_trace_push:nnN` Here `#1` is the module name (`regex`) and `#2` is typically 1. If the module's current tracing level is less than `#2` show nothing, otherwise write `#3` to the terminal.

```

\__regex_trace_pop:nnN
\__regex_trace:nne
8256 \cs_new_protected:Npn \__regex_trace_push:nnN #1#2#3
8257 { \__regex_trace:nne {#1} {#2} { entering~ \token_to_str:N #3 } }
8258 \cs_new_protected:Npn \__regex_trace_pop:nnN #1#2#3
8259 { \__regex_trace:nne {#1} {#2} { leaving~ \token_to_str:N #3 } }
8260 \cs_new_protected:Npn \__regex_trace:nne #1#2#3
8261 {
8262   \int_compare:nNnF
8263   { \int_use:c { g__regex_trace_#1_int } } < {#2}
8264   { \iow_term:e { Trace:~#3 } }
8265 }

```

(End of definition for `__regex_trace_push:nnN`, `__regex_trace_pop:nnN`, and `__regex_trace:nne`.)

`\g__regex_trace_regex_int` No tracing when that is zero.

```

8266 \int_new:N \g__regex_trace_regex_int

```

(End of definition for \g__regex_trace_regex_int.)

`__regex_trace_states:n` This function lists the contents of all states of the NFA, stored in `\toks` from 0 to `\l__-regex_max_state_int` (excluded).

```
8267 \cs_new_protected:Npn \__regex_trace_states:n #1
8268   {
8269     \int_step_inline:nnn
8270       \l__regex_min_state_int
8271       { \l__regex_max_state_int - \c_one_int }
8272     {
8273       \__regex_trace:nne { regex } {#1}
8274       { \iow_char:N \toks ##1 = { \__regex_toks_use:w ##1 } }
8275     }
8276   }
```

(End of definition for __regex_trace_states:n.)

```
8277 \endpackage
```

Chapter 47

l3prg implementation

The following test files are used for this code: `m3prg001.lvt,m3prg002.lvt,m3prg003.lvt`.

```
8278 (*package)
```

47.1 Primitive conditionals

`\if_bool:N` Those two primitive TeX conditionals are synonyms. `\if_bool:N` is defined in `l3basics`, as it's needed earlier to define quark test functions.

```
8279 \cs_new_eq:NN \if_predicate:w \tex_ifodd:D
```

(End of definition for `\if_bool:N` and `\if_predicate:w`. These functions are documented on page 72.)

47.2 Defining a set of conditional functions

These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

(End of definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 63.)

47.3 The boolean data type

```
8280 (@@=bool)
```

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
8281 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
8282 \cs_generate_variant:Nn \bool_new:N { c }
```

(End of definition for `\bool_new:N`. This function is documented on page 66.)

`\bool_const:Nn` A merger between `\tl_const:Nn` and `\bool_set:Nn`.

```
\bool_const:cn
8283 \cs_new_protected:Npn \bool_const:Nn #1#2
8284 {
8285   \__kernel_chk_if_free_cs:N #1
8286   \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
8287 }
8288 \cs_generate_variant:Nn \bool_const:Nn { c }
```


(End of definition for `\bool_const:Nn`. This function is documented on page 66.)

```

\bool_set_true:N Setting is already pretty easy. When check-declarations is active, the definitions are
\bool_set_true:c patched to make sure the boolean exists. This is needed because booleans are not based
\bool_gset_true:N on token lists nor on TEX registers.
\bool_gset_true:c
\bool_set_false:N 8289 \cs_new_protected:Npn \bool_set_true:N #1
\bool_set_false:c 8290 { \cs_set_eq:NN #1 \c_true_bool }
\bool_gset_false:N 8291 \cs_new_protected:Npn \bool_set_false:N #1
\bool_gset_false:c 8292 { \cs_set_eq:NN #1 \c_false_bool }
8293 \cs_new_protected:Npn \bool_gset_true:N #1
8294 { \cs_gset_eq:NN #1 \c_true_bool }
8295 \cs_new_protected:Npn \bool_gset_false:N #1
8296 { \cs_gset_eq:NN #1 \c_false_bool }
8297 \cs_generate_variant:Nn \bool_set_true:N { c }
8298 \cs_generate_variant:Nn \bool_set_false:N { c }
8299 \cs_generate_variant:Nn \bool_gset_true:N { c }
8300 \cs_generate_variant:Nn \bool_gset_false:N { c }

```

(End of definition for `\bool_set_true:N` and others. These functions are documented on page 66.)

```

\bool_set_eq:NN The usual copy code. While it would be cleaner semantically to copy the \cs_set_eq:NN
\bool_set_eq:cN family of functions, we copy \tl_set_eq:NN because that has the correct checking code.
\bool_set_eq:Nc 8301 \cs_new_eq:NN \bool_set_eq:NN \tl_set_eq:NN
\bool_set_eq:cc 8302 \cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN
\bool_gset_eq:NN 8303 \cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }
\bool_gset_eq:cN 8304 \cs_generate_variant:Nn \bool_gset_eq:NN { Nc, cN, cc }
\bool_gset_eq:Nc
\bool_gset_eq:cc

```

(End of definition for `\bool_set_eq:NN` and `\bool_gset_eq:NN`. These functions are documented on page 66.)

```

\bool_set:Nn This function evaluates a boolean expression and assigns the first argument the meaning
\bool_set:cn \c_true_bool or \c_false_bool. Again, we include some checking code. It is important
\bool_gset:Nn to evaluate the expression before applying the \chardef primitive, because that primitive
\bool_gset:cn sets the left-hand side to \scan_stop: before looking for the right-hand side.

```

```

8305 \cs_new_protected:Npn \bool_set:Nn #1#2
8306 {
8307   \exp_last_unbraced:NNNf
8308   \tex_chardef:D #1 = { \bool_if_p:n {#2} }
8309 }
8310 \cs_new_protected:Npn \bool_gset:Nn #1#2
8311 {
8312   \exp_last_unbraced:NNNNf
8313   \tex_global:D \tex_chardef:D #1 = { \bool_if_p:n {#2} }
8314 }
8315 \cs_generate_variant:Nn \bool_set:Nn { c }
8316 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

(End of definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 66.)

```

\bool_set_inverse:N Set to false or true locally or globally.
\bool_set_inverse:c 8317 \cs_new_protected:Npn \bool_set_inverse:N #1
\bool_gset_inverse:N 8318 { \bool_if:NTF #1 { \bool_set_false:N } { \bool_set_true:N } #1 }
\bool_gset_inverse:c 8319 \cs_generate_variant:Nn \bool_set_inverse:N { c }
8320 \cs_new_protected:Npn \bool_gset_inverse:N #1
8321 { \bool_if:NTF #1 { \bool_gset_false:N } { \bool_gset_true:N } #1 }
8322 \cs_generate_variant:Nn \bool_gset_inverse:N { c }

```

(End of definition for `\bool_set_inverse:N` and `\bool_gset_inverse:N`. These functions are documented on page 66.)

47.4 Internal auxiliaries

`\q__bool_recursion_tail` Internal recursion quarks.

```

\q__bool_recursion_stop 8323 \quark_new:N \q__bool_recursion_tail
                        8324 \quark_new:N \q__bool_recursion_stop

```

(End of definition for `\q__bool_recursion_tail` and `\q__bool_recursion_stop`.)

`__bool_use_i_delimit_by_q_recursion_stop:nw` Functions to gobble up to a quark.

```

8325 \cs_new:Npn \__bool_use_i_delimit_by_q_recursion_stop:nw
8326   #1 #2 \q__bool_recursion_stop {#1}

```

(End of definition for `__bool_use_i_delimit_by_q_recursion_stop:nw`.)

`__bool_if_recursion_tail_stop_do:nn` Functions to query recursion quarks.

```

8327 \__kernel_quark_new_test:N \__bool_if_recursion_tail_stop_do:nn

```

(End of definition for `__bool_if_recursion_tail_stop_do:nn`.)

`\bool_if_p:N` Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```

\bool_if_p:c
\bool_if:NTF 8328 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
\bool_if:cTF 8329   {
8330     \if_bool:N #1
8331     \prg_return_true:
8332     \else:
8333     \prg_return_false:
8334     \fi:
8335   }
8336 \prg_generate_conditional_variant:Nnn \bool_if:N { c } { p , T , F , TF }

```

(End of definition for `\bool_if:N`. This function is documented on page 67.)

`\bool_to_str:N` Expands to string literal true or false.

```

\bool_to_str:c 8337 \cs_new:Npe \bool_to_str:N #1
\bool_to_str:n 8338   {
8339     \exp_not:N \bool_if:N {TF} #1
8340     { \tl_to_str:n { true } } { \tl_to_str:n { false } }
8341   }
8342 \cs_generate_variant:Nn \bool_to_str:N { c }
8343 \cs_new:Npe \bool_to_str:n #1
8344   {
8345     \exp_not:N \bool_if:n {TF} {#1}
8346     { \tl_to_str:n { true } } { \tl_to_str:n { false } }
8347   }

```

(End of definition for `\bool_to_str:N` and `\bool_to_str:n`. These functions are documented on page 67.)

\bool_show:n Show the truth value of the boolean.

```
\bool_log:n      8348 \cs_new_protected:Npn \bool_show:n
                  8349   { \__kernel_msg_show_eval:Nn \bool_to_str:n }
                  8350 \cs_new_protected:Npn \bool_log:n
                  8351   { \__kernel_msg_log_eval:Nn \bool_to_str:n }
```

(End of definition for \bool_show:n and \bool_log:n. These functions are documented on page 67.)

\bool_show:N Show the truth value of the boolean, as true or false.

```
\bool_show:c      8352 \cs_new_protected:Npn \bool_show:N { \__bool_show:NN \tl_show:n }
\bool_log:N      8353 \cs_generate_variant:Nn \bool_show:N { c }
\bool_log:c      8354 \cs_new_protected:Npn \bool_log:N { \__bool_show:NN \tl_log:n }
\__bool_show:NN  8355 \cs_generate_variant:Nn \bool_log:N { c }
                  8356 \cs_new_protected:Npn \__bool_show:NN #1#2
                  8357   {
                  8358     \__kernel_chk_defined:NT #2
                  8359     {
                  8360       \token_case_meaning:NnF #2
                  8361       {
                  8362         \c_true_bool { \exp_args:Ne #1 { \token_to_str:N #2 = true } }
                  8363         \c_false_bool { \exp_args:Ne #1 { \token_to_str:N #2 = false } }
                  8364       }
                  8365       {
                  8366         \msg_error:nneee { kernel } { bad-type }
                  8367         { \token_to_str:N #2 } { \token_to_meaning:N #2 } { bool }
                  8368       }
                  8369     }
                  8370   }
```

(End of definition for \bool_show:N, \bool_log:N, and __bool_show:NN. These functions are documented on page 67.)

\l_tmpa_bool A few booleans just if you need them.

```
\l_tmpb_bool      8371 \bool_new:N \l_tmpa_bool
\g_tmpa_bool      8372 \bool_new:N \l_tmpb_bool
\g_tmpb_bool      8373 \bool_new:N \g_tmpa_bool
                  8374 \bool_new:N \g_tmpb_bool
```

(End of definition for \l_tmpa_bool and others. These variables are documented on page 67.)

\bool_if_exist_p:N Copies of the cs functions defined in l3basics.

```
\bool_if_exist_p:c 8375 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
\bool_if_exist:NTF 8376   { TF , T , F , p }
\bool_if_exist:cTF 8377 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
                  8378   { TF , T , F , p }
```

(End of definition for \bool_if_exist:NTF. This function is documented on page 67.)

47.5 Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with (and) for grouping, ! for logical “Not”, && for logical “And” and || for logical “Or”. However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

`\bool_if:nTF`

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a GetNext function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, remove the ! and call a GetNext function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an Eval function, which evaluates it to the boolean value `<true>` or `<false>`.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

`<true>`**And** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

`<false>`**And** Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return `<false>`.

`<true>`**Or** Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return `<true>`.

`<false>`**Or** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

`<true>`**Close** Current truth value is true, Close seen, return `<true>`.

`<false>`**Close** Current truth value is false, Close seen, return `<false>`.

```

8379 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
8380   {
8381     \if_predicate:w \bool_if_p:n {#1}
8382       \prg_return_true:
8383     \else:
8384       \prg_return_false:
8385     \fi:
8386   }

```

(End of definition for \bool_if:nTF. This function is documented on page 69.)

`\bool_if_p:n`
`__bool_if_p:n`
`__bool_if_p_aux:w`

To speed up the case of a single predicate, f-expand and check whether the result is one token (possibly surrounded by spaces), which must be `\c_true_bool` or `\c_false_bool`. We use a version of `\tl_if_single:nTF` optimized for speed since we know that an empty #1 is an error. The auxiliary `__bool_if_p_aux:w` removes the trailing parenthesis and gets rid of any space, then returns `\c_true_bool` or `\c_false_bool` as appropriate. This extra work around is because in a `\bool_set:Nn`, the underlying `\chardef` turns

the `bool` being set temporarily equal to `\relax`, thus assigning a boolean to itself would fail (gh/1055). For the general case, first issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for T_EX. This group is closed after `_bool_get_next:NN` returns `\c_true_bool` or `\c_false_bool`. That function requires the trailing parenthesis to know where the expression ends.

```

8387 \cs_new:Npn \bool_if_p:n { \exp_args:Nf \_bool_if_p:n }
8388 \cs_new:Npn \_bool_if_p:n #1
8389   {
8390     \tl_if_empty:oT { \use_none:nn #1 . } { \_bool_if_p_aux:w }
8391     \group_align_safe_begin:
8392     \exp_after:wN
8393     \group_align_safe_end:
8394     \exp:w \exp_end_continue_f:w % (
8395     \_bool_get_next:NN \use_i:nnnn #1 )
8396   }
8397 \cs_new:Npn \_bool_if_p_aux:w #1 \use_i:nnnn #2#3
8398   { \bool_if:NTF #2 \c_true_bool \c_false_bool }

```

(End of definition for `\bool_if_p:n`, `_bool_if_p:n`, and `_bool_if_p_aux:w`. This function is documented on page 69.)

`_bool_get_next:NN` The GetNext operation. Its first argument is `\use_i:nnnn`, `\use_ii:nnnn`, `\use_iii:nnnn`, or `\use_iv:nnnn` (we call these “states”). In the first state, this function eventually expand to the truth value `\c_true_bool` or `\c_false_bool` of the expression which follows until the next unmatched closing parenthesis. For instance “`_bool_get_next:NN \use_i:nnnn \c_true_bool && \c_true_bool)`” (including the closing parenthesis) expands to `\c_true_bool`. In the second state (after a `!`) the logic is reversed. We call these two states “normal” and the next two “skipping”. In the third state (after `\c_true_bool||`) it always returns `\c_true_bool`. In the fourth state (after `\c_false_bool&&`) it always returns `\c_false_bool` and also stops when encountering `||`, not only parentheses. This code itself is a switch: if what follows is neither `!` nor `(`, we assume it is a predicate.

```

8399 \cs_new:Npn \_bool_get_next:NN #1#2
8400   {
8401     \use:c
8402     {
8403       \_bool_
8404       \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
8405       :Nw
8406     }
8407     #1 #2
8408   }

```

(End of definition for `_bool_get_next:NN`.)

`_bool_!:Nw` The Not operation reverses the logic: it discards the `!` token and calls the GetNext operation with the appropriate first argument. Namely the first and second states are interchanged, but after `\c_true_bool||` or `\c_false_bool&&` the `!` is ignored.

```

8409 \cs_new:cpn { \_bool_!:Nw } #1#2
8410   {
8411     \exp_after:wN \_bool_get_next:NN
8412     #1 \use_ii:nnnn \use_i:nnnn \use_iii:nnnn \use_iv:nnnn
8413   }

```

(End of definition for `__bool_!:Nw`.)

`__bool_(:Nw` The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling `GetNext` (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for `And`, `Or` or `Close` after the group.

```
8414 \cs_new:cpn { __bool_(:Nw } #1#2
8415   {
8416     \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
8417     \int_value:w \__bool_get_next:NN \use_i:nnnn
8418   }
```

(End of definition for `__bool_(:Nw`.)

`__bool_p:Nw` If what follows `GetNext` is neither `!` nor `(`, evaluate the predicate using the primitive `\int_value:w`. The canonical `true` and `false` values have numerical values 1 and 0 respectively. Look for `And`, `Or` or `Close` afterwards.

```
8419 \cs_new:cpn { __bool_p:Nw } #1
8420   { \exp_after:wN \__bool_choose:NNN \exp_after:wN #1 \int_value:w }
```

(End of definition for `__bool_p:Nw`.)

`__bool_choose:NNN` The arguments are `#1`: a function such as `\use_i:nnnn`, `#2`: 0 or 1 encoding the current truth value, `#3`: the next operation, `And`, `Or` or `Close`. We distinguish three cases according to a combination of `#1` and `#2`. Case 2 is when `#1` is `\use_iii:nnnn` (state 3), namely after `\c_true_bool ||`. Case 1 is when `#1` is `\use_i:nnnn` and `#2` is `true` or `__bool_&_0`: when `#1` is `\use_ii:nnnn` and `#2` is `false`, for instance for `!\c_false_bool`. Case 0 includes the same with `true/false` interchanged and the case where `#1` is `\use_iv:nnnn` namely after `\c_false_bool &&`.

`__bool_|_0`: When seeing `)` the current subexpression is done, leave the appropriate boolean.
`__bool_|_1`: When seeing `&` in case 0 go into state 4, equivalent to having seen `\c_false_bool &&`.
`__bool_|_2`: In case 1, namely when the argument is `true` and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an `Or`, continue in the same state. When seeing `|` in case 0, continue in a normal state; in particular stop skipping for `\c_false_bool &&` because that binds more tightly than `||`. In the other two cases start skipping for `\c_true_bool ||`.

```
8421 \cs_new:Npn \__bool_choose:NNN #1#2#3
8422   {
8423     \use:c
8424     {
8425       __bool_ \token_to_str:N #3 _
8426       #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
8427     }
8428   }
8429 \cs_new:cpn { __bool_)_0: } { \c_false_bool }
8430 \cs_new:cpn { __bool_)_1: } { \c_true_bool }
8431 \cs_new:cpn { __bool_)_2: } { \c_true_bool }
8432 \cs_new:cpn { __bool_&_0: } & { \__bool_get_next:NN \use_iv:nnnn }
8433 \cs_new:cpn { __bool_&_1: } & { \__bool_get_next:NN \use_i:nnnn }
8434 \cs_new:cpn { __bool_&_2: } & { \__bool_get_next:NN \use_iii:nnnn }
8435 \cs_new:cpn { __bool_|_0: } | { \__bool_get_next:NN \use_i:nnnn }
8436 \cs_new:cpn { __bool_|_1: } | { \__bool_get_next:NN \use_iii:nnnn }
8437 \cs_new:cpn { __bool_|_2: } | { \__bool_get_next:NN \use_iii:nnnn }
```

(End of definition for `_bool_choose:NNN` and others.)

`\bool_lazy_all_p:n` Go through the list of expressions, stopping whenever an expression is false. If the end is reached without finding any false expression, then the result is true.

```
\bool_lazy_all:nTF  
\_bool_lazy_all:n 8438 \cs_new:Npn \bool_lazy_all_p:n #1  
8439 { \_bool_lazy_all:n #1 \q__bool_recursion_tail \q__bool_recursion_stop }  
8440 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { T , F , TF }  
8441 {  
8442   \if_predicate:w \bool_lazy_all_p:n {#1}  
8443   \prg_return_true:  
8444   \else:  
8445   \prg_return_false:  
8446   \fi:  
8447 }  
8448 \cs_new:Npn \_bool_lazy_all:n #1  
8449 {  
8450   \_bool_if_recursion_tail_stop_do:nn {#1} { \c_true_bool }  
8451   \bool_if:nF {#1}  
8452   { \_bool_use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } }  
8453   \_bool_lazy_all:n  
8454 }
```

(End of definition for `\bool_lazy_all:nTF` and `_bool_lazy_all:n`. This function is documented on page 69.)

`\bool_lazy_and_p:nn` Only evaluate the second expression if the first is true. Note that #2 must be removed as an argument, not just by skipping to the `\else:` branch of the conditional since #2 may contain unbalanced TeX conditionals.

```
\bool_lazy_and:nnTF  
8455 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }  
8456 {  
8457   \if_predicate:w  
8458   \bool_if:nTF {#1} { \bool_if_p:n {#2} } { \c_false_bool }  
8459   \prg_return_true:  
8460   \else:  
8461   \prg_return_false:  
8462   \fi:  
8463 }
```

(End of definition for `\bool_lazy_and:nnTF`. This function is documented on page 69.)

`\bool_lazy_any_p:n` Go through the list of expressions, stopping whenever an expression is true. If the end is reached without finding any true expression, then the result is false.

```
\bool_lazy_any:nTF  
\_bool_lazy_any:n 8464 \cs_new:Npn \bool_lazy_any_p:n #1  
8465 { \_bool_lazy_any:n #1 \q__bool_recursion_tail \q__bool_recursion_stop }  
8466 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { T , F , TF }  
8467 {  
8468   \if_predicate:w \bool_lazy_any_p:n {#1}  
8469   \prg_return_true:  
8470   \else:  
8471   \prg_return_false:  
8472   \fi:  
8473 }  
8474 \cs_new:Npn \_bool_lazy_any:n #1  
8475 {
```

```

8476     \_bool_if_recursion_tail_stop_do:nn {#1} { \c_false_bool }
8477     \bool_if:nT {#1}
8478         { \_bool_use_i_delimit_by_q_recursion_stop:nw { \c_true_bool } }
8479     \_bool_lazy_any:n
8480 }

```

(End of definition for `\bool_lazy_any:nTF` and `_bool_lazy_any:n`. This function is documented on page 69.)

`\bool_lazy_or_p:nn` Only evaluate the second expression if the first is false.

```

\bool_lazy_or:nnTF
8481 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
8482 {
8483     \if_predicate:w
8484         \bool_if:nTF {#1} { \c_true_bool } { \bool_if_p:n {#2} }
8485     \prg_return_true:
8486     \else:
8487         \prg_return_false:
8488     \fi:
8489 }

```

(End of definition for `\bool_lazy_or:nnTF`. This function is documented on page 69.)

`\bool_not_p:n` The Not variant just reverses the outcome of `\bool_if_p:n`. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```

8490 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }

```

(End of definition for `\bool_not_p:n`. This function is documented on page 69.)

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return false, otherwise
`\bool_xor:nnTF` return true.

```

8491 \prg_new_conditional:Npnn \bool_xor:nn #1#2 { p , T , F , TF }
8492 {
8493     \bool_if:nT {#1} \reverse_if:N
8494     \if_predicate:w \bool_if_p:n {#2}
8495     \prg_return_true:
8496     \else:
8497         \prg_return_false:
8498     \fi:
8499 }

```

(End of definition for `\bool_xor:nnTF`. This function is documented on page 70.)

47.6 Logical loops

`\bool_while_do:Nn` A while loop where the boolean is tested before executing the statement. The “while”
`\bool_while_do:cn` version executes the code as long as the boolean is true; the “until” version executes the
`\bool_until_do:Nn` code as long as the boolean is false.
`\bool_until_do:cn`

```

8500 \cs_new:Npn \bool_while_do:Nn #1#2
8501     { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
8502 \cs_new:Npn \bool_until_do:Nn #1#2
8503     { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
8504 \cs_generate_variant:Nn \bool_while_do:Nn { c }
8505 \cs_generate_variant:Nn \bool_until_do:Nn { c }

```


(End of definition for `\bool_while_do:Nn` and `\bool_until_do:Nn`. These functions are documented on page 70.)

`\bool_do_while:Nn` A do-while loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

```

\bool_do_while:cn
\bool_do_until:Nn
\bool_do_until:cn
8506 \cs_new:Npn \bool_do_while:Nn #1#2
8507   { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
8508 \cs_new:Npn \bool_do_until:Nn #1#2
8509   { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
8510 \cs_generate_variant:Nn \bool_do_while:Nn { c }
8511 \cs_generate_variant:Nn \bool_do_until:Nn { c }

```

(End of definition for `\bool_do_while:Nn` and `\bool_do_until:Nn`. These functions are documented on page 70.)

`\bool_while_do:nn` Loop functions with the test either before or after the first body expansion.

```

\bool_do_while:nn
\bool_until_do:nn
\bool_do_until:nn
8512 \cs_new:Npn \bool_while_do:nn #1#2
8513   {
8514     \bool_if:nT {#1}
8515     {
8516       #2
8517       \bool_while_do:nn {#1} {#2}
8518     }
8519   }
8520 \cs_new:Npn \bool_do_while:nn #1#2
8521   {
8522     #2
8523     \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
8524   }
8525 \cs_new:Npn \bool_until_do:nn #1#2
8526   {
8527     \bool_if:nF {#1}
8528     {
8529       #2
8530       \bool_until_do:nn {#1} {#2}
8531     }
8532   }
8533 \cs_new:Npn \bool_do_until:nn #1#2
8534   {
8535     #2
8536     \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
8537   }

```

(End of definition for `\bool_while_do:nn` and others. These functions are documented on page 71.)

`\s__bool_mark` Internal scan marks.

```

\s__bool_stop
8538 \scan_new:N \s__bool_mark
8539 \scan_new:N \s__bool_stop

```

(End of definition for `\s__bool_mark` and `\s__bool_stop`.)

`\bool_case:n` For boolean cases the overall idea is the same as for `\str_case:nnTF` as described in l3str.

```

\bool_case:nTF
__bool_case:NnTF
__bool_case:w
a\__bool_case_end:nw
8540 \cs_new:Npn \bool_case:nTF
8541   { \exp:w __bool_case:nTF }

```

```

8542 \cs_new:Npn \bool_case:nT #1#2
8543   { \exp:w \__bool_case:nTF {#1} {#2} { } }
8544 \cs_new:Npn \bool_case:nF #1
8545   { \exp:w \__bool_case:nTF {#1} { } }
8546 \cs_new:Npn \bool_case:n #1
8547   { \exp:w \__bool_case:nTF {#1} { } { } }
8548 \cs_new:Npn \__bool_case:nTF #1#2#3
8549   {
8550     \__bool_case:w
8551     #1 \c_true_bool { } \s__bool_mark {#2} \s__bool_mark {#3} \s__bool_stop
8552   }
8553 \cs_new:Npn \__bool_case:w #1#2
8554   {
8555     \bool_if:nTF {#1}
8556       { \__bool_case_end:nw {#2} }
8557       { \__bool_case:w }
8558   }
8559 \cs_new:Npn \__bool_case_end:nw #1#2#3 \s__bool_mark #4#5 \s__bool_stop
8560   { \exp_end: #1 #4 }

```

(End of definition for `\bool_case:nTF` and others. This function is documented on page 71.)

47.7 Producing multiple copies

```
8561 (@@=prg)
```

`\prg_replicate:nn` This function uses a cascading csnames technique by David Kastrup (who else :-)

`__prg_replicate:N` The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\exp:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

`__prg_replicate_0:n` This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use, and if necessary, it is possible to write `\prg_replicate:nn {1000} { \prg_replicate:nn {1000} {<code>} }`. An alternative approach is to create a string of m's with `\exp:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```

8562 \cs_new:Npn \prg_replicate:nn #1
8563   {
8564     \exp:w
8565     \exp_after:wN \__prg_replicate_first:N
8566     \int_value:w \int_eval:n {#1}
8567     \cs_end:

```

```

8568   }
8569   \cs_new:Npn \__prg_replicate:N #1
8570     { \cs:w __prg_replicate_#1 :n \__prg_replicate:N }
8571   \cs_new:Npn \__prg_replicate_first:N #1
8572     { \cs:w __prg_replicate_first_#1 :n \__prg_replicate:N }

```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes :n as a parameter.

```

8573   \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
8574   \cs_new:cpn { __prg_replicate_0:n } #1
8575     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
8576   \cs_new:cpn { __prg_replicate_1:n } #1
8577     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
8578   \cs_new:cpn { __prg_replicate_2:n } #1
8579     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
8580   \cs_new:cpn { __prg_replicate_3:n } #1
8581     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
8582   \cs_new:cpn { __prg_replicate_4:n } #1
8583     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
8584   \cs_new:cpn { __prg_replicate_5:n } #1
8585     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
8586   \cs_new:cpn { __prg_replicate_6:n } #1
8587     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
8588   \cs_new:cpn { __prg_replicate_7:n } #1
8589     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
8590   \cs_new:cpn { __prg_replicate_8:n } #1
8591     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
8592   \cs_new:cpn { __prg_replicate_9:n } #1
8593     { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }

```

Users shouldn't ask for something to be replicated once or even not at all but...

```

8594   \cs_new:cpn { __prg_replicate_first_-:n } #1
8595     {
8596       \exp_end:
8597       \msg_expandable_error:nn { prg } { negative-replication }
8598     }
8599   \cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
8600   \cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
8601   \cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
8602   \cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
8603   \cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
8604   \cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
8605   \cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
8606   \cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
8607   \cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
8608   \cs_new:cpn { __prg_replicate_first_9:n } #1
8609     { \exp_end: #1#1#1#1#1#1#1#1#1#1 }

```

(End of definition for \prg_replicate:nm and others. This function is documented on page 71.)

47.8 Detecting T_EX's mode

\mode_if_vertical_p: For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive

conditionals for this and gobbling the `\exp_end:` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
8610 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
8611   { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End of definition for `\mode_if_vertical:TF`. This function is documented on page 72.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```
\mode_if_horizontal:TF 8612 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
8613   { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End of definition for `\mode_if_horizontal:TF`. This function is documented on page 71.)

`\mode_if_inner_p:` For testing inner mode.

```
\mode_if_inner:TF 8614 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
8615   { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End of definition for `\mode_if_inner:TF`. This function is documented on page 72.)

`\mode_if_math_p:` For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```
\mode_if_math:TF 8616 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
8617   { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End of definition for `\mode_if_math:TF`. This function is documented on page 72.)

47.9 Internal programming functions

`\group_align_safe_begin:` `\group_align_safe_end:` T_EX’s alignment structures present many problems. As Knuth says himself in *T_EX: The Program*: “It’s sort of a miracle whenever `\halign` or `\valign` work, [...]” One problem relates to commands that internally issue a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we get some sort of weird error message because the underlying `\futurelet` stores the token at the end of the alignment template. This could be a `&_4` giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that T_EX still thinks it’s on safe ground but at the same time we don’t want to introduce any brace group that may find its way to the output. The following functions help with this by using behaviour documented only in Appendix D of *The T_EXbook*. . . In short evaluating ‘{ and ‘} as numbers will not change the counter T_EX uses to keep track of its state in an alignment, whereas gobbling a brace using `\if_false:` will affect T_EX’s state without producing any real group. We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```
8618 \group_begin:
8619 \tex_catcode:D ‘\^^@ = 2 \exp_stop_f:
8620 \cs_new:Npn \group_align_safe_begin:
8621   { \exp:w \if_false: { \fi: ‘^^@ \exp_stop_f: }
8622 \tex_catcode:D ‘\^^@ = 1 \exp_stop_f:
8623 \cs_new:Npn \group_align_safe_end:
8624   { \exp:w ‘^^@ \if_false: } \fi: \exp_stop_f: }
8625 \group_end:
```

(End of definition for `\group_align_safe_begin:` and `\group_align_safe_end:`. These functions are documented on page 73.)

`\g__kernel_prg_map_int` A nesting counter for mapping.
⁸⁶²⁶ `\int_new:N \g__kernel_prg_map_int`

(End of definition for `\g__kernel_prg_map_int.`)

`\prg_break_point:Nn` `\prg_map_break:Nn` These are defined in `l3basics`, as they are needed “early”. This is just a reminder that is the case!

(End of definition for `\prg_break_point:Nn` and `\prg_map_break:Nn.` These functions are documented on page 72.)

`\prg_break_point:` Also done in `l3basics`.

`\prg_break:`
`\prg_break:n`

(End of definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n.` These functions are documented on page 73.)

⁸⁶²⁷ `\package`

Chapter 48

l3sys implementation

```
8628 (@@=sys)
```

48.1 Kernel code

```
8629 (*package)
```

```
8630 (*tex)
```

```
\l__sys_tmp_tl
```

```
8631 \tl_new:N \l__sys_tmp_tl
```

(End of definition for \l__sys_tmp_tl.)

48.1.1 Detecting the engine

`__sys_const:nn` Set the T, F, TF, p forms of #1 to be constants equal to the result of evaluating the boolean expression #2.

```
8632 \cs_new_protected:Npn \__sys_const:nn #1#2
```

```
8633 {
```

```
8634   \bool_if:nTF {#2}
```

```
8635   {
```

```
8636     \cs_new_eq:cN { #1 :T } \use:n
```

```
8637     \cs_new_eq:cN { #1 :F } \use_none:n
```

```
8638     \cs_new_eq:cN { #1 :TF } \use_i:nn
```

```
8639     \cs_new_eq:cN { #1 _p: } \c_true_bool
```

```
8640   }
```

```
8641   {
```

```
8642     \cs_new_eq:cN { #1 :T } \use_none:n
```

```
8643     \cs_new_eq:cN { #1 :F } \use:n
```

```
8644     \cs_new_eq:cN { #1 :TF } \use_ii:nn
```

```
8645     \cs_new_eq:cN { #1 _p: } \c_false_bool
```

```
8646   }
```

```
8647 }
```

(End of definition for __sys_const:nn.)

`\sys_if_engine luatex_p:` Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive.

```
\sys_if_engine luatex:TF
```

```
\sys_if_engine pdftex_p:
```

```
\sys_if_engine pdftex:TF
```

```
8648 \str_const:Ne \c_sys_engine_str
```

```
\sys_if_engine ptex_p:
```

```
\sys_if_engine ptex:TF
```

```
\sys_if_engine uptex_p:
```

```
\sys_if_engine uptex:TF
```

```
\sys_if_engine xetex_p:
```

```
\sys_if_engine xetex:TF
```

```
\c_sys_engine_str
```

```

8649 {
8650   \cs_if_exist:NT \tex luatexversion:D { luatex }
8651   \cs_if_exist:NT \tex pdftexversion:D { pdftex }
8652   \cs_if_exist:NT \tex kanjiskip:D
8653     {
8654       \cs_if_exist:NTF \tex enablecjktoken:D
8655         { uptex }
8656         { ptex }
8657     }
8658   \cs_if_exist:NT \tex XeTeXversion:D { xetex }
8659 }
8660 \tl_map_inline:nn { { luatex } { pdftex } { ptex } { uptex } { xetex } }
8661 {
8662   \__sys_const:nn { sys_if_engine_ #1 }
8663   { \str_if_eq_p:Vn \c_sys_engine_str {#1} }
8664 }

```

(End of definition for `\sys_if_engine luatex:TF` and others. These functions are documented on page 75.)

`\c_sys_engine_exec_str` Take the functions defined above, and set up the engine and format names. `\c_sys_engine_exec_str` differs from `\c_sys_engine_str` as it is the *actual* engine name, not a “filtered” version. It differs for `ptex` and `uptex`, which have a leading `e`, and for `luatex`, because \LaTeX uses the Lua \LaTeX engine.

`\c_sys_engine_format_str` is quite similar to `\c_sys_engine_str`, except that it differentiates `pdflatex` from `latex` (which is pdf \TeX in DVI mode). This differentiation, however, is reliable only if the user doesn’t change `\tex_pdfoutput:D` before loading this code.

```

8665 \group_begin:
8666   \cs_set_eq:NN \lua_now:e \tex_directlua:D
8667   \str_const:Ne \c_sys_engine_exec_str
8668   {
8669     \sys_if_engine_pdftex:T { pdf }
8670     \sys_if_engine_xetex:T { xe }
8671     \sys_if_engine_ptex:T { ep }
8672     \sys_if_engine_uptex:T { eup }
8673     \sys_if_engine luatex:T
8674     {
8675       lua \lua_now:e
8676       {
8677         if (pcall(require, 'luaharfbuzz')) then ~
8678           tex.print("hb") ~
8679         end
8680       }
8681     }
8682     tex
8683   }
8684 \group_end:
8685 \str_const:Ne \c_sys_engine_format_str
8686 {
8687   \cs_if_exist:NTF \fmtname
8688   {
8689     \bool_lazy_or:nnTF
8690     { \str_if_eq_p:Vn \fmtname { plain } }

```

```

8691     { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
8692     {
8693       \sys_if_engine_pdftex:T
8694         { \int_compare:nNnT { \tex_pdfoutput:D } = { 1 } { pdf } }
8695       \sys_if_engine_xetex:T { xe }
8696       \sys_if_engine_ptex:T { p }
8697       \sys_if_engine_uptex:T { up }
8698       \sys_if_engine luatex:T
8699         {
8700           \int_compare:nNnT { \tex_pdfoutput:D } = { 0 } { dvi }
8701           lua
8702         }
8703       \str_if_eq:VnTF \fmtname { LaTeX2e }
8704       { latex }
8705       {
8706         \bool_lazy_and:nnT
8707           { \sys_if_engine_pdftex_p: }
8708           { \int_compare_p:nNn { \tex_pdfoutput:D } = { 0 } }
8709         { e }
8710       }
8711     }
8712   }
8713   { \fmtname }
8714 }
8715 { unknown }
8716 }

```

(End of definition for `\c_sys_engine_exec_str` and `\c_sys_engine_format_str`. These variables are documented on page 75.)

`\c_sys_engine_version_str` Various different engines, various different ways to extract the data!

```

8717 \str_const:Ne \c_sys_engine_version_str
8718 {
8719   \str_case:on \c_sys_engine_str
8720   {
8721     { pdftex }
8722     {
8723       \int_div_truncate:nn { \tex_pdfptextexversion:D } { 100 }
8724       .
8725       \int_mod:nn { \tex_pdfptextexversion:D } { 100 }
8726       .
8727       \tex_pdfptextexrevision:D
8728     }
8729     { ptex }
8730     {
8731       \cs_if_exist:NT \tex_ptextexversion:D
8732       {
8733         p
8734         \int_use:N \tex_ptextexversion:D
8735         .
8736         \int_use:N \tex_ptextexminorversion:D
8737         \tex_ptextexrevision:D
8738         -
8739         \int_use:N \tex_epTeXversion:D

```



```

8740     }
8741   }
8742   { luatex }
8743   {
8744     \int_div_truncate:nn { \tex_luatexversion:D } { 100 }
8745     .
8746     \int_mod:nn { \tex_luatexversion:D } { 100 }
8747     .
8748     \tex_luatexrevision:D
8749   }
8750   { uptex }
8751   {
8752     \cs_if_exist:NT \tex_ptexversion:D
8753     {
8754       p
8755       \int_use:N \tex_ptexversion:D
8756       .
8757       \int_use:N \tex_ptexminorversion:D
8758       \tex_ptexrevision:D
8759       -
8760       u
8761       \int_use:N \tex_uptexversion:D
8762       \tex_uptexrevision:D
8763       -
8764       \int_use:N \tex_epTeXversion:D
8765     }
8766   }
8767   { xetex }
8768   {
8769     \int_use:N \tex_XeTeXversion:D
8770     \tex_XeTeXrevision:D
8771   }
8772 }
8773 }

```

(End of definition for `\c_sys_engine_version_str`. This variable is documented on page 75.)

48.1.2 Platform

`\sys_if_platform_unix_p:` Setting these up requires the file module (file lookup), so is actually implemented there.

`\sys_if_platform_unix:TF`

(End of definition for `\sys_if_platform_unix:TF`, `\sys_if_platform_windows:TF`, and `\c_sys_platform_str`. These functions are documented on page 76.)

`\sys_if_platform_windows_p:`

`\sys_if_platform_windows:TF`

`\c_sys_platform_str`

48.1.3 Configurations

`\sys_load_backend:n` Loading the backend code is pretty simply: check that the backend is valid, then load it up.

`__sys_load_backend_check:N`

`\c_sys_backend_str`

```

8774 \cs_new_protected:Npn \sys_load_backend:n #1
8775 {
8776   \sys_finalise:
8777   \str_if_exist:NTF \c_sys_backend_str
8778   {
8779     \str_if_eq:VnF \c_sys_backend_str {#1}

```

```

8780     { \msg_error:nn { sys } { backend-set } }
8781   }
8782   {
8783     \tl_if_blank:nF {#1}
8784     { \tl_gset:Nn \g__sys_backend_tl {#1} }
8785     \__sys_load_backend_check:N \g__sys_backend_tl
8786     \str_const:Ne \c_sys_backend_str { \g__sys_backend_tl }
8787     \__kernel_sys_configuration_load:n
8788     { l3backend- \c_sys_backend_str }
8789   }
8790 }
8791 \cs_new_protected:Npn \__sys_load_backend_check:N #1
8792 {
8793   \sys_if_engine_xetex:TF
8794   {
8795     \str_case:VnF #1
8796     {
8797       { dvisvgm } { }
8798       { xdvipdfmx } { \tl_gset:Nn #1 { xetex } }
8799       { xetex } { }
8800     }
8801     {
8802       \msg_error:nnee { sys } { wrong-backend }
8803       #1 { xetex }
8804       \tl_gset:Nn #1 { xetex }
8805     }
8806   }
8807   {
8808     \sys_if_output_pdf:TF
8809     {
8810       \str_if_eq:VnTF #1 { pdfmode }
8811       {
8812         \sys_if_engine_luatex:TF
8813         { \tl_gset:Nn #1 { luatex } }
8814         { \tl_gset:Nn #1 { pdftex } }
8815       }
8816       {
8817         \bool_lazy_or:nnF
8818         { \str_if_eq_p:Vn #1 { luatex } }
8819         { \str_if_eq_p:Vn #1 { pdftex } }
8820         {
8821           \msg_error:nnee { sys } { wrong-backend }
8822           #1 { \sys_if_engine_luatex:TF { luatex } { pdftex } }
8823           \sys_if_engine_luatex:TF
8824           { \tl_gset:Nn #1 { luatex } }
8825           { \tl_gset:Nn #1 { pdftex } }
8826         }
8827       }
8828     }
8829     {
8830       \str_case:VnF #1
8831       {
8832         { dvipdfmx } { }
8833         { dvips } { }

```

```

8834         { dvisvgm } { }
8835     }
8836     {
8837         \msg_error:nnee { sys } { wrong-backend }
8838         #1 { dvips }
8839         \tl_gset:Nn #1 { dvips }
8840     }
8841 }
8842 }
8843 }

```

(End of definition for `\sys_load_backend:n`, `_sys_load_backend_check:N`, and `\c_sys_backend_str`. These functions are documented on page 79.)

`\sys_ensure_backend:` A simple wrapper.

```

8844 \cs_new_protected:Npn \sys_ensure_backend:
8845 {
8846     \str_if_exist:NF \c_sys_backend_str
8847     { \sys_load_backend:n { } }
8848 }

```

(End of definition for `\sys_ensure_backend:.` This function is documented on page 79.)

`\g__sys_debug_bool`

```

8849 \bool_new:N \g__sys_debug_bool

```

(End of definition for `\g__sys_debug_bool`.)

`\sys_load_debug:` The most complicated thing here is that we can only use `_kernel_sys_configuration_load:n` in the preamble in L^AT_EX.

```

8850 \cs_new_protected:Npn \sys_load_debug:
8851 {
8852     \bool_if:NF \g__sys_debug_bool
8853     { \_kernel_sys_configuration_load:n { l3debug } }
8854     \bool_gset_true:N \g__sys_debug_bool
8855 }
8856 \cs_if_exist:NT \@expl@finalise@setup@@
8857 {
8858     \tl_gput_right:Nn \@expl@finalise@setup@@
8859     {
8860         \tl_gput_right:Nn \@kernel@after@begindocument
8861         {
8862             \cs_gset_protected:Npn \sys_load_debug:
8863             { \msg_error:nn { sys } { load-debug-in-preamble } }
8864         }
8865     }
8866 }

```

(End of definition for `\sys_load_debug:.` This function is documented on page 79.)

48.1.4 Access to the shell

`\l__sys_internal_tl`

```
8867 \tl_new:N \l__sys_internal_tl
```

(End of definition for `\l__sys_internal_tl`.)

`\c__sys_marker_tl`

The same idea as the marker for rescanning token lists.

```
8868 \tl_const:Nc \c__sys_marker_tl { : \token_to_str:N : }
```

(End of definition for `\c__sys_marker_tl`.)

`\sys_get_shell:nnN`

Setting using a shell is at this level just a slightly specialised file operation, with an additional check for quotes, as these are not supported.

`\sys_get_shell:nnN`

`__sys_get:nnN`

`__sys_get_do:Nw`

```
8869 \cs_new_protected:Npn \sys_get_shell:nnN #1#2#3
```

```
8870 {
```

```
8871   \sys_get_shell:nnNF {#1} {#2} #3
```

```
8872   { \tl_set:Nn #3 { \q_no_value } }
```

```
8873 }
```

```
8874 \prg_new_protected_conditional:Npnn \sys_get_shell:nnN #1#2#3 { T , F , TF }
```

```
8875 {
```

```
8876   \sys_if_shell:TF
```

```
8877   { \exp_args:No \__sys_get:nnN { \tl_to_str:n {#1} } {#2} #3 }
```

```
8878   { \prg_return_false: }
```

```
8879 }
```

```
8880 \cs_new_protected:Npn \__sys_get:nnN #1#2#3
```

```
8881 {
```

```
8882   \tl_if_in:nnTF {#1} { " }
```

```
8883   {
```

```
8884     \msg_error:nne
```

```
8885     { kernel } { quote-in-shell } {#1}
```

```
8886     \prg_return_false:
```

```
8887   }
```

```
8888   {
```

```
8889     \group_begin:
```

```
8890     \if_false: { \fi:
```

```
8891     \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
```

```
8892     \exp_args:No \tex_everyeof:D { \c__sys_marker_tl }
```

```
8893     #2 \scan_stop:
```

```
8894     \exp_after:wN \__sys_get_do:Nw
```

```
8895     \exp_after:wN #3
```

```
8896     \exp_after:wN \prg_do_nothing:
```

```
8897     \tex_input:D | "#1" \scan_stop:
```

```
8898     \if_false: } \fi:
```

```
8899     \prg_return_true:
```

```
8900   }
```

```
8901 }
```

```
8902 \exp_args:Nno \use:nn
```

```
8903 { \cs_new_protected:Npn \__sys_get_do:Nw #1#2 }
```

```
8904 { \c__sys_marker_tl }
```

```
8905 {
```

```
8906   \group_end:
```

```
8907   \tl_set:No #1 {#2}
```

```
8908 }
```

(End of definition for `\sys_get_shell:nnNTF` and others. These functions are documented on page 77.)

`\c__sys_shell_stream_int` This is not needed for LuaTeX: shell escape there isn't done using a TeX interface.

```
8909 \sys_if_engine luatex:F
8910 { \int_const:Nn \c__sys_shell_stream_int { 18 } }
```

(End of definition for `\c__sys_shell_stream_int`.)

`\sys_shell_now:n` Execute commands through shell escape immediately.
`\sys_shell_now:e` For LuaTeX, we use a pseudo-primitive to do the actual work.

```
8911 </tex>
8912 <lua>
8913 do
8914   local os_exec = os.execute
8915
8916   local function shellescape(cmd)
8917     local status,msg = os_exec(cmd)
8918     if status == nil then
8919       write_nl("log","runsystem(" .. cmd .. ")...(" .. msg .. ")\n")
8920     elseif status == 0 then
8921       write_nl("log","runsystem(" .. cmd .. ")...executed\n")
8922     else
8923       write_nl("log","runsystem(" .. cmd .. ")...failed " .. (msg or "") .. "\n")
8924     end
8925   end
8926   luacmd("__sys_shell_now:e", function()
8927     shellescape(scan_string())
8928   end, "global", "protected")
8929 </lua>
8930 <tex>
8931 \sys_if_engine luatex:TF
8932 {
8933   \cs_new_protected:Npn \sys_shell_now:n #1
8934     { \__sys_shell_now:e { \exp_not:n {#1} } }
8935 }
8936 {
8937   \cs_new_protected:Npn \sys_shell_now:n #1
8938     { \iow_now:Nn \c__sys_shell_stream_int {#1} }
8939 }
8940 \cs_generate_variant:Nn \sys_shell_now:n { e, x }
8941 </tex>
```

(End of definition for `\sys_shell_now:n` and `__sys_shell_now:e`. This function is documented on page 78.)

`\sys_shell_shipout:n` Execute commands through shell escape at shipout.
`\sys_shell_shipout:e` For LuaTeX, we use the same helper as above but delayed using a `late_lua` whatsit.
`\sys_shell_shipout:x` Creating a `late_lua` whatsit works a bit different if we are running under ConTeXt.

```
8942 <lua>
8943   local new_latelua = nodes and nodes.nuts and nodes.nuts.pool and nodes.nuts.pool.latelua
8944   local whatsit_id = node.id'whatsit'
8945   local latelua_sub = node.subtype'late_lua'
8946   local node_new = node.direct.new
8947   local setfield = node.direct.setwhatsitfield or node.direct.setfield
```

```

8948     return function(f)
8949         local n = node_new(whatsit_id, latelua_sub)
8950         setfield(n, 'data', f)
8951         return n
8952     end
8953 end)()
8954 local node_write = node.direct.write
8955
8956 luacmd("__sys_shell_shipout:e", function()
8957     local cmd = scan_string()
8958     node_write(new_latelua(function() shellescape(cmd) end))
8959 end, "global", "protected")
8960 end
8961 </lua>
8962 <*tex>
8963 \sys_if_engine luatex:TF
8964 {
8965     \cs_new_protected:Npn \sys_shell_shipout:n #1
8966     { \__sys_shell_shipout:e { \exp_not:n {#1} } }
8967 }
8968 {
8969     \cs_new_protected:Npn \sys_shell_shipout:n #1
8970     { \iow_shipout:Nn \c__sys_shell_stream_int {#1} }
8971 }
8972 \cs_generate_variant:Nn \sys_shell_shipout:n { e , x }

```

(End of definition for `\sys_shell_shipout:n` and `__sys_shell_shipout:e`. This function is documented on page 78.)

48.2 Dynamic (every job) code

```

\__kernel_sys_everyjob:
  \__sys_everyjob:n
  \g__sys_everyjob_tl
8973 \cs_new_protected:Npn \__kernel_sys_everyjob:
8974 {
8975     \tl_use:N \g__sys_everyjob_tl
8976     \tl_gclear:N \g__sys_everyjob_tl
8977 }
8978 \cs_new_protected:Npn \__sys_everyjob:n #1
8979 { \tl_gput_right:Nn \g__sys_everyjob_tl {#1} }
8980 \tl_new:N \g__sys_everyjob_tl

```

(End of definition for `__kernel_sys_everyjob:`, `__sys_everyjob:n`, and `\g__sys_everyjob_tl`.)

48.2.1 The name of the job

`\c_sys_jobname_str` Inherited from the L^AT_EX3 name for the primitive. This *has* to be the primitive as it's set in `\everyjob`. If the user does

```
pdflatex \input some-file-name
```

then `\everyjob` is inserted *before* `\jobname` is changed from `texput`, and thus we would have the wrong result.

```

8981 \__sys_everyjob:n
8982 { \cs_new_eq:NN \c_sys_jobname_str \tex_jobname:D }

```

(End of definition for `\c_sys_jobname_str`. This variable is documented on page 74.)

48.2.2 Time and date

`\c_sys_minute_int` Copies of the information provided by T_EX. There is a lot of defensive code in package mode: someone may have moved the primitives, and they can only be recovered if we have `\primitive` and it is working correctly. For IniT_EX of course that is all redundant but does no harm.

```
8983 \__sys_everyjob:n
8984 {
8985   \group_begin:
8986   \cs_set:Npn \__sys_tmp:w #1
8987     {
8988     \str_if_eq:eeTF { \cs_meaning:N #1 } { \token_to_str:N #1 }
8989     { #1 }
8990     {
8991     \cs_if_exist:NTF \tex_primitive:D
8992     {
8993     \bool_lazy_and:nnTF
8994     { \sys_if_engine_xetex_p: }
8995     {
8996     \int_compare_p:nNn
8997     { \exp_after:wN \use_none:n \tex_XeTeXrevision:D }
8998     < { 99999 }
8999     }
9000     { 0 }
9001     { \tex_primitive:D #1 }
9002     }
9003     { 0 }
9004   }
9005 }
9006 \int_const:Nn \c_sys_minute_int
9007 { \int_mod:nn { \__sys_tmp:w \time } { 60 } }
9008 \int_const:Nn \c_sys_hour_int
9009 { \int_div_truncate:nn { \__sys_tmp:w \time } { 60 } }
9010 \int_const:Nn \c_sys_day_int { \__sys_tmp:w \day }
9011 \int_const:Nn \c_sys_month_int { \__sys_tmp:w \month }
9012 \int_const:Nn \c_sys_year_int { \__sys_tmp:w \year }
9013 \group_end:
9014 }
```

(End of definition for `\c_sys_minute_int` and others. These variables are documented on page 74.)

`\c_sys_timestamp_str` A simple expansion: LuaT_EX chokes if we use `\pdffeedback` here, hence the direct use of Lua. Notice that the function there is in the pdf library but isn't actually tied to PDF.

```
9015 \__sys_everyjob:n
9016 {
9017   \str_const:Ne \c_sys_timestamp_str
9018   {
9019   \cs_if_exist:NTF \tex_directlua:D
9020   { \tex_directlua:D { tex.print(pdf.getcreationdate()) } }
9021   { \tex_creationdate:D }
9022   }
9023 }
```

(End of definition for `\c_sys_timestamp_str`. This variable is documented on page 74.)

48.2.3 Random numbers

`\sys_rand_seed:` Unpack the primitive.

```
9024 \__sys_everyjob:n
9025 {
9026   \cs_new:Npn \sys_rand_seed: { \tex_the:D \tex_randomseed:D }
9027 }
```

(End of definition for `\sys_rand_seed:`. This function is documented on page 76.)

`\sys_gset_rand_seed:n` The primitive always assigns the seed globally.

```
9028 \__sys_everyjob:n
9029 {
9030   \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9031     { \tex_setrandomseed:D \int_eval:n {#1} \exp_stop_f: }
9032 }
```

(End of definition for `\sys_gset_rand_seed:n`. This function is documented on page 77.)

`\sys_timer:` In LuaTeX, create a pseudo-primitive, otherwise try to locate the real primitive. The elapsed time will be available if this succeeds.

```
\__sys_elapsedtime:
\sys_if_timer_exist_p:
\sys_if_timer_exist:TF
9033 </tex>
9034 <lua>
9035   local gettimeofday = os.gettimeofday
9036   local epoch = gettimeofday() - os.clock()
9037   local write = tex.write
9038   local tointeger = math.tointeger
9039   luacmd('__sys_elapsedtime:', function()
9040     write(tointeger((gettimeofday() - epoch)*65536 // 1))
9041   end, 'global')
9042 </lua>
9043 <*tex>
9044 \sys_if_engine luatex:TF
9045 {
9046   \cs_new:Npn \sys_timer:
9047     { \__sys_elapsedtime: }
9048 }
9049 {
9050   \cs_if_exist:NTF \tex_elapsedtime:D
9051     {
9052       \cs_new:Npn \sys_timer:
9053         { \int_value:w \tex_elapsedtime:D }
9054     }
9055     {
9056       \cs_new:Npn \sys_timer:
9057         {
9058           \int_value:w
9059           \msg_expandable_error:nnn { kernel } { no-elapsed-time }
9060           { \sys_timer: }
9061           \c_zero_int
9062         }
9063     }
```



```

9064 }
9065 \__sys_const:nn { sys_if_timer_exist }
9066 { \cs_if_exist_p:N \tex_elapsedtime:D || \cs_if_exist_p:N \__sys_elapsedtime: }

```

(End of definition for `\sys_timer:`, `__sys_elapsedtime:`, and `\sys_if_timer_exist:TF`. These functions are documented on page 75.)

48.2.4 Access to the shell

`\c_sys_shell_escape_int` Expose the engine's shell escape status to the user.

```

9067 \__sys_everyjob:n
9068 {
9069   \int_const:Nn \c_sys_shell_escape_int
9070   {
9071     \sys_if_engine_luatex:TF
9072     {
9073       \tex_directlua:D
9074       { tex.sprint(status.shell_escape~or~os.execute()) }
9075     }
9076     { \tex_shellescape:D }
9077   }
9078 }

```

(End of definition for `\c_sys_shell_escape_int`. This variable is documented on page 77.)

`\sys_if_shell_p:` Performs a check for whether shell escape is enabled. The first set of functions returns true if either of restricted or unrestricted shell escape is enabled, while the other two sets of functions return true in only one of these two cases.

`\sys_if_shell:TF`

`\sys_if_shell_unrestricted_p:`

`\sys_if_shell_unrestricted:TF`

`\sys_if_shell_restricted_p:`

`\sys_if_shell_restricted:TF`

```

9079 \__sys_everyjob:n
9080 {
9081   \__sys_const:nn { sys_if_shell }
9082   { \int_compare_p:nNn \c_sys_shell_escape_int > 0 }
9083   \__sys_const:nn { sys_if_shell_unrestricted }
9084   { \int_compare_p:nNn \c_sys_shell_escape_int = 1 }
9085   \__sys_const:nn { sys_if_shell_restricted }
9086   { \int_compare_p:nNn \c_sys_shell_escape_int = 2 }
9087 }

```

(End of definition for `\sys_if_shell:TF`, `\sys_if_shell_unrestricted:TF`, and `\sys_if_shell_restricted:TF`. These functions are documented on page 77.)

48.3 System queries

`\sys_get_query:nN` Calling the query system is quite straight-forward: most of the effort is in making the read-back catcode-safe. We also want to trim off the trailing `^^M` from the last line.

`\sys_get_query:nnN`

`\sys_get_query:nnnN`

`__sys_get_query_auxi:nnnN`

`__sys_get_query_auxi:neeN`

`__sys_get_query_auxii:nnnN`

`__sys_get_query_auxii:neeN`

```

9088 \cs_new_protected:Npn \sys_get_query:nN #1#2
9089 { \sys_get_query:nnnN {#1} { } { } #2 }
9090 \cs_new_protected:Npn \sys_get_query:nnN #1#2#3
9091 { \sys_get_query:nnnN {#1} { } {#2} #3 }
9092 \cs_new_protected:Npn \sys_get_query:nnnN #1#2#3#4
9093 {
9094   \tl_clear:N #4
9095   \__sys_get_query_auxi:neeN {#1} {#2} {#3} #4

```

```

9096 }
9097 \cs_new:Npn \__sys_get_query_auxi:nnnN #1#2#3#4
9098 {
9099   \__sys_get_query_auxii:neeN {#1}
9100   { \tl_if_blank:nF {#2} { \tl_to_str:n { ~ #2 } } }
9101   {
9102     \tl_if_blank:nF {#3}
9103     {
9104       \c_space_tl
9105       \sys_if_shell_restricted:F '
9106       \tl_to_str:n {#3}
9107       \sys_if_shell_restricted:F '
9108     }
9109   }
9110   #4
9111 }
9112 \cs_generate_variant:Nn \__sys_get_query_auxi:nnnN { nee }
9113 \cs_new_protected:Npn \__sys_get_query_auxii:nnnN #1#2#3#4
9114 {
9115   \sys_if_shell:T
9116   {
9117     \sys_get_shell:nnN
9118     { l3sys-query~#1 #2 #3 }
9119     {
9120       \int_step_inline:nnn { 0 } { 'A - 1 }
9121       { \char_set_catcode_other:n {##1} }
9122       \int_step_inline:nnn { 'Z + 1 } { 'a - 1 }
9123       { \char_set_catcode_other:n {##1} }
9124       \int_step_inline:nnn { 'z + 1 } { 127 }
9125       { \char_set_catcode_other:n {##1} }
9126       \char_set_catcode_active:n { '\ }
9127       \tex_endlinechar:D 13 \scan_stop:
9128     }
9129     \l__sys_tmp_tl
9130     \tl_if_empty:NF \l__sys_tmp_tl
9131     {
9132       \exp_after:wN \__sys_get_query:Nw \exp_after:wN #4
9133       \l__sys_tmp_tl \q_stop
9134     }
9135   }
9136 }
9137 \cs_generate_variant:Nn \__sys_get_query_auxii:nnnN { nee }
9138 \group_begin:
9139   \tex_lccode:D '\* = 13 \scan_stop:
9140   \tex_lowercase:D
9141   {
9142     \group_end:
9143     \cs_new_protected:Npn \__sys_get_query:Nw #1#2 * \q_stop
9144   }
9145   { \tl_set:Nn #1 {#2} }

```

(End of definition for `\sys_get_query:nN` and others. These functions are documented on page 78.)

`\sys_split_query:nN`
`\sys_split_query:nnN`
`\sys_split_query:nnnN`

A wrapper for convenience.

```

9146 \cs_new_protected:Npn \sys_split_query:nN #1#2
9147   { \sys_split_query:nnnN {#1} { } { } #2 }
9148 \cs_new_protected:Npn \sys_split_query:nnN #1#2#3
9149   { \sys_split_query:nnnN {#1} { } {#2} #3 }
9150 \group_begin:
9151   \tex_lccode:D ‘\* = 13 \scan_stop:
9152   \tex_lowercase:D
9153   {
9154     \group_end:
9155     \cs_new_protected:Npn \sys_split_query:nnnN #1#2#3#4
9156       {
9157         \seq_clear:N #4
9158         \sys_get_query:nnnN {#1} {#2} {#3} \l__sys_tmp_tl
9159         \tl_if_empty:NF \l__sys_tmp_tl
9160           { \seq_set_split:NnV #4 * \l__sys_tmp_tl }
9161       }
9162   }

```

(End of definition for `\sys_split_query:nN`, `\sys_split_query:nnN`, and `\sys_split_query:nnnN`. These functions are documented on page 79.)

48.3.1 Held over from l3file

`\g_file_curr_name_str` See comments about `\c_sys_jobname_str`: here, as soon as there is file input/output, things get “tided up”.

```

9163 \__sys_everyjob:n
9164   { \cs_gset_eq:NN \g_file_curr_name_str \tex_jobname:D }

```

(End of definition for `\g_file_curr_name_str`. This variable is documented on page 99.)

48.4 Last-minute code

`\sys_finalise:` A simple hook to finalise the system-dependent layer. This is forced by the backend loader, which is forced by the main loader, so we do not need to include that here.

```

\__sys_finalise:n
\g__sys_finalise_tl
9165 \cs_new_protected:Npn \sys_finalise:
9166   {
9167     \__kernel_sys_everyjob:
9168     \tl_use:N \g__sys_finalise_tl
9169     \tl_gclear:N \g__sys_finalise_tl
9170   }
9171 \cs_new_protected:Npn \__sys_finalise:n #1
9172   { \tl_gput_right:Nn \g__sys_finalise_tl {#1} }
9173 \tl_new:N \g__sys_finalise_tl

```

(End of definition for `\sys_finalise:`, `__sys_finalise:n`, and `\g__sys_finalise_tl`. This function is documented on page 79.)

48.4.1 Detecting the output

`\sys_if_output_dvi_p:` This is a simple enough concept: the two views here are complementary.

```

\sys_if_output_dvi_p:TF
\sys_if_output_pdf_p:
\sys_if_output_pdf:TF
\c_sys_output_str
9174 \__sys_finalise:n
9175   {
9176     \str_const:Ne \c_sys_output_str

```

```

9177     {
9178       \int_compare:nNnTF
9179         { \cs_if_exist_use:NF \tex_pdfoutput:D { 0 } } > { 0 }
9180         { pdf }
9181         { dvi }
9182     }
9183     \__sys_const:nn { sys_if_output_dvi }
9184     { \str_if_eq_p:Vn \c_sys_output_str { dvi } }
9185     \__sys_const:nn { sys_if_output_pdf }
9186     { \str_if_eq_p:Vn \c_sys_output_str { pdf } }
9187 }

```

(End of definition for `\sys_if_output_dvi:TF`, `\sys_if_output_pdf:TF`, and `\c_sys_output_str`. These functions are documented on page 76.)

48.4.2 Configurations

`\g__sys_backend_tl` As the backend has to be checked and possibly adjusted, the approach here is to create a variable and use that in a one-shot to set a constant.

```

9188 \tl_new:N \g__sys_backend_tl
9189 \__sys_finalise:n
9190 {
9191   \__kernel_tl_gset:Nx \g__sys_backend_tl
9192   {
9193     \sys_if_engine_xetex:TF
9194     { xetex }
9195     {
9196       \sys_if_output_pdf:TF
9197       {
9198         \sys_if_engine_pdftex:TF
9199         { pdftex }
9200         { luatex }
9201       }
9202       { dvips }
9203     }
9204   }
9205 }

```

If there is a class option set, and recognised, we pick it up: these will over-ride anything set automatically but will themselves be over-written if there is a package option.

```

9206 \__sys_finalise:n
9207 {
9208   \cs_if_exist:NT \@classoptionslist
9209   {
9210     \cs_if_eq:NNF \@classoptionslist \scan_stop:
9211     {
9212       \clist_map_inline:Nn \@classoptionslist
9213       {
9214         \str_case:nnT {#1}
9215         {
9216           { dvipdfmx }
9217           { \tl_gset:Nn \g__sys_backend_tl { dvipdfmx } }
9218           { dvips }
9219           { \tl_gset:Nn \g__sys_backend_tl { dvips } }

```

```

9220         { dvisvgn }
9221         { \tl_gset:Nn \g__sys_backend_tl { dvisvgn } }
9222     { pdftex }
9223         { \tl_gset:Nn \g__sys_backend_tl { pdfmode } }
9224     { xetex }
9225         { \tl_gset:Nn \g__sys_backend_tl { xdvipdfmx } }
9226     }
9227     { \clist_remove_all:Nn \@unusedoptionlist {#1} }
9228   }
9229 }
9230 }
9231 }

```

(End of definition for \g__sys_backend_tl.)

```

9232 </tex>
9233 </package>

```

Chapter 49

l3msg implementation

```
9234 (*package)
9235 (@@=msg)
\l__msg_internal_tl A general scratch for the module.
9236 \tl_new:N \l__msg_internal_tl
(End of definition for \l__msg_internal_tl.)
\l__msg_name_str Used to save module info when creating messages.
\l__msg_text_str 9237 \str_new:N \l__msg_name_str
9238 \str_new:N \l__msg_text_str
(End of definition for \l__msg_name_str and \l__msg_text_str.)
```

49.1 Internal auxiliaries

```
\s__msg_mark Internal scan marks.
\s__msg_stop 9239 \scan_new:N \s__msg_mark
9240 \scan_new:N \s__msg_stop
(End of definition for \s__msg_mark and \s__msg_stop.)
\_msg_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.
9241 \cs_new:Npn \_msg_use_none_delimit_by_s_stop:w #1 \s__msg_stop { }
(End of definition for \_msg_use_none_delimit_by_s_stop:w.)
```

49.2 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

```
\c__msg_text_prefix_tl Locations for the text of messages.
\c__msg_more_text_prefix_tl 9242 \tl_const:Nn \c__msg_text_prefix_tl { msg-text~>~ }
9243 \tl_const:Nn \c__msg_more_text_prefix_tl { msg-extra~text~>~ }
```

(End of definition for \c__msg_text_prefix_tl and \c__msg_more_text_prefix_tl.)

\msg_if_exist_p:nn Test whether the control sequence containing the message text exists or not.
\msg_if_exist:nnTF

```
9244 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
9245 {
9246   \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
9247   { \prg_return_true: } { \prg_return_false: }
9248 }
```

(End of definition for \msg_if_exist:nnTF. This function is documented on page 81.)

__msg_chk_if_free:nn This auxiliary is similar to __kernel_chk_if_free_cs:N, and is used when defining messages with \msg_new:nnnn.

```
9249 \cs_new_protected:Npn \__msg_chk_free:nn #1#2
9250 {
9251   \msg_if_exist:nnT {#1} {#2}
9252   {
9253     \msg_error:nnnn { msg } { already-defined }
9254     {#1} {#2}
9255   }
9256 }
```

(End of definition for __msg_chk_if_free:nn.)

\msg_new:nnnn Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```
\msg_new:nnee
\msg_new:nnxx
\msg_new:nnn
\msg_new:nne
\msg_new:nnx
\msg_set:nnnn
\msg_set:nnn
```

```
9257 \cs_new_protected:Npn \msg_new:nnnn #1#2#3#4
9258 {
9259   \__msg_chk_free:nn {#1} {#2}
9260   \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
9261   ##1##2##3##4 {#3}
9262   \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
9263   ##1##2##3##4 {#4}
9264 }
9265 \cs_generate_variant:Nn \msg_new:nnnn { nnee , nnxx }
9266 \cs_new_protected:Npn \msg_new:nnn #1#2#3
9267 { \msg_new:nnnn {#1} {#2} {#3} { } }
9268 \cs_generate_variant:Nn \msg_new:nnn { nne , nnx }
9269 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
9270 {
9271   \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
9272   ##1##2##3##4 {#3}
9273   \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
9274   ##1##2##3##4 {#4}
9275 }
9276 \cs_new_protected:Npn \msg_set:nnn #1#2#3
9277 { \msg_set:nnnn {#1} {#2} {#3} { } }
```

(End of definition for \msg_new:nnnn and others. These functions are documented on page 81.)

49.3 Messages: support functions and text

```

\c__msg_coding_error_text_tl Simple pieces of text for messages.
\c__msg_continue_text_tl     9278 \tl_const:Nn \c__msg_coding_error_text_tl
\c__msg_critical_text_tl     9279 {
\c__msg_fatal_text_tl       9280   This-is-a-coding-error.
\c__msg_help_text_tl       9281   \\\ \\\
\c__msg_no_info_text_tl     9282 }
\c__msg_on_line_text_tl     9283 \tl_const:Nn \c__msg_continue_text_tl
\c__msg_return_text_tl     9284 { Type~<return>~to~continue }
\c__msg_trouble_text_tl    9285 \tl_const:Nn \c__msg_critical_text_tl
                             9286 { Reading~the~current~file~'\g_file_curr_name_str'~will~stop. }
                             9287 \tl_const:Nn \c__msg_fatal_text_tl
                             9288 { This-is-a-fatal-error:-LaTeX-will-abort. }
                             9289 \tl_const:Nn \c__msg_help_text_tl
                             9290 { For~immediate-help~type-H~<return> }
                             9291 \tl_const:Nn \c__msg_no_info_text_tl
                             9292 {
                             9293   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
                             9294   \c__msg_return_text_tl
                             9295 }
                             9296 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
                             9297 \tl_const:Nn \c__msg_return_text_tl
                             9298 {
                             9299   \\\ \\\
                             9300   Try~typing~<return>~to~proceed.
                             9301   \\\
                             9302   If~that~doesn't~work,~type-X~<return>~to~quit.
                             9303 }
                             9304 \tl_const:Nn \c__msg_trouble_text_tl
                             9305 {
                             9306   \\\ \\\
                             9307   More~errors~will~almost~certainly~follow: \\\
                             9308   the~LaTeX~run~should~be~aborted.
                             9309 }

```

(End of definition for \c__msg_coding_error_text_tl and others.)

\msg_line_number: For writing the line number nicely. **\msg_line_context:** was set up earlier, so this is not new.

```

\msg_line_context:
9310 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
9311 \cs_gset:Npn \msg_line_context:
9312 {
9313   \c__msg_on_line_text_tl
9314   \c_space_tl
9315   \msg_line_number:
9316 }

```

(End of definition for \msg_line_number: and \msg_line_context:. These functions are documented on page 82.)

49.4 Showing messages: low level mechanism

`_msg_interrupt:Nnnn`
`_msg_no_more_text:nnnn`

The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TeX's `\errhelp` register before issuing an `\errmessage`. To deal with the various cases of critical or fatal errors with and without help text, there is a bit of argument-passing to do.

```

9317 \cs_new_protected:Npn \_msg_interrupt:NnnnN #1#2#3#4#5
9318   {
9319     \str_set:Ne \l_msg_text_str { #1 {#2} }
9320     \str_set:Ne \l_msg_name_str { \msg_module_name:n {#2} }
9321     \cs_if_eq:cNTF
9322       { \c_msg_more_text_prefix_tl #2 / #3 }
9323       \_msg_no_more_text:nnnn
9324       {
9325         \_msg_interrupt_wrap:nnn
9326         { \use:c { \c_msg_text_prefix_tl #2 / #3 } #4 }
9327         { \c_msg_continue_text_tl }
9328         {
9329           \c_msg_no_info_text_tl
9330           \tl_if_empty:NF #5
9331           { \ \ \ #5 }
9332         }
9333       }
9334     {
9335       \_msg_interrupt_wrap:nnn
9336       { \use:c { \c_msg_text_prefix_tl #2 / #3 } #4 }
9337       { \c_msg_help_text_tl }
9338       {
9339         \use:c { \c_msg_more_text_prefix_tl #2 / #3 } #4
9340         \tl_if_empty:NF #5
9341         { \ \ \ #5 }
9342       }
9343     }
9344   }
9345 \cs_new:Npn \_msg_no_more_text:nnnn #1#2#3#4 { }

```

(End of definition for `_msg_interrupt:Nnnn` and `_msg_no_more_text:nnnn`.)

`_msg_interrupt_wrap:nnn`
`_msg_interrupt_text:n`
`_msg_interrupt_more_text:n`

First setup TeX's `\errhelp` register with the extra help #1, then build a nice-looking error message with #2. Everything is done using e-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `_msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion. We have to split the main text into two parts as only the “message” itself is wrapped with a leader: the generic help is wrapped at full width. We also have to allow for the two characters used by `\errmessage` itself.

```

9346 \cs_new_protected:Npn \_msg_interrupt_wrap:nnn #1#2#3
9347   {
9348     \iow_wrap:nnn { \ \ #3 } { } { } \_msg_interrupt_more_text:n
9349     \group_begin:
9350       \int_sub:Nn \l_iow_line_count_int { 2 }
9351       \iow_wrap:nenN { \l_msg_text_str : ~ #1 }

```

```

9352     {
9353       ( \l__msg_name_str )
9354       \prg_replicate:nn
9355         {
9356           \str_count:N \l__msg_text_str
9357           - \str_count:N \l__msg_name_str
9358             + 2
9359         }
9360       { ~ }
9361     }
9362     { } \__msg_interrupt_text:n
9363     \iow_wrap:nnn { \l__msg_internal_tl \\ \\ #2 } { } { }
9364     \__msg_interrupt:n
9365   }
9366   \cs_new_protected:Npn \__msg_interrupt_text:n #1
9367   {
9368     \group_end:
9369     \tl_set:Nn \l__msg_internal_tl {#1}
9370   }
9371   \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
9372   { \exp_args:Ne \tex_errhelp:D { #1 \iow_newline: } }

```

(End of definition for `__msg_interrupt_wrap:nnn`, `__msg_interrupt_text:n`, and `__msg_interrupt_more_text:n`.)

`__msg_interrupt:n` The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made “invisible”: T_EX’s own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active `!` to call the `\errmessage` primitive, and end its argument with `\use_none:n` `{\spaces}` which fills the output with spaces. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active `!` is closed before producing the message: this ensures that tokens inserted by typing `I` in the command-line are inserted after the message is entirely cleaned up.

The `__kernel_iow_with:Nnn` auxiliary, defined in `l3file`, expects an *(integer variable)*, an integer *(value)*, and some *(code)*. It runs the *(code)* after ensuring that the *(integer variable)* takes the given *(value)*, then restores the former value of the *(integer variable)* if needed. We use it to ensure that the `\newlinechar` is 10, as needed for `\iow_newline:` to work, and that `\errorcontextlines` is `-1`, to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the `\errmessage`, which go in the way of tokens which could be inserted by the user. This is unavoidable.

```

9373   \group_begin:
9374   \char_set_lccode:nn { 38 } { 32 } % &
9375   \char_set_lccode:nn { 46 } { 32 } % .
9376   \char_set_lccode:nn { 123 } { 32 } % {
9377   \char_set_lccode:nn { 125 } { 32 } % }
9378   \char_set_catcode_active:N \&
9379   \tex_lowercase:D
9380   {
9381     \group_end:
9382     \cs_new_protected:Npn \__msg_interrupt:n #1
9383     {

```

```

9384 \iow_term:n { }
9385 \__kernel_iow_with:Nnn \tex_newlinechar:D { '\^^J }
9386 {
9387   \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9388   {
9389     \group_begin:
9390     \cs_set_protected:Npn &
9391     {
9392       \tex_errmessage:D
9393       {
9394         #1
9395         \use_none:n
9396         { ..... }
9397       }
9398     }
9399     \exp_after:wN
9400     \group_end:
9401     &
9402     }
9403   }
9404 }
9405 }

```

(End of definition for `__msg_interrupt:n`.)

49.5 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled.

```

9406 \int_gset:Nn \tex_errorcontextlines:D { -1 }

```

`\msg_fatal_text:n` A function for issuing messages: both the text and order could in principle vary. The module name may be empty for kernel messages, hence the slightly contorted code path for a space.

```

\msg_fatal_text:n
\msg_critical_text:n
\msg_error_text:n
\msg_warning_text:n
\msg_info_text:n
\__msg_text:nn
\__msg_text:n
9407 \cs_new:Npn \msg_fatal_text:n #1
9408 {
9409   Fatal ~
9410   \msg_error_text:n {#1}
9411 }
9412 \cs_new:Npn \msg_critical_text:n #1
9413 {
9414   Critical ~
9415   \msg_error_text:n {#1}
9416 }
9417 \cs_new:Npn \msg_error_text:n #1
9418 { \__msg_text:nn {#1} { Error } }
9419 \cs_new:Npn \msg_warning_text:n #1
9420 { \__msg_text:nn {#1} { Warning } }
9421 \cs_new:Npn \msg_info_text:n #1
9422 { \__msg_text:nn {#1} { Info } }
9423 \cs_new:Npn \__msg_text:nn #1#2
9424 {
9425   \exp_args:Nf \__msg_text:n { \msg_module_type:n {#1} }
9426   \exp_args:Nf \__msg_text:n { \msg_module_name:n {#1} }

```

```

9427     #2
9428   }
9429   \cs_new:Npn \__msg_text:n #1
9430   {
9431     \tl_if_blank:nF {#1}
9432     { #1 ~ }
9433   }

```

(End of definition for `\msg_fatal_text:n` and others. These functions are documented on page 82.)

`\g_msg_module_name_prop` For storing public module information: the kernel data is set up in advance.
`\g_msg_module_type_prop`

```

9434   \prop_new:N \g_msg_module_name_prop
9435   \prop_new:N \g_msg_module_type_prop
9436   \prop_gput:Nnn \g_msg_module_type_prop { LaTeX } { }

```

(End of definition for `\g_msg_module_name_prop` and `\g_msg_module_type_prop`. These variables are documented on page 81.)

`\msg_module_type:n` Contextual footer information, with the potential to give modules an alternative name.

```

9437   \cs_new:Npn \msg_module_type:n #1
9438   {
9439     \prop_if_in:NnTF \g_msg_module_type_prop {#1}
9440     { \prop_item:Nn \g_msg_module_type_prop {#1} }
9441     { Package }
9442   }

```

(End of definition for `\msg_module_type:n`. This function is documented on page 81.)

`\msg_module_name:n` Contextual footer information, with the potential to give modules an alternative name.
`\msg_see_documentation_text:n`

```

9443   \cs_new:Npn \msg_module_name:n #1
9444   {
9445     \prop_if_in:NnTF \g_msg_module_name_prop {#1}
9446     { \prop_item:Nn \g_msg_module_name_prop {#1} }
9447     {#1}
9448   }
9449   \cs_new:Npn \msg_see_documentation_text:n #1
9450   {
9451     See~the~ \msg_module_name:n {#1} ~
9452     documentation~for~further~information.
9453   }

```

(End of definition for `\msg_module_name:n` and `\msg_see_documentation_text:n`. These functions are documented on page 81.)

`__msg_class_new:nn`

```

9454   \group_begin:
9455   \cs_set_protected:Npn \__msg_class_new:nn #1#2
9456   {
9457     \prop_new:c { l__msg_redirect_ #1 _prop }
9458     \cs_new_protected:cpn { __msg_ #1 _code:n }
9459     ##1##2##3##4##5##6 {#2}
9460     \cs_new_protected:cpn { msg_ #1 :nnnnn } ##1##2##3##4##5##6
9461     {
9462       \use:e
9463     }

```

```

9464         \exp_not:n { \_msg_use:nnnnnnn {#1} {##1} {##2} }
9465         { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9466         { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9467     }
9468 }
9469 \cs_new_protected:cpe { msg_ #1 :nnnnn } ##1##2##3##4##5
9470 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9471 \cs_new_protected:cpe { msg_ #1 :nnnn } ##1##2##3##4
9472 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
9473 \cs_new_protected:cpe { msg_ #1 :nnn } ##1##2##3
9474 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
9475 \cs_new_protected:cpe { msg_ #1 :nn } ##1##2
9476 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
9477 \cs_generate_variant:cn { msg_ #1 :nnn }
9478 { nnV , nne , nnx }
9479 \cs_generate_variant:cn { msg_ #1 :nnnn }
9480 { nnVV , nnVn , nnnV , nnne , nnnx , nnee , nxxx }
9481 \cs_generate_variant:cn { msg_ #1 :nnnnn }
9482 { nnnee , nnnxx , nneee , nxxxx }
9483 \cs_generate_variant:cn { msg_ #1 :nnnnnn } { nneeee , nxxxxx }
9484 }

```

(End of definition for _msg_class_new:nn.)

`\msg_fatal:nnnnnn` For fatal errors, after the error message \TeX bails out. We force a bail out rather than using `\end` as this means it does not matter if we are in a context where normally the run cannot end.

```

\msg_fatal:nnnnnn
\msg_fatal:nneeee
\msg_fatal:nxxxxx
\msg_fatal:nnnnnn
\msg_fatal:nneee
\msg_fatal:nxxxx
\msg_fatal:nnnee
\msg_fatal:nnnxx
\msg_fatal:nnnn
\msg_fatal:nnVV
\msg_fatal:nnVn
\msg_fatal:nnnV
\msg_fatal:nnee
\msg_fatal:nnxx
\msg_fatal:nnnx
\msg_fatal:nnne
\msg_fatal:nnn

```

```

9485 \_msg_class_new:nn { fatal }
9486 {
9487     \_msg_interrupt:NnnnN
9488     \msg_fatal_text:n {#1} {#2}
9489     { {#3} {#4} {#5} {#6} }
9490     \c__msg_fatal_text_tl
9491     \_msg_fatal_exit:
9492 }
9493 \cs_new_protected:Npn \_msg_fatal_exit:
9494 {
9495     \tex_batchmode:D
9496     \tex_read:D -1 to \l__msg_internal_tl
9497 }

```

(End of definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page 84.)

`\msg_fatal:nnV` Not quite so bad: just end the current file.

```

\msg_fatal:nnnnnn
\msg_fatal:nne
\msg_fatal:nneeee
\msg_fatal:nnxx
\msg_fatal:nnxxxx
\msg_fatal:nn
\msg_fatal:nnnnnn
\msg_fatal_exit:
\msg_fatal:nneee
\msg_fatal:nnxxx
\msg_fatal:nnnee
\msg_fatal:nnnxx
\msg_fatal:nnnx
\msg_fatal:nnn

```

```

9498 \_msg_class_new:nn { critical }
9499 {
9500     \_msg_interrupt:NnnnN
9501     \msg_critical_text:n {#1} {#2}
9502     { {#3} {#4} {#5} {#6} }
9503     \c__msg_critical_text_tl
9504     \tex_endinput:D
9505 }

```

(End of definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 84.)

```

\msg_critical:nnVV
\msg_critical:nnVn
\msg_critical:nnnV
\msg_critical:nnee
\msg_critical:nnxx
\msg_critical:nnnx
\msg_critical:nnne
\msg_critical:nnn
\msg_critical:nnV
\msg_critical:nne
\msg_critical:nnx
\msg_critical:nn

```

```

\msg_error:nnnnnn
\msg_error:nneeee
\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nneee
\msg_error:nnxxx
\msg_error:nnnee
\msg_error:nnnxx
\msg_error:nnnn
\msg_error:nnVV
\msg_error:nnVn
\msg_error:nnnV
\msg_error:nnee
\msg_error:nnxx
\msg_error:nnx
\__msg_info_aux:NNnnnnn
\msg_error:nnne
\msg_warning:nnnnnn
\msg_error:nnne
\msg_warning:nnxxxx
\msg_error:nnne
\msg_warning:nnnnn
\msg_error:nnnx
\msg_warning:nneee
\msg_error:nn
\msg_warning:nnxxx
\msg_warning:nnnee
\msg_warning:nnnxx
\msg_warning:nnnn
\msg_warning:nnVV
\msg_warning:nnVn
\msg_warning:nnnV
\msg_warning:nnee
\msg_warning:nnxx
\msg_warning:nnnx
\msg_warning:nnne
\msg_warning:nnV
\msg_warning:nne
\msg_warning:nnx
\msg_warning:nn
\msg_note:nnnnnn
\msg_note:nneeee
\msg_note:nnxxxx
\msg_note:nnnnn
\msg_note:nneee
\msg_note:nnxxx
\msg_note:nnnee
\msg_note:nnnxx
\msg_note:nnnn
\msg_note:nnVV
\msg_note:nnVn
\msg_note:nnnV
\msg_note:nnee
\msg_note:nnxx
\msg_note:nnnx
\msg_note:nnne
\msg_note:nnn
\msg_note:nnV
\msg_note:nne
\msg_note:nnx
\msg_note:nn
\msg_info:nnnnnn
\msg_info:nneeee

```

For an error, the interrupt routine is called. We check if there is a “more text” by comparing that control sequence with a permanently empty text. We have to undefine the bootstrap versions here.

```

9506 \cs_undefine:N \msg_error:nnee
9507 \cs_undefine:N \msg_error:nne
9508 \cs_undefine:N \msg_error:nn
9509 \__msg_class_new:nn { error }
9510 {
9511   \__msg_interrupt:NnnnN
9512   \msg_error_text:n {#1} {#2}
9513   { {#3} {#4} {#5} {#6} }
9514   \c_empty_tl
9515 }

```

(End of definition for \msg_error:nnnnnn and others. These functions are documented on page 84.)

Warnings and information messages have no decoration. Warnings are printed to the terminal while information can either go to the log or both log and terminal.

```

9516 \cs_new_protected:Npn \__msg_info_aux:NNnnnnnn #1#2#3#4#5#6#7#8
9517 {
9518   \str_set:Ne \l__msg_text_str { #2 {#3} }
9519   \str_set:Ne \l__msg_name_str { \msg_module_name:n {#3} }
9520   #1 { }
9521   \iow_wrap:nenN
9522   {
9523     \l__msg_text_str : ~
9524     \use:c { \c__msg_text_prefix_tl #3 / #4 } {#5} {#6} {#7} {#8}
9525   }
9526   {
9527     ( \l__msg_name_str )
9528     \prg_replicate:nn
9529     {
9530       \str_count:N \l__msg_text_str
9531       - \str_count:N \l__msg_name_str
9532     }
9533     { ~ }
9534   }
9535   { } #1
9536   #1 { }
9537 }
9538 \__msg_class_new:nn { warning }
9539 {
9540   \__msg_info_aux:NNnnnnnn \iow_term:n \msg_warning_text:n
9541   {#1} {#2} {#3} {#4} {#5} {#6}
9542 }
9543 \__msg_class_new:nn { note }
9544 {
9545   \__msg_info_aux:NNnnnnnn \iow_term:n \msg_info_text:n
9546   {#1} {#2} {#3} {#4} {#5} {#6}
9547 }
9548 \__msg_class_new:nn { info }
9549 {
9550   \__msg_info_aux:NNnnnnnn \iow_log:n \msg_info_text:n
9551   {#1} {#2} {#3} {#4} {#5} {#6}

```

```
9552     }
```

(End of definition for `_msg_info_aux:NNnnnnnn` and others. These functions are documented on page 85.)

```
\msg_term:nnnnnn "Log" data is very similar to information, but with no extras added. "Term" is used
\msg_term:nneeee for communicating with the user through the terminal, like diagnostic messages, and
\msg_term:nnxxxx debugging. This is similar to "log" messages, but uses the terminal output.
\msg_term:nnnnn 9553   \_msg_class_new:nn { log }
\msg_term:nneee 9554   {
\msg_term:nnxxx 9555     \iow_wrap:nnnN
\msg_term:nnnee 9556     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_term:nnnxx 9557     { } { } \iow_log:n
\msg_term:nnnn 9558   }
\msg_term:nnVV 9559   \_msg_class_new:nn { term }
\msg_term:nnVn 9560   {
\msg_term:nnnV 9561     \iow_wrap:nnnN
\msg_term:nnee 9562     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_term:nnxx 9563     { } { } \iow_term:n
\msg_term:nnnx 9564   }
\msg_term:nnne (End of definition for \msg_term:nnnnnn and others. These functions are documented on page 86.)
```

```
\msg_term:nnn  The none message type is needed so that input can be gobbled.
\msg_none:nnnnnn 9565   \_msg_class_new:nn { none } { }
```

(End of definition for `\msg_none:nnnnnn` and others. These functions are documented on page 86.)

```
\msg_term:nnn  The show message type is used for \seq_show:N and similar complicated data structures.
\msg_none:nnnnnn Wrap the given text with a trailing dot (important later) then pass it to \_msg_show:n.
\msg_term:nnV If there is \>~ (or if the whole thing starts with >~) we split there, print the first part
\msg_none:nneeee and show the second part using \showtokens (the \exp_after:wN ensure a nice display).
\msg_term:nnee Note that this primitive adds a leading >~ and trailing dot. That is why we included a
\msg_none:nnxxxx trailing dot before wrapping and removed it afterwards. If there is no \>~ do the same
\msg_term:nnnxx but with an empty second part which adds a spurious but inevitable >~.
\msg_none:nnnnn 9566   \_msg_class_new:nn { show }
\msg_term:nn 9567   {
\msg_none:nneee \msg_ggshog:nnnnnn 9568     \iow_wrap:nnnN
\msg_none:nnxxx \msg_ggshog:nneeee 9569     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_ggshog:nnee \msg_ggshog:nnxxxx 9570     { } { } \_msg_show:n
\msg_ggshog:nnxxx \msg_ggshog:nnnnn 9571   }
\msg_ggshog:nnee \msg_ggshog:nnnn 9572   \cs_new_protected:Npn \_msg_show:n #1
\msg_ggshog:nnV \msg_ggshog:nnee 9573   {
\msg_ggshog:nnVn \msg_ggshog:nnxx 9574     \tl_if_in:nnTF { ^^J #1 } { ^^J > ~ }
\msg_ggshog:nnnee \msg_ggshog:nnV 9575     {
\msg_ggshog:nnV \msg_ggshog:nnxx 9576       \tl_if_in:nnTF { #1 \s__msg_mark } { . \s__msg_mark }
\msg_ggshog:nnV \msg_ggshog:nne 9577       { \_msg_show_dot:w } { \_msg_show:w }
\msg_ggshog:nnV \msg_ggshog:nne 9578       ^^J #1 \s__msg_stop
\msg_ggshog:nnV \msg_ggshog:nnx 9579     }
\msg_ggshog:nnV \msg_ggshog:nn 9580     { \_msg_show:nn { ? #1 } { } }
\_msg_show:n 9581   }
\_msg_show:w 9582   \cs_new:Npn \_msg_show_dot:w #1 ^^J > ~ #2 . \s__msg_stop
\_msg_show_dot:w 9583   { \_msg_show:nn {#1} {#2} }
\_msg_show:nn 9584   \cs_new:Npn \_msg_show:w #1 ^^J > ~ #2 \s__msg_stop
```

```

9585     { \_msg_show:nn {#1} {#2} }
9586 \cs_new_protected:Npn \_msg_show:nn #1#2
9587     {
9588     \tl_if_empty:nF {#1}
9589     { \exp_args:No \iow_term:n { \use_none:n #1 } }
9590     \tl_set:Nn \l_msg_internal_tl {#2}
9591     \_kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
9592     {
9593     \_kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9594     {
9595     \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
9596     { \exp_after:wN \l_msg_internal_tl }
9597     }
9598     }
9599     }

```

(End of definition for `\msg_show:nnnnn` and others. These functions are documented on page 87.)

End the group to eliminate `_msg_class_new:nn`.

```

9600 \group_end:

```

`\msg_show_item:n` Each item in the variable is formatted using one of the following functions. We cannot use `\l` and so on because these short-hands cannot be used inside the arguments of messages, only when defining the messages. We need to use `^^J` here directly as `l3file` is not yet loaded.

```

\msg_show_item_unbraced:n
\msg_show_item:nn
\msg_show_item_unbraced:nn
9601 \cs_new:Npe \msg_show_item:n #1
9602 { ^^J > ~ \c_space_tl \exp_not:N \tl_to_str:n { {#1} } }
9603 \cs_new:Npe \msg_show_item_unbraced:n #1
9604 { ^^J > ~ \c_space_tl \exp_not:N \tl_to_str:n {#1} }
9605 \cs_new:Npe \msg_show_item:nn #1#2
9606 {
9607     ^^J > \use:nn { ~ } { ~ }
9608     \exp_not:N \tl_to_str:n { {#1} }
9609     \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
9610     \exp_not:N \tl_to_str:n { {#2} }
9611 }
9612 \cs_new:Npe \msg_show_item_unbraced:nn #1#2
9613 {
9614     ^^J > \use:nn { ~ } { ~ }
9615     \exp_not:N \tl_to_str:n {#1}
9616     \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
9617     \exp_not:N \tl_to_str:n {#2}
9618 }

```

(End of definition for `\msg_show_item:n` and others. These functions are documented on page 87.)

`_msg_class_chk_exist:nT` Checking that a message class exists. We build this from `\cs_if_free:cTF` rather than `\cs_if_exist:cTF` because that avoids reading the second argument earlier than necessary.

```

9619 \cs_new:Npn \_msg_class_chk_exist:nT #1
9620 {
9621     \cs_if_free:cTF { \_msg_ #1 _code:nnnnn }
9622     { \msg_error:nnn { msg } { class-unknown } {#1} }
9623 }

```


(End of definition for `__msg_class_chk_exist:nT`.)

`\l__msg_class_tl` Support variables needed for the redirection system.
`\l__msg_current_class_tl` 9624 `\tl_new:N \l__msg_class_tl`
9625 `\tl_new:N \l__msg_current_class_tl`

(End of definition for `\l__msg_class_tl` and `\l__msg_current_class_tl`.)

`\l__msg_redirect_prop` For redirection of individually-named messages
9626 `\prop_new:N \l__msg_redirect_prop`

(End of definition for `\l__msg_redirect_prop`.)

`\l__msg_hierarchy_seq` During redirection, split the message name into a sequence: `{/module/submodule}`, `{/module}`, and `{}`.

9627 `\seq_new:N \l__msg_hierarchy_seq`

(End of definition for `\l__msg_hierarchy_seq`.)

`\l__msg_class_loop_seq` Classes encountered when following redirections to check for loops.

9628 `\seq_new:N \l__msg_class_loop_seq`

(End of definition for `\l__msg_class_loop_seq`.)

`__msg_use:nnnnnnn` Actually using a message is a multi-step process. First, some safety checks on the message
`__msg_use_redirect_name:n` and class requested. The code and arguments are then stored to avoid passing them
`__msg_use_hierarchy:nwN` around. The assignment to `__msg_use_code:` is similar to `\tl_set:Nn`. The message
`__msg_use_redirect_module:n` is eventually produced with whatever `\l__msg_class_tl` is when `__msg_use_code:`
`__msg_use_code:` is called. Here is also a good place to suppress tracing output if the trace package is loaded
since all (non-expandable) messages go through this auxiliary.

```
9629 \cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
9630   {
9631     \cs_if_exist_use:N \conditionally@traceoff
9632     \msg_if_exist:nnTF {#2} {#3}
9633     {
9634       \__msg_class_chk_exist:nT {#1}
9635       {
9636         \tl_set:Nn \l__msg_current_class_tl {#1}
9637         \cs_set_protected:Npe \__msg_use_code:
9638         {
9639           \exp_not:n
9640           {
9641             \use:c { __msg_ \l__msg_class_tl _code:nnnnnn }
9642             {#2} {#3} {#4} {#5} {#6} {#7}
9643           }
9644         }
9645       }
9646       \__msg_use_redirect_name:n { #2 / #3 }
9647     }
9648     { \msg_error:nnnn { msg } { unknown } {#2} {#3} }
9649     \cs_if_exist_use:N \conditionally@traceon
9650   }
9651 \cs_new_protected:Npn \__msg_use_code: { }
```

The first check is for a individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into $\langle module \rangle$, $\langle submodule \rangle$ and $\langle message \rangle$ (with an arbitrary number of slashes), and store $\{/module/submodule\}$, $\{/module\}$ and $\{\}$ into $\backslash l_msg_hierarchy_seq$. We then map through this sequence, applying the most specific redirection.

```

9652 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
9653 {
9654   \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
9655   { \__msg_use_code: }
9656   {
9657     \seq_clear:N \l__msg_hierarchy_seq
9658     \__msg_use_hierarchy:nwN { }
9659     #1 \s__msg_mark \__msg_use_hierarchy:nwN
9660     / \s__msg_mark \__msg_use_none_delimit_by_s_stop:w
9661     \s__msg_stop
9662     \__msg_use_redirect_module:n { }
9663   }
9664 }
9665 \cs_new_protected:Npn \__msg_use_hierarchy:nwN #1#2 / #3 \s__msg_mark #4
9666 {
9667   \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
9668   #4 { #1 / #2 } #3 \s__msg_mark #4
9669 }

```

At this point, the items of $\backslash l_msg_hierarchy_seq$ are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of $\backslash _msg_use_redirect_module:n$ are not attempted. This argument is empty for a class redirection, $/module$ for a module redirection, *etc.* Loop through the sequence to find the most specific redirection, with module **##1**. The loop is interrupted after testing for a redirection for **##1** equal to the argument **#1** (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as **##1**.

```

9670 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
9671 {
9672   \seq_map_inline:Nn \l__msg_hierarchy_seq
9673   {
9674     \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
9675     {##1} \l__msg_class_tl
9676     {
9677       \seq_map_break:n
9678       {
9679         \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
9680         { \__msg_use_code: }
9681         {
9682           \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
9683           \__msg_use_redirect_module:n {##1}
9684         }
9685       }
9686     }
9687   {
9688     \str_if_eq:nnT {##1} {#1}

```

```

9689         {
9690         \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
9691         \seq_map_break:n { \__msg_use_code: }
9692         }
9693     }
9694 }
9695 }

```

(End of definition for `__msg_use:nnnnnn` and others.)

`\msg_redirect_name:nnn` Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

9696 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
9697 {
9698   \tl_if_empty:nTF {#3}
9699   { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
9700   {
9701     \__msg_class_chk_exist:nT {#3}
9702     { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
9703   }
9704 }

```

(End of definition for `\msg_redirect_name:nnn`. This function is documented on page 89.)

`\msg_redirect_class:nn` If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in `\l__msg_current_class_tl`.

`\msg_redirect_module:nnn`
`__msg_redirect:nnn`
`__msg_redirect_loop_chk:nnn`
`__msg_redirect_loop_list:n`

```

9705 \cs_new_protected:Npn \msg_redirect_class:nn
9706 { \__msg_redirect:nnn { } }
9707 \cs_new_protected:Npn \msg_redirect_module:nnn #1
9708 { \__msg_redirect:nnn { / #1 } }
9709 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
9710 {
9711   \__msg_class_chk_exist:nT {#2}
9712   {
9713     \tl_if_empty:nTF {#3}
9714     { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
9715     {
9716       \__msg_class_chk_exist:nT {#3}
9717       {
9718         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
9719         \tl_set:Nn \l__msg_current_class_tl {#2}
9720         \seq_clear:N \l__msg_class_loop_seq
9721         \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
9722       }
9723     }
9724   }
9725 }

```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, #1, and not submodules. After some initialization above, follow redirections with `\l__msg_class_tl`, and keep track in `\l__msg_class_loop_seq` of the various classes encountered. A redirection from a class to

itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from #2, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```

9726 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
9727 {
9728   \seq_put_right:Nn \l__msg_class_loop_seq {#1}
9729   \prop_get:cnNT { l__msg_redirect_ #1_prop } {#3} \l__msg_class_tl
9730   {
9731     \str_if_eq:VnF \l__msg_class_tl {#1}
9732     {
9733       \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
9734       {
9735         \prop_put:cnn { l__msg_redirect_ #2_prop } {#3} {#2}
9736         \msg_warning:nneeee
9737         { msg } { redirect-loop }
9738         { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9739         { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
9740         {#3}
9741         {
9742           \seq_map_function:NN \l__msg_class_loop_seq
9743           \__msg_redirect_loop_list:n
9744           { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9745         }
9746       }
9747       { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
9748     }
9749   }
9750 }
9751 \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
9752 \cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }

```

(End of definition for `\msg_redirect_class:nn` and others. These functions are documented on page 89.)

49.6 Kernel-specific functions

`__kernel_msg_show_eval:Nn` A short-hand used for `\int_show:n` and similar functions that passes to `\tl_show:n` the result of applying #1 (a function such as `\int_eval:n`) to the expression #2. The use of f-expansion ensures that #1 is expanded in the scope in which the show command is called, rather than in the group created by `\iow_wrap:nnnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgrouptype`. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```

9753 \cs_new_protected:Npn \__kernel_msg_show_eval:Nn #1#2
9754 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
9755 \cs_new_protected:Npn \__kernel_msg_log_eval:Nn #1#2
9756 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
9757 \cs_new_protected:Npn \__msg_show_eval:nnN #1#2#3 { #3 { #2 = #1 } }

```

(End of definition for `_kernel_msg_show_eval:Nn`, `_kernel_msg_log_eval:Nn`, and `_msg_show_eval:nnN`.)

These are all retained purely for older xparse support.

```
\_kernel_msg_new:nnnn
\_kernel_msg_new:nnn 9758 \cs_new_protected:Npn \_kernel_msg_new:nnnn #1
                      9759   { \msg_new:nnnn { LaTeX / #1 } }
                      9760 \cs_new_protected:Npn \_kernel_msg_new:nnn #1
                      9761   { \msg_new:nnn { LaTeX / #1 } }
```

(End of definition for `_kernel_msg_new:nnnn` and `_kernel_msg_new:nnn`.)

```
\_kernel_msg_info:nxxx
\_kernel_msg_warning:nxx 9762 \cs_new_protected:Npn \_kernel_msg_info:nxxx #1
\_kernel_msg_warning:nxxx 9763   { \msg_info:nnee { LaTeX / #1 } }
\_kernel_msg_error:nxx 9764 \cs_new_protected:Npn \_kernel_msg_warning:nxx #1
\_kernel_msg_error:nxxx 9765   { \msg_warning:nne { LaTeX / #1 } }
\_kernel_msg_error:nxxxx 9766 \cs_new_protected:Npn \_kernel_msg_warning:nxxx #1
9767   { \msg_warning:nnee { LaTeX / #1 } }
9768 \cs_new_protected:Npn \_kernel_msg_error:nxx #1
9769   { \msg_error:nne { LaTeX / #1 } }
9770 \cs_new_protected:Npn \_kernel_msg_error:nxxx #1
9771   { \msg_error:nnee { LaTeX / #1 } }
9772 \cs_new_protected:Npn \_kernel_msg_error:nxxxx #1
9773   { \msg_error:nnee { LaTeX / #1 } }
```

(End of definition for `_kernel_msg_info:nxxx` and others.)

```
\_kernel_msg_expandable_error:nnn
\_kernel_msg_expandable_error:nmf 9774 \cs_new:Npn \_kernel_msg_expandable_error:nnn #1
\_kernel_msg_expandable_error:nnff 9775   { \msg_expandable_error:nnn { LaTeX / #1 } }
9776 \cs_new:Npn \_kernel_msg_expandable_error:nmf #1
9777   { \msg_expandable_error:nmf { LaTeX / #1 } }
9778 \cs_new:Npn \_kernel_msg_expandable_error:nnff #1
9779   { \msg_expandable_error:nnff { LaTeX / #1 } }
```

(End of definition for `_kernel_msg_expandable_error:nnn` and `_kernel_msg_expandable_error:nnff`.)

49.7 Internal messages

Error messages needed to actually implement the message system itself.

```
9780 \msg_new:nnnn { msg } { already-defined }
9781   { Message~'#2'~for~module~'#1'~already-defined. }
9782   {
9783     \c_msg_coding_error_text_tl
9784     LaTeX~was~asked~to~define~a~new~message~called~'#2'~\
9785     by~the~module~'#1':~this~message~already~exists.
9786     \c_msg_return_text_tl
9787   }
9788 \msg_new:nnnn { msg } { unknown }
9789   { Unknown~message~'#2'~for~module~'#1'. }
9790   {
9791     \c_msg_coding_error_text_tl
```

```

9792 LaTeX-was-asked-to-display-a-message-called-#2'\
9793 by-the-module-#1':-this-message-does-not-exist.
9794 \c_msg_return_text_tl
9795 }
9796 \msg_new:nnnn { msg } { class-unknown }
9797 { Unknown-message-class-#1'. }
9798 {
9799 LaTeX-has-been-asked-to-redirect-messages-to-a-class-#1':\
9800 this-was-never-defined.
9801 \c_msg_return_text_tl
9802 }
9803 \msg_new:nnnn { msg } { redirect-loop }
9804 {
9805 Message-redirect-loop-caused-by- {#1} ~=>~ {#2}
9806 \tl_if_empty:nF {#3} { ~for-module~ \use_none:n #3 ' } .
9807 }
9808 {
9809 Adding-the-message-redirect-loop- {#1} ~=>~ {#2}
9810 \tl_if_empty:nF {#3} { ~for-the-module~ \use_none:n #3 ' } ~
9811 created-an-infinite-loop\\
9812 \iow_indent:n { #4 \\ \\ }
9813 }

```

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.

```

9814 \msg_new:nnnn { kernel } { bad-number-of-arguments }
9815 { Function-#1'-cannot-be-defined-with-#2-arguments. }
9816 {
9817 \c_msg_coding_error_text_tl
9818 LaTeX-has-been-asked-to-define-a-function-#1'-with-
9819 #2-arguments.-
9820 TeX-allows-between-0-and-9-arguments-for-a-single-function.
9821 }
9822 \msg_new:nnnn { kernel } { command-already-defined }
9823 { Control-sequence-#1-already-defined. }
9824 {
9825 \c_msg_coding_error_text_tl
9826 LaTeX-has-been-asked-to-create-a-new-control-sequence-#1'~
9827 but-this-name-has-already-been-used-elsewhere. \\ \\
9828 The-current-meaning-is:\
9829 \ \ #2
9830 }
9831 \msg_new:nnnn { kernel } { command-not-defined }
9832 { Control-sequence-#1-undefined. }
9833 {
9834 \c_msg_coding_error_text_tl
9835 LaTeX-has-been-asked-to-use-a-control-sequence-#1':\
9836 this-has-not-been-defined-yet.
9837 }
9838 \msg_new:nnnn { kernel } { empty-search-pattern }
9839 { Empty-search-pattern. }
9840 {
9841 \c_msg_coding_error_text_tl
9842 LaTeX-has-been-asked-to-replace-an-empty-pattern-by-#1':-that-
9843 would-lead-to-an-infinite-loop!
9844 }

```

```

9845 \cs_if_exist:NF \tex_elapsedtime:D
9846 {
9847   \msg_new:nnnn { kernel } { no-elapsed-time }
9848   { No-clock-detected-for-#1. }
9849   { The-current-engine-provides-no-way-to-access-the-system-time. }
9850 }
9851 \msg_new:nnnn { kernel } { non-base-function }
9852 { Function-#1'-is-not-a-base-function }
9853 {
9854   \c__msg_coding_error_text_tl
9855   Functions-defined-through-\iow_char:N\cs_new:Nn-must-have-
9856   a-signature-consisting-of-only-normal-arguments-'N'-and-'n'.~
9857   The-signature-#2'-of-#1'-contains-other-arguments-#3'.~
9858   To-define-variants-use-\iow_char:N\cs_generate_variant:Nn~
9859   and-to-define-other-functions-use-\iow_char:N\cs_new:Npn.
9860 }
9861 \msg_new:nnnn { kernel } { missing-colon }
9862 { Function-#1'-contains-no-':'. }
9863 {
9864   \c__msg_coding_error_text_tl
9865   Code-level-functions-must-contain-':'-to-separate-the-
9866   argument-specification-from-the-function-name.~This-is-
9867   needed-when-defining-conditionals-or-variants,~or-when-building-a-
9868   parameter-text-from-the-number-of-arguments-of-the-function.
9869 }
9870 \msg_new:nnnn { kernel } { overflow }
9871 { Integers-larger-than-2^{30}-1-cannot-be-stored-in-arrays. }
9872 {
9873   An-attempt-was-made-to-store-#3~
9874   \tl_if_empty:nF {#2} { at-position-#2- } in-the-array-#1'.~
9875   The-largest-allowed-value-#4-will-be-used-instead.
9876 }
9877 \msg_new:nnnn { kernel } { out-of-bounds }
9878 { Access-to-an-entry-beyond-an-array's-bounds. }
9879 {
9880   An-attempt-was-made-to-access-or-store-data-at-position-#2-of-the-
9881   array-#1',~but-this-array-has-entries-at-positions-from-1-to-#3.
9882 }
9883 \msg_new:nnnn { kernel } { protected-predicate }
9884 { Predicate-#1'-must-be-expandable. }
9885 {
9886   \c__msg_coding_error_text_tl
9887   LaTeX-has-been-asked-to-define-#1'-as-a-protected-predicate.~
9888   Only-expandable-tests-can-have-a-predicate-version.
9889 }
9890 \msg_new:nnn { kernel } { randint-backward-range }
9891 { Wrong-order-of-bounds-in-\iow_char:N\int_rand:nn{#1}{#2}. }
9892 \msg_new:nnnn { kernel } { conditional-form-unknown }
9893 { Conditional-form-#1'-for-function-#2'-unknown. }
9894 {
9895   \c__msg_coding_error_text_tl
9896   LaTeX-has-been-asked-to-define-the-conditional-form-#1'-of-
9897   the-function-#2',~but-only-'TF',~'T',~'F',~and-'p'-forms-exist.
9898 }

```

```

9899 \msg_new:nnnn { kernel } { variant-too-long }
9900 { Variant-form-#1'-longer-than-base-signature-of-#2'. }
9901 {
9902   \c_msg_coding_error_text_tl
9903   LaTeX-has-been-asked-to-create-a-variant-of-the-function-#2'~
9904   with-a-signature-starting-with-#1',-but-that-is-longer-than~
9905   the-signature-(part-after-the-colon)-of-#2'.
9906 }
9907 \msg_new:nnnn { kernel } { invalid-variant }
9908 { Variant-form-#1'-invalid-for-base-form-#2'. }
9909 {
9910   \c_msg_coding_error_text_tl
9911   LaTeX-has-been-asked-to-create-a-variant-of-the-function-#2'~
9912   with-a-signature-starting-with-#1',-but-cannot-change-an-argument~
9913   from-type-#3'-to-type-#4'.
9914 }
9915 \msg_new:nnnn { kernel } { invalid-exp-args }
9916 { Invalid-variant-specifier-#1'-in-#2'. }
9917 {
9918   \c_msg_coding_error_text_tl
9919   LaTeX-has-been-asked-to-create-an-\iow_char:N\exp_args:N...~
9920   function-with-signature-N#2'~but-#1'-is-not-a-valid-argument~
9921   specifier.
9922 }
9923 \msg_new:nnn { kernel } { deprecated-variant }
9924 {
9925   Variant-form-#1'-deprecated-for-base-form-#2'.~
9926   One-should-not-change-an-argument-from-type-#3'-to-type-#4'
9927   \str_case:nnF {#3}
9928   {
9929     { n } { :-use-a-\token_if_eq_charcode:NNTF #4 c v V'-variant? }
9930     { N } { :-base-form-only-accepts-a-single-token-argument. }
9931     {#4} { :-base-form-is-already-a-variant. }
9932   } { . }
9933 }
9934 \msg_new:nnn { char } { active }
9935 { Cannot-generate-active-chars. }
9936 \msg_new:nnn { char } { invalid-catcode }
9937 { Invalid-catcode-for-char-generation. }
9938 \msg_new:nnn { char } { null-space }
9939 { Cannot-generate-null-char-as-a-space. }
9940 \msg_new:nnn { char } { out-of-range }
9941 { Charcode-requested-out-of-engine-range. }
9942 \msg_new:nnn { dim } { zero-unit }
9943 { Zero-unit-in-conversion. }
9944 \msg_new:nnnn { kernel } { quote-in-shell }
9945 { Quotes-in-shell-command-#1'. }
9946 { Shell-commands-cannot-contain-quotes-("). }
9947 \msg_new:nnnn { keys } { no-property }
9948 { No-property-given-in-definition-of-key-#1'. }
9949 {
9950   \c_msg_coding_error_text_tl
9951   Inside-\keys_define:nn each-key-name~
9952   needs-a-property: \ \ \

```



```

9953 \iow_indent:n { #1 .<property> } \ \ \
9954 LaTeX-did-not-find-a'.'~to-indicate-the-start-of-a-property.
9955 }
9956 \msg_new:nnnn { keys } { property-boolean-values-only }
9957 { The-property-#1'-accepts-boolean-values-only. }
9958 {
9959 \c__msg_coding_error_text_tl
9960 The-property-#1'-only-accepts-the-values-'true'~and-'false'.
9961 }
9962 \msg_new:nnnn { keys } { property-requires-value }
9963 { The-property-#1'-requires-a-value. }
9964 {
9965 \c__msg_coding_error_text_tl
9966 LaTeX-was-asked-to-set-property-#1'-for-key-#2'.\ \
9967 No-value-was-given-for-the-property,~and-one-is-required.
9968 }
9969 \msg_new:nnnn { keys } { property-unknown }
9970 { The-key-property-#1'-is-unknown. }
9971 {
9972 \c__msg_coding_error_text_tl
9973 LaTeX-has-been-asked-to-set-the-property-#1'-for-key-#2':~
9974 this-property-is-not-defined.
9975 }
9976 \msg_new:nnnn { quark } { invalid-function }
9977 { Quark-test-function-#1'-is-invalid. }
9978 {
9979 \c__msg_coding_error_text_tl
9980 LaTeX-has-been-asked-to-create-quark-test-function-#1'~
9981 \tl_if_empty:nTF {#2}
9982 { but-that-name~ }
9983 { with-signature-#2',~but-that-signature~ }
9984 is-not-valid.
9985 }
9986 \__kernel_msg_new:nnn { quark } { invalid }
9987 { Invalid-quark-variable-#1'. }
9988 \msg_new:nnnn { scanmark } { already-defined }
9989 { Scan-mark-#1-already-defined. }
9990 {
9991 \c__msg_coding_error_text_tl
9992 LaTeX-has-been-asked-to-create-a-new-scan-mark-#1'~
9993 but-this-name-has-already-been-used-for-a-scan-mark.
9994 }
9995 \msg_new:nnnn { seq } { item-too-large }
9996 { Sequence-#1'-does-not-have-an-item-#3 }
9997 {
9998 An-attempt-was-made-to-push-or-pop-the-item-at-position-#3~
9999 of-#1',~but-this~
10000 \int_compare:nTF { #3 = 0 }
10001 { position-does-not-exist. }
10002 { sequence-only-has-#2-item \int_compare:nF { #2 = 1 } {s}. }
10003 }
10004 \msg_new:nnnn { seq } { shuffle-too-large }
10005 { The-sequence-#1-is-too-long-to-be-shuffled-by-TeX. }
10006 {

```

```

10007 TeX-has~ \int_eval:n { \c_max_register_int + 1 } ~
10008 toks-registers:~this-only-allows-to-shuffle-up-to~
10009 \int_use:N \c_max_register_int \ items.~
10010 The-list-will-not-be-shuffled.
10011 }
10012 \msg_new:nnnn { kernel } { variable-not-defined }
10013 { Variable-#1-undefined. }
10014 {
10015   \c_msg_coding_error_text_tl
10016   LaTeX-has-been-asked-to-show-a-variable-#1,~but-this-has-not~
10017   been-defined-yet.
10018 }
10019 \msg_new:nnnn { kernel } { bad-type }
10020 { Variable-'#1'-is-not-a-valid-#3. }
10021 {
10022   \c_msg_coding_error_text_tl
10023   The-variable-'#1'-with-\tl_if_empty:nTF {#4} {meaning} {value}\\\\
10024   \iow_indent:n {#2}\\\\
10025   should-be-a-#3-variable,~but~
10026   \tl_if_empty:nTF {#4}
10027     { it-is-not \str_if_eq:nnF {#3} { bool } { ~a-short-macro } . }
10028     {
10029       it~does~not~have~the~correct~
10030       \str_if_eq:nnTF {#2} {#4}
10031       { category-codes. }
10032       { internal-structure:\\\\\iow_indent:n {#4} }
10033     }
10034 }
10035 \msg_new:nnnn { prop } { bad-link }
10036 { Variable-'#1'-is-not-a-valid-(linked)-prop. }
10037 {
10038   \c_msg_coding_error_text_tl
10039   The-variable-'#1'-has-an-incorrect-internal-structure.~
10040   Its-internal-entry-'#2'-points-to-'#3',~whose-name-is-not-of-the~
10041   form-'#4-<key>'.
10042 }
10043 \msg_new:nnnn { clist } { non-clist }
10044 { Variable-'#1'-is-not-a-valid-clist. }
10045 {
10046   \c_msg_coding_error_text_tl
10047   The-variable-'#1'-with-value\\\\
10048   \iow_indent:n {#2}\\\\
10049   should-be-a-clist-variable,~but-it~includes~empty-or-blank-items~
10050   without-braces.
10051 }
10052 \msg_new:nnnn { prop } { misused }
10053 { A-property-list-was-misused. }
10054 {
10055   \c_msg_coding_error_text_tl
10056   A-property-list-variable-was-used-without-an-accessor-function.~
10057   It~
10058   \tl_if_empty:nTF {#1}
10059     { is-empty. }
10060     { contains-the-key-value-pairs \use_none:n #1 . }

```

```

10061 }
10062 \msg_new:nnnn { prop } { inner-make }
10063 { '#1'~ cannot~ be~ used~ in~ a~ group. }
10064 {
10065   \c_msg_coding_error_text_tl
10066   The~ command~ '#1'~ was~ applied~ to~ the~ property~ list~
10067   variable~ '#2', but~ the~ storage~ type~ can~ only~ be~ changed~
10068   at~ the~ outermost~ group~ level.
10069 }

```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```

10070 \msg_new:nnn { kernel } { bad-exp-end-f }
10071 { Misused~\exp_end_continue_f:w or~:nw }
10072 \msg_new:nnn { kernel } { bad-variable }
10073 { Erroneous~variable~#1 used! }
10074 \msg_new:nnn { seq } { misused }
10075 { A~sequence~was~misused. }
10076 \msg_new:nnn { prg } { negative-replication }
10077 { Negative~argument~for~\iow_char:N\prg_replicate:nn. }
10078 \msg_new:nnn { prop } { prop-keyval }
10079 { Missing~'=~in~'#1'~(in~'..._keyval:Nn') }
10080 \msg_new:nnn { kernel } { unknown-comparison }
10081 { Relation~'#1'~not~among~=<,>==,!=,<=,>=. }
10082 \msg_new:nnn { kernel } { zero-step }
10083 { Zero~step~size~for~function~#1. }

```

Messages used by the "show" functions.

```

10084 \msg_new:nnn { clist } { show }
10085 {
10086   The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
10087   \tl_if_empty:nTF {#2}
10088   { is~empty \>>~ . }
10089   { contains~the~items~(without~outer~braces): #2 . }
10090 }
10091 \msg_new:nnn { intarray } { show }
10092 { The~integer~array~#1~contains~#2~items: \>> #3 . }
10093 \msg_new:nnn { prop } { show }
10094 {
10095   The~ \str_if_eq:nnF {#3} { flat } { #3~ }
10096   property~list~#1~
10097   \tl_if_empty:nTF {#2}
10098   { is~empty \>>~ . }
10099   { contains~the~pairs~(without~outer~braces): #2 . }
10100 }
10101 \msg_new:nnn { seq } { show }
10102 {
10103   The~sequence~#1~
10104   \tl_if_empty:nTF {#2}
10105   { is~empty \>>~ . }
10106   { contains~the~items~(without~outer~braces): #2 . }
10107 }
10108 \msg_new:nnn { kernel } { show-streams }
10109 {
10110   \tl_if_empty:nTF {#2} { No~ } { The~following~

```

```

10111 \str_case:nn {#1}
10112 {
10113   { ior } { input ~ }
10114   { iow } { output ~ }
10115 }
10116 streams-are~
10117 \tl_if_empty:nTF {#2} { open } { in-use: #2 . }
10118 }
System layer messages
10119 \msg_new:nnnn { sys } { backend-set }
10120 { Backend-configuration-already-set. }
10121 {
10122   Run-time-backend-selection-may-only-be-carried-out-once-during-a-run.~
10123   This-second-attempt-to-set-them-will-be-ignored.
10124 }
10125 \msg_new:nnnn { sys } { load-debug-in-preamble }
10126 { Load-debug-support-in-the-preamble. }
10127 {
10128   Debugging-requires-support-loaded-in-the-preamble: \\
10129   Use-\sys_load_debug:~before-\begin{document}.
10130 }
10131 \msg_new:nnnn { sys } { wrong-backend }
10132 { Backend-request-inconsistent-with-engine:~using~'#2'~backend. }
10133 {
10134   You-have-requested-backend~'#1',~but-this-is-not-suitable-for-use-with-the-
10135   active-engine.~LaTeX-will-use-the~'#2'~backend-instead.
10136 }

```

49.8 Expandable errors

`_msg_expandable_error:nn` In expansion only context, we cannot use the normal means of reporting errors. Instead, we rely on a low-level \TeX error caused by expanding a macro `\???` with parameter text “?” (this could be any token) which we used followed by something else (here, a space). This shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \???
! mypkg Error: The error message.

```

In other words, \TeX is processing the argument of `\use:n`, which is `\??? <space> ! <error type> : <error message>`.

```

10137 \cs_set_protected:Npn \_msg_tmp:w #1
10138 {
10139   \cs_new:Npn #1 ? { }
10140   \cs_new:Npn \_msg_expandable_error:nn ##1##2
10141   {
10142     \exp_after:wN \exp_after:wN
10143     \exp_after:wN \_msg_use_none_delimit_by_s_stop:w
10144     \use:n { #1 ~ ! ~ ##2 : ~ ##1 } \s_msg_stop
10145   }
10146 }
10147 \exp_args:Nc \_msg_tmp:w { ??? }

```

(End of definition for `_msg_expandable_error:nn`.)

`\msg_expandable_error:nnnnnn` The command built from the csname `\c__msg_text_prefix_tl #1 / #2` takes four arguments and builds the error text, which is fed to `__msg_expandable_error:n` with appropriate expansion: just as for usual messages the arguments are first turned to strings, then the message is fully expanded. The module name also has to be determined.

```

10148 \exp_args_generate:n { oooo }
10149 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
10150 {
10151   \exp_args:Nee \__msg_expandable_error:nn
10152   {
10153     \exp_args:Nc \exp_args:Noooo
10154     { \c__msg_text_prefix_tl #1 / #2 }
10155     { \tl_to_str:n {#3} }
10156     { \tl_to_str:n {#4} }
10157     { \tl_to_str:n {#5} }
10158     { \tl_to_str:n {#6} }
10159   }
10160   { \msg_error_text:n {#1} }
10161 }
10162 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
10163 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
10164 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
10165 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
10166 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
10167 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
10168 \cs_new:Npn \msg_expandable_error:nn #1#2
10169 { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
10170 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
10171 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnfff }
10172 \cs_generate_variant:Nn \msg_expandable_error:nnnnn { nnff }
10173 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnf }

```

(End of definition for `\msg_expandable_error:nnnnnn` and others. These functions are documented on page 88.)

49.9 Message formatting

```

10174 \prop_gput:Nnn \g_msg_module_name_prop { kernel } { LaTeX }
10175 \prop_gput:Nnn \g_msg_module_type_prop { kernel } { }
10176 \clist_map_inline:nn
10177 {
10178   char , clist , coffin , debug , deprecation , dim , msg ,
10179   quark , prg , prop , scanmark , seq , sys
10180 }
10181 {
10182   \prop_gput:Nnn \g_msg_module_name_prop {#1} { LaTeX }
10183   \prop_gput:Nnn \g_msg_module_type_prop {#1} { }
10184 }
10185 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX / cmd } { LaTeX }
10186 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX / cmd } { }
10187 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX / ltcmd } { LaTeX }
10188 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX / ltcmd } { }
10189 </package>

```

Chapter 50

l3file implementation

The following test files are used for this code: *m3file001*.

```
10190 <*package>
```

50.1 Input operations

```
10191 <@@=ior>
```

50.1.1 Variables and constants

`\l__ior_internal_tl` Used as a short-term scratch variable.

```
10192 \tl_new:N \l__ior_internal_tl
```

(End of definition for \l__ior_internal_tl.)

`\c__ior_term_ior` Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

```
10193 \int_const:Nn \c__ior_term_ior { 16 }
```

(End of definition for \c__ior_term_ior.)

`\g__ior_streams_seq` A list of the currently-available input streams to be used as a stack.

```
10194 \seq_new:N \g__ior_streams_seq
```

(End of definition for \g__ior_streams_seq.)

`\l__ior_stream_tl` Used to recover the raw stream number from the stack.

```
10195 \tl_new:N \l__ior_stream_tl
```

(End of definition for \l__ior_stream_tl.)

`\g__ior_streams_prop` The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For $\text{\LaTeX} 2_{\epsilon}$ and plain \TeX this data is stored in `\count16`: with the `etex` package loaded we need to subtract 1 as the register holds the number of the next stream to use. In `ConTeXt`, we need to look at `\count38` but there is no subtraction: like the original plain $\text{\TeX}/\text{\LaTeX} 2_{\epsilon}$ mechanism it holds the value of the *last* stream allocated.

```
10196 \prop_new:N \g__ior_streams_prop
```

```

10197 \int_step_inline:nnn
10198   { 0 }
10199   {
10200     \cs_if_exist:NTF \contextversion
10201     { \tex_count:D 38 ~ }
10202     {
10203       \tex_count:D 16 ~ %
10204       \cs_if_exist:NT \loccount { - 1 }
10205     }
10206   }
10207   {
10208     \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by-format }
10209   }

```

(End of definition for `\g__ior_streams_prop`.)

50.1.2 Stream management

`\ior_new:N` Reserving a new stream is done by defining the name as equal to using the terminal.

```

\ior_new:c 10210 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c__ior_term_ior }
10211 \cs_generate_variant:Nn \ior_new:N { c }

```

(End of definition for `\ior_new:N`. This function is documented on page 91.)

`\g_tmpa_ior` The usual scratch space.

```

\g_tmpb_ior 10212 \ior_new:N \g_tmpa_ior
10213 \ior_new:N \g_tmpb_ior

```

(End of definition for `\g_tmpa_ior` and `\g_tmpb_ior`. These variables are documented on page 99.)

`\ior_open:Nn` Use the conditional version, with an error if the file is not found.

```

\ior_open:cn 10214 \cs_new_protected:Npn \ior_open:Nn #1#2
10215   { \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
10216 \cs_generate_variant:Nn \ior_open:Nn { c }

```

(End of definition for `\ior_open:Nn`. This function is documented on page 91.)

`\l__ior_file_name_tl` Data storage.

```

10217 \tl_new:N \l__ior_file_name_tl

```

(End of definition for `\l__ior_file_name_tl`.)

`\ior_open:NnTF` An auxiliary searches for the file in the \TeX , $\text{\LaTeX} 2_{\epsilon}$ and $\text{\LaTeX} 3$ paths. Then pass the file found to the lower-level function which deals with streams. The `full_name` is empty when the file is not found.

`\ior_open:cnTF`

```

10218 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
10219   {
10220     \file_get_full_name:nNTF {#2} \l__ior_file_name_tl
10221     {
10222       \__kernel_ior_open:No #1 \l__ior_file_name_tl
10223       \prg_return_true:
10224     }
10225     { \prg_return_false: }
10226   }
10227 \prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }

```

(End of definition for `\ior_open:NnTF`. This function is documented on page 91.)

`__ior_new:N` Streams are reserved using `\newread` before they can be managed by `ior`. To prevent `ior` from being affected by redefinitions of `\newread` (such as done by the third-party package `morewrites`), this macro is saved here under a private name. The complicated code ensures that `__ior_new:N` is not `\outer` despite plain TeX's `\newread` being `\outer`. For ConTeXt, we have to deal with the fact that `\newread` works like our own: it actually checks before altering definition.

```

10228 \exp_args:NNf \cs_new_protected:Npn \__ior_new:N
10229   { \exp_args:Nnc \exp_after:wN \exp_stop_f: { newread } }
10230 \cs_if_exist:NT \contextversion
10231   {
10232     \cs_new_eq:NN \__ior_new_aux:N \__ior_new:N
10233     \cs_gset_protected:Npn \__ior_new:N #1
10234       {
10235         \cs_undefine:N #1
10236         \__ior_new_aux:N #1
10237       }
10238   }

```

(End of definition for `__ior_new:N`.)

`__kernel_ior_open:Nn` The stream allocation itself uses the fact that there is a list of all of those available. Life gets more complex as it's important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by `ior` but are now free are tracked, so we first try those. If that fails, ask plain TeX or L^AT_ΕX 2_ε for a new stream and use that number (after a bit of conversion).

`__kernel_ior_open:No`

`__ior_open_stream:Nn`

```

10239 \cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
10240   {
10241     \ior_close:N #1
10242     \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
10243     { \__ior_open_stream:Nn #1 {#2} }
10244     {
10245       \__ior_new:N #1
10246       \__kernel_tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
10247       \__ior_open_stream:Nn #1 {#2}
10248     }
10249   }
10250 \cs_generate_variant:Nn \__kernel_ior_open:Nn { No }

```

Here, we act defensively in case LuaTeX is in use with an extensionless file name.

```

10251 \cs_new_protected:Npe \__ior_open_stream:Nn #1#2
10252   {
10253     \tex_global:D \tex_chardef:D #1 = \exp_not:N \l__ior_stream_tl \scan_stop:
10254     \prop_gput:NVn \exp_not:N \g__ior_streams_prop #1 {#2}
10255     \tex_openin:D #1
10256     \sys_if_engine luatex:TF
10257     { {#2} }
10258     { \exp_not:N \__kernel_file_name_quote:n {#2} \scan_stop: }
10259   }

```

(End of definition for `__kernel_ior_open:Nn` and `__ior_open_stream:Nn`.)

`\ior_shell_open:Nn` Actually much easier than either the standard `open` or `input` versions! When calling `__ior_shell_open:nN` `__kernel_ior_open:Nn` the file the pipe is added to signal a shell command, but the quotes are not added yet—they are added later by `__kernel_file_name_quote:n`.

```

10260 \cs_new_protected:Npn \ior_shell_open:Nn #1#2
10261 {
10262   \sys_if_shell:TF
10263     { \__ior_shell_open:oN { \tl_to_str:n {#2} } #1 }
10264     { \msg_error:nn { kernel } { pipe-failed } }
10265 }
10266 \cs_new_protected:Npn \__ior_shell_open:nN #1#2
10267 {
10268   \tl_if_in:nnTF {#1} { " }
10269   {
10270     \msg_error:nne
10271       { kernel } { quote-in-shell } {#1}
10272   }
10273   { \__kernel_ior_open:Nn #2 { |#1 } }
10274 }
10275 \cs_generate_variant:Nn \__ior_shell_open:nN { o }
10276 \msg_new:nnnn { kernel } { pipe-failed }
10277 { Cannot~run~piped~system~commands. }
10278 {
10279   LaTeX~tried~to~call~a~system~process~but~this~was~not~possible.\\
10280   Try~the~"--shell-escape"~(or~"--enable-pipes")~option.
10281 }

```

(End of definition for `\ior_shell_open:Nn` and `__ior_shell_open:nN`. This function is documented on page 91.)

`\ior_close:N` Closing a stream means getting rid of it at the T_EX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0, 15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

```

10282 \cs_new_protected:Npn \ior_close:N #1
10283 {
10284   \int_compare:nT { -1 < #1 < \c__ior_term_ior }
10285   {
10286     \tex_closein:D #1
10287     \prop_gremove:NV \g__ior_streams_prop #1
10288     \seq_if_in:NVF \g__ior_streams_seq #1
10289       { \seq_gpush:NV \g__ior_streams_seq #1 }
10290     \cs_gset_eq:NN #1 \c__ior_term_ior
10291   }
10292 }
10293 \cs_generate_variant:Nn \ior_close:N { c }

```

(End of definition for `\ior_close:N`. This function is documented on page 92.)

`\ior_show:N` Seek the stream in the `\g__ior_streams_prop` list, then show the stream as open or closed accordingly.

```

10294 \cs_new_protected:Npn \ior_show:N { \__ior_show:NN \tl_show:n }
10295 \cs_generate_variant:Nn \ior_show:N { c }
10296 \cs_new_protected:Npn \ior_log:N { \__ior_show:NN \tl_log:n }
10297 \cs_generate_variant:Nn \ior_log:N { c }

```

```

10298 \cs_new_protected:Npn \__ior_show:NN #1#2
10299 {
10300   \__kernel_chk_defined:NT #2
10301   {
10302     \prop_get:NVNTF \g__ior_streams_prop #2 \l__ior_internal_tl
10303     {
10304       \exp_args:Ne #1
10305       { \token_to_str:N #2 ~ open: ~ \l__ior_internal_tl }
10306     }
10307     { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
10308   }
10309 }

```

(End of definition for `\ior_show:N`, `\ior_log:N`, and `__ior_show:NN`. These functions are documented on page 92.)

`\ior_show_list:` Show the property lists, but with some “pretty printing”. See the `l3msg` module. The first argument of the message is `ior` (as opposed to `iow`) and the second is empty if no read stream is open and non-empty (the list of streams formatted using `\msg_show_item_unbraced:nn`) otherwise. The code of the message `show-streams` takes care of translating `ior/iow` to English.

```

10310 \cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nneeee }
10311 \cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nneeee }
10312 \cs_new_protected:Npn \__ior_list:N #1
10313 {
10314   #1 { kernel } { show-streams }
10315   { ior }
10316   {
10317     \prop_map_function:NN \g__ior_streams_prop
10318     \msg_show_item_unbraced:nn
10319   }
10320   { } { }
10321 }

```

(End of definition for `\ior_show_list:`, `\ior_log_list:`, and `__ior_list:N`. These functions are documented on page 92.)

50.1.3 Reading input

`\if_eof:w` The primitive conditional

```

10322 \cs_new_eq:NN \if_eof:w \tex_ifeof:D

```

(End of definition for `\if_eof:w`. This function is documented on page 99.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.
`\ior_if_eof:NTF` The primitive test can only deal with numbers in the range $[0, 15]$ so we catch outliers (they are exhausted).

```

10323 \prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
10324 {
10325   \if_int_compare:w -1 < #1
10326   \if_int_compare:w #1 < \c__ior_term_ior
10327   \if_eof:w #1
10328   \prg_return_true:
10329   \else:

```

```

10330         \prg_return_false:
10331         \fi:
10332     \else:
10333         \prg_return_true:
10334         \fi:
10335     \else:
10336         \prg_return_true:
10337     \fi:
10338 }

```

(End of definition for `\ior_if_eof:NTF`. This function is documented on page 95.)

`\ior_get:NN` And here we read from files.

```

\__ior_get:NN 10339 \cs_new_protected:Npn \ior_get:NN #1#2
\ior_get:NNTF 10340 { \ior_get:NNTF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
10341 \cs_new_protected:Npn \__ior_get:NN #1#2
10342 { \tex_read:D #1 to #2 }
10343 \prg_new_protected_conditional:Npnn \ior_get:NN #1#2 { T , F , TF }
10344 {
10345     \ior_if_eof:NTF #1
10346     { \prg_return_false: }
10347     {
10348         \__ior_get:NN #1 #2
10349         \prg_return_true:
10350     }
10351 }

```

(End of definition for `\ior_get:NN`, `__ior_get:NN`, and `\ior_get:NNTF`. These functions are documented on page 93.)

`\ior_str_get:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline
`__ior_str_get:NN` character and restore it afterwards.

```

\ior_str_get:NNTF 10352 \cs_new_protected:Npn \ior_str_get:NN #1#2
10353 { \ior_str_get:NNTF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
10354 \cs_new_protected:Npn \__ior_str_get:NN #1#2
10355 {
10356     \exp_args:Nno \use:n
10357     {
10358         \int_set:Nn \tex_endlinechar:D { -1 }
10359         \tex_readline:D #1 to #2
10360         \int_set:Nn \tex_endlinechar:D
10361     } { \int_use:N \tex_endlinechar:D }
10362 }
10363 \prg_new_protected_conditional:Npnn \ior_str_get:NN #1#2 { T , F , TF }
10364 {
10365     \ior_if_eof:NTF #1
10366     { \prg_return_false: }
10367     {
10368         \__ior_str_get:NN #1 #2
10369         \prg_return_true:
10370     }
10371 }

```

(End of definition for `\ior_str_get:NN`, `__ior_str_get:NN`, and `\ior_str_get:NNTF`. These functions are documented on page 93.)

`\c__ior_term_noprompt_ior` For reading without a prompt.
10372 `\int_const:Nn \c__ior_term_noprompt_ior { -1 }`

(End of definition for `\c__ior_term_noprompt_ior`.)

`\ior_get_term:nN` Getting from the terminal is better with pretty-printing.

`\ior_str_get_term:nN`
`__ior_get_term:NnN`

```

10373 \cs_new_protected:Npn \ior_get_term:nN #1#2
10374 { \__ior_get_term:NnN \__ior_get:NN {#1} #2 }
10375 \cs_new_protected:Npn \ior_str_get_term:nN #1#2
10376 { \__ior_get_term:NnN \__ior_str_get:NN {#1} #2 }
10377 \cs_new_protected:Npn \__ior_get_term:NnN #1#2#3
10378 {
10379   \group_begin:
10380   \tex_escapechar:D = -1 \scan_stop:
10381   \tl_if_blank:nTF {#2}
10382     { \exp_args:NNC #1 \c__ior_term_noprompt_ior }
10383     { \exp_args:NNC #1 \c__ior_term_ior }
10384     {#2}
10385   \exp_args:NNNv \group_end:
10386   \tl_set:Nn #3 {#2}
10387 }

```

(End of definition for `\ior_get_term:nN`, `\ior_str_get_term:nN`, and `__ior_get_term:NnN`. These functions are documented on page 96.)

`\ior_map_break:` Usual map breaking functions.

`\ior_map_break:n`

```

10388 \cs_new:Npn \ior_map_break:
10389 { \prg_map_break:Nn \ior_map_break: { } }
10390 \cs_new:Npn \ior_map_break:n
10391 { \prg_map_break:Nn \ior_map_break: }

```

(End of definition for `\ior_map_break:` and `\ior_map_break:n`. These functions are documented on page 95.)

`\ior_map_inline:Nn` Mapping over an input stream can be done on either a token or a string basis, hence the
`\ior_str_map_inline:Nn` set up. Within that, there is a check to avoid reading past the end of a file, hence the two
`__ior_map_inline:NNn` applications of `\ior_if_eof:N` and its lower-level analogue `\if_eof:w`. This mapping
`__ior_map_inline:NNNn` cannot be nested with twice the same stream, as the stream has only one “current line”.
`__ior_map_inline_loop:NNN`

```

10392 \cs_new_protected:Npn \ior_map_inline:Nn
10393 { \__ior_map_inline:NNn \__ior_get:NN }
10394 \cs_new_protected:Npn \ior_str_map_inline:Nn
10395 { \__ior_map_inline:NNn \__ior_str_get:NN }
10396 \cs_new_protected:Npn \__ior_map_inline:NNn
10397 {
10398   \int_gincr:N \g__kernel_prg_map_int
10399   \exp_args:Nc \__ior_map_inline:NNNn
10400     { \__ior_map_ \int_use:N \g__kernel_prg_map_int :n }
10401 }
10402 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
10403 {
10404   \cs_gset_protected:Npn #1 ##1 {#4}
10405   \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
10406   \prg_break_point:Nn \ior_map_break:
10407     { \int_gdecr:N \g__kernel_prg_map_int }

```

```

10408 }
10409 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
10410 {
10411   #2 #3 \l__ior_internal_tl
10412   \if_eof:w #3
10413     \exp_after:wN \ior_map_break:
10414   \fi:
10415   \exp_args:No #1 \l__ior_internal_tl
10416   \__ior_map_inline_loop:NNN #1#2#3
10417 }

```

(End of definition for `\ior_map_inline:Nn` and others. These functions are documented on page 94.)

```

\ior_map_variable:NNn
\ior_str_map_variable:NNn
\__ior_map_variable:NNNn
  \__ior_map_variable_loop:NNNn

```

Since the TeX primitive (`\read` or `\readline`) assigns the tokens read in the same way as a token list assignment, we simply call the appropriate primitive. The end-of-loop is checked using the primitive conditional for speed.

```

10418 \cs_new_protected:Npn \ior_map_variable:NNn
10419   { \__ior_map_variable:NNNn \ior_get:NN }
10420 \cs_new_protected:Npn \ior_str_map_variable:NNn
10421   { \__ior_map_variable:NNNn \ior_str_get:NN }
10422 \cs_new_protected:Npn \__ior_map_variable:NNNn #1#2#3#4
10423   {
10424     \ior_if_eof:NF #2 { \__ior_map_variable_loop:NNNn #1#2#3 {#4} }
10425     \prg_break_point:Nn \ior_map_break: { }
10426   }
10427 \cs_new_protected:Npn \__ior_map_variable_loop:NNNn #1#2#3#4
10428   {
10429     #1 #2 #3
10430     \if_eof:w #2
10431       \exp_after:wN \ior_map_break:
10432     \fi:
10433     #4
10434     \__ior_map_variable_loop:NNNn #1#2#3 {#4}
10435   }

```

(End of definition for `\ior_map_variable:NNn` and others. These functions are documented on page 94.)

50.2 Output operations

```
10436 <@@=iow>
```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

50.2.1 Variables and constants

`\l__iow_internal_tl` Used as a short-term scratch variable.

```
10437 \tl_new:N \l__iow_internal_tl
```

(End of definition for `\l__iow_internal_tl`.)

`\c_log_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`) and to both the terminal and transcript file (`\c_term_iow`). Recent LuaTeX provide 128

write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value depends on the engine.

```

10438 \int_const:Nn \c_log_iow { -1 }
10439 \int_const:Nn \c_term_iow
10440 {
10441   \bool_lazy_and:nnTF
10442     { \sys_if_engine_luatex_p: }
10443     { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
10444     { 128 }
10445     { 16 }
10446 }

```

(End of definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 99.)

`\g__iow_streams_seq` A list of the currently-available output streams to be used as a stack.

```
10447 \seq_new:N \g__iow_streams_seq
```

(End of definition for `\g__iow_streams_seq`.)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

```
10448 \tl_new:N \l__iow_stream_tl
```

(End of definition for `\l__iow_stream_tl`.)

`\g__iow_streams_prop` As for reads with the appropriate adjustment of the register numbers to check on.

```

10449 \prop_new:N \g__iow_streams_prop
10450 \int_step_inline:nnn
10451   { 0 }
10452   {
10453     \cs_if_exist:NTF \contextversion
10454     { \tex_count:D 39 ~ }
10455     {
10456       \tex_count:D 17 ~
10457       \cs_if_exist:NT \loccount { - 1 }
10458     }
10459   }
10460   {
10461     \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved-by-format }
10462   }

```

(End of definition for `\g__iow_streams_prop`.)

50.2.2 Internal auxiliaries

`\s__iow_mark` Internal scan marks.

```

\s__iow_stop 10463 \scan_new:N \s__iow_mark
10464 \scan_new:N \s__iow_stop

```

(End of definition for `\s__iow_mark` and `\s__iow_stop`.)

`__iow_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```
10465 \cs_new:Npn \__iow_use_i_delimit_by_s_stop:nw #1 #2 \s__iow_stop {#1}
```

(End of definition for `__iow_use_i_delimit_by_s_stop:nw`.)

`\q__iow_nil` Internal quarks.
 10466 `\quark_new:N \q__iow_nil`
 (End of definition for `\q__iow_nil`.)

50.3 Stream management

`\iow_new:N` Reserving a new stream is done by defining the name as equal to writing to the terminal:
`\iow_new:c` odd but at least consistent.

```
10467 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
10468 \cs_generate_variant:Nn \iow_new:N { c }
```

(End of definition for `\iow_new:N`. This function is documented on page 91.)

`\g_tmpa_iow` The usual scratch space.

`\g_tmpb_iow`
 10469 `\iow_new:N \g_tmpa_iow`
 10470 `\iow_new:N \g_tmpb_iow`

(End of definition for `\g_tmpa_iow` and `\g_tmpb_iow`. These variables are documented on page 99.)

`__iow_new:N` As for read streams, copy `\newwrite`, making sure that it is not `\outer`. For `ConTeXt`, we have to deal with the fact that `\newwrite` works like our own: it actually checks before altering definition.

```
10471 \exp_args:NNf \cs_new_protected:Npn \__iow_new:N
10472 { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
10473 \cs_if_exist:NT \contextversion
10474 {
10475   \cs_new_eq:NN \__iow_new_aux:N \__iow_new:N
10476   \cs_gset_protected:Npn \__iow_new:N #1
10477     {
10478       \cs_undefine:N #1
10479       \__iow_new_aux:N #1
10480     }
10481 }
```

(End of definition for `__iow_new:N`.)

`\l__iow_file_name_tl` Data storage.

```
10482 \tl_new:N \l__iow_file_name_tl
```

(End of definition for `\l__iow_file_name_tl`.)

`\iow_open:Nn` The same idea as for reading, but without the path and without the need to allow for a conditional version.

`\iow_open:NV`
`\iow_open:cn`
`\iow_open:cV`
 10483 `\cs_new_protected:Npn \iow_open:Nn #1#2`
 10484 {

```
10485   \__kernel_tl_set:Nx \l__iow_file_name_tl
10486   { \__kernel_file_name_sanitize:n {#2} }
10487   \__kernel_iow_open:No #1 \l__iow_file_name_tl
10488 }
10489 \cs_generate_variant:Nn \iow_open:Nn { NV , c , cV }
10490 \cs_new_protected:Npn \__kernel_iow_open:Nn #1#2
10491 {
```

```

10492 \iow_close:N #1
10493 \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
10494 { \__iow_open_stream:Nn #1 {#2} }
10495 {
10496   \__iow_new:N #1
10497   \__kernel_tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
10498   \__iow_open_stream:Nn #1 {#2}
10499 }
10500 }
10501 \cs_generate_variant:Nn \__kernel_iow_open:Nn { No }
10502 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
10503 {
10504   \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
10505   \prop_gput:NVn \g__iow_streams_prop #1 {#2}
10506   \tex_immediate:D \tex_openout:D
10507     #1 \__kernel_file_name_quote:n {#2} \scan_stop:
10508 }
10509 \cs_generate_variant:Nn \__iow_open_stream:Nn { NV }

```

(End of definition for `\iow_open:Nn`, `__kernel_iow_open:Nn`, and `__iow_open_stream:Nn`. This function is documented on page 91.)

`\iow_shell_open:Nn`
`__iow_shell_open:nN`
`__iow_shell_open:oN`

Very similar to the `ior` version

```

10510 \cs_new_protected:Npn \iow_shell_open:Nn #1#2
10511 {
10512   \sys_if_shell:TF
10513     { \__iow_shell_open:oN { \tl_to_str:n {#2} } #1 }
10514     { \msg_error:nn { kernel } { pipe-failed } }
10515 }
10516 \cs_new_protected:Npn \__iow_shell_open:nN #1#2
10517 {
10518   \tl_if_in:nnTF {#1} { " }
10519   {
10520     \msg_error:nne
10521       { kernel } { quote-in-shell } {#1}
10522   }
10523   { \__kernel_iow_open:Nn #2 { |#1 } }
10524 }
10525 \cs_generate_variant:Nn \__iow_shell_open:nN { o }

```

(End of definition for `\iow_shell_open:Nn` and `__iow_shell_open:nN`. This function is documented on page 91.)

`\iow_close:N`
`\iow_close:c`

Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```

10526 \cs_new_protected:Npn \iow_close:N #1
10527 {
10528   \int_compare:nT { \c_log_iow < #1 < \c_term_iow }
10529   {
10530     \tex_immediate:D \tex_closeout:D #1
10531     \prop_gremove:NV \g__iow_streams_prop #1
10532     \seq_if_in:NVF \g__iow_streams_seq #1
10533       { \seq_gpush:NV \g__iow_streams_seq #1 }
10534     \cs_gset_eq:NN #1 \c_term_iow

```



```

10535     }
10536   }
10537 \cs_generate_variant:Nn \iow_close:N { c }

```

(End of definition for `\iow_close:N`. This function is documented on page 92.)

`\iow_show:N` Seek the stream in the `\g__iow_streams_prop` list, then show the stream as open or closed accordingly.

```

\iow_log:N
\__iow_show:NN
10538 \cs_new_protected:Npn \iow_show:N { \__iow_show:NN \tl_show:n }
10539 \cs_generate_variant:Nn \iow_show:N { c }
10540 \cs_new_protected:Npn \iow_log:N { \__iow_show:NN \tl_log:n }
10541 \cs_generate_variant:Nn \iow_log:N { c }
10542 \cs_new_protected:Npn \__iow_show:NN #1#2
10543   {
10544     \__kernel_chk_defined:NT #2
10545     {
10546       \prop_get:NVNTF \g__iow_streams_prop #2 \l__iow_internal_tl
10547       {
10548         \exp_args:Ne #1
10549         { \token_to_str:N #2 ~ open:~ \l__iow_internal_tl }
10550       }
10551       { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
10552     }
10553   }

```

(End of definition for `\iow_show:N`, `\iow_log:N`, and `__iow_show:NN`. These functions are documented on page 92.)

`\iow_show_list:` Done as for input, but with a copy of the auxiliary so the name is correct.

```

\iow_log_list:
\__iow_list:N
10554 \cs_new_protected:Npn \iow_show_list: { \__iow_list:N \msg_show:nneeee }
10555 \cs_new_protected:Npn \iow_log_list: { \__iow_list:N \msg_log:nneeee }
10556 \cs_new_protected:Npn \__iow_list:N #1
10557   {
10558     #1 { kernel } { show-streams }
10559     { iow }
10560     {
10561       \prop_map_function:NN \g__iow_streams_prop
10562       \msg_show_item_unbraced:nn
10563     }
10564     { } { }
10565   }

```

(End of definition for `\iow_show_list:`, `\iow_log_list:`, and `__iow_list:N`. These functions are documented on page 92.)

50.3.1 Deferred writing

`\iow_shipout_e:Nn` First the easy part, this is the primitive, which expects its argument to be braced.

```

\iow_shipout_e:Ne
\iow_shipout_e:cn
\iow_shipout_e:ce
10566 \cs_new_protected:Npn \iow_shipout_e:Nn #1#2
10567   { \tex_write:D #1 {#2} }
10568 \cs_generate_variant:Nn \iow_shipout_e:Nn { Ne , c, ce }

```

(End of definition for `\iow_shipout_e:Nn`. This function is documented on page 97.)

`\iow_shipout:Nn` With ε -TeX available deferred writing without expansion is easy.

```

\iow_shipout:Ne 10569 \cs_new_protected:Npn \iow_shipout:Nn #1#2
\iow_shipout:Nx 10570 { \tex_write:D #1 { \exp_not:n {#2} } }
\iow_shipout:cn 10571 \cs_generate_variant:Nn \iow_shipout:Nn { Ne , c , ce }
\iow_shipout:ce 10572 \cs_generate_variant:Nn \iow_shipout:Nn { Nx , cx }
\iow_shipout:cx

```

(End of definition for `\iow_shipout:Nn`. This function is documented on page 96.)

50.3.2 Immediate writing

`__kernel_iow_with:Nnn` If the integer #1 is equal to #2, just leave #3 in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

```

10573 \cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2
10574 {
10575   \int_compare:nNnTF {#1} = {#2}
10576     { \use:n }
10577     { \__iow_with:oNnn { \int_use:N #1 } #1 {#2} }
10578 }
10579 \cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4
10580 {
10581   \int_set:Nn #2 {#3}
10582   #4
10583   \int_set:Nn #2 {#1}
10584 }
10585 \cs_generate_variant:Nn \__iow_with:nNnn { o }

```

(End of definition for `__kernel_iow_with:Nnn` and `__iow_with:nNnn`.)

`\iow_now:Nn` This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don't use the expansion done by `\iow_now:NV` `\write` to get the `Nx` variant, because it differs in subtle ways from `x`-expansion, namely, macro parameter characters would not need to be doubled. We set the `\newlinechar` to 10 using `__kernel_iow_with:Nnn` to support formats such as plain TeX: otherwise, `\iow_now:cn` `\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_shipout_x:Nn`, as TeX looks at the value of the `\newlinechar` at shipout time in those cases.

```

10586 \cs_new_protected:Npn \iow_now:Nn #1#2
10587 {
10588   \__kernel_iow_with:Nnn \tex_newlinechar:D { '\^^J }
10589   { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
10590 }
10591 \cs_generate_variant:Nn \iow_now:Nn { NV , Ne , c , cV , ce }
10592 \cs_generate_variant:Nn \iow_now:Nn { Nx , cx }

```

(End of definition for `\iow_now:Nn`. This function is documented on page 96.)

`\iow_log:n` Writing to the log and the terminal directly are relatively easy; as we need the two `e`-type variants for bootstrapping, they are redefinitions here.

```

\iow_log:e 10593 \cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
\iow_log:x 10594 \cs_set_protected:Npn \iow_log:e { \iow_now:Ne \c_log_iow }
\iow_term:n 10595 \cs_generate_variant:Nn \iow_log:n { x }
\iow_term:e
\iow_term:x

```

```

10596 \cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
10597 \cs_set_protected:Npn \iow_term:e { \iow_now:Ne \c_term_iow }
10598 \cs_generate_variant:Nn \iow_term:n { x }

```

(End of definition for `\iow_log:n` and `\iow_term:n`. These functions are documented on page 96.)

50.3.3 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream.

```

10599 \cs_new:Npn \iow_newline: { ^^J }

```

(End of definition for `\iow_newline:`. This function is documented on page 97.)

`\iow_char:N` Function to write any escaped char to an output stream.

```

10600 \cs_new_eq:NN \iow_char:N \cs_to_str:N

```

(End of definition for `\iow_char:N`. This function is documented on page 97.)

50.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

`\l_iow_line_count_int` This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by T_EX Live and MiK_TE_X.

```

10601 \int_new:N \l_iow_line_count_int
10602 \int_set:Nn \l_iow_line_count_int { 78 }

```

(End of definition for `\l_iow_line_count_int`. This variable is documented on page 99.)

`\l__iow_newline_tl` The token list inserted to produce a new line, with the *⟨run-on text⟩*.

```

10603 \tl_new:N \l__iow_newline_tl

```

(End of definition for `\l__iow_newline_tl`.)

`\l_iow_line_target_int` This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```

10604 \int_new:N \l_iow_line_target_int

```

(End of definition for `\l_iow_line_target_int`.)

`__iow_set_indent:n` **`__iow_unindent:w`** **`\l__iow_one_indent_tl`** **`\l_iow_one_indent_int`** The `one_indent` variables hold one indentation marker and its length. The `__iow_unindent:w` auxiliary removes one indentation. The function `__iow_set_indent:n` (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```

10605 \tl_new:N \l__iow_one_indent_tl
10606 \int_new:N \l_iow_one_indent_int
10607 \cs_new:Npn \__iow_unindent:w { }
10608 \cs_new_protected:Npn \__iow_set_indent:n #1
10609 {
10610   \__kernel_tl_set:Nx \l__iow_one_indent_tl
10611   { \exp_args:No \__kernel_str_to_other_fast:n { \tl_to_str:n {#1} } } }
10612 \int_set:Nn \l_iow_one_indent_int

```

```

10613     { \str_count:N \l__iow_one_indent_tl }
10614     \exp_last_unbraced:NNo
10615     \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_tl { }
10616   }
10617 \exp_args:Ne \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }

```

(End of definition for __iow_set_indent:n and others.)

`\l__iow_indent_tl` `\l__iow_indent_int` The current indentation (some copies of `\l__iow_one_indent_tl`) and its number of characters.

```

10618 \tl_new:N \l__iow_indent_tl
10619 \int_new:N \l__iow_indent_int

```

(End of definition for \l__iow_indent_tl and \l__iow_indent_int.)

`\l__iow_line_tl` `\l__iow_line_part_tl` These hold the current line of text and a partial line to be added to it, respectively.

```

10620 \tl_new:N \l__iow_line_tl
10621 \tl_new:N \l__iow_line_part_tl

```

(End of definition for \l__iow_line_tl and \l__iow_line_part_tl.)

`\l__iow_line_break_bool` Indicates whether the line was broken precisely at a chunk boundary.

```

10622 \bool_new:N \l__iow_line_break_bool

```

(End of definition for \l__iow_line_break_bool.)

`\l__iow_wrap_tl` Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```

10623 \tl_new:N \l__iow_wrap_tl

```

(End of definition for \l__iow_wrap_tl.)

`\c__iow_wrap_marker_tl` `\c__iow_wrap_end_marker_tl` `\c__iow_wrap_newline_marker_tl` `\c__iow_wrap_allow_break_marker_tl` `\c__iow_wrap_indent_marker_tl` `\c__iow_wrap_unindent_marker_tl` Every special action of the wrapping code starts with the same recognizable string, `\c__iow_wrap_marker_tl`. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of `\escapechar` here is not very important, but makes `\c__iow_wrap_marker_tl` look marginally nicer.

```

10624 \group_begin:
10625   \int_set:Nn \tex_escapechar:D { -1 }
10626   \tl_const:Ne \c__iow_wrap_marker_tl
10627     { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
10628 \group_end:
10629 \tl_map_inline:nn
10630 { { end } { newline } { allow_break } { indent } { unindent } }
10631 {
10632   \tl_const:ce { c__iow_wrap_#1 _marker_tl }
10633   {
10634     \c__iow_wrap_marker_tl
10635     #1
10636     \c_catcode_other_space_tl
10637   }
10638 }

```

(End of definition for \c__iow_wrap_marker_tl and others.)

`\iow_wrap_allow_break:` We set `\iow_wrap_allow_break:n` to produce an error when outside messages. Within wrapped message, it is set to `__iow_wrap_allow_break:` when valid and otherwise to `__iow_wrap_allow_break_error:`. The second produces an error expandably.

```

10639 \cs_new_protected:Npn \iow_wrap_allow_break:
10640 {
10641   \msg_error:nnnn { kernel } { iow-indent }
10642   { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
10643 }
10644 \cs_new:Npe \__iow_wrap_allow_break: { \c__iow_wrap_allow_break_marker_tl }
10645 \cs_new:Npn \__iow_wrap_allow_break_error:
10646 {
10647   \msg_expandable_error:nnnn { kernel } { iow-indent }
10648   { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
10649 }

```

(End of definition for `\iow_wrap_allow_break:`, `__iow_wrap_allow_break:`, and `__iow_wrap_allow_break_error:`. This function is documented on page 98.)

`\iow_indent:n` We set `\iow_indent:n` to produce an error when outside messages. Within wrapped message, it is set to `__iow_indent:n` when valid and otherwise to `__iow_indent_error:n`. The first places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```

10650 \cs_new_protected:Npn \iow_indent:n #1
10651 {
10652   \msg_error:nnnnn { kernel } { iow-indent }
10653   { \iow_wrap:nnnN } { \iow_indent:n } {#1}
10654   #1
10655 }
10656 \cs_new:Npe \__iow_indent:n #1
10657 {
10658   \c__iow_wrap_indent_marker_tl
10659   #1
10660   \c__iow_wrap_unindent_marker_tl
10661 }
10662 \cs_new:Npn \__iow_indent_error:n #1
10663 {
10664   \msg_expandable_error:nnnnn { kernel } { iow-indent }
10665   { \iow_wrap:nnnN } { \iow_indent:n } {#1}
10666   #1
10667 }

```

(End of definition for `\iow_indent:n`, `__iow_indent:n`, and `__iow_indent_error:n`. This function is documented on page 98.)

`\iow_wrap:nnnN` The main wrapping function works as follows. First give `\`, `_` and other formatting commands the correct definition for messages and perform the given setup #3. The definition of `_` uses an “other” space rather than a normal space, because the latter might be absorbed by `TEX` to end a number or other f-type expansions. Use `\conditionally@traceoff` if defined; it is introduced by the trace package and suppresses uninteresting tracing of the wrapping code.

```

10668 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4

```

```

10669 {
10670   \group_begin:
10671   \cs_if_exist_use:N \conditionally@traceoff
10672   \int_set:Nn \tex_escapechar:D { -1 }
10673   \cs_set:Npe \{ { \token_to_str:N \{ }
10674   \cs_set:Npe \# { \token_to_str:N \# }
10675   \cs_set:Npe \} { \token_to_str:N \} }
10676   \cs_set:Npe \% { \token_to_str:N \% }
10677   \cs_set:Npe \~ { \token_to_str:N \~ }
10678   \int_set:Nn \tex_escapechar:D { 92 }
10679   \cs_set_eq:NN \ \ \iow_newline:
10680   \cs_set_eq:NN \ \c_catcode_other_space_tl
10681   \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break:
10682   \cs_set_eq:NN \iow_indent:n \__iow_indent:n
10683   #3

```

Then fully-expand the input: in package mode, the expansion uses L^AT_EX 2_ε's `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnnN`.

```

10684   \cs_set_eq:NN \protect \token_to_str:N
10685   \__kernel_tl_set:Nx \l__iow_wrap_tl {#1}
10686   \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break_error:
10687   \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n

```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l_iow_line_count_int` instead).

```

10688   \__kernel_tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
10689   \__kernel_tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
10690   \int_set:Nn \l__iow_line_target_int
10691   { \l_iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }

```

Sanity check.

```

10692   \int_compare:nNnT { \l__iow_line_target_int } < 0
10693   {
10694     \tl_set:Nn \l__iow_newline_tl { \iow_newline: }
10695     \int_set:Nn \l__iow_line_target_int
10696     { \l_iow_line_count_int + 1 }
10697   }

```

There is then a loop over the input, which stores the wrapped result in `\l__iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the “other” spaces back to normal spaces. The f-expansion removes a leading space from `\l__iow_wrap_tl`.

```

10698   \__iow_wrap_do:
10699   \exp_args:NNf \group_end:
10700   #4 { \tl_to_str:N \l__iow_wrap_tl }
10701   }
10702   \cs_generate_variant:Nn \iow_wrap:nnnN { ne }

```

(End of definition for `\iow_wrap:nnnN`. This function is documented on page 98.)

```

\__iow_wrap_do: Escape spaces and change newlines to \c__iow_wrap_newline_marker_tl. Set up a
\__iow_wrap_fix_newline:w few variables, in particular the initial value of \l__iow_wrap_tl: the space stops the
\__iow_wrap_start:w

```

f-expansion of the main wrapping function and `\use_none:n` removes a newline marker inserted by later code. The main loop consists of repeatedly calling the `chunk` auxiliary to wrap chunks delimited by (newline or indentation) markers.

```

10703 \cs_new_protected:Npn \__iow_wrap_do:
10704 {
10705   \__kernel_tl_set:Nx \l__iow_wrap_tl
10706   {
10707     \exp_args:No \__kernel_str_to_other_fast:n \l__iow_wrap_tl
10708     \c__iow_wrap_end_marker_tl
10709   }
10710   \__kernel_tl_set:Nx \l__iow_wrap_tl
10711   {
10712     \exp_after:wN \__iow_wrap_fix_newline:w \l__iow_wrap_tl
10713     ^^J \q__iow_nil ^^J \s__iow_stop
10714   }
10715   \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
10716 }
10717 \cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
10718 {
10719   #1
10720   \if_meaning:w \q__iow_nil #2
10721   \__iow_use_i_delimit_by_s_stop:nw
10722   \fi:
10723   \c__iow_wrap_newline_marker_tl
10724   \__iow_wrap_fix_newline:w #2 ^^J
10725 }
10726 \cs_new_protected:Npn \__iow_wrap_start:w
10727 {
10728   \bool_set_false:N \l__iow_line_break_bool
10729   \tl_clear:N \l__iow_line_tl
10730   \tl_clear:N \l__iow_line_part_tl
10731   \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
10732   \int_zero:N \l__iow_indent_int
10733   \tl_clear:N \l__iow_indent_tl
10734   \__iow_wrap_chunk:nw { \l__iow_line_count_int }
10735 }

```

(End of definition for `__iow_wrap_do:`, `__iow_wrap_fix_newline:w`, and `__iow_wrap_start:w`.)

```

\__iow_wrap_chunk:nw
\__iow_wrap_next:nw

```

The `chunk` and `next` auxiliaries are defined indirectly to obtain the expansions of `\c_catcode_other_space_tl` and `\c__iow_wrap_marker_tl` in their definition. The `next` auxiliary calls a function corresponding to the type of marker (its `##2`), which can be `newline` or `indent` or `unindent` or `end`. The first argument of the `chunk` auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call `next`. Otherwise, set up a call to `__iow_wrap_line:nw`, including the indentation if the current line is empty, and including a trailing space (`#1`) before the `__iow_wrap_end_chunk:w` auxiliary.

```

10736 \cs_set_protected:Npn \__iow_tmp:w #1#2
10737 {
10738   \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
10739   {
10740     \tl_if_empty:nTF {##2}
10741     {

```

```

10742         \tl_clear:N \l__iow_line_part_tl
10743         \__iow_wrap_next:nw {##1}
10744     }
10745     {
10746         \tl_if_empty:NTF \l__iow_line_tl
10747         {
10748             \__iow_wrap_line:nw
10749             { \l__iow_indent_tl }
10750             ##1 - \l__iow_indent_int ;
10751         }
10752         { \__iow_wrap_line:nw { } ##1 ; }
10753         ##2 #1
10754         \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \s__iow_stop
10755     }
10756 }
10757 \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
10758 { \use:c { __iow_wrap_##2:n } {##1} }
10759 }
10760 \exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl

```

(End of definition for `__iow_wrap_chunk:nw` and `__iow_wrap_next:nw`.)

```

\__iow_wrap_line:nw
\__iow_wrap_line_loop:w
\__iow_wrap_line_aux:Nw
\__iow_wrap_line_seven:nnnnnnn
\__iow_wrap_line_end:NnnnnnnnN
\__iow_wrap_line_end:nw
\__iow_wrap_end_chunk:w

```

This is followed by `{⟨string⟩}⟨int expr⟩`; It stores the `⟨string⟩` and up to `⟨int expr⟩` characters from the current chunk into `\l__iow_line_part_tl`. Characters are grabbed 8 at a time and left in `\l__iow_line_part_tl` by the `line_loop` auxiliary. When $k < 8$ remain to be found, the `line_aux` auxiliary calls the `line_end` auxiliary followed by (the single digit) k , then $7 - k$ empty brace groups, then the chunk's remaining characters. The `line_end` auxiliary leaves k characters from the chunk in the line part, then ends the assignment. Ignore the `\use_none:nnnnn` line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the `end_chunk` auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments #2-#9 of the `line_loop` auxiliary or as one of the arguments #2-#8 of the `line_end` auxiliary. In both cases stop the assignment and work out how many characters are still needed. Notice that when we have exactly seven arguments to clean up, a `\exp_stop_f:` has to be inserted to stop the `\exp:w`. The weird `\use_none:nnnnn` ensures that the required data is in the right place.

```

10761 \cs_new_protected:Npn \__iow_wrap_line:nw #1
10762 {
10763     \tex_edef:D \l__iow_line_part_tl { \if_false: } \fi:
10764     #1
10765     \exp_after:wN \__iow_wrap_line_loop:w
10766     \int_value:w \int_eval:w
10767 }
10768 \cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
10769 {
10770     \if_int_compare:w #1 < 8 \exp_stop_f:
10771     \__iow_wrap_line_aux:Nw #1
10772     \fi:
10773     #2 #3 #4 #5 #6 #7 #8 #9
10774     \exp_after:wN \__iow_wrap_line_loop:w

```



```

10775     \int_value:w \int_eval:w #1 - 8 ;
10776   }
10777 \cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
10778   {
10779     #2
10780     \exp_after:wN \__iow_wrap_line_end:NnnnnnnN
10781     \exp_after:wN #1
10782     \exp:w \exp_end_continue_f:w
10783     \exp_after:wN \exp_after:wN
10784     \if_case:w #1 \exp_stop_f:
10785       \prg_do_nothing:
10786     \or: \use_none:n
10787     \or: \use_none:nn
10788     \or: \use_none:nnn
10789     \or: \use_none:nnnn
10790     \or: \use_none:nnnnn
10791     \or: \use_none:nnnnnn
10792     \or: \__iow_wrap_line_seven:nnnnnnn
10793     \fi:
10794     { } { } { } { } { } { } { } { } #3
10795   }
10796 \cs_new:Npn \__iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
10797 \cs_new:Npn \__iow_wrap_line_end:NnnnnnnN #1#2#3#4#5#6#7#8#9
10798   {
10799     #2 #3 #4 #5 #6 #7 #8
10800     \use_none:nnnnn \int_eval:w 8 - ; #9
10801     \token_if_eq_charcode:NNTF \c_space_token #9
10802     { \__iow_wrap_line_end:nw { } }
10803     { \if_false: { \fi: } \__iow_wrap_break:w #9 }
10804   }
10805 \cs_new:Npn \__iow_wrap_line_end:nw #1
10806   {
10807     \if_false: { \fi: }
10808     \__iow_wrap_store_do:n {#1}
10809     \__iow_wrap_next_line:w
10810   }
10811 \cs_new:Npn \__iow_wrap_end_chunk:w
10812   #1 \int_eval:w #2 - #3 ; #4#5 \s__iow_stop
10813   {
10814     \if_false: { \fi: }
10815     \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
10816   }

```

(End of definition for __iow_wrap_line:nw and others.)

`__iow_wrap_break:w` Functions here are defined indirectly: `__iow_tmp:w` is eventually called with an “other” space as its argument. The goal is to remove from `\l__iow_line_part_tl` the part after the last space. In most cases this is done by repeatedly calling the `break_loop` auxiliary, which leaves “words” (delimited by spaces) until it hits the trailing space: then its argument `##3` is `__iow_wrap_break_end:w` instead of a single token, and that `break_end` auxiliary leaves in the assignment the line until the last space, then calls `__iow_wrap_line_end:nw` to finish up the line and move on to the next. If there is no space in `\l__iow_line_part_tl` then the `break_first` auxiliary calls the `break_none` auxiliary. In that case, if the current line is empty, the complete word (including

##4, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).

```

10817 \cs_set_protected:Npn \__iow_tmp:w #1
10818 {
10819   \cs_new:Npn \__iow_wrap_break:w
10820   {
10821     \tex_edef:D \l__iow_line_part_tl
10822     { \if_false: } \fi:
10823     \exp_after:wN \__iow_wrap_break_first:w
10824     \l__iow_line_part_tl
10825     #1
10826     { ? \__iow_wrap_break_end:w }
10827     \s__iow_mark
10828   }
10829   \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
10830   {
10831     \use_none:nn ##2 \__iow_wrap_break_none:w
10832     \__iow_wrap_break_loop:w ##1 #1 ##2
10833   }
10834   \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \s__iow_mark ##4 #1
10835   {
10836     \tl_if_empty:NTF \l__iow_line_tl
10837     { ##2 ##4 \__iow_wrap_line_end:nw { } }
10838     { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
10839   }
10840   \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
10841   {
10842     \use_none:n ##3
10843     ##1 #1
10844     \__iow_wrap_break_loop:w ##2 #1 ##3
10845   }
10846   \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
10847   { ##1 \__iow_wrap_line_end:nw { } ##3 }
10848 }
10849 \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl

```

(End of definition for __iow_wrap_break:w and others.)

`__iow_wrap_next_line:w` The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call `__iow_wrap_line:nw` to find characters for the next line (remembering to account for the indentation).

```

10850 \cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \s__iow_stop
10851 {
10852   \tl_clear:N \l__iow_line_tl
10853   \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
10854   {
10855     \tl_clear:N \l__iow_line_part_tl
10856     \bool_set_true:N \l__iow_line_break_bool
10857     \__iow_wrap_next:nw { \l__iow_line_target_int }
10858   }
10859   {
10860     \__iow_wrap_line:nw
10861     { \l__iow_indent_tl }

```

```

10862         \l__iow_line_target_int - \l__iow_indent_int ;
10863         #1 #2 \s__iow_stop
10864     }
10865 }

```

(End of definition for __iow_wrap_next_line:w.)

`__iow_wrap_allow_break:n` This is called after a chunk has been wrapped. The `\l__iow_line_part_tl` typically ends with a space (except at the beginning of a line?), which we remove since the `allow_break` marker should not insert a space. Then move on with the next chunk, making sure to adjust the target number of characters for the line in case we did remove a space.

```

10866 \cs_new_protected:Npn \__iow_wrap_allow_break:n #1
10867 {
10868     \__kernel_tl_set:Nx \l__iow_line_tl
10869     { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }
10870     \bool_set_false:N \l__iow_line_break_bool
10871     \tl_if_empty:NTF \l__iow_line_part_tl
10872     { \__iow_wrap_chunk:nw {#1} }
10873     { \exp_args:Nf \__iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
10874 }

```

(End of definition for __iow_wrap_allow_break:n.)

`__iow_wrap_indent:n` `__iow_wrap_unindent:n` These functions are called after a chunk has been wrapped, when encountering `indent/unindent` markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

```

10875 \cs_new_protected:Npn \__iow_wrap_indent:n #1
10876 {
10877     \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
10878     \bool_set_false:N \l__iow_line_break_bool
10879     \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10880     \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
10881     \__iow_wrap_chunk:nw {#1}
10882 }
10883 \cs_new_protected:Npn \__iow_wrap_unindent:n #1
10884 {
10885     \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
10886     \bool_set_false:N \l__iow_line_break_bool
10887     \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10888     \__kernel_tl_set:Nx \l__iow_indent_tl
10889     { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
10890     \__iow_wrap_chunk:nw {#1}
10891 }

```

(End of definition for __iow_wrap_indent:n and __iow_wrap_unindent:n.)

`__iow_wrap_newline:n` `__iow_wrap_end:n` These functions are called after a chunk has been line-wrapped, when encountering a `newline/end` marker. Unless we just took a line-break, store the line part and the line so far into the whole `\l__iow_wrap_tl`, trimming a trailing space. In the `newline` case look for a new line (of length `\l__iow_line_target_int`) in a new chunk.

```

10892 \cs_new_protected:Npn \__iow_wrap_newline:n #1
10893 {

```

```

10894     \bool_if:NF \l__iow_line_break_bool
10895         { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
10896     \bool_set_false:N \l__iow_line_break_bool
10897     \__iow_wrap_chunk:nw { \l__iow_line_target_int }
10898 }
10899 \cs_new_protected:Npn \__iow_wrap_end:n #1
10900 {
10901     \bool_if:NF \l__iow_line_break_bool
10902         { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
10903     \bool_set_false:N \l__iow_line_break_bool
10904 }

```

(End of definition for __iow_wrap_newline:n and __iow_wrap_end:n.)

__iow_wrap_store_do:n First add the last line part to the line, then append it to \l__iow_wrap_tl with the appropriate new line (with “run-on” text), possibly with its last space removed (#1 is empty or __iow_wrap_trim:N).

```

10905 \cs_new_protected:Npn \__iow_wrap_store_do:n #1
10906 {
10907     \__kernel_tl_set:Nx \l__iow_line_tl
10908     { \l__iow_line_tl \l__iow_line_part_tl }
10909     \__kernel_tl_set:Nx \l__iow_wrap_tl
10910     {
10911         \l__iow_wrap_tl
10912         \l__iow_newline_tl
10913         #1 \l__iow_line_tl
10914     }
10915     \tl_clear:N \l__iow_line_tl
10916 }

```

(End of definition for __iow_wrap_store_do:n.)

__iow_wrap_trim:N Remove one trailing “other” space from the argument if present.

```

\__iow_wrap_trim:w
\__iow_wrap_trim_aux:w
10917 \cs_set_protected:Npn \__iow_tmp:w #1
10918 {
10919     \cs_new:Npn \__iow_wrap_trim:N ##1
10920     { \exp_after:wN \__iow_wrap_trim:w ##1 \s__iow_mark #1 \s__iow_mark \s__iow_stop }
10921     \cs_new:Npn \__iow_wrap_trim:w ##1 #1 \s__iow_mark
10922     { \__iow_wrap_trim_aux:w ##1 \s__iow_mark }
10923     \cs_new:Npn \__iow_wrap_trim_aux:w ##1 \s__iow_mark ##2 \s__iow_stop {##1}
10924 }
10925 \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl

```

(End of definition for __iow_wrap_trim:N, __iow_wrap_trim:w, and __iow_wrap_trim_aux:w.)

10926 <@@=file>

50.4 File operations

\l__file_internal_tl Used as a short-term scratch variable.

```

10927 \tl_new:N \l__file_internal_tl

```

(End of definition for \l__file_internal_tl.)

`\g_file_curr_dir_str` `\g_file_curr_ext_str` `\g_file_curr_name_str` The name of the current file should be available at all times: the name itself is set dynamically.

```
10928 \str_new:N \g_file_curr_dir_str
10929 \str_new:N \g_file_curr_ext_str
10930 \str_new:N \g_file_curr_name_str
```

(End of definition for `\g_file_curr_dir_str`, `\g_file_curr_ext_str`, and `\g_file_curr_name_str`. These variables are documented on page 99.)

`\g__file_stack_seq` The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by L^AT_EX 2_ε (we must be in the preamble and loaded using `\usepackage` or `\RequirePackage`). As L^AT_EX 2_ε doesn't store directory and name separately, we stick to the same convention here. In pre-loading, `\@currnamestack` is empty so is skipped.

```
10931 \seq_new:N \g__file_stack_seq
10932 \group_begin:
10933   \cs_set_protected:Npn \__file_tmp:w #1#2#3
10934     {
10935       \tl_if_blank:nTF {#1}
10936         {
10937           \cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \s_file_stop
10938             { { } {##2} { } }
10939           \seq_gput_right:Ne \g__file_stack_seq
10940             {
10941               \exp_after:wN \__file_tmp:w \tex_jobname:D
10942                 " \tex_jobname:D " \s_file_stop
10943             }
10944         }
10945       {
10946         \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
10947         \__file_tmp:w
10948       }
10949     }
10950   \cs_if_exist:NT \@currnamestack
10951     {
10952       \tl_if_empty:NF \@currnamestack
10953         { \exp_after:wN \__file_tmp:w \@currnamestack }
10954     }
10955 \group_end:
```

(End of definition for `\g__file_stack_seq`.)

`\g__file_record_seq` The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! We will eventually copy the contents of `\@filelist`.

```
10956 \seq_new:N \g__file_record_seq
```

(End of definition for `\g__file_record_seq`.)

`\l__file_base_name_tl` For storing the basename and full path whilst passing data internally.

```
\l__file_full_name_tl
10957 \tl_new:N \l__file_base_name_tl
10958 \tl_new:N \l__file_full_name_tl
```

(End of definition for `\l__file_base_name_tl` and `\l__file_full_name_tl`.)

`\l__file_dir_str` Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

`\l__file_ext_str`

`\l__file_name_str`

```

10959 \str_new:N \l__file_dir_str
10960 \str_new:N \l__file_ext_str
10961 \str_new:N \l__file_name_str

```

(End of definition for `\l__file_dir_str`, `\l__file_ext_str`, and `\l__file_name_str`.)

`\l_file_search_path_seq` The current search path.

```

10962 \seq_new:N \l_file_search_path_seq

```

(End of definition for `\l_file_search_path_seq`. This variable is documented on page 100.)

`\l__file_tmp_seq` Scratch space for comma list conversion.

```

10963 \seq_new:N \l__file_tmp_seq

```

(End of definition for `\l__file_tmp_seq`.)

50.4.1 Internal auxiliaries

`\s__file_stop` Internal scan marks.

```

10964 \scan_new:N \s__file_stop

```

(End of definition for `\s__file_stop`.)

`\q__file_nil` Internal quarks.

```

10965 \quark_new:N \q__file_nil

```

(End of definition for `\q__file_nil`.)

`__file_quark_if_nil_p:n` Branching quark conditional.

`__file_quark_if_nil:nTF`

```

10966 \__kernel_quark_new_conditional:Nn \__file_quark_if_nil:n { TF }

```

(End of definition for `__file_quark_if_nil:nTF`.)

`\q__file_recursion_tail` Internal recursion quarks.

`\q__file_recursion_stop`

```

10967 \quark_new:N \q__file_recursion_tail
10968 \quark_new:N \q__file_recursion_stop

```

(End of definition for `\q__file_recursion_tail` and `\q__file_recursion_stop`.)

`_file_if_recursion_tail_break:NM` Functions to query recursion quarks.

`_file_if_recursion_tail_stop_do:Nn`

```

10969 \__kernel_quark_new_test:N \_file_if_recursion_tail_stop:N
10970 \__kernel_quark_new_test:N \_file_if_recursion_tail_stop_do:nn

```

(End of definition for `_file_if_recursion_tail_break:NM` and `_file_if_recursion_tail_stop_do:Nn`.)

```

\_kernel_file_name_sanitize:n
\__file_name_expand:n
\_file_name_expand_cleanup:Nw
\_file_name_expand_cleanup:w
\__file_name_expand_end:
\__file_name_expand_error:Nw
\_file_name_expand_error_aux:Nw
\__file_name_strip_quotes:n
\_file_name_strip_quotes:nnw
\_file_name_strip_quotes:nnn
\__file_name_trim_spaces:n
\__file_name_trim_spaces:nw
\_file_name_trim_spaces_aux:n
\_file_name_trim_spaces_aux:w

```

Expanding the file name uses a `\csname`-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using `\ifcsname`. This is less conservative than the token-by-token approach used before, but it is much faster.

```

10971 \cs_new:Npn \__kernel_file_name_sanitize:n #1
10972 {
10973   \exp_args:Ne \__file_name_trim_spaces:n
10974   {
10975     \exp_args:Ne \__file_name_strip_quotes:n
10976     { \__file_name_expand:n {#1} }
10977   }
10978 }

```

We'll use `\cs:w` to start expanding the file name, and to avoid creating csnames equal to `\relax` with “common” names, there's a prefix `__file_name=` to the csname. There's also a guard token at the end so we can check if there was an error during the process and (try to) clean up gracefully.

```

10979 \cs_new:Npn \__file_name_expand:n #1
10980 {
10981   \exp_after:wN \__file_name_expand_cleanup:Nw
10982   \cs:w __file_name = #1 \cs_end:
10983   \__file_name_expand_end:
10984 }

```

With the csname built, we grab it, and grab the remaining tokens delimited by `__file_name_expand_end:`. If there are any remaining tokens, something bad happened, so we'll call the error procedure `__file_name_expand_error:Nw`. If everything went according to plan, then use `\token_to_str:N` on the csname built, and call `__file_name_expand_cleanup:w` to remove the prefix we added a while back. `__file_name_expand_cleanup:w` takes a leading argument so we don't have to bother about the value of `\tex_escapechar:D`.

```

10985 \cs_new:Npn \__file_name_expand_cleanup:Nw #1 #2 \__file_name_expand_end:
10986 {
10987   \tl_if_empty:nF {#2}
10988   { \__file_name_expand_error:Nw #2 \__file_name_expand_end: }
10989   \exp_after:wN \__file_name_expand_cleanup:w \token_to_str:N #1
10990 }
10991 \exp_last_unbraced:NNNNo
10992 \cs_new:Npn \__file_name_expand_cleanup:w #1 \tl_to_str:n { __file_name = } { }

```

In non-error cases `__file_name_expand_end:` should not expand. It will only do so in case there is a `\csname` too much in the file name, so it will throw an error (while expanding), then insert the missing `\cs_end:` and yet another `__file_name_expand_end:` that will be used as a delimiter by `__file_name_expand_cleanup:Nw` (or that will expand again if yet another `\endcsname` is missing).

```

10993 \cs_new:Npn \__file_name_expand_end:
10994 {
10995   \msg_expandable_error:nn
10996   { kernel } { filename-missing-endcsname }
10997   \cs_end: \__file_name_expand_end:
10998 }

```

Now to the error case. `__file_name_expand_error:Nw` adds an extra `\cs_end:` so that in case there was an extra `\csname` in the file name, then `__file_name_expand_error_aux:Nw` throws the error.

```

10999 \cs_new:Npn \__file_name_expand_error:Nw #1 #2 \__file_name_expand_end:
11000   { \__file_name_expand_error_aux:Nw #1 #2 \cs_end: \__file_name_expand_end: }
11001 \cs_new:Npn \__file_name_expand_error_aux:Nw #1 #2 \cs_end: #3
11002   \__file_name_expand_end:
11003   {
11004     \msg_expandable_error:nfff
11005     { kernel } { filename-chars-lost }
11006     { \token_to_str:N #1 } { \exp_stop_f: #2 }
11007   }

```

Quoting file name uses basically the same approach as for `luaquotejobname:` count the " tokens and remove them.

```

11008 \cs_new:Npn \__file_name_strip_quotes:n #1
11009   {
11010     \__file_name_strip_quotes:nw { 0 }
11011     #1 " \q__file_recursion_tail " \q__file_recursion_stop {#1}
11012   }
11013 \cs_new:Npn \__file_name_strip_quotes:nw #1#2 "
11014   {
11015     \if_meaning:w \q__file_recursion_tail #2
11016     \__file_name_strip_quotes_end:w n w n
11017     \fi:
11018     #2
11019     \__file_name_strip_quotes:nw { #1 + 1 }
11020   }
11021 \cs_new:Npn \__file_name_strip_quotes_end:w n w n \fi: #1
11022   \__file_name_strip_quotes:nw #2 \q__file_recursion_stop #3
11023   {
11024     \fi:
11025     \int_if_odd:nT {#2}
11026     {
11027       \msg_expandable_error:nnn
11028       { kernel } { unbalanced-quote-in-filename } {#3}
11029     }
11030   }

```

Spaces need to be trimmed from the start of the name and from the end of any extension. However, the name we are passed might not have an extension: that means we have to look for one. If there is no extension, we still use the standard trimming function but deliberately prevent any spaces being removed at the end.

```

11031 \cs_new:Npn \__file_name_trim_spaces:n #1
11032   { \__file_name_trim_spaces:nw {#1} #1 . \q__file_nil . \s__file_stop }
11033 \cs_new:Npn \__file_name_trim_spaces:nw #1#2 . #3 . #4 \s__file_stop
11034   {
11035     \__file_quark_if_nil:nTF {#3}
11036     {
11037       \tl_trim_spaces_apply:nN { #1 \s__file_stop }
11038       \__file_name_trim_spaces_aux:n
11039     }
11040     { \tl_trim_spaces:n {#1} }
11041   }

```



```

11042 \cs_new:Npn \__file_name_trim_spaces_aux:n #1
11043   { \__file_name_trim_spaces_aux:w #1 }
11044 \cs_new:Npn \__file_name_trim_spaces_aux:w #1 \s__file_stop {#1}

```

(End of definition for __kernel_file_name_sanitize:n and others.)

```

\__kernel_file_name_quote:n
  \__file_name_quote:nw
11045 \cs_new:Npn \__kernel_file_name_quote:n #1
11046   { \__file_name_quote:nw {#1} #1 ~ \q__file_nil \s__file_stop }
11047 \cs_new:Npn \__file_name_quote:nw #1 #2 ~ #3 \s__file_stop
11048   {
11049     \__file_quark_if_nil:nTF {#3}
11050     { #1 }
11051     { "#1" }
11052   }

```

(End of definition for __kernel_file_name_quote:n and __file_name_quote:nw.)

\c__file_marker_tl The same idea as the marker for rescanning token lists: this pair of tokens cannot appear in a file that is being input.

```

11053 \tl_const:Ne \c__file_marker_tl { : \token_to_str:N : }

```

(End of definition for \c__file_marker_tl.)

\file_get:nnNTF The approach here is similar to that for \tl_set_rescan:Nnn. The file contents are grabbed as an argument delimited by \c__file_marker_tl. A few subtleties: braces in **\file_get:VnNTF** \if_false: ... \fi: to deal with possible alignment tabs, \tracingnesting to avoid a warning about a group being closed inside the \scantokens, and \prg_return_true: is placed after the end-of-file marker.

\file_get:nnN
 __file_get_aux:nnN
 __file_get_do:Nw

```

11054 \cs_new_protected:Npn \file_get:nnN #1#2#3
11055   {
11056     \file_get:nnNF {#1} {#2} #3
11057     { \tl_set:Nn #3 { \q_no_value } }
11058   }
11059 \cs_generate_variant:Nn \file_get:nnN { V }
11060 \prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
11061   {
11062     \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11063     {
11064       \exp_args:NV \__file_get_aux:nnN
11065       \l__file_full_name_tl
11066       {#2} #3
11067       \prg_return_true:
11068     }
11069     { \prg_return_false: }
11070   }
11071 \prg_generate_conditional_variant:Nnn \file_get:nnN { V } { T , F , TF }
11072 \cs_new_protected:Npe \__file_get_aux:nnN #1#2#3
11073   {
11074     \exp_not:N \if_false: { \exp_not:N \fi:
11075     \group_begin:
11076       \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
11077       \exp_not:N \exp_args:No \tex_everyeof:D
11078       { \exp_not:N \c__file_marker_tl }

```

```

11079     #2 \scan_stop:
11080     \exp_not:N \exp_after:wN \exp_not:N \__file_get_do:Nw
11081     \exp_not:N \exp_after:wN #3
11082     \exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
11083     \exp_not:N \tex_input:D
11084     \sys_if_engine_luatex:TF
11085     { {#1} }
11086     { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
11087     \exp_not:N \if_false: } \exp_not:N \fi:
11088   }
11089 \exp_args:Nno \use:nn
11090 { \cs_new_protected:Npn \__file_get_do:Nw #1#2 }
11091 { \c__file_marker_tl }
11092 {
11093   \group_end:
11094   \tl_set:No #1 {#2}
11095 }

```

(End of definition for `\file_get:nnNTF` and others. These functions are documented on page 103.)

`__file_size:n` A copy of the primitive where it's available.

```

11096 \cs_new_eq:NN \__file_size:n \tex_filesize:D

```

(End of definition for `__file_size:n`.)

`\file_full_name:n` File searching can be carried out if the `\pdffilesize` primitive or an equivalent is available. That of course means we need to arrange for everything else to here to be done by expansion too. We start off by sanitizing the name and quoting if required: we may need to remove those quotes, so the raw name is passed too.

```

\__file_full_name:n
\__file_full_name_aux:n
\__file_full_name_auxi:nn
\__file_full_name_auxii:nn
\__file_full_name_slash:n
\__file_full_name_slash:w
\__file_full_name_aux:nN
\__file_full_name_aux:nnN
\__file_name_cleanup:w
\__file_name_end:
\__file_name_ext_check:nn
\__file_name_ext_check:nnw
\__file_name_ext_check:nnnw
\__file_name_ext_check:nnnn
\__file_name_ext_check:nnnn

```

```

11097 \cs_new:Npn \file_full_name:n #1
11098 {
11099   \exp_args:Ne \__file_full_name:n
11100   { \__kernel_file_name_sanitize:n {#1} }
11101 }
11102 \cs_generate_variant:Nn \file_full_name:n { V }

```

First, we check if the file is just here: no mapping so we do not need the break part of the broader auxiliary. We are using the fact that the primitive here returns nothing if the file is entirely absent. To avoid unnecessary filesystem lookups, the result of `\pdffilesize` is kept available as an argument. For package mode, `\input@path` is a token list not a sequence.

```

11103 \cs_new:Npn \__file_full_name:n #1
11104 {
11105   \tl_if_blank:nF {#1}
11106   { \exp_args:Nne \__file_full_name_auxii:nn {#1} { \__file_full_name_aux:n {#1} } }
11107 }

```

To avoid repeated reading of files we need to cache the loading: this is important as the code here is used by *all* file checks. The same marker is used in the L^AT_EX_{2 ϵ} kernel, meaning that we get a double-saving with for example `\IfFileExists`. As this is all about performance, we use the low-level approach for the conditionals. For a file already seen, the size is reported as `-1` so it's distinct from any non-cached ones.

```

11108 \cs_new:Npn \__file_full_name_aux:n #1
11109 {

```

```

11110 \if_cs_exist:w __file_seen_ \tl_to_str:n {#1} : \cs_end:
11111 -1
11112 \else:
11113 \exp_args:Ne \__file_full_name_auxi:nn { \__file_size:n {#1} } {#1}
11114 \fi:
11115 }

```

We will need the size of files later, and we have to avoid the `\scan_stop:` causing issues if we are raising the flag. Thus there is a slightly odd gobble here.

```

11116 \cs_new:Npn \__file_full_name_auxi:nn #1#2
11117 {
11118 \if:w \scan_stop: #1 \scan_stop:
11119 \else:
11120 \exp_after:wN \use_none:n
11121 \cs:w __file_seen_ \tl_to_str:n {#2} : \cs_end:
11122 #1
11123 \fi:
11124 }
11125 \cs_new:Npn \__file_full_name_auxii:nn #1 #2
11126 {
11127 \tl_if_blank:nTF {#2}
11128 {
11129 \seq_map_tokens:Nn \l_file_search_path_seq
11130 { \__file_full_name_aux:Nnn \seq_map_break:n {#1} }
11131 \cs_if_exist:NT \input@path
11132 {
11133 \tl_map_tokens:Nn \input@path
11134 { \__file_full_name_aux:Nnn \tl_map_break:n {#1} }
11135 }
11136 \__file_name_end:
11137 }
11138 { \__file_ext_check:nn {#1} {#2} }
11139 }

```

Two pars to the auxiliary here so we can avoid doing quoting twice in the event we find the right file.

```

11140 \cs_new:Npn \__file_full_name_aux:Nnn #1#2#3
11141 {
11142 \exp_args:Ne \__file_full_name_aux:nN
11143 { \__file_full_name_slash:n {#3} #2 }
11144 #1
11145 }
11146 \cs_new:Npn \__file_full_name_slash:n #1
11147 {
11148 \__file_full_name_slash:nw {#1} #1 \q_nil / \q_nil / \q_nil \q_stop
11149 }
11150 \cs_new:Npn \__file_full_name_slash:nw #1#2 / \q_nil / #3 \q_stop
11151 {
11152 \quark_if_nil:nTF {#3}
11153 { #1 / }
11154 { #2 / }
11155 }
11156 \cs_new:Npn \__file_full_name_aux:nN #1
11157 { \exp_args:Nne \__file_full_name_aux:nnN {#1} { \__file_full_name_aux:n {#1} } }
11158 \cs_new:Npn \__file_full_name_aux:nnN #1 #2 #3

```

```

11159 {
11160   \tl_if_blank:nF {#2}
11161   {
11162     #3
11163     {
11164       \__file_ext_check:nn {#1} {#2}
11165       \__file_name_cleanup:w
11166     }
11167   }
11168 }
11169 \cs_new:Npn \__file_name_cleanup:w #1 \__file_name_end: { }
11170 \cs_new:Npn \__file_name_end: { }

```

As T_EX automatically adds .tex if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

```

11171 \cs_new:Npn \__file_ext_check:nn #1 #2
11172 { \__file_ext_check:nnw {#2} { / } #1 / \q__file_nil / \s__file_stop }
11173 \cs_new:Npn \__file_ext_check:nnw #1 #2 #3 / #4 / #5 \s__file_stop
11174 {
11175   \__file_quark_if_nil:nTF {#4}
11176   {
11177     \exp_args:No \__file_ext_check:nnnw
11178     { \use_none:n #2 } {#1} {#3} #3 . \q__file_nil . \s__file_stop
11179   }
11180   { \__file_ext_check:nnw {#1} { #2 #3 / } #4 / #5 \s__file_stop }
11181 }
11182 \cs_new:Npe \__file_ext_check:nnnw #1#2#3#4 . #5 . #6 \s__file_stop
11183 {
11184   \exp_not:N \__file_quark_if_nil:nTF {#5}
11185   {
11186     \exp_not:N \__file_ext_check:nnn
11187     { #1 #3 \tl_to_str:n { .tex } } { #1 #3 } {#2}
11188   }
11189   { #1 #3 }
11190 }
11191 \cs_new:Npn \__file_ext_check:nnn #1
11192 { \exp_args:Nne \__file_ext_check:nnnn {#1} { \__file_full_name_aux:n {#1} } }
11193 \cs_new:Npn \__file_ext_check:nnnn #1#2#3#4
11194 {
11195   \tl_if_blank:nTF {#2}
11196   {#3}
11197   {
11198     \bool_lazy_or:nnTF
11199     { \int_compare_p:nNn {#4} = {#2} }
11200     { \int_compare_p:nNn {#2} = { -1 } }
11201     {#1}
11202     {#3}
11203   }
11204 }

```

(End of definition for \file_full_name:n and others. This function is documented on page 102.)

\file_get_full_name:nN These functions pre-date using \tex_filesize:D for file searching, so are get functions with protection. To avoid having different search set ups, they are simply wrappers

\file_get_full_name:VN

\file_get_full_name:nNTF

\file_get_full_name:VNTF

__file_get_full_name_search:nN

around the code above.

```

11205 \cs_new_protected:Npn \file_get_full_name:nN #1#2
11206 {
11207   \file_get_full_name:nNF {#1} #2
11208   { \tl_set:Nn #2 { \q_no_value } }
11209 }
11210 \cs_generate_variant:Nn \file_get_full_name:nN { V }
11211 \prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
11212 {
11213   \__kernel_tl_set:Nx #2
11214   { \file_full_name:n {#1} }
11215   \tl_if_empty:NTF #2
11216   { \prg_return_false: }
11217   { \prg_return_true: }
11218 }
11219 \prg_generate_conditional_variant:Nnn \file_get_full_name:nN
11220 { V } { T , F , TF }

```

(End of definition for `\file_get_full_name:nN`, `\file_get_full_name:nNTF`, and `__file_get_full_name_search:nN`. These functions are documented on page 102.)

`\g__file_internal_ior` A reserved stream to test for opening a shell.

```

11221 \ior_new:N \g__file_internal_ior

```

(End of definition for `\g__file_internal_ior`.)

`\file_md5five_hash:n` Getting file details by expansion is relatively easy if a bit repetitive. As the MD5 function has a slightly different syntax from the other commands, there is a little cleaning up to do.

```

\file_md5five_hash:n
\file_md5five_hash:V
\file_size:n
\file_size:V
\file_timestamp:n
\file_timestamp:V
\__file_details:nn
\__file_details_aux:nn
\__file_md5five_hash:n
11222 \cs_new:Npn \file_size:n #1
11223 { \__file_details:nn {#1} { size } }
11224 \cs_generate_variant:Nn \file_size:n { V }
11225 \cs_new:Npn \file_timestamp:n #1
11226 { \__file_details:nn {#1} { moddate } }
11227 \cs_generate_variant:Nn \file_timestamp:n { V }
11228 \cs_new:Npn \__file_details:nn #1#2
11229 {
11230   \exp_args:Ne \__file_details_aux:nn
11231   { \file_full_name:n {#1} } {#2}
11232 }
11233 \cs_new:Npn \__file_details_aux:nn #1#2
11234 {
11235   \tl_if_blank:nF {#1}
11236   { \use:c { tex_file #2 :D } {#1} }
11237 }
11238 \cs_new:Npn \file_md5five_hash:n #1
11239 { \exp_args:Ne \__file_md5five_hash:n { \file_full_name:n {#1} } }
11240 \cs_generate_variant:Nn \file_md5five_hash:n { V }
11241 \cs_new:Npn \__file_md5five_hash:n #1
11242 { \tex_md5fivesum:D file {#1} }

```

(End of definition for `\file_md5five_hash:n` and others. These functions are documented on page 101.)

```

\file_hex_dump:nnn
\file_hex_dump:Vnn
  \_file_hex_dump_auxi:nnn
  \_file_hex_dump_auxii:nnnn
  \_file_hex_dump_auxiii:nnnn
  \_file_hex_dump_auxiiv:nnn
  \file_hex_dump:n
  \file_hex_dump:V
  \_file_hex_dump:n

```

These are separate as they need multiple arguments *or* the file size. For LuaTeX, the emulation does not need the file size so we save a little on expansion.

```

11243 \cs_new:Npn \file_hex_dump:nnn #1#2#3
11244 {
11245   \exp_args:Neee \_file_hex_dump_auxi:nnn
11246   { \file_full_name:n {#1} }
11247   { \int_eval:n {#2} }
11248   { \int_eval:n {#3} }
11249 }
11250 \cs_generate_variant:Nn \file_hex_dump:nnn { V }
11251 \cs_new:Npn \_file_hex_dump_auxi:nnn #1#2#3
11252 {
11253   \bool_lazy_any:nF
11254   {
11255     { \tl_if_blank_p:n {#1} }
11256     { \int_compare_p:nNn {#2} = 0 }
11257     { \int_compare_p:nNn {#3} = 0 }
11258   }
11259   {
11260     \exp_args:Ne \_file_hex_dump_auxii:nnnn
11261     { \_file_details_aux:nn {#1} { size } }
11262     {#1} {#2} {#3}
11263   }
11264 }
11265 \cs_new:Npn \_file_hex_dump_auxii:nnnn #1#2#3#4
11266 {
11267   \int_compare:nNnTF {#3} > 0
11268   { \_file_hex_dump_auxiii:nnnn {#3} }
11269   {
11270     \exp_args:Ne \_file_hex_dump_auxiii:nnnn
11271     { \int_eval:n { #1 + #3 } }
11272   }
11273   {#1} {#2} {#4}
11274 }
11275 \cs_new:Npn \_file_hex_dump_auxiii:nnnn #1#2#3#4
11276 {
11277   \int_compare:nNnTF {#4} > 0
11278   { \_file_hex_dump_auxiv:nnn {#4} }
11279   {
11280     \exp_args:Ne \_file_hex_dump_auxiv:nnn
11281     { \int_eval:n { #2 + #4 } }
11282   }
11283   {#1} {#3}
11284 }
11285 \cs_new:Npn \_file_hex_dump_auxiv:nnn #1#2#3
11286 {
11287   \tex_filedump:D
11288   offset ~ \int_eval:n { #2 - 1 } ~
11289   length ~ \int_eval:n { #1 - #2 + 1 }
11290   {#3}
11291 }
11292 \cs_new:Npn \file_hex_dump:n #1
11293 { \exp_args:Ne \_file_hex_dump:n { \file_full_name:n {#1} } }
11294 \cs_generate_variant:Nn \file_hex_dump:n { V }

```

```

11295 \sys_if_engine luatex:TF
11296 {
11297   \cs_new:Npn \__file_hex_dump:n #1
11298     {
11299       \tl_if_blank:nF {#1}
11300       { \tex_dump:D whole {#1} {#1} }
11301     }
11302   }
11303 {
11304   \cs_new:Npn \__file_hex_dump:n #1
11305     {
11306       \tl_if_blank:nF {#1}
11307       { \tex_dump:D length \tex_filesize:D {#1} {#1} }
11308     }
11309 }

```

(End of definition for `\file_hex_dump:nnn` and others. These functions are documented on page 100.)

```

\file_get_hex_dump:nN Non-expandable wrappers around the above in the case where appropriate primitive
\file_get_hex_dump:VN support exists.
\file_get_hex_dump:nNTF 11310 \cs_new_protected:Npn \file_get_hex_dump:nN #1#2
\file_get_hex_dump:VNTF 11311 { \file_get_hex_dump:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\file_get_md5five_hash:nN 11312 \cs_generate_variant:Nn \file_get_hex_dump:nN { V }
\file_get_md5five_hash:VN 11313 \cs_new_protected:Npn \file_get_md5five_hash:nN #1#2
\file_get_md5five_hash:nNTF 11314 { \file_get_md5five_hash:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\file_get_md5five_hash:VNTF 11315 \cs_generate_variant:Nn \file_get_md5five_hash:nN { V }
\file_get_size:nN 11316 \cs_new_protected:Npn \file_get_size:nN #1#2
\file_get_size:VN 11317 { \file_get_size:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\file_get_size:nNTF 11318 \cs_generate_variant:Nn \file_get_size:nN { V }
\file_get_size:VNTF 11319 \cs_new_protected:Npn \file_get_timestamp:nN #1#2
\file_get_timestamp:nN 11320 { \file_get_timestamp:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\file_get_timestamp:VN 11321 \cs_generate_variant:Nn \file_get_timestamp:nN { V }
\file_get_timestamp:nNTF 11322 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nN #1#2 { T , F , TF }
\file_get_timestamp:VNTF 11323 { \__file_get_details:nnN {#1} { hex_dump } #2 }
\__file_get_details:nnN 11324 \prg_generate_conditional_variant:Nnn \file_get_hex_dump:nN
11325 { V } { T , F , TF }
11326 \prg_new_protected_conditional:Npnn \file_get_md5five_hash:nN #1#2 { T , F , TF }
11327 { \__file_get_details:nnN {#1} { md5five_hash } #2 }
11328 \prg_generate_conditional_variant:Nnn \file_get_md5five_hash:nN
11329 { V } { T , F , TF }
11330 \prg_new_protected_conditional:Npnn \file_get_size:nN #1#2 { T , F , TF }
11331 { \__file_get_details:nnN {#1} { size } #2 }
11332 \prg_generate_conditional_variant:Nnn \file_get_size:nN
11333 { V } { T , F , TF }
11334 \prg_new_protected_conditional:Npnn \file_get_timestamp:nN #1#2 { T , F , TF }
11335 { \__file_get_details:nnN {#1} { timestamp } #2 }
11336 \prg_generate_conditional_variant:Nnn \file_get_timestamp:nN
11337 { V } { T , F , TF }
11338 \cs_new_protected:Npn \__file_get_details:nnN #1#2#3
11339 {
11340   \__kernel_tl_set:Nx #3
11341   { \use:c { file_ #2 :n } {#1} }
11342   \tl_if_empty:NTF #3
11343   { \prg_return_false: }

```

```

11344     { \prg_return_true: }
11345 }

```

(End of definition for `\file_get_hex_dump:nNTF` and others. These functions are documented on page 100.)

Custom code due to the additional arguments.

```

\file_get_hex_dump:nnnN
\file_get_hex_dump:VnnN
\file_get_hex_dump:nnnNTF
\file_get_hex_dump:VnnNTF
11346 \cs_new_protected:Npn \file_get_hex_dump:nnnN #1#2#3#4
11347 {
11348   \file_get_hex_dump:nnnNF {#1} {#2} {#3} #4
11349   { \tl_set:Nn #4 { \q_no_value } }
11350 }
11351 \cs_generate_variant:Nn \file_get_hex_dump:nnnN { V }
11352 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nnnN #1#2#3#4
11353 { T , F , TF }
11354 {
11355   \__kernel_tl_set:Nx #4
11356   { \file_hex_dump:nnn {#1} {#2} {#3} }
11357   \tl_if_empty:NTF #4
11358   { \prg_return_false: }
11359   { \prg_return_true: }
11360 }
11361 \prg_generate_conditional_variant:Nnn \file_get_hex_dump:nnnN
11362 { V } { T , F , TF }

```

(End of definition for `\file_get_hex_dump:nnnNTF`. This function is documented on page 100.)

`__file_str_cmp:nn` As we are doing a fixed-length “big” integer comparison, it is easiest to use the low-level behavior of string comparisons.

```

11363 \cs_new_eq:NN \__file_str_cmp:nn \tex_strcmp:D

```

(End of definition for `__file_str_cmp:nn`.)

Comparison of file date can be done by using the low-level nature of the string comparison functions.

```

\file_compare_timestamp:p:nNn
\file_compare_timestamp:p:nNV
\file_compare_timestamp:p:VnN
\file_compare_timestamp:p:VNV
\file_compare_timestamp:nNnTF
\file_compare_timestamp:nNVTF
\file_compare_timestamp:VnNNTF
\file_compare_timestamp:VNVNTF
\__file_compare_timestamp:nnN
\__file_timestamp:n
11364 \prg_new_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
11365 { p , T , F , TF }
11366 {
11367   \exp_args:Nee \__file_compare_timestamp:nnN
11368   { \file_full_name:n {#1} }
11369   { \file_full_name:n {#3} }
11370   #2
11371 }
11372 \prg_generate_conditional_variant:Nnn \file_compare_timestamp:nNn
11373 { nNV , V , VNV } { p , T , F , TF }
11374 \cs_new:Npn \__file_compare_timestamp:nnN #1#2#3
11375 {
11376   \tl_if_blank:nTF {#1}
11377   {
11378     \if_charcode:w #3 <
11379     \prg_return_true:
11380     \else:
11381     \prg_return_false:
11382     \fi:
11383   }

```



```

11384     {
11385         \tl_if_blank:nTF {#2}
11386         {
11387             \if_charcode:w #3 >
11388             \prg_return_true:
11389             \else:
11390             \prg_return_false:
11391             \fi:
11392         }
11393     {
11394         \if_int_compare:w
11395         \__file_str_cmp:nn
11396         { \__file_timestamp:n {#1} }
11397         { \__file_timestamp:n {#2} }
11398         #3 \c_zero_int
11399         \prg_return_true:
11400         \else:
11401         \prg_return_false:
11402         \fi:
11403     }
11404 }
11405 }
11406 \cs_new_eq:NN \__file_timestamp:n \tex_filemoddate:D

```

(End of definition for `\file_compare_timestamp:nNnTF`, `__file_compare_timestamp:nnN`, and `__file_timestamp:n`. This function is documented on page 102.)

`\file_if_exist_p:n` The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located

`\file_if_exist_p:V` then the return value is empty.

`\file_if_exist:nTF`

`\file_if_exist:VTF`

```

11407 \prg_new_conditional:Npnn \file_if_exist:n #1 { p , T , F , TF }
11408 {
11409     \tl_if_blank:eTF { \file_full_name:n {#1} }
11410     { \prg_return_false: }
11411     { \prg_return_true: }
11412 }
11413 \prg_generate_conditional_variant:Nnn \file_if_exist:n { V } { p , T , F , TF }

```

(End of definition for `\file_if_exist:nTF`. This function is documented on page 100.)

`\file_if_exist_input:n` Input of a file with a test for existence. We do not define the T or TF variants because the most useful place to place the `<true code>` would be inconsistent with other conditionals.

`\file_if_exist_input:V`

`\file_if_exist_input:nF`

`\file_if_exist_input:VF`

```

11414 \cs_new_protected:Npn \file_if_exist_input:n #1
11415 {
11416     \file_get_full_name:nNT {#1} \l__file_full_name_tl
11417     { \__file_input:V \l__file_full_name_tl }
11418 }
11419 \cs_generate_variant:Nn \file_if_exist_input:n { V }
11420 \cs_new_protected:Npn \file_if_exist_input:nF #1#2
11421 {
11422     \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11423     { \__file_input:V \l__file_full_name_tl }
11424     {#2}
11425 }
11426 \cs_generate_variant:Nn \file_if_exist_input:nF { V }

```

(End of definition for `\file_if_exist_input:n` and `\file_if_exist_input:nF`. These functions are documented on page 103.)

`\file_input_stop:` A simple rename.

```
11427 \cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }
```

(End of definition for `\file_input_stop:`. This function is documented on page 104.)

`__kernel_file_missing:n` An error message for a missing file, also used in `\ior_open:Nn`.

```
11428 \cs_new_protected:Npn \__kernel_file_missing:n #1
11429 {
11430   \msg_error:nne { kernel } { file-not-found }
11431   { \__kernel_file_name_sanitize:n {#1} }
11432 }
```

(End of definition for `__kernel_file_missing:n`.)

`\file_input:n` Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g__file_stack_seq`, and add it to the file list, either `\g__file_record_seq`, or `\@filelist` in package mode.

`\file_input:V`

`__file_input:n`

`__file_input:V`

`__file_input_push:n`

`__kernel_file_input_push:n`

`__file_input_pop:`

`__kernel_file_input_pop:`

`__file_input_pop:nnn`

```
11433 \cs_new_protected:Npn \file_input:n #1
11434 {
11435   \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11436   { \__file_input:V \l__file_full_name_tl }
11437   { \__kernel_file_missing:n {#1} }
11438 }
11439 \cs_generate_variant:Nn \file_input:n { V }
11440 \cs_new_protected:Npe \__file_input:n #1
11441 {
11442   \exp_not:N \clist_if_exist:NTF \exp_not:N \@filelist
11443   { \exp_not:N \@addtofilelist {#1} }
11444   { \seq_gput_right:Nn \exp_not:N \g__file_record_seq {#1} }
11445   \exp_not:N \__file_input_push:n {#1}
11446   \exp_not:N \tex_input:D
11447   \sys_if_engine luatex:TF
11448   { {#1} }
11449   { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
11450   \exp_not:N \__file_input_pop:
11451 }
11452 \cs_generate_variant:Nn \__file_input:n { V }
```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```
11453 \cs_new_protected:Npn \__file_input_push:n #1
11454 {
11455   \seq_gpush:Ne \g__file_stack_seq
11456   {
11457     { \g_file_curr_dir_str }
11458     { \g_file_curr_name_str }
11459     { \g_file_curr_ext_str }
11460   }
11461   \file_parse_full_name:nNNN {#1}
11462   \l__file_dir_str \l__file_name_str \l__file_ext_str
11463   \str_gset_eq:NN \g_file_curr_dir_str \l__file_dir_str
11464   \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
```

```

11465 \str_gset_eq:NN \g_file_curr_ext_str \l__file_ext_str
11466 }
11467 \cs_new_eq:NN \__kernel_file_input_push:n \__file_input_push:n
11468 \cs_new_protected:Npn \__file_input_pop:
11469 {
11470 \seq_gpop:NN \g_file_stack_seq \l__file_internal_tl
11471 \exp_after:wN \__file_input_pop:nnn \l__file_internal_tl
11472 }
11473 \cs_new_eq:NN \__kernel_file_input_pop: \__file_input_pop:
11474 \cs_new_protected:Npn \__file_input_pop:nnn #1#2#3
11475 {
11476 \str_gset:Nn \g_file_curr_dir_str {#1}
11477 \str_gset:Nn \g_file_curr_name_str {#2}
11478 \str_gset:Nn \g_file_curr_ext_str {#3}
11479 }

```

(End of definition for `\file_input:n` and others. This function is documented on page 103.)

```

\file_input_raw:n No error checking, no tracking.
\file_input_raw:V
\__file_input_raw:nn
11480 \cs_new:Npn \file_input_raw:n #1
11481 { \exp_args:Ne \__file_input_raw:nn { \file_full_name:n {#1} } {#1} }
11482 \cs_generate_variant:Nn \file_input_raw:n { V }
11483 \cs_new:Npe \__file_input_raw:nn #1#2
11484 {
11485 \exp_not:N \tl_if_blank:nTF {#1}
11486 {
11487 \exp_not:N \exp_args:Nnne \exp_not:N \msg_expandable_error:nnn
11488 { kernel } { file-not-found }
11489 { \exp_not:N \__kernel_file_name_sanitize:n {#2} }
11490 }
11491 {
11492 \exp_not:N \tex_input:D
11493 \sys_if_engine luatex:TF
11494 { {#1} }
11495 { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
11496 }
11497 }
11498 \exp_args_generate:n { nne }

```

(End of definition for `\file_input_raw:n` and `__file_input_raw:nn`. This function is documented on page 103.)

`\file_parse_full_name:n` The main parsing macro `\file_parse_full_name_apply:nN` passes the file name #1 through `__kernel_file_name_sanitize:n` so that we have a single normalised way to treat files internally. `\file_parse_full_name:n` uses the former, with `\prg_do_nothing:` to leave each part of the name within a pair of braces.

```

\file_parse_full_name:V
\file_parse_full_name_apply:nN
\file_parse_full_name_apply:VN
11499 \cs_new:Npn \file_parse_full_name:n #1
11500 {
11501 \file_parse_full_name_apply:nN {#1}
11502 \prg_do_nothing:
11503 }
11504 \cs_generate_variant:Nn \file_parse_full_name:n { V }
11505 \cs_new:Npn \file_parse_full_name_apply:nN #1
11506 {

```

```

11507 \exp_args:Ne \_file_parse_full_name_auxi:nN
11508 { \_kernel_file_name_sanitize:n {#1} }
11509 }
11510 \cs_generate_variant:Nn \file_parse_full_name_apply:nN { V }

```

_file_parse_full_name_area:nw splits the file name into chunks separated by /, until the last one is reached. The last chunk is the file name plus the extension, and everything before that is the path. When _file_parse_full_name_area:nw is done, it leaves the path within braces after the scan mark \s__file_stop and proceeds parsing the actual file name.

```

\_file_parse_full_name_auxi:nN
\_file_parse_full_name_area:nw
11511 \cs_new:Npn \_file_parse_full_name_auxi:nN #1
11512 {
11513   \_file_parse_full_name_area:nw { } #1
11514   / \s__file_stop
11515 }
11516 \cs_new:Npn \_file_parse_full_name_area:nw #1 #2 / #3 \s__file_stop
11517 {
11518   \tl_if_empty:nTF {#3}
11519     { \_file_parse_full_name_base:nw { } #2 . \s__file_stop {#1} }
11520     { \_file_parse_full_name_area:nw { #1 / #2 } #3 \s__file_stop }
11521 }

```

_file_parse_full_name_base:nw does roughly the same as above, but it separates the chunks at each period. However here there's some extra complications: In case #1 is empty, it is assumed that the extension is actually empty, and the file name is #2. Besides, an extra . has to be added to #2 because it is later removed in _file_parse_full_name_tidy:nnnN. In any case, if there's an extension, it is returned with a leading ..

```

\_file_parse_full_name_base:nw
11522 \cs_new:Npn \_file_parse_full_name_base:nw #1 #2 . #3 \s__file_stop
11523 {
11524   \tl_if_empty:nTF {#3}
11525     {
11526       \tl_if_empty:nTF {#1}
11527         {
11528           \tl_if_empty:nTF {#2}
11529             { \_file_parse_full_name_tidy:nnnN { } { } }
11530             { \_file_parse_full_name_tidy:nnnN { .#2 } { } }
11531         }
11532         { \_file_parse_full_name_tidy:nnnN {#1} { .#2 } }
11533     }
11534     { \_file_parse_full_name_base:nw { #1 . #2 } #3 \s__file_stop }
11535 }

```

Now we just need to tidy some bits left loose before. The loop used in the two macros above start with a leading / and . in the file path an name, so here we need to remove them, except in the path, if it is a single /, in which case it's left as is. After all's done, pass to #4.

```

11536 \cs_new:Npn \_file_parse_full_name_tidy:nnnN #1 #2 #3 #4
11537 {
11538   \exp_args:Nee #4
11539   {
11540     \str_if_eq:nnF {#3} { / } { \use_none:n }
11541     #3 \prg_do_nothing:

```

```

11542     }
11543     { \use_none:n #1 \prg_do_nothing: }
11544     {#2}
11545   }

```

(End of definition for `\file_parse_full_name:n` and others. These functions are documented on page 103.)

`\file_parse_full_name:nNNN`
`\file_parse_full_name:VNNN`

```

11546 \cs_new_protected:Npn \file_parse_full_name:nNNN #1 #2 #3 #4
11547   {
11548     \file_parse_full_name_apply:nN {#1}
11549     \__file_full_name_assign:nnnNNN #2 #3 #4
11550   }
11551 \cs_new_protected:Npn \__file_full_name_assign:nnnNNN #1 #2 #3 #4 #5 #6
11552   {
11553     \str_set:Nn #4 {#1}
11554     \str_set:Nn #5 {#2}
11555     \str_set:Nn #6 {#3}
11556   }
11557 \cs_generate_variant:Nn \file_parse_full_name:nNNN { V }

```

(End of definition for `\file_parse_full_name:nNNN`. This function is documented on page 102.)

`\file_show_list:` A function to list all files used to the log, without duplicates. In package mode, if
`\file_log_list:` `\@filelist` is still defined, we need to take this list of file names into account (we
`__file_list:N` capture it `\AtBeginDocument` into `\g__file_record_seq`), turning it to a string (this
`__file_list_aux:n` does not affect the commas of this comma list).

```

11558 \cs_new_protected:Npn \file_show_list: { \__file_list:N \msg_show:nneeee }
11559 \cs_new_protected:Npn \file_log_list: { \__file_list:N \msg_log:nneeee }
11560 \cs_new_protected:Npn \__file_list:N #1
11561   {
11562     \seq_clear:N \l__file_tmp_seq
11563     \clist_if_exist:NT \@filelist
11564     {
11565       \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
11566         { \tl_to_str:N \@filelist }
11567     }
11568     \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
11569     \seq_remove_duplicates:N \l__file_tmp_seq
11570     #1 { kernel } { file-list }
11571     { \seq_map_function:NN \l__file_tmp_seq \__file_list_aux:n }
11572     { } { } { }
11573   }
11574 \cs_new:Npn \__file_list_aux:n #1 { \iow_newline: #1 }

```

(End of definition for `\file_show_list:` and others. These functions are documented on page 104.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in `\@filelist` must be turned to strings before being added to `\g__file_record_seq`.

```

11575 \cs_if_exist:NT \@filelist
11576   {
11577     \AtBeginDocument
11578     {

```

```

11579     \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
11580     { \tl_to_str:N \@filelist }
11581     \seq_gconcat:NNN
11582     \g__file_record_seq
11583     \g__file_record_seq
11584     \l__file_tmp_seq
11585   }
11586 }

```

50.5 GetIdInfo

`\GetIdInfo` As documented in `expl3.dtx` this function extracts file name etc from an SVN Id line. This used to be how we got version number and so on in all modules, so it had to be defined in `l3bootstrap`. Now it's more convenient to define it after we have set up quite a lot of tools, and `l3file` seems the least unreasonable place for it.

The idea here is to extract out the information needed from a standard SVN Id line, but to avoid a line that would get changed when the file is checked in. Hence the fact that none of the lines here include both a dollar sign and the Id keyword!

```

11587 \cs_new_protected:Npn \GetIdInfo
11588 {
11589   \tl_clear_new:N \ExplFileDescription
11590   \tl_clear_new:N \ExplFileDate
11591   \tl_clear_new:N \ExplFileName
11592   \tl_clear_new:N \ExplFileExtension
11593   \tl_clear_new:N \ExplFileVersion
11594   \group_begin:
11595   \char_set_catcode_space:n { 32 }
11596   \exp_after:wN
11597   \group_end:
11598   \__file_id_info_auxi:w
11599 }

```

A first check for a completely empty SVN field. If that is not the case, there is a second case when a file created using `svn cp` but has not been checked in. That leaves a special marker `-1` version, which has no further data. Dealing correctly with that is the reason for the space in the line to use `__file_id_info_auxii:w`.

```

11600 \cs_new_protected:Npn \__file_id_info_auxi:w $ #1 $ #2
11601 {
11602   \tl_set:Nn \ExplFileDescription {#2}
11603   \str_if_eq:nnTF {#1} { Id }
11604   {
11605     \tl_set:Nn \ExplFileDate { 0000/00/00 }
11606     \tl_set:Nn \ExplFileName { [unknown] }
11607     \tl_set:Nn \ExplFileExtension { [unknown~extension] }
11608     \tl_set:Nn \ExplFileVersion {-1}
11609   }
11610   { \__file_id_info_auxii:w #1 ~ \s_file_stop }
11611 }

```

Here, `#1` is Id, `#2` is the file name, `#3` is the extension, `#4` is the version, `#5` is the check in date and `#6` is the check in time and user, plus some trailing spaces. If `#4` is the marker `-1` value then `#5` and `#6` are empty.

```

11612 \cs_new_protected:Npn \__file_id_info_auxii:w

```

```

11613     #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s__file_stop
11614   {
11615     \tl_set:Nn \ExplFileName {#2}
11616     \tl_set:Nn \ExplFileExtension {#3}
11617     \tl_set:Nn \ExplFileVersion {#4}
11618     \str_if_eq:nnTF {#4} {-1}
11619     { \tl_set:Nn \ExplFileDate { 0000/00/00 } }
11620     { \__file_id_info_auxiii:w #5 - 0 - 0 - \s__file_stop }
11621   }

```

Convert an SVN-style date into a L^AT_EX-style one.

```

11622 \cs_new_protected:Npn \__file_id_info_auxiii:w #1 - #2 - #3 - #4 \s__file_stop
11623   { \tl_set:Nn \ExplFileDate { #1/#2/#3 } }

```

(End of definition for `\GetIdInfo` and others. This function is documented on page 10.)

50.6 Checking the version of kernel dependencies

```

\__kernel_dependency_version_check:Nn
\__kernel_dependency_version_check:mn
\__file_kernel_dependency_compare:nnm
\__file_parse_version:w

```

This function is responsible for checking if dependencies of the L^AT_EX₃ kernel match the version preloaded in the L^AT_EX_{2_ε} kernel. If versions don't match, the function attempts to tell why by searching for a possible stray format file.

The function starts by checking that the kernel date is defined, and if not zero is used to force the error route. The kernel date is then compared with the argument requested date (usually the packaging date of the dependency). If the kernel date is less than the required date, it's an error and the loading should abort.

```

11624 \cs_new_protected:Npn \__kernel_dependency_version_check:Nn #1
11625   { \exp_args:NV \__kernel_dependency_version_check:mn #1 }
11626 \cs_new_protected:Npn \__kernel_dependency_version_check:mn #1
11627   {
11628     \cs_if_exist:NTF \c__kernel_expl_date_tl
11629     {
11630       \exp_args:NV \__file_kernel_dependency_compare:nnm
11631       \c__kernel_expl_date_tl {#1}
11632     }
11633     { \__file_kernel_dependency_compare:nnm { 0000-00-00 } {#1} }
11634   }
11635 \cs_new_protected:Npn \__file_kernel_dependency_compare:nnm #1 #2 #3
11636   {
11637     \int_compare:nNnT
11638     { \__file_parse_version:w #1 \s__file_stop } <
11639     { \__file_parse_version:w #2 \s__file_stop }
11640     { \__file_mismatched_dependency_error:nn {#2} {#3} }
11641   }
11642 \cs_new:Npn \__file_parse_version:w #1 - #2 - #3 \s__file_stop {#1#2#3}

```

If the versions differ, then we try to give the user some guidance. This function starts by taking the engine name `\c_sys_engine_str` and replacing `tex` by `latex`, then building a command of the form: `kpsewhich -all -engine=<engine> <format>[-dev].fmt` to query the format files available. A shell is opened and each line is read into a sequence.

```

\__file_mismatched_dependency_error:nn

```

```

11643 \cs_new_protected:Npn \__file_mismatched_dependency_error:nn #1 #2
11644   {
11645     \exp_args:NNe \ior_shell_open:Nn \g__file_internal_ior
11646     {

```

```

11647     kpsewhich ~ --all ~
11648     --engine = \c_sys_engine_exec_str
11649     \c_space_tl \c_sys_engine_format_str
11650     \bool_lazy_and:nnT
11651     { \tl_if_exist_p:N \development@branch@name }
11652     { ! \tl_if_empty_p:N \development@branch@name }
11653     { -dev } .fmt
11654   }
11655   \seq_clear:N \l__file_tmp_seq
11656   \ior_map_inline:Nn \g__file_internal_ior
11657   { \seq_put_right:Nn \l__file_tmp_seq {##1} }
11658   \ior_close:N \g__file_internal_ior
11659   \msg_error:nnnn { kernel } { mismatched-support-file }
11660   {#1} {#2}

```

And finish by ending the current file.

```

11661   \tex_endinput:D
11662 }

```

Now define the actual error message:

```

11663 \msg_new:nnnn { kernel } { mismatched-support-file }
11664 {
11665   Mismatched~LaTeX~support~files~detected. \\
11666   Loading~'##2'~aborted!

```

\c__kernel_expl_date_tl may not exist, due to an older format, so only print the dates when the sentinel token list exists:

```

11667   \tl_if_exist:NT \c__kernel_expl_date_tl
11668   {
11669     \\ \\
11670     The~L3~programming~layer~in~the~LaTeX~format \\
11671     is~dated~\c__kernel_expl_date_tl,~but~in~your~TeX~
11672     tree~the~files~require \\ at~least~#1.
11673   }
11674 }
11675 {

```

The sequence containing the format files should have exactly one item: the format file currently being run. If that's the case, the cause of the error is not that, so print a generic help with some possible causes. If more than one format file was found, then print the list to the user, with appropriate indications of what's in the system and what's in the user tree.

```

11676   \int_compare:nNnTF { \seq_count:N \l__file_tmp_seq } > 1
11677   {
11678     The~cause~seems~to~be~an~old~format~file~in~the~user~tree. \\
11679     LaTeX~found~these~files:
11680     \seq_map_tokens:Nn \l__file_tmp_seq { \\---\use:n } \\
11681     Try~deleting~the~file~in~the~user~tree~then~run~LaTeX~again.
11682   }
11683   {
11684     The~most~likely~causes~are:
11685     \\---A~recent~format~generation~failed;
11686     \\---A~stray~format~file~in~the~user~tree~which~needs~
11687     to~be~removed~or~rebuilt;
11688     \\---You~are~running~a~manually~installed~version~of~#2 \\

```



```

11689     \ \ \ which~is~incompatible~with~the~version~in~LaTeX. \ \
11690     }
11691     \ \
11692     LaTeX~will~abort~loading~the~incompatible~support~files~
11693     but~this~may~lead~to \ \ later~errors.~Please~ensure~that~
11694     your~LaTeX~format~is~correctly~regenerated.
11695     }

```

(End of definition for `_kernel_dependency_version_check:Nn` and others.)

50.7 Messages

```

11696 \msg_new:nnnn { kernel } { file-not-found }
11697 { File~'#1'~not~found. }
11698 {
11699     The~requested~file~could~not~be~found~in~the~current~directory,~
11700     in~the~TeX~search~path~or~in~the~LaTeX~search~path.
11701 }
11702 \msg_new:nnn { kernel } { file-list }
11703 {
11704     >~File~List~<
11705     #1 \ \
11706     .....
11707 }
11708 \msg_new:nnnn { kernel } { filename-chars-lost }
11709 { #1~invalid~in~file~name.~Lost:~#2. }
11710 {
11711     There~was~an~invalid~token~in~the~file~name~that~caused~
11712     the~characters~following~it~to~be~lost.
11713 }
11714 \msg_new:nnnn { kernel } { filename-missing-endcsname }
11715 { Missing~\iow_char:N\endcsname~inserted~in~filename. }
11716 {
11717     The~file~name~had~more~\iow_char:N\csname~commands~than~
11718     \iow_char:N\endcsname~ones.~LaTeX~will~add~the~missing~
11719     \iow_char:N\endcsname~and~try~to~continue~as~best~as~it~can.
11720 }
11721 \msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
11722 { Unbalanced~quotes~in~file~name~'#1'. }
11723 {
11724     File~names~must~contain~balanced~numbers~of~quotes~(").
11725 }
11726 \msg_new:nnnn { kernel } { iow-indent }
11727 { Only~#1~allows~#2 }
11728 {
11729     The~command~#2~can~only~be~used~in~messages~
11730     which~will~be~wrapped~using~#1.
11731     \tl_if_empty:nF {#3} { ~ It~was~called~with~argument~'#3'. }
11732 }

```

50.8 Functions delayed from earlier modules

<@@=sys>

`\c_sys_platform_str` Detecting the platform on LuaTeX is easy: for other engines, we use the fact that the two common cases have special null files. It is possible to probe further (see package `platform`), but that requires shell escape and seems unlikely to be useful. This is set up here as it requires file searching.

```

11733 \sys_if_engine luatex:TF
11734 {
11735   \str_const:Ne \c_sys_platform_str
11736   { \tex_directlua:D { tex.print(os.type) } }
11737 }
11738 {
11739   \file_if_exist:nTF { nul: }
11740   {
11741     \file_if_exist:nF { /dev/null }
11742     { \str_const:Nn \c_sys_platform_str { windows } }
11743   }
11744   {
11745     \file_if_exist:nT { /dev/null }
11746     { \str_const:Nn \c_sys_platform_str { unix } }
11747   }
11748 }
11749 \cs_if_exist:NF \c_sys_platform_str
11750 { \str_const:Nn \c_sys_platform_str { unknown } }

```

(End of definition for `\c_sys_platform_str`. This variable is documented on page 76.)

`\sys_if_platform_unix_p:` We can now set up the tests.

```

\sys_if_platform_unix:TF 11751 \clist_map_inline:nn { unix , windows }
\sys_if_platform_windows_p: 11752 {
\sys_if_platform_windows:TF 11753   \__file_const:nn { sys_if_platform_ #1 }
11754   { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
11755 }

```

(End of definition for `\sys_if_platform_unix:TF` and `\sys_if_platform_windows:TF`. These functions are documented on page 76.)

```

11756 </package>

```

Chapter 51

l3luatex implementation

```
11757 (*package)
```

51.1 Breaking out to Lua

```
11758 (*tex)
```

```
11759 (@@=lua)
```

```
\__lua_escape:n Copies of primitives.
  \__lua_now:n   11760 \cs_new_eq:NN \__lua_escape:n \tex_luaescapestring:D
\__lua_shipout:n 11761 \cs_new_eq:NN \__lua_now:n \tex_directlua:D
                  11762 \cs_new_eq:NN \__lua_shipout:n \tex_latelua:D
```

(End of definition for __lua_escape:n, __lua_now:n, and __lua_shipout:n.)

These functions are set up in l3str for bootstrapping: we want to replace them with a “proper” version at this stage, so clean up.

```
11763 \cs_undefine:N \lua_escape:e
```

```
11764 \cs_undefine:N \lua_now:e
```

```
\lua_now:n Wrappers around the primitives.
  \lua_now:e 11765 \cs_new:Npn \lua_now:e #1 { \__lua_now:n {#1} }
\lua_shipout_e:n 11766 \cs_new:Npn \lua_now:n #1 { \lua_now:e { \exp_not:n {#1} } }
  \lua_shipout:n 11767 \cs_new_protected:Npn \lua_shipout_e:n #1 { \__lua_shipout:n {#1} }
  \lua_escape:n 11768 \cs_new_protected:Npn \lua_shipout:n #1
  \lua_escape:e 11769 { \lua_shipout_e:n { \exp_not:n {#1} } }
                  11770 \cs_new:Npn \lua_escape:e #1 { \__lua_escape:n {#1} }
                  11771 \cs_new:Npn \lua_escape:n #1 { \lua_escape:e { \exp_not:n {#1} } }
```

(End of definition for \lua_now:n and others. These functions are documented on page 105.)

```
\lua_load_module:n Wrapper around require'(module)'.
```

```
11772 \str_new:N \l__lua_err_msg_str
11773 \cs_new_protected:Npn \lua_load_module:n #1
11774 {
11775   \bool_if:nF { \__lua_load_module_p:n { #1 } }
11776   {
11777     \msg_error:nnnV
11778     { luatex } { module-not-found } { #1 } \l__lua_err_msg_str
11779   }
11780 }
```

(End of definition for `\lua_load_module:n`. This function is documented on page 106.)

As with engines other than LuaTeX these have to be macros, we give them the same status in all cases. When LuaTeX is not in use, simply give an error message/

```

11781 \sys_if_engine luatex:F
11782 {
11783   \clist_map_inline:nn
11784     {
11785       \lua_escape:n , \lua_escape:e ,
11786       \lua_now:n , \lua_now:e
11787     }
11788     {
11789       \cs_gset:Npn #1 ##1
11790         {
11791           \msg_expandable_error:nnn
11792             { luatex } { luatex-required } { #1 }
11793         }
11794     }
11795   \clist_map_inline:nn
11796     { \lua_shipout_e:n , \lua_shipout:n , \lua_load_module:n }
11797     {
11798       \cs_gset_protected:Npn #1 ##1
11799         {
11800           \msg_error:nnn
11801             { luatex } { luatex-required } { #1 }
11802         }
11803     }
11804 }

```

51.2 Messages

```

11805 \msg_new:nnnn { luatex } { luatex-required }
11806   { LuaTeX-engine-not-in-use!~Ignoring~#1. }
11807   {
11808     The~feature~you~are~using~is~only~available~
11809     with~the~LuaTeX~engine.~LaTeX3~ignored~'~#1'~.
11810   }
11811
11812 \msg_new:nnnn { luatex } { module-not-found }
11813   { Lua~module~'~#1'~not~found. }
11814   {
11815     The~file~'~#1.lua'~could~not~be~found.~Please~ensure~
11816     that~the~file~was~properly~installed~and~that~the~
11817     filename~database~is~current. \\ \\
11818     The~Lua~loader~provided~this~additional~information: \\
11819     #2
11820   }
11821
11822 \prop_gput:Nnn \g_msg_module_name_prop { luatex } { LaTeX }
11823 \prop_gput:Nnn \g_msg_module_type_prop { luatex } { }
11824 </tex>

```

51.3 Lua functions for internal use

```
11825 (*lua)
```

Most of the emulation of pdfTeX here is based heavily on Heiko Oberdiek's pdfTeX-cmds package.

`ltx.utils` Create a table for the kernel's own use.

```
11826 ltx = ltx or {utils={}}
11827 ltx.utils = ltx.utils or { }
11828 local ltxutils = ltx.utils
```

(End of definition for `ltx.utils`. This function is documented on page 106.)

Local copies of global tables.

```
11829 local io      = io
11830 local kpse    = kpse
11831 local lfs     = lfs
11832 local math    = math
11833 local md5     = md5
11834 local os      = os
11835 local string  = string
11836 local tex     = tex
11837 local texio   = texio
11838 local tonumber = tonumber
```

Local copies of standard functions.

```
11839 local abs      = math.abs
11840 local byte     = string.byte
11841 local floor    = math.floor
11842 local format   = string.format
11843 local gsub     = string.gsub
11844 local lfs_attr = lfs.attributes
11845 local open     = io.open
11846 local os_date  = os.date
11847 local setcatcode = tex.setcatcode
11848 local sprint   = tex.sprint
11849 local cprint   = tex.cprint
11850 local write    = tex.write
11851 local write_nl = texio.write_nl
11852 local utf8_char = utf8.char
11853 local package_loaded = package.loaded
11854 local package_searchers = package.searchers
11855 local table_concat = table.concat
11856
11857 local scan_int      = token.scan_int or token.scan_integer
11858 local scan_string  = token.scan_string
11859 local scan_keyword = token.scan_keyword
11860 local put_next     = token.put_next
11861 local token_create = token.create
11862 local token_new    = token.new
11863 local set_macro    = token.set_macro
```

Since `token.create` only returns useful values after the tokens has been added to TeX's hash table, we define a variant which defines it first if necessary.

```
11864 local token_create_safe
11865 do
11866   local is_defined = token.is_defined
11867   local set_char   = token.set_char
```

```

11868 local runtoks = tex.runtoks
11869 local let_token = token_create'let'
11870
11871 function token_create_safe(s)
11872   local orig_token = token_create(s)
11873   if is_defined(s, true) then
11874     return orig_token
11875   end
11876   set_char(s, 0)
11877   local new_token = token_create(s)
11878   runtoks(function()
11879     put_next(let_token, new_token, orig_token)
11880   end)
11881   return new_token
11882 end
11883 end
11884
11885 local true_tok = token_create_safe'prg_return_true:'
11886 local false_tok = token_create_safe'prg_return_false:'

```

In ConT_EXt `lmtx.token.command_id` does not exist, but it can easily be emulated with ConT_EXt's `tokens.commands`.

```

11887 local command_id = token.command_id
11888 if not command_id and tokens and tokens.commands then
11889   local id_map = tokens.commands
11890   function command_id(name)
11891     return id_map[name]
11892   end
11893 end

```

Deal with ConT_EXt: doesn't use `kpse` library.

```

11894 local kpse_find = (resolvers and resolvers.findfile) or kpse.find_file

```

`escapehex` An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in `pdftexcmds` but is not currently required here.

```

11895 local function escapehex(str)
11896   return (gsub(str, ".",
11897     function (ch) return format("%02X", byte(ch)) end))
11898 end

```

(End of definition for `escapehex`.)

`ltx.utils.filedump` Similar comments here to the next function: read the file in binary mode to avoid any line-end weirdness.

```

11899 local function filedump(name,offset,length)
11900   local file = kpse_find(name,"tex",true)
11901   if not file then return end
11902   local f = open(file,"rb")
11903   if not f then return end
11904   if offset and offset > 0 then
11905     f:seek("set", offset)
11906   end
11907   local data = f:read(length or 'a')
11908   f:close()

```

```

11909   return escapehex(data)
11910 end
11911 ltxutils.filedump = filedump

```

(End of definition for `ltx.utils.filedump`. This function is documented on page 106.)

`md5.HEX` Hash a string and return the hash in uppercase hexadecimal format. In some engines, this is built-in. For traditional LuaTeX, the conversion to hexadecimal has to be done by us.

```

11912 local md5_HEX = md5.HEX
11913 if not md5_HEX then
11914   local md5_sum = md5.sum
11915   function md5_HEX(data)
11916     return escapehex(md5_sum(data))
11917   end
11918   md5.HEX = md5_HEX
11919 end

```

(End of definition for `md5.HEX`.)

`ltx.utils.filemd5sum` Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see `pdftexcmds` and how it handles strings that have passed through LuaTeX). The file is read in binary mode so that no line ending normalisation occurs.

```

11920 local function filemd5sum(name)
11921   local file = kpse_find(name, "tex", true) if not file then return end
11922   local f = open(file, "rb") if not f then return end
11923
11924   local data = f:read("*a")
11925   f:close()
11926   return md5_HEX(data)
11927 end
11928 ltxutils.filemd5sum = filemd5sum

```

(End of definition for `ltx.utils.filemd5sum`. This function is documented on page 106.)

`ltx.utils.filemoddate` There are two cases: If the C standard library is C99 compliant, we can use `%z` to get the timezone in almost the right format. We only have to add primes and replace a zero or missing offset with Z.

Of course this would be boring, so Windows does things differently. There we have to manually calculate the offset. See procedure `makepdftime` in `utils.c` of `pdfTeX`.

```

11929 local filemoddate
11930 if os_date'%z':match'^[+-]%d%d%d$d$' then
11931   local pattern = lpeg.Cs(16 *
11932     (lpeg.Cg(lpeg.S'+-' * '0000' * lpeg.Cc'Z')
11933     + 3 * lpeg.Cc'"' * 2 * lpeg.Cc'"'
11934     + lpeg.Cc'Z')
11935     * -1)
11936   function filemoddate(name)
11937     local file = kpse_find(name, "tex", true)
11938     if not file then return end
11939     local date = lfs_attr(file, "modification")
11940     if not date then return end
11941     return pattern:match(os_date("D:%Y%m%d%H%M%S%z", date))

```

```

11942 end
11943 else
11944 local function filemoddate(name)
11945     local file = kpse_find(name, "tex", true)
11946     if not file then return end
11947     local date = lfs_attr(file, "modification")
11948     if not date then return end
11949     local d = os_date("*t", date)
11950     local u = os_date("!*t", date)
11951     local off = 60 * (d.hour - u.hour) + d.min - u.min
11952     if d.year ~= u.year then
11953         if d.year > u.year then
11954             off = off + 1440
11955         else
11956             off = off - 1440
11957         end
11958     elseif d.yday ~= u.yday then
11959         if d.yday > u.yday then
11960             off = off + 1440
11961         else
11962             off = off - 1440
11963         end
11964     end
11965     local timezone
11966     if off == 0 then
11967         timezone = "Z"
11968     else
11969         if off < 0 then
11970             timezone = "-"
11971             off = -off
11972         else
11973             timezone = "+"
11974         end
11975         timezone = format("%s%02d'%02d'", timezone, hours // 60, hours % 60)
11976     end
11977     return format("D:%04d%02d%02d%02d%02d%02d%s",
11978         d.year, d.month, d.day, d.hour, d.min, d.sec, timezone)
11979 end
11980 end
11981 ltxutils.filemoddate = filemoddate

```

(End of definition for ltx.utils.filemoddate. This function is documented on page 106.)

ltx.utils.filesize A simple disk lookup.

```

11982 local function filesize(name)
11983     local file = kpse_find(name, "tex", true)
11984     if file then
11985         local size = lfs_attr(file, "size")
11986         if size then
11987             return size
11988         end
11989     end
11990 end
11991 ltxutils.filesize = filesize

```


(End of definition for `ltx.utils.filesize`. This function is documented on page 107.)

`luaedef` An internal function for defining control sequences from Lua which behave like primitives. This acts as a wrapper around `token.set_lua` which accepts a function instead of an index into the functions table.

```
11992 local luacmd do
11993   local set_lua = token.set_lua
11994   local undefined_cs = command_id'undefined_cs'
11995
11996   if not context and not luatexbase then require'ltluaux' end
11997   if luatexbase then
11998     local new_luafunction = luatexbase.new_luafunction
11999     local functions = lua.get_functions_table()
12000     function luacmd(name, func, ...)
12001       local id
12002       local tok = token_create(name)
12003       if tok.command == undefined_cs then
12004         id = new_luafunction(name)
12005         set_lua(name, id, ...)
12006       else
12007         id = tok.index or tok.mode
12008       end
12009       functions[id] = func
12010     end
12011   elseif context then
12012     local register = context.functions.register
12013     local functions = context.functions.known
12014     function luacmd(name, func, ...)
12015       local tok = token_create(name)
12016       if tok.command == undefined_cs then
12017         token.set_lua(name, register(func), ...)
12018       else
12019         functions[tok.index or tok.mode] = func
12020       end
12021     end
12022   end
12023 end
```

(End of definition for `luaedef`.)

`try_require` Loads a Lua module. This function loads the module similarly to the standard Lua global function `require`, with a few differences. On success, `try_require` returns `true`, `module`. If the module cannot be found, it returns `false`, `err_msg`. If the module is found, but something goes wrong when loading it, the function throws an error.

```
12024 local function try_require(name)
12025   if package_loaded[name] then
12026     return true, package_loaded[name]
12027   end
12028
12029   local failure_details = {}
12030   for _, searcher in ipairs(package_searchers) do
12031     local loader, data = searcher(name)
12032     if type(loader) == 'function' then
12033       package_loaded[name] = loader(name, data) or true

```

```

12034     return true, package_loaded[name]
12035   elseif type(loader) == 'string' then
12036     failure_details[#failure_details + 1] = loader
12037   end
12038 end
12039
12040 return false, table_concat(failure_details, '\n')
12041 end

```

(End of definition for `try_require`.)

`_lua_load_module_p:n` Check to see if we can load a module using `require`. If we can load the module, then we load it immediately. Otherwise, we save the error message in `\l_@@_err_msg_str`.

```

12042 local char_given = command_id'char_given'
12043 local c_true_bool = token_create(1, char_given)
12044 local c_false_bool = token_create(0, char_given)
12045 local c_str_cctab = token_create('c_str_cctab').mode
12046
12047 luacmd('\_lua_load_module_p:n', function()
12048   local success, result = try_require(scan_string())
12049   if success then
12050     set_macro(c_str_cctab, 'l\_lua_err_msg_str', '')
12051     put_next(c_true_bool)
12052   else
12053     set_macro(c_str_cctab, 'l\_lua_err_msg_str', result)
12054     put_next(c_false_bool)
12055   end
12056 end)

```

(End of definition for `_lua_load_module_p:n`.)

51.4 Preserving iniTeX Lua data for runs

```
12057 (@@=lua)
```

The Lua state is not dumped when a format is written, therefore any Lua variables filled doing format building need to be restored in order to be accessible during normal runs.

We provide some kernel-internal helpers for this. They will only be available if `luatexbase` is available. This is not a big restriction though, because `ConTeXt` (which does not use `luatexbase`) does not load `expl3` in the format.

```

12058 local register_luadata, get_luadata
12059
12060 if luatexbase then
12061   local register = token_create'expl@luadata@bytecode'.index
12062   if status.ini_version then

```

`register_luadata` `register_luadata` is only available during format generation. It accept a string which uniquely identifies the data object and has to be provided to retrieve it later. Additionally it accepts a function which is called in the `pre_dump` callback and which has to return a string that evaluates to a valid Lua object to be preserved.

```

12063     local luadata, luadata_order = {}, {}
12064
12065     function register_luadata(name, func)

```

```

12066     if luadata[name] then
12067         error(format("LaTeX error: data name %q already in use", name))
12068     end
12069     luadata[name] = func
12070     luadata_order[#luadata_order + 1] = func and name
12071 end

```

(End of definition for register_luadata.)

The actual work is done in `pre_dump`. The `luadata_order` is used to ensure that the order is consistent over multiple runs.

```

12072     luatexbase.add_to_callback("pre_dump", function()
12073         if next(luadata) then
12074             local str = "return {"
12075             for i=1, #luadata_order do
12076                 local name = luadata_order[i]
12077                 str = format('%s[%q]=%s,', str, name, luadata[name]())
12078             end
12079             lua.bytecode[register] = assert(load(str .. "}")
12080         end
12081     end, "ltx.luadata")
12082 else

```

`get_luadata` `get_luadata` is only available if data should be restored. It accept the identifier which was used when the data object was registered and returns the associated object. Every object can only be retrieved once.

```

12083     local luadata = lua.bytecode[register]
12084     if luadata then
12085         lua.bytecode[register] = nil
12086         luadata = luadata()
12087     end
12088     function get_luadata(name)
12089         if not luadata then return end
12090         local data = luadata[name]
12091         luadata[name] = nil
12092         return data
12093     end
12094 end
12095 end

```

(End of definition for get_luadata.)

```

12096 </lua>
12097 </package>

```

Chapter 52

13legacy implementation

```
12098 <*package>
12099 <@@=legacy>

\legacy_if_p:n A friendly wrapper. We need to use the \if:w approach here, rather than testing
\legacy_if:nTF against \iftrue/\iffalse as the latter approach fails for primitive conditionals such
as \ifmmode. The \reverse_if:N here means that we get a slightly more useful error if
the name is undefined.
```

```
12100 \prg_new_conditional:Npnm \legacy_if:n #1 { p , T , F , TF }
12101 {
12102   \exp_after:wN \reverse_if:N
12103   \cs:w if#1 \cs_end:
12104   \prg_return_false:
12105   \else:
12106   \prg_return_true:
12107   \fi:
12108 }
```

(End of definition for \legacy_if:nTF. This function is documented on page 108.)

```
\legacy_if_set_true:n A friendly wrapper.
\legacy_if_set_false:n 12109 \cs_new_protected:Npn \legacy_if_set_true:n #1
\legacy_if_gset_true:n 12110 { \cs_set_eq:cN { if#1 } \if_true: }
\legacy_if_gset_false:n 12111 \cs_new_protected:Npn \legacy_if_set_false:n #1
12112 { \cs_set_eq:cN { if#1 } \if_false: }
12113 \cs_new_protected:Npn \legacy_if_gset_true:n #1
12114 { \cs_gset_eq:cN { if#1 } \if_true: }
12115 \cs_new_protected:Npn \legacy_if_gset_false:n #1
12116 { \cs_gset_eq:cN { if#1 } \if_false: }
```

(End of definition for \legacy_if_set_true:n and others. These functions are documented on page 108.)

```
\legacy_if_set:nn A more elaborate wrapper.
\legacy_if_gset:nn 12117 \cs_new_protected:Npn \legacy_if_set:nn #1#2
12118 {
12119   \bool_if:nTF {#2} \legacy_if_set_true:n \legacy_if_set_false:n
12120   {#1}
12121 }
```

```
12122 \cs_new_protected:Npn \legacy_if_gset:nn #1#2
12123   {
12124     \bool_if:nTF {#2} \legacy_if_gset_true:n \legacy_if_gset_false:n
12125     {#1}
12126   }
```

(End of definition for \legacy_if_set:nn and \legacy_if_gset:nn. These functions are documented on page 108.)

```
12127 \endpackage
```

Chapter 53

l3tl implementation

```
12128 (*package)
12129 (@@=tl)
```

A token list variable is a \TeX macro that holds tokens. By using the ε - \TeX primitive `\unexpanded` inside a \TeX `\edef` it is possible to store any tokens, including `#`, in this way.

53.1 Functions

`__kernel_tl_set:Nx` These two are supplied to get better performance for macros which would otherwise use `\tl_set:Ne` or `\tl_gset:Ne` internally.

```
12130 \cs_new_eq:NN \__kernel_tl_set:Nx \cs_set_nopar:Npe
12131 \cs_new_eq:NN \__kernel_tl_gset:Nx \cs_gset_nopar:Npe
```

(End of definition for `__kernel_tl_set:Nx` and `__kernel_tl_gset:Nx`.)

`\tl_new:N` Creating new token list variables is a case of checking for an existing definition and doing the definition.

`\tl_new:c`

```
12132 \cs_new_protected:Npn \tl_new:N #1
12133 {
12134   \__kernel_chk_if_free_cs:N #1
12135   \cs_gset_eq:NN #1 \c_empty_tl
12136 }
12137 \cs_generate_variant:Nn \tl_new:N { c }
```

(End of definition for `\tl_new:N`. This function is documented on page 110.)

`\tl_const:Nn` Constants are also easy to generate. They use `\cs_gset_nopar:Npe` instead of `__kernel_tl_gset:Nx` so that the correct scope checking for `c`, instead of for `g`, is applied when `\debug_on:n { check-declarations }` is used. Constant assignment functions are patched specially in `l3debug` to apply such checks.

`\tl_const:Nx`
`\tl_const:cn`
`\tl_const:ce`
`\tl_const:cx`

```
12138 \cs_new_protected:Npn \tl_const:Nn #1#2
12139 {
12140   \__kernel_chk_if_free_cs:N #1
12141   \cs_gset_nopar:Npe #1 { \__kernel_exp_not:w {#2} }
12142 }
12143 \cs_generate_variant:Nn \tl_const:Nn { Ne , c , ce }
12144 \cs_generate_variant:Nn \tl_const:Nn { Nx , cx }
```

(End of definition for `\tl_const:Nn`. This function is documented on page 111.)

`\tl_clear:N` Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\tl_clear:c
\tl_gclear:N
\tl_gclear:c
12145 \cs_new_protected:Npn \tl_clear:N #1
12146   { \tex_let:D #1 = ~ \c_empty_tl }
12147 \cs_new_protected:Npn \tl_gclear:N #1
12148   { \tex_global:D \tex_let:D #1 ~ \c_empty_tl }
12149 \cs_generate_variant:Nn \tl_clear:N { c }
12150 \cs_generate_variant:Nn \tl_gclear:N { c }
```

(End of definition for `\tl_clear:N` and `\tl_gclear:N`. These functions are documented on page 111.)

`\tl_clear_new:N` Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\tl_clear_new:c
\tl_gclear_new:N
\tl_gclear_new:c
12151 \cs_new_protected:Npn \tl_clear_new:N #1
12152   { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
12153 \cs_new_protected:Npn \tl_gclear_new:N #1
12154   { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
12155 \cs_generate_variant:Nn \tl_clear_new:N { c }
12156 \cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(End of definition for `\tl_clear_new:N` and `\tl_gclear_new:N`. These functions are documented on page 111.)

`\tl_set_eq:NN` For setting token list variables equal to each other. To allow for patching, the arguments have to be explicit. In addition this ensures that a braced second argument will not cause problems.

```
\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cc
\tl_gset_eq:NN
\tl_gset_eq:Nc
\tl_gset_eq:cN
\tl_gset_eq:cc
12157 \cs_new_protected:Npn \tl_set_eq:NN #1#2
12158   { \tex_let:D #1 = ~ #2 }
12159 \cs_new_protected:Npn \tl_gset_eq:NN #1#2
12160   { \tex_global:D \tex_let:D #1 = ~ #2 }
12161 \cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
12162 \cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }
```

(End of definition for `\tl_set_eq:NN` and `\tl_gset_eq:NN`. These functions are documented on page 111.)

`\tl_concat:NNN` Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list #2 or #3 equal to `\scan_stop:` would lead to problems later on.

```
\tl_concat:ccc
\tl_gconcat:NNN
\tl_gconcat:ccc
12163 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
12164   {
12165     \__kernel_tl_set:Nx #1
12166     {
12167       \__kernel_exp_not:w \exp_after:wN {#2}
12168       \__kernel_exp_not:w \exp_after:wN {#3}
12169     }
12170   }
12171 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
12172   {
12173     \__kernel_tl_gset:Nx #1
12174     {
12175       \__kernel_exp_not:w \exp_after:wN {#2}
12176       \__kernel_exp_not:w \exp_after:wN {#3}

```

```

12177     }
12178   }
12179 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
12180 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

```

(End of definition for `\tl_concat:NNN` and `\tl_gconcat:NNN`. These functions are documented on page 111.)

`\tl_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

12181 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
12182 \prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }

```

`\tl_if_exist:NTF`

`\tl_if_exist:cTF`

(End of definition for `\tl_if_exist:NTF`. This function is documented on page 111.)

53.2 Constant token lists

`\c_empty_tl` Never full. We need to define that constant before using `\tl_new:N`.

```

12183 \tl_const:Nn \c_empty_tl { }

```

(End of definition for `\c_empty_tl`. This variable is documented on page 125.)

`\c_novalue_tl` A special marker: as we don't have `\char_generate:nn` yet, has to be created the old-fashioned way.

```

12184 \group_begin:
12185 \tex_catcode:D '- = 11 ~
12186 \tl_const:Ne \c_novalue_tl { - NoValue \token_to_str:N - }
12187 \group_end:

```

(End of definition for `\c_novalue_tl`. This variable is documented on page 126.)

`\c_space_tl` A space as a token list (as opposed to as a character).

```

12188 \tl_const:Nn \c_space_tl { ~ }

```

(End of definition for `\c_space_tl`. This variable is documented on page 126.)

53.3 Adding to token list variables

`\tl_set:Nn` By using `\exp_not:n` token list variables can contain `#` tokens, which makes the token list registers provided by `TEX` more or less redundant. The `\tl_set:No` version is done by hand as it is used quite a lot.

```

12189 \cs_new_protected:Npn \tl_set:Nn #1#2
12190   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w {#2} } }
12191 \cs_new_protected:Npn \tl_set:No #1#2
12192   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
12193 \cs_new_protected:Npn \tl_gset:Nn #1#2
12194   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w {#2} } }
12195 \cs_new_protected:Npn \tl_gset:No #1#2
12196   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
12197 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Ne , Nf }
12198 \cs_generate_variant:Nn \tl_set:Nn { c , cV , cv , ce , cf }
12199 \cs_generate_variant:Nn \tl_set:No { c }
12200 \cs_generate_variant:Nn \tl_set:Nn { Nx , cx }
12201 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Ne , Nf }

```

`\tl_gset:NV`

`\tl_gset:Nv`

`\tl_gset:No`

`\tl_gset:Ne`

`\tl_gset:Nf`

`\tl_gset:Nx`

`\tl_gset:cn`

`\tl_gset:cV`

`\tl_gset:cv`

`\tl_gset:co`


```

12202 \cs_generate_variant:Nn \tl_gset:Nn { c, cV , cv , ce , cf }
12203 \cs_generate_variant:Nn \tl_gset:No { c }
12204 \cs_generate_variant:Nn \tl_gset:Nn { Nx , cx }

```

(End of definition for \tl_set:Nn and \tl_gset:Nn. These functions are documented on page 111.)

```

\tl_put_left:Nn Adding to the left is done directly to gain a little performance.
\tl_put_left:NV 12205 \cs_new_protected:Npn \tl_put_left:Nn #1#2
\tl_put_left:Nv 12206 {
\tl_put_left:Ne 12207   \__kernel_tl_set:Nx #1
\tl_put_left:No 12208   { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
\tl_put_left:Nx 12209 }
\tl_put_left:cn 12210 \cs_new_protected:Npn \tl_put_left:Nv #1#2
\tl_put_left:cV 12211 {
\tl_put_left:cv 12212   \__kernel_tl_set:Nx #1
\tl_put_left:ce 12213   { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
\tl_put_left:co 12214 }
\tl_put_left:cx 12215 \cs_new_protected:Npn \tl_put_left:Nv #1#2
\tl_gput_left:Nn 12216 {
\tl_gput_left:NV 12217   \__kernel_tl_set:Nx #1
\tl_gput_left:Nv 12218   { \exp_not:v {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
\tl_gput_left:Ne 12219 }
\tl_gput_left:No 12220 \cs_new_protected:Npn \tl_gput_left:Ne #1#2
\tl_gput_left:Nx 12221 {
\tl_gput_left:cn 12222   \__kernel_tl_set:Nx #1
\tl_gput_left:cV 12223   {
\tl_gput_left:cv 12224     \__kernel_exp_not:w \tex_expanded:D { {#2} }
\tl_gput_left:ce 12225     \__kernel_exp_not:w \exp_after:wN {#1}
\tl_gput_left:co 12226   }
\tl_gput_left:cx 12227 }
\tl_gput_left:co 12228 \cs_new_protected:Npn \tl_gput_left:No #1#2
\tl_gput_left:cx 12229 {
12230   \__kernel_tl_set:Nx #1
12231   {
12232     \__kernel_exp_not:w \exp_after:wN {#2}
12233     \__kernel_exp_not:w \exp_after:wN {#1}
12234   }
12235 }
12236 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
12237 {
12238   \__kernel_tl_gset:Nx #1
12239   { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
12240 }
12241 \cs_new_protected:Npn \tl_gput_left:Nv #1#2
12242 {
12243   \__kernel_tl_gset:Nx #1
12244   { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
12245 }
12246 \cs_new_protected:Npn \tl_gput_left:Nv #1#2
12247 {
12248   \__kernel_tl_gset:Nx #1
12249   { \exp_not:v {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
12250 }
12251 \cs_new_protected:Npn \tl_gput_left:Ne #1#2

```

```

12252 {
12253   \__kernel_tl_gset:Nx #1
12254   {
12255     \__kernel_exp_not:w \tex_expanded:D { {#2} }
12256     \__kernel_exp_not:w \exp_after:wN {#1}
12257   }
12258 }
12259 \cs_new_protected:Npn \tl_gput_left:No #1#2
12260 {
12261   \__kernel_tl_gset:Nx #1
12262   {
12263     \__kernel_exp_not:w \exp_after:wN {#2}
12264     \__kernel_exp_not:w \exp_after:wN {#1}
12265   }
12266 }
12267 \cs_generate_variant:Nn \tl_put_left:Nn { c }
12268 \cs_generate_variant:Nn \tl_put_left:NV { c }
12269 \cs_generate_variant:Nn \tl_put_left:Nv { c }
12270 \cs_generate_variant:Nn \tl_put_left:Ne { c }
12271 \cs_generate_variant:Nn \tl_put_left:No { c }
12272 \cs_generate_variant:Nn \tl_put_left:Nn { Nx, cx }
12273 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
12274 \cs_generate_variant:Nn \tl_gput_left:NV { c }
12275 \cs_generate_variant:Nn \tl_gput_left:Nv { c }
12276 \cs_generate_variant:Nn \tl_gput_left:Ne { c }
12277 \cs_generate_variant:Nn \tl_gput_left:No { c }
12278 \cs_generate_variant:Nn \tl_gput_left:Nn { Nx , cx }

```

(End of definition for `\tl_put_left:Nn` and `\tl_gput_left:Nn`. These functions are documented on page 111.)

`\tl_put_right:Nn` The same on the right.

```

\tl_put_right:NV 12279 \cs_new_protected:Npn \tl_put_right:Nn #1#2
\tl_put_right:Nv 12280 { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
\tl_put_right:Ne 12281 \cs_new_protected:Npn \tl_put_right:NV #1#2
\tl_put_right:No 12282 {
\tl_put_right:Nx 12283   \__kernel_tl_set:Nx #1
\tl_put_right:cn 12284   { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:v #2 }
\tl_put_right:cV 12285 }
\tl_put_right:cv 12286 \cs_new_protected:Npn \tl_put_right:Nv #1#2
\tl_put_right:ce 12287 {
\tl_put_right:co 12288   \__kernel_tl_set:Nx #1
\tl_put_right:cx 12289   { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:v {#2} }
12290 }
\tl_gput_right:Nn 12291 \cs_new_protected:Npn \tl_put_right:Ne #1#2
\tl_gput_right:NV 12292 {
\tl_gput_right:Nv 12293   \__kernel_tl_set:Nx #1
\tl_gput_right:Ne 12294   {
\tl_gput_right:No 12295     \__kernel_exp_not:w \exp_after:wN {#1}
\tl_gput_right:Nx 12296     \__kernel_exp_not:w \tex_expanded:D { {#2} }
12297   }
\tl_gput_right:cn 12298 }
\tl_gput_right:cV 12299 \cs_new_protected:Npn \tl_put_right:No #1#2
\tl_gput_right:cv 12300 {
\tl_gput_right:ce
\tl_gput_right:co
\tl_gput_right:cx

```

```

12301     \__kernel_tl_set:Nx #1
12302     {
12303         \__kernel_exp_not:w \exp_after:wN {#1}
12304         \__kernel_exp_not:w \exp_after:wN {#2}
12305     }
12306 }
12307 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
12308 { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
12309 \cs_new_protected:Npn \tl_gput_right:NV #1#2
12310 {
12311     \__kernel_tl_gset:Nx #1
12312     { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
12313 }
12314 \cs_new_protected:Npn \tl_gput_right:Nv #1#2
12315 {
12316     \__kernel_tl_gset:Nx #1
12317     { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:v {#2} }
12318 }
12319 \cs_new_protected:Npn \tl_gput_right:Ne #1#2
12320 {
12321     \__kernel_tl_gset:Nx #1
12322     {
12323         \__kernel_exp_not:w \exp_after:wN {#1}
12324         \__kernel_exp_not:w \tex_expanded:D { {#2} }
12325     }
12326 }
12327 \cs_new_protected:Npn \tl_gput_right:No #1#2
12328 {
12329     \__kernel_tl_gset:Nx #1
12330     {
12331         \__kernel_exp_not:w \exp_after:wN {#1}
12332         \__kernel_exp_not:w \exp_after:wN {#2}
12333     }
12334 }
12335 \cs_generate_variant:Nn \tl_put_right:Nn { c }
12336 \cs_generate_variant:Nn \tl_put_right:NV { c }
12337 \cs_generate_variant:Nn \tl_put_right:Nv { c }
12338 \cs_generate_variant:Nn \tl_put_right:Ne { c }
12339 \cs_generate_variant:Nn \tl_put_right:No { c }
12340 \cs_generate_variant:Nn \tl_put_right:Nn { Nx , cx }
12341 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
12342 \cs_generate_variant:Nn \tl_gput_right:NV { c }
12343 \cs_generate_variant:Nn \tl_gput_right:Nv { c }
12344 \cs_generate_variant:Nn \tl_gput_right:Ne { c }
12345 \cs_generate_variant:Nn \tl_gput_right:No { c }
12346 \cs_generate_variant:Nn \tl_gput_right:Nn { Nx, cx }

```

(End of definition for \tl_put_right:Nn and \tl_gput_right:Nn. These functions are documented on page 112.)

53.4 Internal quarks and quark-query functions

```

\q__tl_nil Internal quarks.
\q__tl_mark
\q__tl_stop

```

```

12347 \quark_new:N \q__tl_nil
12348 \quark_new:N \q__tl_mark
12349 \quark_new:N \q__tl_stop

```

(End of definition for `\q__tl_nil`, `\q__tl_mark`, and `\q__tl_stop`.)

```

\q__tl_recursion_tail Internal recursion quarks.
\q__tl_recursion_stop 12350 \quark_new:N \q__tl_recursion_tail
                      12351 \quark_new:N \q__tl_recursion_stop

```

(End of definition for `\q__tl_recursion_tail` and `\q__tl_recursion_stop`.)

```

\_tl_if_recursion_tail break:nN Functions to query recursion quarks.
\_tl_if_recursion_tail_stop p:n 12352 \__kernel_quark_new_test:N \_tl_if_recursion_tail_break:nN
\_tl_if_recursion_tail_stop:nTF 12353 \__kernel_quark_new_conditional:Nn \_tl_quark_if_nil:n { TF }

```

(End of definition for `_tl_if_recursion_tail_break:nN` and `_tl_if_recursion_tail_stop:nTF`.)

53.5 Reassigning token list category codes

`\c__tl_rescan_marker_tl` The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

```

12354 \tl_const:Ne \c__tl_rescan_marker_tl { : \token_to_str:N : }

```

(End of definition for `\c__tl_rescan_marker_tl`.)

```

\tl_set_rescan:Nnn In a group, after some initial setup explained below and the user setup #3 (followed by
\tl_set_rescan:NnV \scan_stop: to be safe), there is a call to \__tl_set_rescan:nNN. This shared auxiliary
\tl_set_rescan:Nne defined later distinguishes single-line and multi-line “files”. In the simplest case of multi-
\tl_set_rescan:Nno line files, it calls (with the same arguments) \__tl_set_rescan_multi:nNN, whose code
\tl_set_rescan:Nnx is included here to help understand the approach. This function rescans its argument #1,
\tl_set_rescan:cnm closes the group, and performs the assignment.
\tl_set_rescan:cnV
\tl_set_rescan:cne
\tl_set_rescan:cno
\tl_set_rescan:cnx

```

One difficulty when rescanning is that `\scantokens` treats the argument as a file, and without the correct settings a `TeX` error occurs:

```

! File ended while scanning definition of ...

```

```

\tl_gset_rescan:Nnn A related minor issue is a warning due to opening a group before the \scantokens and
\tl_gset_rescan:NnV closing it inside that temporary file; we avoid that by setting \tracingnesting. The
\tl_gset_rescan:Nne standard solution to the “File ended” error is to grab the rescanned tokens as a delimited
\tl_gset_rescan:Nno argument of an auxiliary, here \_tl_rescan:NNw, that performs the assignment, then let
\tl_gset_rescan:Nnx TeX “execute” the end of file marker. As usual in delimited arguments we use \prg_do_
\tl_gset_rescan:cnm nothing: to avoid stripping an outer set braces: this is removed by using o-expanding
\tl_gset_rescan:cnV assignments. The delimiter cannot appear within the rescanned token list because it
\tl_gset_rescan:cne contains twice the same character, with different catcodes.
\tl_gset_rescan:cno
\tl_gset_rescan:cnx

```

For `\tl_rescan:nn` we cannot simply call `__tl_set_rescan:NNnn \prg_do_nothing: \use:n` because that would leave the end-of-file marker *after* the result of rescanning. If that rescanned result is code that looks further in the input stream for arguments, it would break.

For multi-line files the only subtlety is that `\newlinechar` should be equal to `\endlinechar` because `\newlinechar` characters become new lines and then become

```

\tl_rescan:nn
\tl_rescan:nV
\_tl_rescan_aux:
\_tl_set_rescan:NNnn
\_tl_set_rescan_multi:nNN
\_tl_rescan:NNw

```

`\endlinechar` characters when writing to an abstract file and reading back. This equality is ensured by setting `\newlinechar` equal to `\endlinechar`. Prior to this, `\endlinechar` is set to `-1` if it was `32` (in particular true after `\ExplSyntaxOn`) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode 10) are ignored at the beginning of lines, and spaces and tabs (character code 32 and 9) are ignored at the end of lines.

The two `\if_false: ... \fi:` are there to prevent alignment tabs to cause a change of tabular cell while rescanning. We put the “opening” one after `\group_begin:` so that if one accidentally f-expands `\tl_set_rescan:Nnn` braces remain balanced. This is essential in e-type arguments when `\expanded` is not available.

```

12355 \cs_new_protected:Npn \tl_rescan:nn #1#2
12356   {
12357     \tl_set_rescan:Nnn \l__tl_internal_a_tl {#1} {#2}
12358     \exp_after:wN \__tl_rescan_aux:
12359     \l__tl_internal_a_tl
12360   }
12361 \cs_generate_variant:Nn \tl_rescan:nn { nV }
12362 \exp_args:NNo \cs_new_protected:Npn \__tl_rescan_aux:
12363   { \tl_clear:N \l__tl_internal_a_tl }
12364 \cs_new_protected:Npn \tl_set_rescan:Nnn
12365   { \__tl_set_rescan:NNnn \tl_set:No }
12366 \cs_new_protected:Npn \tl_gset_rescan:Nnn
12367   { \__tl_set_rescan:NNnn \tl_gset:No }
12368 \cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
12369   {
12370     \group_begin:
12371     \if_false: { \fi:
12372       \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
12373       \int_compare:nNnT \tex_endlinechar:D = { 32 }
12374       { \int_set:Nn \tex_endlinechar:D { -1 } }
12375       \int_set_eq:NN \tex_newlinechar:D \tex_endlinechar:D
12376       #3 \scan_stop:
12377       \exp_args:No \__tl_set_rescan:nNN { \tl_to_str:n {#4} } #1 #2
12378     \if_false: } \fi:
12379   }
12380 \cs_new_protected:Npn \__tl_set_rescan_multi:nNN #1#2#3
12381   {
12382     \tex_everyeof:D \exp_after:wN { \c__tl_rescan_marker_tl }
12383     \exp_after:wN \__tl_rescan:NNw
12384     \exp_after:wN #2
12385     \exp_after:wN #3
12386     \exp_after:wN \prg_do_nothing:
12387     \tex_scantokens:D {#1}
12388   }
12389 \exp_args:Nno \use:nn
12390   { \cs_new:Npn \__tl_rescan:NNw #1#2#3 } \c__tl_rescan_marker_tl
12391   {
12392     \group_end:
12393     #1 #2 {#3}
12394   }
12395 \cs_generate_variant:Nn \tl_set_rescan:Nnn { NnV , Nne , c , cnV , cne }
12396 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx , cno , cnx }

```

```

12397 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { NnV , Nne , c , cnV , cne }
12398 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx , cno , cnx }

```

(End of definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 125.)

```

\__tl_set_rescan:nNN
\__tl_set_rescan_single:nnNN
\__tl_set_rescan_single_aux:nnnNN
\__tl_set_rescan_single_aux:w

```

The function `__tl_set_rescan:nNN` calls `__tl_set_rescan_multi:nNN` or `__tl_set_rescan_single:nnNN { ' }` depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a `\newlinechar` character. If `\newlinechar` is out of range, the argument is assumed to be a single line.

For a single line, no `\endlinechar` should be added, so it is set to `-1`, and spaces should not be removed. Trailing spaces and tabs are a difficult matter, as `TEX` removes these at a very low level. The only way to preserve them is to rescan not the argument but the argument followed by a character with a reasonable category code. Here, `11` (letter) and `12` (other) are accepted, as these are convenient, suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from `'` (ASCII 39) to `~` (ASCII 127). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the number of steps needed by the loop (most often just a single one). If no valid character is found (very rare), fall-back on `__tl_set_rescan_multi:nNN`.

Otherwise, once a valid character is found (let us use `'` in this explanation) run some code very similar to `__tl_set_rescan_multi:nNN` but with `'` added at both ends of the input. Of course, we need to define the auxiliary `__tl_set_rescan_single:NNww` on the fly to remove the additional `'` that is just before `::` (by which we mean `\c__tl_set_rescan_marker_tl`). Note that the argument must be delimited by `'` with the current catcode; this is done thanks to `\char_generate:nn`. Yet another issue is that the rescanned token list may contain a comment character, in which case the `'` we expected is not there. We fix this as follows: rather than just `::` we set `\everyeof to ::{<code1>}'::{<code2>}` `\s__tl_stop`. The auxiliary `__tl_set_rescan_single:NNww` runs the `o`-expanding assignment, expanding either `<code1>` or `<code2>` before its the main argument `#3`. In the typical case without comment character, `<code1>` is expanded, removing the leading `'`. In the rarer case with comment character, `<code2>` is expanded, calling `__tl_set_rescan_single_aux:w`, which removes the trailing `::{<code1>}` and the leading `'`.

```

12399 \cs_new_protected:Npn \__tl_set_rescan:nNN #1
12400   {
12401     \int_compare:nNnTF \tex_newlinechar:D < 0
12402       { \use_ii:nn }
12403       {
12404         \exp_args:Nnf \tl_if_in:nnTF {#1}
12405           { \char_generate:nn { \tex_newlinechar:D } { 12 } }
12406       }
12407     { \__tl_set_rescan_multi:nNN }
12408     {
12409       \int_set:Nn \tex_endlinechar:D { -1 }
12410       \__tl_set_rescan_single:nnNN { ' ' }
12411     }
12412     {#1}
12413   }
12414 \cs_new_protected:Npn \__tl_set_rescan_single:nnNN #1

```

```

12415 {
12416   \int_compare:nNnTF
12417     { \char_value_catcode:n {#1} / 2 } = 6
12418     {
12419       \exp_args:Nof \__tl_set_rescan_single_aux:nnnNN
12420         \c__tl_rescan_marker_tl
12421         { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
12422     }
12423     {
12424       \int_compare:nNnTF {#1} < { '\~ }
12425       {
12426         \exp_args:Nf \__tl_set_rescan_single:nnNN
12427           { \int_eval:n { #1 + 1 } }
12428       }
12429       { \__tl_set_rescan_multi:nNN }
12430     }
12431 }
12432 \cs_new_protected:Npn \__tl_set_rescan_single_aux:nnnNN #1#2#3#4#5
12433 {
12434   \tex_everyeof:D
12435   {
12436     #1 \use_none:n
12437     #2 #1 { \exp:w \__tl_set_rescan_single_aux:w }
12438     \s__tl_stop
12439   }
12440   \cs_set:Npn \__tl_rescan:NNw ##1##2##3 #2 #1 ##4 ##5 \s__tl_stop
12441   {
12442     \group_end:
12443     ##1 ##2 { ##4 ##3 }
12444   }
12445   \exp_after:wN \__tl_rescan:NNw
12446   \exp_after:wN #4
12447   \exp_after:wN #5
12448   \tex_scantokens:D { #2 #3 #2 }
12449 }
12450 \exp_args:Nno \use:nn
12451 { \cs_new:Npn \__tl_set_rescan_single_aux:w #1 }
12452 \c__tl_rescan_marker_tl #2
12453 { \use_i:nn \exp_end: #1 }

```

(End of definition for `__tl_set_rescan:nNN` and others.)

53.6 Modifying token list variables

`\tl_replace_once:Nnn` All of the replace functions call `__tl_replace:NnNNNnn` with appropriate arguments. `\tl_replace_once:NVn` The first two arguments are explained later. The next controls whether the replacement function calls itself (`__tl_replace_next:w`) or stops (`__tl_replace_wrap:w`) after the first replacement. Next comes an e-type assignment function `\tl_set:Ne` or `\tl_gset:Ne` for local or global replacements. Finally, the three arguments `<tl var>` `{<pattern>}` `{<replacement>}` provided by the user. When describing the auxiliary functions below, we denote the contents of the `<tl var>` by `<token list>`.

```

12454 \cs_new_protected:Npn \tl_replace_once:Nnn
12455   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_set:Nx }

```

`\tl_replace_once:NnV`
`\tl_replace_once:Nen`
`\tl_replace_once:Nne`
`\tl_replace_once:Nee`
`\tl_replace_once:Nxn`
`\tl_replace_once:Nnx`
`\tl_replace_once:Nxx`
`\tl_replace_once:cnn`
`\tl_replace_once:cVn`
`\tl_replace_once:cnV`
`\tl_replace_once:cen`
`\tl_replace_once:cne`
`\tl_replace_once:cee`
`\tl_replace_once:cxn`
`\tl_replace_once:cnx`
`\tl_replace_once:cxx`
`\tl_greplace_once:Nnn`

```

12456 \cs_new_protected:Npn \tl_greplace_once:Nnn
12457   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_gset:Nx }
12458 \cs_new_protected:Npn \tl_replace_all:Nnn
12459   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_set:Nx }
12460 \cs_new_protected:Npn \tl_greplace_all:Nnn
12461   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_gset:Nx }
12462 \cs_generate_variant:Nn \tl_replace_once:Nnn
12463   { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
12464 \cs_generate_variant:Nn \tl_replace_once:Nnn
12465   { Nx , Nnx , Nxx , cxn , cnx , cxx }
12466 \cs_generate_variant:Nn \tl_greplace_once:Nnn
12467   { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
12468 \cs_generate_variant:Nn \tl_greplace_once:Nnn
12469   { Nx , Nnx , Nxx , cxn , cnx , cxx }
12470 \cs_generate_variant:Nn \tl_replace_all:Nnn
12471   { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
12472 \cs_generate_variant:Nn \tl_replace_all:Nnn
12473   { Nx , Nnx , Nxx , cxn , cnx , cxx }
12474 \cs_generate_variant:Nn \tl_greplace_all:Nnn
12475   { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
12476 \cs_generate_variant:Nn \tl_greplace_all:Nnn
12477   { Nx , Nnx , Nxx , cxn , cnx , cxx }

```

(End of definition for `\tl_replace_once:Nnn` and others. These functions are documented on page 123.)

```

\__tl_replace:NnNNNnn
\__tl_replace_auxi:NnnNNNnn
\__tl_replace_auxii:nNNNnn
\__tl_replace_next:w
\__tl_replace_next_aux:w
\__tl_replace_wrap:w

```

To implement the actual replacement auxiliary `__tl_replace_auxii:nNNNnn` we need a *delimiter* with the following properties:

- all occurrences of the `<pattern> #6` in “`<token list> <delimiter>`” belong to the `<token list>` and have no overlap with the `<delimiter>`,
- the first occurrence of the `<delimiter>` in “`<token list> <delimiter>`” is the trailing `<delimiter>`.

We first find the building blocks for the *delimiter*, namely two tokens `<A>` and `` such that `<A>` does not appear in `#6` and `#6` is not `` (this condition is trivial if `#6` has more than one token). Then we consider the delimiters “`<A>`” and “`<A> <A>n <A>n `”, for $n \geq 1$, where `<A>n` denotes n copies of `<A>`, and we choose as our *delimiter* the first one which is not in the `<token list>`.

Every delimiter in the set obeys the first condition: `#6` does not contain `<A>` hence cannot be overlapping with the `<token list>` and the *delimiter*, and it cannot be within the *delimiter* since it would have to be in one of the two `` hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the *delimiter* we choose does not appear in the `<token list>`. Additionally, the set of delimiters is such that a `<token list>` of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a *delimiter* with at most $O(n^{1/2})$ tokens in a time at most $O(n^{3/2})$. Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “`<A>`” in the list of delimiters to try, so that the *delimiter* is simply `\q__tl_mark` in the most common situation where neither the `<token list>` nor the `<pattern>` contains `\q__tl_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty `<pattern> #6` is an error, and if `#1` is absent from both the `<token list> #5`

and the $\langle pattern \rangle$ #6 then we can use it as the $\langle delimiter \rangle$ through `__tl_replace_auxii:nNNNnn {#1}`. Otherwise, we end up calling `__tl_replace:NnNNNnn` repeatedly with the first two arguments `\q__tl_mark {?}`, `\? {??}`, `\?? {???`, and so on, until #6 does not contain the control sequence #1, which we take as our $\langle A \rangle$. The argument #2 only serves to collect ? characters for #1. Note that the order of the tests means that the first two are done every time, which is wasteful (for instance, we repeatedly test for the emptiness of #6). However, this is rare enough not to matter. Finally, choose $\langle B \rangle$ to be `\q__tl_nil` or `\q__tl_stop` such that it is not equal to #6.

The `__tl_replace_auxi:NnnNNNnn` auxiliary receives $\{\langle A \rangle\}$ and $\{\langle A \rangle^n \langle B \rangle\}$ as its arguments, initially with $n = 1$. If “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ” is in the $\langle token list \rangle$ then increase n and try again. Once it is not anymore in the $\langle token list \rangle$ we take it as our $\langle delimiter \rangle$ and pass this to the `auxii` auxiliary.

```

12478 \cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
12479 {
12480   \tl_if_empty:nTF {#6}
12481     {
12482       \msg_error:nne { kernel } { empty-search-pattern }
12483       { \tl_to_str:n {#7} }
12484     }
12485     {
12486       \tl_if_in:onTF { #5 #6 } {#1}
12487       {
12488         \tl_if_in:nnTF {#6} {#1}
12489         { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
12490         {
12491           \__tl_quark_if_nil:nTF {#6}
12492           { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q__tl_stop } }
12493           { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q__tl_nil } }
12494         }
12495       }
12496       { \__tl_replace_auxii:nNNNnn {#1} }
12497       #3#4#5 {#6} {#7}
12498     }
12499 }
12500 \cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
12501 {
12502   \tl_if_in:NnTF #1 { #2 #3 #3 }
12503   { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
12504   { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
12505 }

```

The auxiliary `__tl_replace_auxii:nNNNnn` receives the following arguments:

$$\{\langle delimiter \rangle\} \langle function \rangle \langle assignment \rangle$$

$$\langle tl var \rangle \{\langle pattern \rangle\} \{\langle replacement \rangle\}$$

All of its work is done between `\group_align_safe_begin:` and `\group_align_safe_end:` to avoid issues in alignments. It does the actual replacement within #3 #4 {...}, an e-expanding $\langle assignment \rangle$ #3 to the $\langle tl var \rangle$ #4. The auxiliary `__tl_replace_next:w` is called, followed by the $\langle token list \rangle$, some tokens including the $\langle delimiter \rangle$ #1, followed by the $\langle pattern \rangle$ #5. This auxiliary finds an argument delimited by #5 (the presence of a trailing #5 avoids runaway arguments) and calls `__tl_replace_wrap:w` to test whether this #5 is found within the $\langle token list \rangle$ or is the trailing one.

If on the one hand it is found within the $\langle token list \rangle$, then $\##1$ cannot contain the $\langle delimiter \rangle \##1$ that we worked so hard to obtain, thus $_tl_replace_wrap:w$ gets $\##1$ as its own argument $\##1$, and protects it against the e-expanding assignment. It also finds $_exp_not:n$ as $\##2$ and does nothing to it, thus letting through $_exp_not:n \{ \langle replacement \rangle \}$ into the assignment. Note that $_tl_replace_next:w$ and $_tl_replace_wrap:w$ are always called followed by two empty brace groups. These are safe because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of $_exp_not:o$ and $_use_none:nn$, rather than simply $_exp_not:n$. Afterwards, $_tl_replace_next:w$ is called to repeat the replacement, or $_tl_replace_wrap:w$ if we only want a single replacement. In this second case, $\##1$ is the $\langle remaining tokens \rangle$ in the $\langle token list \rangle$ and $\##2$ is some $\langle ending code \rangle$ which ends the assignment and removes the trailing tokens $\##5$ using some $_if_false: \{ _fi: \}$ trickery because $\##5$ may contain any delimiter.

If on the other hand the argument $\##1$ of $_tl_replace_next:w$ is delimited by the trailing $\langle pattern \rangle \##5$, then $\##1$ is “ $\{ \} \{ \} \langle token list \rangle \langle delimiter \rangle \{ \langle ending code \rangle \}$ ”, hence $_tl_replace_wrap:w$ finds “ $\{ \} \{ \} \langle token list \rangle$ ” as $\##1$ and the $\langle ending code \rangle$ as $\##2$. It leaves the $\langle token list \rangle$ into the assignment and unbraces the $\langle ending code \rangle$ which removes what remains (essentially the $\langle delimiter \rangle$ and $\langle replacement \rangle$).

```

12506 \cs_new_protected:Npn \_tl\_replace\_auxii:nNNNnn #1#2#3#4#5#6
12507 {
12508   \group\_align\_safe\_begin:
12509   \cs\_set:Npn \_tl\_replace\_wrap:w ##1 #1 ##2
12510     { \_kernel\_exp\_not:w \exp\_after:wN { \_use\_none:nn ##1 } ##2 }
12511   \cs\_set:Npe \_tl\_replace\_next:w ##1 #5
12512   {
12513     \exp\_not:N \_tl\_replace\_wrap:w ##1
12514     \exp\_not:n { #1 }
12515     \exp\_not:n { \exp\_not:n {#6} }
12516     \exp\_not:n { #2 { } { } }
12517   }
12518   #3 #4
12519   {
12520     \exp\_after:wN \_tl\_replace\_next\_aux:w
12521     #4
12522     #1
12523     {
12524       \_if\_false: { \_fi: }
12525       \exp\_after:wN \_use\_none:n \exp\_after:wN { \_if\_false: } \_fi:
12526     }
12527     #5
12528   }
12529   \group\_align\_safe\_end:
12530 }
12531 \cs\_new:Npn \_tl\_replace\_next\_aux:w { \_tl\_replace\_next:w { } { } }
12532 \cs\_new\_eq:NN \_tl\_replace\_wrap:w ?
12533 \cs\_new\_eq:NN \_tl\_replace\_next:w ?

```

(End of definition for $_tl_replace:NnNNNnn$ and others.)

$_tl_remove_once:Nn$ Removal is just a special case of replacement.

```

\tl\_remove\_once:NV 12534 \cs\_new_protected:Npn \tl\_remove\_once:Nn #1#2
\tl\_remove\_once:Ne 12535 { \tl\_replace\_once:Nnn #1 {#2} { } }
\tl\_remove\_once:cn 12536 \cs\_new_protected:Npn \tl\_gremove\_once:Nn #1#2
\tl\_remove\_once:cV
\tl\_remove\_once:ce
\tl\_gremove\_once:Nn
\tl\_gremove\_once:NV
\tl\_gremove\_once:cn
\tl\_gremove\_once:cV

```

```

12537 { \tl_greplce_once:Nnn #1 {#2} { } }
12538 \cs_generate_variant:Nn \tl_remove_once:Nn { NV , Ne , c , cV , ce }
12539 \cs_generate_variant:Nn \tl_gremove_once:Nn { NV , Ne , c , cV , ce }

```

(End of definition for `\tl_remove_once:Nn` and `\tl_gremove_once:Nn`. These functions are documented on page 124.)

```

\tl_remove_all:Nn Removal is just a special case of replacement.
\tl_remove_all:NV 12540 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
\tl_remove_all:Ne 12541 { \tl_replace_all:Nnn #1 {#2} { } }
\tl_remove_all:Nx 12542 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
\tl_remove_all:cn 12543 { \tl_greplce_all:Nnn #1 {#2} { } }
\tl_remove_all:cV 12544 \cs_generate_variant:Nn \tl_remove_all:Nn { NV , Ne , c , cV , ce }
\tl_remove_all:ce 12545 \cs_generate_variant:Nn \tl_remove_all:Nn { Nx , cx }
\tl_remove_all:cx 12546 \cs_generate_variant:Nn \tl_gremove_all:Nn { NV , Ne , c , cV , ce }
\tl_gremove_all:Nn 12547 \cs_generate_variant:Nn \tl_gremove_all:Nn { Nx , cx }

```

(End of definition for `\tl_remove_all:Nn` and `\tl_gremove_all:Nn`. These functions are documented on page 124.)

53.7 Token list conditionals

These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```

\tl_if_empty_p:N 12548 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
\tl_if_empty_p:cx 12549 {
\tl_if_empty_p:c 12550   \if_meaning:w #1 \c_empty_tl
12551   \prg_return_true:
12552   \else:
12553   \prg_return_false:
12554   \fi:
12555 }
\tl_if_empty:NTF 12556 \prg_generate_conditional_variant:Nnn \tl_if_empty:N
\tl_if_empty:cTF 12557 { c } { p , T , F , TF }

```

(End of definition for `\tl_if_empty:NTF`. This function is documented on page 112.)

`\tl_if_empty_p:n` The `\if:w` triggers the expansion of `\tl_to_str:n` which converts the argument to a string: this is empty if and only if the argument is. Then `\if:w \scan_stop: ... \scan_stop:` is true if and only if the string ... is empty. It could be tempting to use `\if:w \scan_stop: #1 \scan_stop:` directly. But this fails on a token list expanding to anything starting with `\scan_stop:` leaving everything that follows in the input stream.

```

12558 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
12559 {
12560   \if:w \scan_stop: \tl_to_str:n {#1} \scan_stop:
12561   \prg_return_true:
12562   \else:
12563   \prg_return_false:
12564   \fi:
12565 }
12566 \prg_generate_conditional_variant:Nnn \tl_if_empty:n
12567 { V , e } { p , TF , T , F }

```

(End of definition for `\tl_if_empty:nTF`. This function is documented on page 112.)

`\tl_if_empty_p:o` The auxiliary function `__tl_if_empty_if:o` is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on `\tl_if_empty:nTF`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in several places. We don't put `\prg_return_true:` and so on in the definition of the auxiliary, because that would prevent an optimization applied to conditionals that end with this code. Also the `\@@_if_empty_if:o` is expanded once in `\tl_if_empty:oTF` for efficiency as well (and to reduce code doubling).

```

12568 \cs_new:Npn \__tl_if_empty_if:o #1
12569   {
12570     \if:w \scan_stop: \__kernel_tl_to_str:w \exp_after:wN {#1} \scan_stop:
12571   }
12572 \exp_args:Nno \use:n
12573 { \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F } }
12574 {
12575   \__tl_if_empty_if:o {#1}
12576   \prg_return_true:
12577   \else:
12578   \prg_return_false:
12579   \fi:
12580 }

```

(End of definition for `\tl_if_empty:nTF` and `__tl_if_empty_if:o`. This function is documented on page 112.)

`\tl_if_blank_p:n` TeX skips spaces when reading a non-delimited arguments. Thus, a `<token list>` is blank if and only if `\use_none:n <token list> ?` is empty after one expansion. The auxiliary `__tl_if_empty_if:o` is a fast emptiness test, converting its argument to a string (after one expansion) and using the test `\if:w \scan_stop: ... \scan_stop:..`

```

12581 \exp_args:Nno \use:n
12582 { \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF } }
12583 {
12584   \__tl_if_empty_if:o { \use_none:n #1 ? }
12585   \prg_return_true:
12586   \else:
12587   \prg_return_false:
12588   \fi:
12589 }
12590 \prg_generate_conditional_variant:Nnn \tl_if_blank:n
12591 { e , V , o } { p , T , F , TF }

```

(End of definition for `\tl_if_blank:nTF` and `__tl_if_blank_p:NNw`. This function is documented on page 112.)

`\tl_if_eq_p:NN` Returns `\c_true_bool` if and only if the two token list variables are equal.

```

12592 \prg_new_eq_conditional:NNn \tl_if_eq:NN \cs_if_eq:NN { p , T , F , TF }
12593 \prg_generate_conditional_variant:Nnn \tl_if_eq:NN
12594 { Nc , c , cc } { p , TF , T , F }

```

(End of definition for `\tl_if_eq:NNTF`. This function is documented on page 112.)

`\tl_if_eq:NcTF`
`\tl_if_eq:cNTF`
`\tl_if_eq:ccTF`

`\l__tl_internal_a_tl` Temporary storage.
`\l__tl_internal_b_tl`

```

12595 \tl_new:N \l__tl_internal_a_tl
12596 \tl_new:N \l__tl_internal_b_tl

```

(End of definition for `\l__tl_internal_a_tl` and `\l__tl_internal_b_tl`.)

`\tl_if_eq:NnTF` A simple store and compare routine.

```

12597 \prg_new_protected_conditional:Npnn \tl_if_eq:Nn #1#2 { T , F , TF }
12598 {
12599   \group_begin:
12600     \tl_set:Nn \l__tl_internal_b_tl {#2}
12601     \exp_after:wN
12602   \group_end:
12603   \if_meaning:w #1 \l__tl_internal_b_tl
12604     \prg_return_true:
12605   \else:
12606     \prg_return_false:
12607   \fi:
12608 }
12609 \prg_generate_conditional_variant:Nnn \tl_if_eq:Nn { c } { TF , T , F }

```

(End of definition for `\tl_if_eq:NnTF`. This function is documented on page 112.)

`\tl_if_eq:mnTF` A simple store and compare routine.

```

12610 \prg_new_protected_conditional:Npnn \tl_if_eq:mn #1#2 { T , F , TF }
12611 {
12612   \group_begin:
12613     \tl_set:Nn \l__tl_internal_a_tl {#1}
12614     \tl_set:Nn \l__tl_internal_b_tl {#2}
12615     \exp_after:wN
12616   \group_end:
12617   \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
12618     \prg_return_true:
12619   \else:
12620     \prg_return_false:
12621   \fi:
12622 }
12623 \prg_generate_conditional_variant:Nnn \tl_if_eq:mn
12624 { nV , ne , nx , V , e , ee , x , xx }
12625 { TF , T , F }

```

(End of definition for `\tl_if_eq:mnTF`. This function is documented on page 113.)

`\tl_if_in:NnTF` See `\tl_if_in:nnTF` for further comments. Here we simply expand the token list variable and pass it to `\tl_if_in:nnTF`.

```

12626 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
12627 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
12628 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
12629 \prg_generate_conditional_variant:Nnn \tl_if_in:Nn
12630 { NV , No , c , cV , co } { T , F , TF }

```

(End of definition for `\tl_if_in:NnTF`. This function is documented on page 113.)

`\tl_if_in:nnTF` Once more, the test relies on the emptiness test for robustness. The function `__tl_tmp:w` removes tokens until the first occurrence of #2. If this does not appear in #1, then the final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and the test is false. See `\tl_if_empty:nTF` for details on the emptiness test.

Treating correctly cases like `\tl_if_in:nnTF {a state}{states}`, where #1#2 contains #2 before the end, requires special care. To cater for this case, we insert `{}` between the two token lists. This marker may not appear in #2 because of T_EX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments. The `\if_false:` constructions are a faster way to do `\group_align_safe_begin:` and `\group_align_safe_end:`. The `\scan_stop:` ensures that f-expanding `\tl_if_in:nnTF` does not lead to unbalanced braces.

```

12631 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
12632 {
12633   \scan_stop:
12634   \if_false: { \fi:
12635     \cs_set:Npn \__tl_tmp:w ##1 #2 { }
12636     \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
12637     { \prg_return_false: } { \prg_return_true: }
12638     \if_false: } \fi:
12639   }
12640 \prg_generate_conditional_variant:Nnn \tl_if_in:nn
12641 { V , WV , o , oo , nV , no } { T , F , TF }

```

(End of definition for `\tl_if_in:nnTF`. This function is documented on page 113.)

`\tl_if_novalue_p:n` Tests whether ##1 matches -NoValue- exactly (with suitable catcodes): this is similar to `\quark_if_nil:nTF`. The first argument of `__tl_if_novalue:w` is empty if and only if ##1 starts with -NoValue-, while the second argument is empty if ##1 is exactly -NoValue- or if it has a question mark just following -NoValue-. In this second case, however, the material after the first ?! remains and makes the emptiness test return false.

```

12642 \cs_set_protected:Npn \__tl_tmp:w #1
12643 {
12644   \prg_new_conditional:Npnn \tl_if_novalue:n ##1
12645   { p , T , F , TF }
12646   {
12647     \__tl_if_empty_if:o { \__tl_if_novalue:w } ##1 {} ? ! #1 ? ? ! }
12648     \prg_return_true:
12649     \else:
12650     \prg_return_false:
12651     \fi:
12652   }
12653   \cs_new:Npn \__tl_if_novalue:w ##1 #1 ##2 ? ##3 ? ! { ##1 ##2 }
12654 }
12655 \exp_args:No \__tl_tmp:w { \c_novalue_tl }

```

(End of definition for `\tl_if_novalue:nTF` and `__tl_if_novalue:w`. This function is documented on page 113.)

`\tl_if_single_p:N` Expand the token list and feed it to `\tl_if_single:nTF`.

```

12656 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
12657 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
12658 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }

```

```

12659 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }
12660 \prg_generate_conditional_variant:Nnn \tl_if_single:N {c} { p , T , F , TF }

```

(End of definition for `\tl_if_single:NTF`. This function is documented on page 113.)

`\tl_if_single_p:n` This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields an empty result if #1 is blank, a single ? if #1 has a single item, and otherwise yields some tokens ending with ??. Then, `__kernel_tl_to_str:w` makes sure there are no odd category codes. An earlier version would compare the result to a single ? using string comparison, but the Lua call is slow in LuaTeX. Instead, `__tl_if_single:nnw` picks the second token in front of it. If #1 is empty, this token is the trailing ? and the `\if:w` test yields false. If #1 has a single item, the token is `\scan_stop:` and the `\if:w` test yields true. Otherwise, it is one of the characters resulting from `\tl_to_str:n`, and the `\if:w` test yields false. Note that `\if:w` and `__kernel_tl_to_str:w` are primitives that take care of expansion.

```

12661 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
12662 {
12663   \if:w \scan_stop: \exp_after:wN \__tl_if_single:nnw
12664     \__kernel_tl_to_str:w
12665     \exp_after:wN { \use_none:nn #1 ?? } \scan_stop: ? \s__tl_stop
12666   \prg_return_true:
12667   \else:
12668     \prg_return_false:
12669   \fi:
12670 }
12671 \cs_new:Npn \__tl_if_single:nnw #1#2#3 \s__tl_stop {#2}

```

(End of definition for `\tl_if_single:nTF` and `__tl_if_single:nnw`. This function is documented on page 113.)

`\tl_if_single_token_p:n` There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying f-expansion yields an empty result if and only if the token list is a single space.

`\tl_if_single_token:nTF`

```

12672 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
12673 {
12674   \tl_if_head_is_N_type:nTF {#1}
12675     { \__tl_if_empty_if:o { \use_none:n #1 } }
12676     {
12677       \tl_if_empty:nTF {#1}
12678       { \if_false: }
12679       { \__tl_if_empty_if:o { \exp:w \exp_end_continue_f:w #1 } }
12680     }
12681   \prg_return_true:
12682   \else:
12683     \prg_return_false:
12684   \fi:
12685 }

```

(End of definition for `\tl_if_single_token:nTF`. This function is documented on page 113.)

53.8 Mapping over token lists

`\tl_map_function:nN` Expandable loop macro for token lists. We use the internal scan mark `\s__tl_stop` (defined later), which is not allowed to show up in the token list #1 since it is internal to l3tl. This allows us a very fast test of whether some `<item>` is the end-marker `\s__tl_stop`, namely call `__tl_use_none_delimit_by_s_stop:w <item> <function> \s__tl_stop`, which calls `<function>` if the `<item>` is the end-marker. To speed up the loop even more, only test one out of eight items, and once we hit one of the eight end-markers, go more slowly through the last few items of the list using `__tl_map_function_end:w`.

```

12686 \cs_new:Npn \tl_map_function:nN #1#2
12687   {
12688     \__tl_map_function:Nnnnnnnnn #2 #1
12689     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12690     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12691     \prg_break_point:Nn \tl_map_break: { }
12692   }
12693 \cs_new:Npn \tl_map_function:NN
12694   { \exp_args:No \tl_map_function:nN }
12695 \cs_generate_variant:Nn \tl_map_function:NN { c }
12696 \cs_new:Npn \__tl_map_function:Nnnnnnnnn #1#2#3#4#5#6#7#8#9
12697   {
12698     \__tl_use_none_delimit_by_s_stop:w
12699     #9 \__tl_map_function_end:w \s__tl_stop
12700     #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
12701     \__tl_map_function:Nnnnnnnnn #1
12702   }
12703 \cs_new:Npn \__tl_map_function_end:w \s__tl_stop #1#2
12704   {
12705     \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
12706     #1 {#2}
12707     \__tl_map_function_end:w \s__tl_stop
12708   }
12709 \cs_new:Npn \__tl_use_none_delimit_by_s_stop:w #1 \s__tl_stop { }

```

(End of definition for `\tl_map_function:nN` and others. These functions are documented on page 118.)

`\tl_map_inline:nn` The inline functions are straight forward by now. We use a little trick with the counter `\g__kernel_prg_map_int` to make them nestable. We can also make use of `__tl_map_function:Nnnnnnnnn` from before.

```

12710 \cs_new_protected:Npn \tl_map_inline:nn #1#2
12711   {
12712     \int_gincr:N \g__kernel_prg_map_int
12713     \cs_gset_protected:cpn
12714     { __tl_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
12715     \exp_args:Nc \__tl_map_function:Nnnnnnnnn
12716     { __tl_map_ \int_use:N \g__kernel_prg_map_int :w }
12717     #1
12718     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12719     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12720     \prg_break_point:Nn \tl_map_break:
12721     { \int_gdecr:N \g__kernel_prg_map_int }
12722   }
12723 \cs_new_protected:Npn \tl_map_inline:Nn

```



```

12724 { \exp_args:No \tl_map_inline:nn }
12725 \cs_generate_variant:Nn \tl_map_inline:Nn { c }

```

(End of definition for `\tl_map_inline:nn` and `\tl_map_inline:Nn`. These functions are documented on page 118.)

`\tl_map_tokens:nn` Much like the function mapping.

`\tl_map_tokens:Nn`

`\tl_map_tokens:cn`

`__tl_map_tokens:nnnnnnnnn`

`__tl_map_tokens_end:w`

```

12726 \cs_new:Npn \tl_map_tokens:nn #1#2
12727 {
12728   \__tl_map_tokens:nnnnnnnnn {#2} #1
12729   \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12730   \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12731   \prg_break_point:Nn \tl_map_break: { }
12732 }
12733 \cs_new:Npn \tl_map_tokens:Nn
12734 { \exp_args:No \tl_map_tokens:nn }
12735 \cs_generate_variant:Nn \tl_map_tokens:Nn { c }
12736 \cs_new:Npn \__tl_map_tokens:nnnnnnnnn #1#2#3#4#5#6#7#8#9
12737 {
12738   \__tl_use_none_delimit_by_s_stop:w
12739   #9 \__tl_map_tokens_end:w \s__tl_stop
12740   \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
12741   \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
12742   \__tl_map_tokens:nnnnnnnnn {#1}
12743 }
12744 \cs_new:Npn \__tl_map_tokens_end:w \s__tl_stop \use:n #1#2
12745 {
12746   \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
12747   #1 {#2}
12748   \__tl_map_tokens_end:w \s__tl_stop
12749 }

```

(End of definition for `\tl_map_tokens:nn` and others. These functions are documented on page 118.)

`\tl_map_variable:nNn`

`\tl_map_variable:NNn`

`\tl_map_variable:cNn`

`__tl_map_variable:Nnn`

`\tl_map_variable:nNn` `{(token list)}` `<tl var>` `{(action)}` assigns `<tl var>` to each element and executes `<action>`. The assignment to `<tl var>` is done after the quark test so that this variable does not get set to a quark.

```

12750 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
12751 { \tl_map_tokens:nn {#1} { \__tl_map_variable:Nnn #2 {#3} } }
12752 \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
12753 { \tl_set:Nn #1 {#3} #2 }
12754 \cs_new_protected:Npn \tl_map_variable:NNn
12755 { \exp_args:No \tl_map_variable:nNn }
12756 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End of definition for `\tl_map_variable:nNn`, `\tl_map_variable:NNn`, and `__tl_map_variable:Nnn`. These functions are documented on page 119.)

`\tl_map_break:`

`\tl_map_break:n`

The break statements use the general `\prg_map_break:Nn`.

```

12757 \cs_new:Npn \tl_map_break:
12758 { \prg_map_break:Nn \tl_map_break: { } }
12759 \cs_new:Npn \tl_map_break:n
12760 { \prg_map_break:Nn \tl_map_break: }

```

(End of definition for `\tl_map_break:` and `\tl_map_break:n`. These functions are documented on page 119.)

53.9 Using token lists

`\tl_to_str:n` Another name for a primitive: defined in `l3basics`.
`\tl_to_str:o` 12761 `\cs_generate_variant:Nn \tl_to_str:n { o , V , v , e }`
`\tl_to_str:V`
`\tl_to_str:v` (End of definition for `\tl_to_str:n`. This function is documented on page 115.)
`\tl_to_str:e`
`\tl_to_str:N` These functions return the replacement text of a token list as a string.
`\tl_to_str:c` 12762 `\cs_new:Npn \tl_to_str:N #1 { __kernel_tl_to_str:w \exp_after:wN {#1} }`
12763 `\cs_generate_variant:Nn \tl_to_str:N { c }`
(End of definition for `\tl_to_str:N`. This function is documented on page 115.)

`\tl_use:N` Token lists which are simply not defined give a clear \TeX error here. No such luck for
`\tl_use:c` ones equal to `\scan_stop`: so instead a test is made and if there is an issue an error is forced.

```
12764 \cs_new:Npn \tl_use:N #1
12765   {
12766     \tl_if_exist:NTF #1 {#1}
12767     {
12768       \msg_expandable_error:nnn
12769         { kernel } { bad-variable } {#1}
12770     }
12771   }
12772 \cs_generate_variant:Nn \tl_use:N { c }
```

(End of definition for `\tl_use:N`. This function is documented on page 115.)

53.10 Working with the contents of token lists

`\tl_count:n` Count number of elements within a token list or token list variable. Brace groups within
`\tl_count:V` the list are read as a single element. Spaces are ignored. `__tl_count:n` grabs the
`\tl_count:v` element and replaces it by +1. The 0 ensures that it works on an empty list.
`\tl_count:e` 12773 `\cs_new:Npn \tl_count:n #1`
`\tl_count:o` 12774 `{`
`\tl_count:N` 12775 `\int_eval:n`
`\tl_count:c` 12776 `{ 0 \tl_map_function:nN {#1} __tl_count:n }`
`__tl_count:n` 12777 `}`
12778 `\cs_new:Npn \tl_count:N #1`
12779 `{`
12780 `\int_eval:n`
12781 `{ 0 \tl_map_function:NN #1 __tl_count:n }`
12782 `}`
12783 `\cs_new:Npn __tl_count:n #1 { + 1 }`
12784 `\cs_generate_variant:Nn \tl_count:n { V , v , e , o }`
12785 `\cs_generate_variant:Nn \tl_count:N { c }`

(End of definition for `\tl_count:n`, `\tl_count:N`, and `__tl_count:n`. These functions are documented on page 116.)

`\tl_count_tokens:n` The token count is computed through an `\int_eval:n` construction. Each `1+` is output to the *left*, into the integer expression, and the sum is ended by the `\exp_end:` inserted by `__tl_act_end:wn` (which is technically implemented as `\c_zero_int`). Somewhat a hack!

```

12786 \cs_new:Npn \tl_count_tokens:n #1
12787   {
12788     \int_eval:n
12789     {
12790       \__tl_act:NNNn
12791       \__tl_act_count_normal:N
12792       \__tl_act_count_group:n
12793       \__tl_act_count_space:
12794       {#1}
12795     }
12796   }
12797 \cs_new:Npn \__tl_act_count_normal:N #1 { 1 + }
12798 \cs_new:Npn \__tl_act_count_space: { 1 + }
12799 \cs_new:Npn \__tl_act_count_group:n #1 { 2 + \tl_count_tokens:n {#1} + }

```

(End of definition for `\tl_count_tokens:n` and others. This function is documented on page 116.)

`\tl_reverse_items:n` Reversal of a token list is done by taking one item at a time and putting it after `\s__tl_stop`.

```

\__tl_reverse_items:nwNwn
\__tl_reverse_items:wn
12800 \cs_new:Npn \tl_reverse_items:n #1
12801   {
12802     \__tl_reverse_items:nwNwn #1 ?
12803     \s__tl_mark \__tl_reverse_items:nwNwn
12804     \s__tl_mark \__tl_reverse_items:wn
12805     \s__tl_stop { }
12806   }
12807 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \s__tl_mark #3 #4 \s__tl_stop #5
12808   {
12809     #3 #2
12810     \s__tl_mark \__tl_reverse_items:nwNwn
12811     \s__tl_mark \__tl_reverse_items:wn
12812     \s__tl_stop { {#1} #5 }
12813   }
12814 \cs_new:Npn \__tl_reverse_items:wn #1 \s__tl_stop #2
12815   { \__kernel_exp_not:w \exp_after:wN { \use_none:nn #2 } }

```

(End of definition for `\tl_reverse_items:n`, `__tl_reverse_items:nwNwn`, and `__tl_reverse_items:wn`. This function is documented on page 116.)

`\tl_trim_spaces:n` Trimming spaces from around the input is deferred to an internal function whose first argument is the token list to trim, augmented by an initial `__tl_trim_mark:`, and whose second argument is a *continuation*, which receives as a braced argument `__tl_trim_mark: <trimmed token list>`. The control sequence `__tl_trim_mark:` expands to nothing in a single expansion. In the case at hand, we take `__kernel_exp_not:w` `\exp_after:wN` as our continuation, so that space trimming behaves correctly within an e-type or x-type expansion.

```

\tl_trim_spaces:V
\tl_trim_spaces:v
\tl_trim_spaces:e
\tl_trim_spaces:o
\tl_trim_spaces_apply:nN
\tl_trim_spaces_apply:oN
\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c
12816 \cs_new:Npn \tl_trim_spaces:n #1
12817   {
12818     \__tl_trim_spaces:nn
12819     { \__tl_trim_mark: #1 }

```

```

12820     { \_kernel_exp_not:w \exp_after:wN }
12821   }
12822 \cs_generate_variant:Nn \tl_trim_spaces:n { V , v , e , o }
12823 \cs_new:Npn \tl_trim_spaces_apply:nN #1#2
12824   { \_tl_trim_spaces:nn { \_tl_trim_mark: #1 } { \exp_args:No #2 } }
12825 \cs_generate_variant:Nn \tl_trim_spaces_apply:nN { o }
12826 \cs_new_protected:Npn \tl_trim_spaces:N #1
12827   { \_kernel_tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
12828 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
12829   { \_kernel_tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
12830 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
12831 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in `_tl_tmp:w`, which then receives a single space as its argument: `#1` is `␣`. Removing leading spaces is done with `_tl_trim_spaces_auxi:w`, which loops until `_tl_trim_mark:␣` matches the end of the token list: then `##1` is the token list and `##3` is `_tl_trim_spaces_auxii:w`. This hands the relevant tokens to the loop `_tl_trim_spaces_auxiii:w`, responsible for trimming trailing spaces. The end is reached when `␣\s__tl_nil` matches the one present in the definition of `\tl_trim_spaces:n`. Then `_tl_trim_spaces_auxiv:w` puts the token list into a group, with a lingering `_tl_trim_mark:` at the start (which will expand to nothing in one step of expansion), and feeds this to the *continuation*.

```

\_tl_trim_spaces:nn
\_tl_trim_spaces_auxi:w
\_tl_trim_spaces_auxii:w
\_tl_trim_spaces_auxiii:w
\_tl_trim_spaces_auxiv:w
\_tl_trim_mark:
12832 \cs_set_protected:Npn \_tl_tmp:w #1
12833   {
12834     \cs_new:Npn \_tl_trim_spaces:nn ##1
12835       {
12836         \_tl_trim_spaces_auxi:w
12837         ##1
12838         \s__tl_nil
12839         \_tl_trim_mark: #1 { }
12840         \_tl_trim_mark: \_tl_trim_spaces_auxii:w
12841         \_tl_trim_spaces_auxiii:w
12842         #1 \s__tl_nil
12843         \_tl_trim_spaces_auxiv:w
12844         \s__tl_stop
12845       }
12846     \cs_new:Npn
12847       \_tl_trim_spaces_auxi:w ##1 \_tl_trim_mark: #1 ##2 \_tl_trim_mark: ##3
12848       {
12849         ##3
12850         \_tl_trim_spaces_auxi:w
12851         \_tl_trim_mark:
12852         ##2
12853         \_tl_trim_mark: #1 {##1}
12854       }
12855     \cs_new:Npn \_tl_trim_spaces_auxii:w
12856       \_tl_trim_spaces_auxi:w \_tl_trim_mark: \_tl_trim_mark: ##1
12857       {
12858         \_tl_trim_spaces_auxiii:w
12859         ##1
12860       }
12861     \cs_new:Npn \_tl_trim_spaces_auxiii:w ##1 #1 \s__tl_nil ##2

```

```

12862     {
12863         ##2
12864         ##1 \s__tl_nil
12865         \__tl_trim_spaces_auxiii:w
12866     }
12867     \cs_new:Npn \__tl_trim_spaces_auxiv:w ##1 \s__tl_nil ##2 \s__tl_stop ##3
12868     { ##3 { ##1 } }
12869     \cs_new:Npn \__tl_trim_mark: {}
12870 }
12871 \__tl_tmp:w { ~ }

```

(End of definition for `\tl_trim_spaces:n` and others. These functions are documented on page 117.)

```

\tl_sort:Nn    Implemented in l3sort.
\tl_sort:cn
\tl_gsort:Nn   (End of definition for \tl_sort:Nn, \tl_gsort:Nn, and \tl_sort:nN. These functions are documented
\tl_gsort:cn   on page 123.)
\tl_sort:nN

```

53.11 The first token from a token list

```

\tl_head:N    Finding the head of a token list expandably always strips braces, which is fine as this is
\tl_head:n    consistent with for example mapping over a list. The empty brace groups in \tl_head:n
\tl_head:V    ensure that a blank argument gives an empty result. The result is returned within the
\tl_head:v    \unexpanded primitive. The approach here is to use \if_false: to allow us to use } as
\tl_head:f    the closing delimiter: this is the only safe choice, as any other token would not be able
\__tl_head_auxi:nw  to parse it's own code. More detail in http://tex.stackexchange.com/a/70168.
\__tl_head_auxii:n 12872 \cs_new:Npn \tl_head:n #1
\tl_head:w    12873 {
\__tl_tl_head:w 12874     \__kernel_exp_not:w \tex_expanded:D
\tl_tail:N    12875     { { \if_false: { \fi: \__tl_head_aux:n #1 { } } } }
\tl_tail:n    12876 }
\tl_tail:V    12877 \cs_new:Npn \__tl_head_aux:n #1
\tl_tail:v    12878 {
\tl_tail:f    12879     \__kernel_exp_not:w {#1}
12880     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
12881 }
12882 \cs_generate_variant:Nn \tl_head:n { V , v , f }
12883 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
12884 \cs_new:Npn \__tl_tl_head:w #1#2 \s__tl_stop {#1}
12885 \cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }

```

To correctly leave the tail of a token list, it's important *not* to absorb any of the tail part as an argument. For example, the simple definition

```

\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop

```

would give the wrong result for `\tl_tail:n { a { bc } }` (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (*i.e.* that the argument is not blank) and assuming there is to dispose of the first item. As with `\tl_head:n`, the result is protected from further expansion by `\unexpanded`. While we could optimise the test here, this would leave some tokens “banned” in the input, which we do not have with this definition.

```

12886 \exp_args:Nno \use:n { \cs_new:Npn \tl_tail:n #1 }
12887 {
12888   \exp_after:wN \__kernel_exp_not:w
12889   \tl_if_blank:nTF {#1}
12890   { { } }
12891   { \exp_after:wN { \use_none:n #1 } }
12892 }
12893 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
12894 \cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }

```

(End of definition for `\tl_head:N` and others. These functions are documented on page 120.)

```

\tl_if_head_eq_meaning_p:nN
\tl_if_head_eq_meaning_p:VN
\tl_if_head_eq_meaning_p:eN
\tl_if_head_eq_meaning:nNTF
\tl_if_head_eq_meaning:VNTF
\tl_if_head_eq_meaning:eNTF
\tl_if_head_eq_charcode_p:nN
\tl_if_head_eq_charcode_p:VN
\tl_if_head_eq_charcode_p:eN
\tl_if_head_eq_charcode_p:fN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode:VNTF
\tl_if_head_eq_charcode:eNTF
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode_p:VN
\tl_if_head_eq_catcode_p:eN
\tl_if_head_eq_catcode_p:oN
\tl_if_head_eq_catcode:nNTF
\tl_if_head_eq_catcode:VNTF
\tl_if_head_eq_catcode:eNTF
\tl_if_head_eq_catcode:oNTF
  \__tl_head_exp_not:w
  \__tl_if_head_eq_empty_arg:w

```

Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```

\if_charcode:w
  \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
  \exp_not:N #2

```

The two first special cases are detected by testing if the token list starts with an N-type token (the extra ? sends empty token lists to the true branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument results in `\tl_head:w` leaving two token: `^` and `__tl_if_head_eq_empty_arg:w` which will result in the `\if_charcode:w` test being false and remove `\exp_not:N` and `#2`.

```

12895 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
12896 {
12897   \if_charcode:w
12898     \tl_if_head_is_N_type:nTF { #1 ? }
12899     { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
12900     { \str_head:n {#1} }
12901     \exp_not:N #2
12902     \prg_return_true:
12903   \else:
12904     \prg_return_false:
12905   \fi:
12906 }
12907 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_charcode:nN
12908 { V , e , f } { p , TF , T , F }

```

For `\tl_if_head_eq_catcode:nN`, again we detect special cases with a `\tl_if_head_is_N_type:n`. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either `\c_group_begin_token` or `\c_space_token`. Again, for an empty argument, a hack is used, removing the token given by the user and leaving two tokens in the input stream which will make the `\if_catcode:w` test return false.

```

12909 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
12910 {
12911   \if_catcode:w
12912     \tl_if_head_is_N_type:nTF { #1 ? }

```

```

12913         { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
12914         {
12915             \tl_if_head_is_group:nTF {#1}
12916             \c_group_begin_token
12917             \c_space_token
12918         }
12919         \exp_not:N #2
12920     \prg_return_true:
12921 \else:
12922     \prg_return_false:
12923 \fi:
12924 }
12925 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_catcode:nN
12926 { V , e , o } { p , TF , T , F }

```

For `\tl_if_head_eq_meaning:nN`, again, detect special cases. In the normal case, use `\tl_head:w`, with no `\exp_not:N` this time, since `\if_meaning:w` causes no expansion. With an empty argument, the test is true, and `\use_none:nnn` removes #2 and `\prg_return_true:` and `\else:` (it is safe this way here as in this case `\prg_new_conditional:Npnn` didn't optimize these two away). In the special cases, we know that the first token is a character, hence `\if_charcode:w` and `\if_catcode:w` together are enough. We combine them in some order, hopefully faster than the reverse. Tests are not nested because the arguments may contain unmatched primitive conditionals.

```

12927 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
12928 {
12929     \tl_if_head_is_N_type:nTF { #1 ? }
12930     \__tl_if_head_eq_meaning_normal:nN
12931     \__tl_if_head_eq_meaning_special:nN
12932     {#1} #2
12933 }
12934 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_meaning:nN
12935 { V , e } { p , TF , T , F }
12936 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
12937 {
12938     \exp_after:wN \if_meaning:w
12939     \__tl_tl_head:w #1 { ?? \use_none:nnn } \s__tl_stop #2
12940     \prg_return_true:
12941     \else:
12942     \prg_return_false:
12943     \fi:
12944 }
12945 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
12946 {
12947     \if_charcode:w \str_head:n {#1} \exp_not:N #2
12948     \exp_after:wN \use_ii:nn
12949     \else:
12950     \prg_return_false:
12951     \fi:
12952     \use_none:n
12953     {
12954         \if_catcode:w \exp_not:N #2
12955         \tl_if_head_is_group:nTF {#1}
12956         { \c_group_begin_token }
12957         { \c_space_token }

```

```

12958     \prg_return_true:
12959     \else:
12960     \prg_return_false:
12961     \fi:
12962   }
12963 }

```

Both `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` will need to get the first token of their argument and apply `\exp_not:N` to it. `__tl_head_exp_not:w` does exactly that.

```

12964 \cs_new:Npn \__tl_head_exp_not:w #1 #2 \s__tl_stop
12965   { \exp_not:N #1 }

```

If the argument of `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` was empty `__tl_if_head_eq_empty_arg:w` will be left in the input stream. This macro has to remove `\exp_not:N` and the following token from the input stream to make sure no unbalanced if-construct is created and leave tokens there which make the two tests return false.

```

12966 \cs_new:Npn \__tl_if_head_eq_empty_arg:w \exp_not:N #1
12967   { ? }

```

(End of definition for `\tl_if_head_eq_meaning:nNTF` and others. These functions are documented on page 114.)

```

\tl_if_head_is_N_type_p:n
\tl_if_head_is_N_type:nTF
  \__tl_if_head_is_N_type_auxi:w
  \__tl_if_head_is_N_type_auxii:n

```

A token list can be empty, can start with an explicit space character (catcode 10 and charcode 32), can start with a begin-group token (catcode 1), or start with an N-type argument. In the first two cases, and when `#1~` starts with `{}`, `__tl_if_head_is_N_type_auxi:w` receives an empty argument hence produces `f` and removes everything before the first `\scan_stop:.` In the third case (except when `#1~` starts with `{}`), the second auxiliary removes the first copy of `#1` that was used for the space test, then expands `\token_to_str:N` which hits the leading begin-group token, leaving a single closing brace to be compared with `\scan_stop:.` In the last case, `\token_to_str:N` does not change the brace balance so that only `\scan_stop: \scan_stop:` remain, making the character code test true. One cannot optimize by moving one of the `\scan_stop:` to the beginning: if `#1` contains primitive conditionals, all of its occurrences must be dealt with before the `\if:w` tries to skip the true branch of the conditional.

```

12968 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
12969   {
12970     \if:w
12971       \if_false: { \fi: \__tl_if_head_is_N_type_auxi:w #1 ~ }
12972       { \exp_after:wN { \token_to_str:N #1 } }
12973       \scan_stop: \scan_stop:
12974       \prg_return_true:
12975     \else:
12976       \prg_return_false:
12977     \fi:
12978   }
12979 \exp_args:Nno \use:n { \cs_new:Npn \__tl_if_head_is_N_type_auxi:w #1 ~ }
12980   {
12981     \tl_if_empty:nTF {#1}
12982     { f \exp_after:wN \use_none:nn }
12983     { \exp_after:wN \__tl_if_head_is_N_type_auxii:n }
12984     \exp_after:wN { \if_false: } \fi:
12985   }

```



```

12986 \cs_new:Npn \__tl_if_head_is_N_type_auxii:n #1
12987 { \exp_after:wN \use_none:n \exp_after:wN }

```

(End of definition for `\tl_if_head_is_N_type:nTF`, `__tl_if_head_is_N_type_auxi:w`, and `__tl_if_head_is_N_type_auxii:n`. This function is documented on page 114.)

`\tl_if_head_is_group_p:n` Pass the first token of #1 through `\token_to_str:N`, then check for the brace balance.
`\tl_if_head_is_group:nTF` The extra ? caters for an empty argument. This could be made faster, but we need all brace tricks to happen in one step of expansion, keeping the token list brace balanced at all times.
`__tl_if_head_is_group_fi_false:w`

```

12988 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
12989 {
12990   \if:w
12991     \exp_after:wN \use_none:n
12992     \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
12993     \scan_stop: \scan_stop:
12994     \__tl_if_head_is_group_fi_false:w
12995   \fi:
12996   \if_true:
12997     \prg_return_true:
12998   \else:
12999     \prg_return_false:
13000   \fi:
13001 }
13002 \cs_new:Npn \__tl_if_head_is_group_fi_false:w \fi: \if_true: { \fi: \if_false: }

```

(End of definition for `\tl_if_head_is_group:nTF` and `__tl_if_head_is_group_fi_false:w`. This function is documented on page 114.)

`\tl_if_head_is_space_p:n` The auxiliary's argument is all that is before the first explicit space in `\prg_do_nothing:#1?~`.
`\tl_if_head_is_space:nTF` If that is a single `\prg_do_nothing:` the test yields true. Otherwise, that is more than one token, and the test yields false. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from \TeX in a table, and to allow for removing what remains of the token list after its first space. The use of `\if:w` ensures that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).
`__tl_if_head_is_space:w`

```

13003 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
13004 {
13005   \if:w
13006     \if_false: { \fi: \__tl_if_head_is_space:w \prg_do_nothing: #1 ? ~ }
13007     \scan_stop: \scan_stop:
13008     \prg_return_true:
13009   \else:
13010     \prg_return_false:
13011   \fi:
13012 }
13013 \exp_args:Nno \use:n { \cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
13014 {
13015   \__tl_if_empty_if:o {#1} \else: f \fi:
13016   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
13017 }

```

(End of definition for `\tl_if_head_is_space:nTF` and `__tl_if_head_is_space:w`. This function is documented on page 115.)

53.12 Token by token changes

`\s__tl_act_stop` The `__tl_act_...` functions may be applied to any token list. Hence, we use a private quark, to allow any token, even quarks, in the token list. Only `\s__tl_act_stop` may not appear in the token lists manipulated by `__tl_act:NNNn` functions.

```
13018 \scan_new:N \s__tl_act_stop
```

(End of definition for `\s__tl_act_stop`.)

```

__tl_act:NNNn To help control the expansion, \__tl_act:NNNn should always be preceded by \exp:w
__tl_act_output:n and ends by producing \exp_end: once the result has been obtained. This way no internal
__tl_act_reverse_output:n token of it can be accidentally end up in the input stream. Because \s__tl_act_stop
__tl_act_loop:w can't appear without braces around it in the argument #1 of \__tl_act_loop:w, we can
__tl_act_normal:NwNNN use this marker to set up a fast test for leading spaces.
__tl_act_group:nwNNN 13019 \cs_set_protected:Npn \__tl_tmp:w #1
__tl_act_space:wwNNN 13020 {
__tl_act_end:wn 13021 \cs_new:Npn \__tl_act_if_head_is_space:nTF ##1
__tl_act_if_head_is_space:nTF 13022 {
__tl_act_if_head_is_space:w 13023 \__tl_act_if_head_is_space:w
__tl_act_if_head_is_space_true:w 13024 \s__tl_act_stop ##1 \s__tl_act_stop \__tl_act_if_head_is_space_true:w
__tl_use_none_delimit_by_q_act_stop:w 13025 \s__tl_act_stop #1 \s__tl_act_stop \use_ii:nn
13026 }
13027 \cs_new:Npn \__tl_act_if_head_is_space:w
13028 ##1 \s__tl_act_stop #1 ##2 \s__tl_act_stop
13029 {}
13030 \cs_new:Npn \__tl_act_if_head_is_space_true:w
13031 \s__tl_act_stop #1 \s__tl_act_stop \use_ii:nn ##1 ##2
13032 {##1}
13033 }
13034 \__tl_tmp:w { ~ }

```

(We expand the definition `__tl_act_if_head_is_space:nTF` when setting up `__tl_act_loop:w`, so we can then undefine the auxiliary.) In the loop, we check how the token list begins and act accordingly. In the “group” case, we may have reached `\s__tl_act_stop`, the end of the list. Then leave `\exp_end:` and the result in the input stream, to terminate the expansion of `\exp:w`. Otherwise, apply the relevant function to the “arguments”, #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with an N-type or with a space, making sure that `__tl_act_space:wwNNN` gobbles the space.

```

13035 \exp_args:Nne \use:n { \cs_new:Npn \__tl_act_loop:w #1 \s__tl_act_stop }
13036 {
13037 \exp_not:o { \__tl_act_if_head_is_space:nTF {#1} }
13038 \exp_not:N \__tl_act_space:wwNNN
13039 {
13040 \exp_not:o { \tl_if_head_is_group:nTF {#1} }
13041 \exp_not:N \__tl_act_group:nwNNN
13042 \exp_not:N \__tl_act_normal:NwNNN
13043 }
13044 \exp_not:n {#1} \s__tl_act_stop
13045 }
13046 \cs_undefine:N \__tl_act_if_head_is_space:nTF
13047 \cs_new:Npn \__tl_act_normal:NwNNN #1 #2 \s__tl_act_stop #3
13048 {

```

```

13049     #3 #1
13050     \_t1_act_loop:w #2 \s__t1_act_stop
13051     #3
13052   }
13053 \cs_new:Npn \_t1_use_none_delimit_by_s_act_stop:w #1 \s__t1_act_stop { }
13054 \cs_new:Npn \_t1_act_end:wn #1 \_t1_act_result:n #2
13055   { \group_align_safe_end: \exp_end: #2 }
13056 \cs_new:Npn \_t1_act_group:nwNNN #1 #2 \s__t1_act_stop #3#4#5
13057   {
13058     \_t1_use_none_delimit_by_s_act_stop:w #1 \_t1_act_end:wn \s__t1_act_stop
13059     #5 {#1}
13060     \_t1_act_loop:w #2 \s__t1_act_stop
13061     #3 #4 #5
13062   }
13063 \exp_last_unbraced:NNo
13064 \cs_new:Npn \_t1_act_space:wwNNN \c_space_t1 #1 \s__t1_act_stop #2#3
13065   {
13066     #3
13067     \_t1_act_loop:w #1 \s__t1_act_stop
13068     #2 #3
13069   }

```

_t1_act:NNNn loops over tokens, groups, and spaces in #4. {\s@@_act_stop} serves as the end of token list marker, the ? after it avoids losing outer braces. The result is stored as an argument for the dummy function _t1_act_result:n.

```

13070 \cs_new:Npn \_t1_act:NNNn #1#2#3#4
13071   {
13072     \group_align_safe_begin:
13073     \_t1_act_loop:w #4 { \s__t1_act_stop } ? \s__t1_act_stop
13074     #1 #3 #2
13075     \_t1_act_result:n { }
13076   }

```

Typically, the output is done to the right of what was already output, using _t1_act_output:n, but for the _t1_act_reverse functions, it should be done to the left.

```

13077 \cs_new:Npn \_t1_act_output:n #1 #2 \_t1_act_result:n #3
13078   { #2 \_t1_act_result:n { #3 #1 } }
13079 \cs_new:Npn \_t1_act_reverse_output:n #1 #2 \_t1_act_result:n #3
13080   { #2 \_t1_act_result:n { #1 #3 } }

```

(End of definition for _t1_act:NNNn and others.)

\t1_reverse:n The goal here is to reverse without losing spaces nor braces. This is done using the general internal function _t1_act:NNNn. Spaces and “normal” tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group.

```

\t1_reverse:e 13081 \cs_new:Npn \t1_reverse:n #1
\_t1_reverse_normal:nN 13082   {
\_t1_reverse_group_preserve:nn 13083     \_kernel_exp_not:w \exp_after:wN
\_t1_reverse_space:n 13084     {
13085       \exp:w
13086       \_t1_act:NNNn
13087       \_t1_reverse_normal:N
13088       \_t1_reverse_group_preserve:n
13089       \_t1_reverse_space:

```

```

13090         {#1}
13091     }
13092 }
13093 \cs_generate_variant:Nn \tl_reverse:n { o , V , f , e }
13094 \cs_new:Npn \__tl_reverse_normal:N
13095   { \__tl_act_reverse_output:n }
13096 \cs_new:Npn \__tl_reverse_group_preserve:n #1
13097   { \__tl_act_reverse_output:n { {#1} } }
13098 \cs_new:Npn \__tl_reverse_space:
13099   { \__tl_act_reverse_output:n { ~ } }

```

(End of definition for `\tl_reverse:n` and others. This function is documented on page 116.)

```

\tl_reverse:N This reverses the list, leaving \exp_stop_f: in front, which stops the f-expansion.
\tl_reverse:c 13100 \cs_new_protected:Npn \tl_reverse:N #1
\tl_greverse:N 13101   { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\tl_greverse:c 13102 \cs_new_protected:Npn \tl_greverse:N #1
                13103   { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
                13104 \cs_generate_variant:Nn \tl_reverse:N { c }
                13105 \cs_generate_variant:Nn \tl_greverse:N { c }

```

(End of definition for `\tl_reverse:N` and `\tl_greverse:N`. These functions are documented on page 116.)

53.13 Using a single item

```

\tl_item:nn The idea here is to find the offset of the item from the left, then use a loop to grab
\tl_item:Nn the correct item. If the resulting offset is too large, then \__tl_if_recursion_tail_
\tl_item:cn break:nN terminates the loop, and returns nothing at all.
__tl_item_aux:nn 13106 \cs_new:Npn \tl_item:nn #1#2
__tl_item:nn     13107   {
                13108     \exp_args:Nf \__tl_item:nn
                13109     { \exp_args:Nf \__tl_item_aux:nn { \int_eval:n {#2} } {#1} }
                13110     #1
                13111     \q_tl_recursion_tail
                13112     \prg_break_point:
                13113   }
                13114 \cs_new:Npn \__tl_item_aux:nn #1#2
                13115   {
                13116     \int_compare:nNnTF {#1} < 0
                13117     { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
                13118     {#1}
                13119   }
                13120 \cs_new:Npn \__tl_item:nn #1#2
                13121   {
                13122     \__tl_if_recursion_tail_break:nN {#2} \prg_break:
                13123     \int_compare:nNnTF {#1} = 1
                13124     { \prg_break:n { \exp_not:n {#2} } }
                13125     { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
                13126   }
                13127 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
                13128 \cs_generate_variant:Nn \tl_item:Nn { c }

```

(End of definition for `\tl_item:nn` and others. These functions are documented on page 121.)

`\tl_rand_item:n` Importantly `\tl_item:nn` only evaluates its argument once.

```

\tl_rand_item:n 13129 \cs_new:Npn \tl_rand_item:n #1
\tl_rand_item:N 13130 {
\tl_rand_item:c 13131   \tl_if_blank:nF {#1}
                  13132     { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
                  13133   }
13134 \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
13135 \cs_generate_variant:Nn \tl_rand_item:N { c }

```

(End of definition for `\tl_rand_item:n` and `\tl_rand_item:N`. These functions are documented on page 121.)

`\tl_range:Nnn` To avoid checking for the end of the token list at every step, start by counting the number

`\tl_range:cnn` `l` of items and “normalizing” the bounds, namely clamping them to the interval $[0, l]$ and

`\tl_range:nnn` dealing with negative indices. More precisely, `__tl_range_items:nnNn` receives the

`__tl_range:Nnnn` number of items to skip at the beginning of the token list, the index of the last item

`__tl_range:nnnNn` to keep, a function which is either `__tl_range:w` or the token list itself. If nothing

`__tl_range:nnNn` should be kept, leave `{}`: this stops the f-expansion of `\tl_head:f` and that function

`__tl_range_skip:w` produces an empty result. Otherwise, repeatedly call `__tl_range_skip:w` to delete `#1`

`__tl_range:w` items from the input stream (the extra brace group avoids an off-by-one shift). For the

`__tl_range_skip_spaces:n` braced version `__tl_range_braced:w` sets up `__tl_range_collect_braced:w` which

`__tl_range_collect:nn` stores items one by one in an argument after the semicolon. Depending on the first token

`__tl_range_collect:ff` of the tail, either just move it (if it is a space) or also decrement the number of items left

`__tl_range_collect_space:nw` to find. Eventually, the result is a brace group followed by the rest of the token list, and

`__tl_range_collect_N:nN` `\tl_head:f` cleans up and gives the result in `\exp_not:n`.

`__tl_range_collect_group:nN`

```

13136 \cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
13137 \cs_generate_variant:Nn \tl_range:Nnn { c }
13138 \cs_new:Npn \tl_range:nnn { \__tl_range:Nnnn \__tl_range:w }
13139 \cs_new:Npn \__tl_range:Nnnn #1#2#3#4
13140 {
13141   \tl_head:f
13142   {
13143     \exp_args:Nf \__tl_range:nnnNn
13144       { \tl_count:n {#2} } {#3} {#4} #1 {#2}
13145   }
13146 }
13147 \cs_new:Npn \__tl_range:nnnNn #1#2#3
13148 {
13149   \exp_args:Nff \__tl_range:nnNn
13150   {
13151     \exp_args:Nf \__tl_range_normalize:nn
13152       { \int_eval:n { #2 - 1 } } {#1}
13153   }
13154   {
13155     \exp_args:Nf \__tl_range_normalize:nn
13156       { \int_eval:n {#3} } {#1}
13157   }
13158 }
13159 \cs_new:Npn \__tl_range:nnNn #1#2#3#4
13160 {
13161   \if_int_compare:w #2 > #1 \exp_stop_f: \else:
13162     \exp_after:wN { \exp_after:wN }
13163   \fi:

```

```

13164 \exp_after:wN #3
13165 \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
13166 \exp_after:wN { \exp:w \_tl_range_skip:w #1 ; { } #4 }
13167 }
13168 \cs_new:Npn \_tl_range_skip:w #1 ; #2
13169 {
13170 \if_int_compare:w #1 > \c_zero_int
13171 \exp_after:wN \_tl_range_skip:w
13172 \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
13173 \else:
13174 \exp_after:wN \exp_end:
13175 \fi:
13176 }
13177 \cs_new:Npn \_tl_range:w #1 ; #2
13178 {
13179 \exp_args:Nf \_tl_range_collect:nn
13180 { \_tl_range_skip_spaces:n {#2} } {#1}
13181 }
13182 \cs_new:Npn \_tl_range_skip_spaces:n #1
13183 {
13184 \tl_if_head_is_space:nTF {#1}
13185 { \exp_args:Nf \_tl_range_skip_spaces:n {#1} }
13186 { { } #1 }
13187 }
13188 \cs_new:Npn \_tl_range_collect:nn #1#2
13189 {
13190 \int_compare:nNnTF {#2} = 0
13191 {#1}
13192 {
13193 \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
13194 {
13195 \exp_args:Nf \_tl_range_collect:nn
13196 { \_tl_range_collect_space:nw #1 }
13197 {#2}
13198 }
13199 {
13200 \_tl_range_collect:ff
13201 {
13202 \exp_args:No \tl_if_head_is_N_type:nTF { \use_none:n #1 }
13203 { \_tl_range_collect_N:nN }
13204 { \_tl_range_collect_group:nn }
13205 #1
13206 }
13207 { \int_eval:n { #2 - 1 } }
13208 }
13209 }
13210 }
13211 \cs_new:Npn \_tl_range_collect_space:nw #1 ~ { { #1 ~ } }
13212 \cs_new:Npn \_tl_range_collect_N:nN #1#2 { { #1 #2 } }
13213 \cs_new:Npn \_tl_range_collect_group:nn #1#2 { { #1 {#2} } }
13214 \cs_generate_variant:Nn \_tl_range_collect:nn { ff }

```

(End of definition for `\tl_range:Nnn` and others. These functions are documented on page 122.)

`_tl_range_normalize:nn` This function converts an `<index>` argument into an explicit position in the token list

(a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the $\langle index \rangle$ #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

```

13215 \cs_new:Npn \__tl_range_normalize:nn #1#2
13216 {
13217   \int_eval:n
13218   {
13219     \if_int_compare:w #1 < \c_zero_int
13220     \if_int_compare:w #1 < -#2 \exp_stop_f:
13221     0
13222     \else:
13223     #1 + #2 + 1
13224     \fi:
13225   \else:
13226     \if_int_compare:w #1 < #2 \exp_stop_f:
13227     #1
13228     \else:
13229     #2
13230     \fi:
13231   \fi:
13232 }
13233 }
```

(End of definition for `__tl_range_normalize:nn`.)

53.14 Viewing token lists

```

\tl_show:N Showing token list variables is done after checking that the variable is defined (see
\tl_show:c \__kernel_register_show:N).
\tl_log:N
\tl_log:c
\__tl_show:NN
13234 \cs_new_protected:Npn \tl_show:N { \__tl_show:NN \tl_show:n }
13235 \cs_generate_variant:Nn \tl_show:N { c }
13236 \cs_new_protected:Npn \tl_log:N { \__tl_show:NN \tl_log:n }
13237 \cs_generate_variant:Nn \tl_log:N { c }
13238 \cs_new_protected:Npn \__tl_show:NN #1#2
13239 {
13240   \__kernel_chk_defined:NT #2
13241   {
13242     \exp_args:Nf \tl_if_empty:nTF
13243     { \cs_prefix_spec:N #2 \cs_parameter_spec:N #2 }
13244     {
13245       \exp_args:Ne #1
13246       { \token_to_str:N #2 = \__kernel_exp_not:w \exp_after:wN {#2} }
13247     }
13248     {
13249       \msg_error:nneee { kernel } { bad-type }
13250       { \token_to_str:N #2 } { \token_to_meaning:N #2 } { tl }
13251     }
13252   }
13253 }
```

(End of definition for `\tl_show:N`, `\tl_log:N`, and `__tl_show:NN`. These functions are documented on page 117.)

`\tl_show:n` Many show functions are based on `\tl_show:n`. The argument of `\tl_show:n` is line-wrapped using `\iow_wrap:nnnN` but with a leading `>~` and trailing period, both removed before passing the wrapped text to the `\showtokens` primitive. This primitive shows the result with a leading `>~` and trailing period.

`__tl_show:n` The token list `\l__tl_internal_a_tl` containing the result of all these manipulations is displayed to the terminal using `\tex_showtokens:D` and an odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to `__kernel_iow_with:Nnn` ensure that the `\newlinechar` is set to 10 so that the `\iow_newline:` inserted by the line-wrapping code are correctly recognized by T_EX, and that `\errorcontextlines` is `-1` to avoid printing irrelevant context.

```

13254 \cs_new_protected:Npn \tl_show:n #1
13255   { \iow_wrap:nnnN { >~ \tl_to_str:n {#1} . } { } { } \__tl_show:n }
13256 \cs_generate_variant:Nn \tl_show:n { e , x }
13257 \cs_new_protected:Npn \__tl_show:n #1
13258   {
13259     \tl_set:Nf \l__tl_internal_a_tl { \__tl_show:w #1 \s_tl_stop }
13260     \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
13261     {
13262       \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
13263       {
13264         \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
13265         { \exp_after:wN \l__tl_internal_a_tl }
13266       }
13267     }
13268   }
13269 \cs_new:Npn \__tl_show:w #1 > #2 . \s_tl_stop {#2}

```

(End of definition for `\tl_show:n`, `__tl_show:n`, and `__tl_show:w`. This function is documented on page 117.)

`\tl_log:n` Logging is much easier, simply line-wrap. The `>~` and trailing period is there to match the output of `\tl_show:n`.

```

\__tl_log:e
\__tl_log:x
13270 \cs_new_protected:Npn \tl_log:n #1
13271   { \iow_wrap:nnnN { > ~ \tl_to_str:n {#1} . } { } { } \iow_log:n }
13272 \cs_generate_variant:Nn \tl_log:n { e , x }

```

(End of definition for `\tl_log:n`. This function is documented on page 117.)

`__kernel_chk_tl_type:NnnT` Helper for checking that `#1` has the correct internal structure to be of a certain type. Make sure that it is defined and that it is a token list, namely a macro with no `\long` nor `\protected` prefix. Then compare `#1` to an attempt at reconstructing a valid structure of the given type using `#2` (see implementation of `\seq_show:N` for instance). If that is successful run the requested code `#4`.

```

13273 \cs_new_protected:Npn \__kernel_chk_tl_type:NnnT #1#2#3#4
13274   {
13275     \__kernel_chk_defined:NT #1
13276     {
13277       \exp_args:Nf \tl_if_empty:nTF
13278       { \cs_prefix_spec:N #1 \cs_parameter_spec:N #1 }
13279       {
13280         \tl_set:Ne \l__tl_internal_a_tl {#3}
13281         \tl_if_eq:NNTF #1 \l__tl_internal_a_tl
13282         {#4}

```



```

13283         {
13284             \msg_error:nneeee { kernel } { bad-type }
13285             { \token_to_str:N #1 } { \tl_to_str:N #1 }
13286             {#2} { \tl_to_str:N \l__tl_internal_a_tl }
13287         }
13288     }
13289     {
13290         \msg_error:nneee { kernel } { bad-type }
13291         { \token_to_str:N #1 } { \token_to_meaning:N #1 } {#2}
13292     }
13293 }
13294 }

```

(End of definition for `__kernel_chk_tl_type:NnnT`.)

53.15 Internal scan marks

`\s__tl_nil` Internal scan marks. These are defined here at the end because the code for `\scan_new:N` depends on some `|3tl` functions.

```

\s__tl_mark
\s__tl_stop
13295 \scan_new:N \s__tl_nil
13296 \scan_new:N \s__tl_mark
13297 \scan_new:N \s__tl_stop

```

(End of definition for `\s__tl_nil`, `\s__tl_mark`, and `\s__tl_stop`.)

53.16 Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```

13298 \tl_new:N \g_tmpa_tl
13299 \tl_new:N \g_tmpb_tl

```

(End of definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 126.)

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```

13300 \tl_new:N \l_tmpa_tl
13301 \tl_new:N \l_tmpb_tl

```

(End of definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 126.)

We finally clean up a temporary control sequence that we have used at various points to set up some definitions.

```

13302 \cs_undefine:N \__tl_tmp:w
13303 </package>

```

Chapter 54

l3tl-build implementation

```
13304 (*package)
```

```
13305 (@@=tl)
```

Between `\tl_build_begin:N⟨tl var⟩` and `\tl_build_end:N⟨tl var⟩`, the `⟨tl var⟩` has the structure

```
\exp_end: ... \exp_end: \_tl_build_last:NNn ⟨assignment⟩ ⟨next tl⟩
  {⟨left⟩} ⟨right⟩
```

where `⟨right⟩` is not braced. The “data” it represents is `⟨left⟩` followed by the “data” of `⟨next tl⟩` followed by `⟨right⟩`. The `⟨next tl⟩` is a token list variable whose name is that of `⟨tl var⟩` followed by `'`. There are between 0 and 4 `\exp_end:` to keep track of when `⟨left⟩` and `⟨right⟩` should be put into the `⟨next tl⟩`. The `⟨assignment⟩` is `\cs_set_nopar:Npe` if the variable is local, and `\cs_gset_nopar:Npe` if it is global.

```
\tl_build_begin:N
\tl_build_gbegin:N
\_tl_build_begin:NN
\_tl_build_begin:NNN
```

First construct the `⟨next tl⟩`: using a prime here conflicts with the usual `expl3` convention but we need a name that can be derived from `#1` without any external data such as a counter. Empty that `⟨next tl⟩` and setup the structure. The local and global versions only differ by a single function `\cs_(g)set_nopar:Npe` used for all assignments: this is important because only that function is stored in the `⟨tl var⟩` and `⟨next tl⟩` for subsequent assignments. In principle `_tl_build_begin:NNN` could use `\tl_(g)clear_new:N` to empty `#1` and make sure it is defined, but logging the definition does not seem useful so we just do `#3 #1 {}` to clear it locally or globally as appropriate.

```
13306 \cs_new_protected:Npn \tl_build_begin:N #1
13307   { \_tl_build_begin:NN \cs_set_nopar:Npe #1 }
13308 \cs_new_protected:Npn \tl_build_gbegin:N #1
13309   { \_tl_build_begin:NN \cs_gset_nopar:Npe #1 }
13310 \cs_new_protected:Npn \_tl_build_begin:NN #1#2
13311   { \exp_args:Nc \_tl_build_begin:NNN { \cs_to_str:N #2 ' } #2 #1 }
13312 \cs_new_protected:Npn \_tl_build_begin:NNN #1#2#3
13313   {
13314     #3 #1 { }
13315     #3 #2
13316     {
13317       \exp_not:n { \exp_end: \exp_end: \exp_end: \exp_end: }
13318       \exp_not:n { \_tl_build_last:NNn #3 #1 { } }
13319     }
13320   }
```

(End of definition for `\tl_build_begin:N` and others. These functions are documented on page 127.)

`\tl_build_put_right:Nn` Similar to `\tl_put_right:Nn`, but apply `\exp:w` to #1. Most of the time this just removes one `\exp_end:`. When there are none left, `__tl_build_last:NNn` is expanded instead.
`\tl_build_put_right:Ne` It resets the definition of the `\tl var` by ending the `\exp_not:n` and the definition
`\tl_build_put_right:Nx` early. Then it makes sure the `\next tl` (its argument #1) is set-up and starts a new
`\tl_build_gput_right:Nn` definition. Then `__tl_build_put:nn` and `__tl_build_put:nw` place the `\left` part
`\tl_build_gput_right:Ne` of the original `\tl var` as appropriate for the definition of the `\next tl` (the `\right`)
`\tl_build_gput_right:Nx` part is left in the right place without ever becoming a macro argument). We use `\exp_`
`__tl_build_last:NNn` `after:wN` rather than some `\exp_args:No` to avoid reading arguments that are likely
`__tl_build_put:nn` very long token lists. We use `\cs_(g)set_nopar:Npe` rather than `\tl_(g)set:Ne` partly
`__tl_build_put:nw` for the same reason and partly because the assignments are interrupted by brace tricks,
which implies that the assignment does not simply set the token list to an e-expansion
of the second argument.

```

13321 \cs_new_protected:Npn \tl_build_put_right:Nn #1#2
13322   {
13323     \cs_set_nopar:Npe #1
13324     { \__kernel_exp_not:w \exp_after:wN { \exp:w #1 #2 } }
13325   }
13326 \cs_generate_variant:Nn \tl_build_put_right:Nn { Ne , Nx }
13327 \cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
13328   {
13329     \cs_gset_nopar:Npe #1
13330     { \__kernel_exp_not:w \exp_after:wN { \exp:w #1 #2 } }
13331   }
13332 \cs_generate_variant:Nn \tl_build_gput_right:Nn { Ne , Nx }
13333 \cs_new_protected:Npn \__tl_build_last:NNn #1#2
13334   {
13335     \if_false: { { \fi:
13336       \exp_end: \exp_end: \exp_end: \exp_end: \exp_end:
13337       \__tl_build_last:NNn #1 #2 { }
13338     }
13339   }
13340 \if_meaning:w \c_empty_tl #2
13341   \__tl_build_begin:NN #1 #2
13342 \fi:
13343 #1 #2
13344   {
13345     \__kernel_exp_not:w \exp_after:wN
13346     {
13347       \exp:w \if_false: } } \fi:
13348     \exp_after:wN \__tl_build_put:nn \exp_after:wN {#2}
13349   }
13350 \cs_new_protected:Npn \__tl_build_put:nn #1#2 { \__tl_build_put:nw {#2} #1 }
13351 \cs_new_protected:Npn \__tl_build_put:nw #1#2 \__tl_build_last:NNn #3#4#5
13352   { #2 \__tl_build_last:NNn #3 #4 { #1 #5 } }

```

(End of definition for `\tl_build_put_right:Nn` and others. These functions are documented on page 127.)

`\tl_build_put_left:Nn` See `\tl_build_put_right:Nn` for all the machinery. We could easily provide `\tl_`
`\tl_build_put_left:Ne` `build_put_left_right:NNn`, by just adding the `\right` material after the `{\left}` in
`\tl_build_put_left:Nx` the e-expanding assignment.
`\tl_build_gput_left:Nn`
`\tl_build_gput_left:Ne`
`\tl_build_gput_left:Nx`
`__tl_build_put_left:NNn`

```

13353 \cs_new_protected:Npn \tl_build_put_left:Nn #1
13354 { \__tl_build_put_left:NNn \cs_set_nopar:Npe #1 }
13355 \cs_generate_variant:Nn \tl_build_put_left:Nn { Ne , Nx }
13356 \cs_new_protected:Npn \tl_build_gput_left:Nn #1
13357 { \__tl_build_put_left:NNn \cs_gset_nopar:Npe #1 }
13358 \cs_generate_variant:Nn \tl_build_gput_left:Nn { Ne , Nx }
13359 \cs_new_protected:Npn \__tl_build_put_left:NNn #1#2#3
13360 {
13361   #1 #2
13362   {
13363     \__kernel_exp_not:w \exp_after:wN
13364     {
13365       \exp:w \exp_after:wN \__tl_build_put:nn
13366       \exp_after:wN {#2} {#3}
13367     }
13368   }
13369 }

```

(End of definition for `\tl_build_put_left:Nn`, `\tl_build_gput_left:Nn`, and `__tl_build_put_left:NNn`. These functions are documented on page 127.)

`\tl_build_end:N` Get the data then clear the `<next tl>` recursively until finding an empty one. It is perhaps wasteful to repeatedly use `\cs_to_str:N`. The local/global scope is checked by `\tl_set:Ne` or `\tl_gset:Ne`.

```

13370 \cs_new_protected:Npn \tl_build_end:N #1
13371 {
13372   \__tl_build_get:NNN \__kernel_tl_set:Nx #1 #1
13373   \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_clear:N
13374 }
13375 \cs_new_protected:Npn \tl_build_gend:N #1
13376 {
13377   \__tl_build_get:NNN \__kernel_tl_gset:Nx #1 #1
13378   \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_gclear:N
13379 }
13380 \cs_new_protected:Npn \__tl_build_end_loop:NN #1#2
13381 {
13382   \if_meaning:w \c_empty_tl #1
13383   \exp_after:wN \use_none:nnnnnn
13384   \fi:
13385   #2 #1
13386   \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } #2
13387 }

```

(End of definition for `\tl_build_end:N`, `\tl_build_gend:N`, and `__tl_build_end_loop:NN`. These functions are documented on page 128.)

`\tl_build_get_intermediate:NN`

```

13388 \cs_new_protected:Npn \tl_build_get_intermediate:NN
13389 { \__tl_build_get:NNN \__kernel_tl_set:Nx }

```

(End of definition for `\tl_build_get_intermediate:NN`. This function is documented on page 128.)

`__tl_build_get:NNN` The idea is to expand the `<tl var>` then the `<next tl>` and so on, all within an expanding assignment, and wrap as appropriate in `\exp_not:n`. The various `<left>` parts are left in the assignment as we go, which enables us to expand the `<next tl>` at

the right place. The various *<right>* parts are eventually picked up in one last `\exp_not:n`, with a brace trick to wrap all the *<right>* parts together.

```

13390 \cs_new_protected:Npn \__tl_build_get:NNN #1#2#3
13391   { #1 #3 { \if_false: { \fi: \exp_after:wN \__tl_build_get:w #2 } } }
13392 \cs_new:Npn \__tl_build_get:w #1 \__tl_build_last:NNn #2#3#4
13393   {
13394     \exp_not:n {#4}
13395     \if_meaning:w \c_empty_tl #3
13396       \exp_after:wN \__tl_build_get_end:w
13397     \fi:
13398     \exp_after:wN \__tl_build_get:w #3
13399   }
13400 \cs_new:Npn \__tl_build_get_end:w #1#2#3
13401   { \__kernel_exp_not:w \exp_after:wN { \if_false: } \fi: }

```

(End of definition for `__tl_build_get:NNN`, `__tl_build_get:w`, and `__tl_build_get_end:w`.)

```

13402 </package>

```

Chapter 55

l3str implementation

```
13403 (*package)
```

```
13404 (@@=str)
```

55.1 Internal auxiliaries

`\s__str_mark` Internal scan marks.

```
\s__str_stop 13405 \scan_new:N \s__str_mark
13406 \scan_new:N \s__str_stop
```

(End of definition for \s__str_mark and \s__str_stop.)

`_str_use_none_delimit_by_s_stop:w` Functions to gobble up to a scan mark.

```
\_str_use_i_delimit_by_s_stop:nw 13407 \cs_new:Npn \_str_use_none_delimit_by_s_stop:w #1 \s__str_stop { }
13408 \cs_new:Npn \_str_use_i_delimit_by_s_stop:nw #1 #2 \s__str_stop {#1}
```

(End of definition for _str_use_none_delimit_by_s_stop:w and _str_use_i_delimit_by_s_stop:nw.)

`\q__str_recursion_tail` Internal recursion quarks.

```
\q__str_recursion_stop 13409 \quark_new:N \q__str_recursion_tail
13410 \quark_new:N \q__str_recursion_stop
```

(End of definition for \q__str_recursion_tail and \q__str_recursion_stop.)

`_str_if_recursion_tail_break:NN` Functions to query recursion quarks.

```
\_str_if_recursion_tail_stop_do:Nn 13411 \__kernel_quark_new_test:N \_str_if_recursion_tail_break:NN
13412 \__kernel_quark_new_test:N \_str_if_recursion_tail_stop_do:Nn
```

(End of definition for _str_if_recursion_tail_break:NN and _str_if_recursion_tail_stop_do:Nn.)

55.2 Creating and setting string variables

`\str_new:N` A string is simply a token list. The full mapping system isn't set up yet so do things by hand.

```

\str_new:c
\str_use:N
\str_use:c
\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c
\str_clear_new:N
\str_clear_new:c
\str_gclear_new:N
\str_gclear_new:c
\str_set_eq:NN
\str_set_eq:cN
\str_set_eq:Nc
\str_set_eq:cc
\str_gset_eq:NN
\str_gset_eq:cN
\str_gset_eq:Nc
\str_gset_eq:cc
\str_concat:NNN
\str_concat:ccc
\str_gconcat:NNN
\str_gconcat:ccc
13413 \group_begin:
13414   \cs_set_protected:Npn \__str_tmp:n #1
13415     {
13416       \tl_if_blank:nF {#1}
13417         {
13418           \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
13419           \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
13420           \__str_tmp:n
13421         }
13422       }
13423   \__str_tmp:n
13424     { new }
13425     { use }
13426     { clear }
13427     { gclear }
13428     { clear_new }
13429     { gclear_new }
13430     { }
13431 \group_end:
13432 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
13433 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
13434 \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }
13435 \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
13436 \cs_new_eq:NN \str_concat:NNN \tl_concat:NNN
13437 \cs_new_eq:NN \str_gconcat:NNN \tl_gconcat:NNN
13438 \cs_generate_variant:Nn \str_concat:NNN { ccc }
13439 \cs_generate_variant:Nn \str_gconcat:NNN { ccc }

```

(End of definition for `\str_new:N` and others. These functions are documented on page 130.)

`\str_set:Nn` Similar to corresponding `\tl` base functions, except that `__kernel_exp_not:w` is replaced with `__kernel_tl_to_str:w`. Just like token list, string constants use `\cs_gset_nopar:Npe` instead of `__kernel_tl_gset:Nx` so that the scope checking for `c` is applied when `\l3debug` is used. To maintain backward compatibility, in `\str_(g)put_left:Nn` and `\str_(g)put_right:Nn`, contents of string variables are wrapped in `__kernel_exp_not:w` to prevent further expansion.

```

\str_set:cV
\str_set:ce
\str_set:cx
13440 \cs_new_protected:Npn \str_set:Nn #1#2
13441   { \__kernel_tl_set:Nx #1 { \__kernel_tl_to_str:w {#2} } }
\str_gset:Nn
\str_gset:NV
\str_gset:Ne
\str_gset:Nx
\str_gset:cn
\str_gset:cV
\str_gset:ce
\str_gset:cx
13442 \cs_gset_protected:Npn \str_gset:Nn #1#2
13443   { \__kernel_tl_gset:Nx #1 { \__kernel_tl_to_str:w {#2} } }
13444 \cs_new_protected:Npn \str_const:Nn #1#2
13445   {
13446     \__kernel_chk_if_free_cs:N #1
13447     \cs_gset_nopar:Npe #1 { \__kernel_tl_to_str:w {#2} }
13448   }
13449 \cs_new_protected:Npn \str_put_left:Nn #1#2
13450   {
13451     \__kernel_tl_set:Nx #1
13452     { \__kernel_tl_to_str:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
13453   }
\str_const:Nx
\str_const:cn
\str_const:cV
\str_const:ce
\str_const:cx
\str_put_left:Nn
\str_put_left:NV
\str_put_left:Ne
\str_put_left:Nx
\str_put_left:cn

```

```

13454 \cs_new_protected:Npn \str_gput_left:Nn #1#2
13455   {
13456     \__kernel_tl_gset:Nx #1
13457     { \__kernel_tl_to_str:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
13458   }
13459 \cs_new_protected:Npn \str_put_right:Nn #1#2
13460   {
13461     \__kernel_tl_set:Nx #1
13462     { \__kernel_exp_not:w \exp_after:wN {#1} \__kernel_tl_to_str:w {#2} }
13463   }
13464 \cs_new_protected:Npn \str_gput_right:Nn #1#2
13465   {
13466     \__kernel_tl_gset:Nx #1
13467     { \__kernel_exp_not:w \exp_after:wN {#1} \__kernel_tl_to_str:w {#2} }
13468   }
13469 \cs_generate_variant:Nn \str_set:Nn      { NV , Ne , Nx , c , cV , ce , cx }
13470 \cs_generate_variant:Nn \str_gset:Nn     { NV , Ne , Nx , c , cV , ce , cx }
13471 \cs_generate_variant:Nn \str_const:Nn    { NV , Ne , Nx , c , cV , ce , cx }
13472 \cs_generate_variant:Nn \str_put_left:Nn { NV , Ne , Nx , c , cV , ce , cx }
13473 \cs_generate_variant:Nn \str_gput_left:Nn { NV , Ne , Nx , c , cV , ce , cx }
13474 \cs_generate_variant:Nn \str_put_right:Nn { NV , Ne , Nx , c , cV , ce , cx }
13475 \cs_generate_variant:Nn \str_gput_right:Nn { NV , Ne , Nx , c , cV , ce , cx }

```

(End of definition for `\str_set:Nn` and others. These functions are documented on page 131.)

55.3 Modifying string variables

`\str_replace_all:Nnn` Start by applying `\tl_to_str:n` to convert the old and new token lists to strings, and `\str_replace_all:cnn` also apply `\tl_to_str:N` to avoid any issues if we are fed a token list variable. Then `\str_greplace_all:Nnn` the code is a much simplified version of the token list code because neither the delimiter `\str_greplace_all:cnn` nor the replacement can contain macro parameters or braces. The delimiter `\s_str_`
`\str_replace_once:Nnn` `mark` cannot appear in the string to edit so it is used in all cases. Some e-expansion is unnecessary. There is no need to avoid losing braces nor to protect against expansion. `\str_replace_once:cnn` The ending code is much simplified and does not need to hide in braces.

```

13476 \cs_new_protected:Npn \str_replace_once:Nnn
13477   { \__str_replace:NNNnn \prg_do_nothing: \__kernel_tl_set:Nx }
13478 \cs_new_protected:Npn \str_greplace_once:Nnn
13479   { \__str_replace:NNNnn \prg_do_nothing: \__kernel_tl_gset:Nx }
13480 \cs_new_protected:Npn \str_replace_all:Nnn
13481   { \__str_replace:NNNnn \__str_replace_next:w \__kernel_tl_set:Nx }
13482 \cs_new_protected:Npn \str_greplace_all:Nnn
13483   { \__str_replace:NNNnn \__str_replace_next:w \__kernel_tl_gset:Nx }
13484 \cs_generate_variant:Nn \str_replace_once:Nnn { c }
13485 \cs_generate_variant:Nn \str_greplace_once:Nnn { c }
13486 \cs_generate_variant:Nn \str_replace_all:Nnn { c }
13487 \cs_generate_variant:Nn \str_greplace_all:Nnn { c }
13488 \cs_new_protected:Npn \__str_replace:NNNnn #1#2#3#4#5
13489   {
13490     \tl_if_empty:nTF {#4}
13491     {
13492       \msg_error:nne { kernel } { empty-search-pattern } {#5}
13493     }
13494     {

```



```

13495     \use:e
13496     {
13497         \exp_not:n { \__str_replace_aux:NNNnnn #1 #2 #3 }
13498         { \tl_to_str:N #3 }
13499         { \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
13500     }
13501 }
13502 }
13503 \cs_new_protected:Npn \__str_replace_aux:NNNnnn #1#2#3#4#5#6
13504 {
13505     \cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
13506     #2 #3
13507     {
13508         \__str_replace_next:w
13509         #4
13510         \__str_use_none_delimit_by_s_stop:w
13511         #5
13512         \s__str_stop
13513     }
13514 }
13515 \cs_new_eq:NN \__str_replace_next:w ?

```

(End of definition for `\str_replace_all:Nnn` and others. These functions are documented on page 138.)

```

\str_remove_once:Nn Removal is just a special case of replacement.
\str_remove_once:cn 13516 \cs_new_protected:Npn \str_remove_once:Nn #1#2
\str_gremove_once:Nn 13517 { \str_replace_once:Nnn #1 {#2} { } }
\str_gremove_once:cn 13518 \cs_new_protected:Npn \str_gremove_once:Nn #1#2
13519 { \str_greplace_once:Nnn #1 {#2} { } }
13520 \cs_generate_variant:Nn \str_remove_once:Nn { c }
13521 \cs_generate_variant:Nn \str_gremove_once:Nn { c }

```

(End of definition for `\str_remove_once:Nn` and `\str_gremove_once:Nn`. These functions are documented on page 138.)

```

\str_remove_all:Nn Removal is just a special case of replacement.
\str_remove_all:cn 13522 \cs_new_protected:Npn \str_remove_all:Nn #1#2
\str_gremove_all:Nn 13523 { \str_replace_all:Nnn #1 {#2} { } }
\str_gremove_all:cn 13524 \cs_new_protected:Npn \str_gremove_all:Nn #1#2
13525 { \str_greplace_all:Nnn #1 {#2} { } }
13526 \cs_generate_variant:Nn \str_remove_all:Nn { c }
13527 \cs_generate_variant:Nn \str_gremove_all:Nn { c }

```

(End of definition for `\str_remove_all:Nn` and `\str_gremove_all:Nn`. These functions are documented on page 138.)

55.4 String comparisons

```

\str_if_empty_p:N More copy-paste!
\str_if_empty_p:c 13528 \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N
\str_if_empty:NTF 13529 { p , T , F , TF }
\str_if_empty:cTF 13530 \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c
\str_if_empty_p:n 13531 { p , T , F , TF }
\str_if_empty:nTF 13532 \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N
\str_if_exist_p:N
\str_if_exist_p:c
\str_if_exist:NTF
\str_if_exist:cTF

```

```

13533 { p , T , F , TF }
13534 \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c
13535 { p , T , F , TF }
13536 \prg_new_eq_conditional:NNn \str_if_empty:n \tl_if_empty:n
13537 { p , T , F , TF }

```

(End of definition for `\str_if_empty:NTF`, `\str_if_empty:nTF`, and `\str_if_exist:NTF`. These functions are documented on page 131.)

`__str_if_eq:nn` String comparisons rely on the primitive `\(pdf)strcmp`, so we define a new name for it.

```

13538 \cs_new_eq:NN \__str_if_eq:nn \tex_strcmp:D

```

(End of definition for `__str_if_eq:nn`.)

`\str_compare_p:nNn` Simply rely on `__str_if_eq:nn`, which expands to -1, 0 or 1. The `ee` version is created directly because it is more efficient.

`\str_compare_p:eNe`

`\str_compare:nNnTF`

`\str_compare:eNeTF`

```

13539 \prg_new_conditional:Npnn \str_compare:nNn #1#2#3 { p , T , F , TF }
13540 {
13541   \if_int_compare:w
13542     \__str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#3} }
13543     #2 \c_zero_int
13544     \prg_return_true: \else: \prg_return_false: \fi:
13545 }
13546 \prg_new_conditional:Npnn \str_compare:eNe #1#2#3 { p , T , F , TF }
13547 {
13548   \if_int_compare:w \__str_if_eq:nn {#1} {#3} #2 \c_zero_int
13549   \prg_return_true: \else: \prg_return_false: \fi:
13550 }

```

(End of definition for `\str_compare:nNnTF`. This function is documented on page 133.)

`\str_if_eq_p:nn` Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore makes life a bit clearer. The `nn` and `ee` versions are created directly as this is most efficient. Since `__str_if_eq:nn` will expand to 0 as an explicit character with category 12 if the two lists match (and either -1 or 1 if they don't) we can use `\if:w` here which is faster than using `\if_int_compare:w`.

`\str_if_eq_p:Vn`

`\str_if_eq_p:on`

`\str_if_eq_p:nV`

`\str_if_eq_p:no`

`\str_if_eq_p:VV`

`\str_if_eq_p:ee`

`\str_if_eq:nnTF`

`\str_if_eq:VnTF`

`\str_if_eq:onTF`

`\str_if_eq:nVTF`

`\str_if_eq:noTF`

`\str_if_eq:VVTF`

`\str_if_eq:eeTF`

```

13551 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
13552 {
13553   \if:w 0 \__str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
13554   \prg_return_true: \else: \prg_return_false: \fi:
13555 }
13556 \prg_generate_conditional_variant:Nnn \str_if_eq:nn
13557 { V , v , o , nV , no , VV , nv } { p , T , F , TF }
13558 \prg_new_conditional:Npnn \str_if_eq:ee #1#2 { p , T , F , TF }
13559 {
13560   \if:w 0 \__str_if_eq:nn {#1} {#2}
13561   \prg_return_true: \else: \prg_return_false: \fi:
13562 }

```

(End of definition for `\str_if_eq:nnTF`. This function is documented on page 132.)

`\str_if_eq_p:NN` Note that `\str_if_eq:NNTF` is different from `\tl_if_eq:NNTF` because it needs to ignore category codes.

`\str_if_eq_p:Nc`

`\str_if_eq_p:cN`

`\str_if_eq_p:cc`

`\str_if_eq:NNTF`

`\str_if_eq:NcTF`

`\str_if_eq:cNTF`

`\str_if_eq:ccTF`

```

13563 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
13564 {

```

```

13565     \if:w 0 \__str_if_eq:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
13566     \prg_return_true: \else: \prg_return_false: \fi:
13567   }
13568 \prg_generate_conditional_variant:Nnn \str_if_eq:NN
13569   { c , Nc , cc } { T , F , TF , p }

```

(End of definition for `\str_if_eq:NNTF`. This function is documented on page 131.)

`\str_if_in:NnTF` Everything here needs to be detokenized but beyond that it is a simple token list test.
`\str_if_in:cnTF` It would be faster to fine-tune the T, F, TF variants by calling the appropriate variant of
`\str_if_in:nnTF` `\tl_if_in:nnTF` directly but that takes more code.

```

13570 \prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 { T , F , TF }
13571 {
13572   \use:e
13573   { \tl_if_in:nnTF { \tl_to_str:N #1 } { \tl_to_str:n {#2} } }
13574   { \prg_return_true: } { \prg_return_false: }
13575 }
13576 \prg_generate_conditional_variant:Nnn \str_if_in:Nn
13577   { c } { T , F , TF }
13578 \prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 { T , F , TF }
13579 {
13580   \use:e
13581   { \tl_if_in:nnTF { \tl_to_str:n {#1} } { \tl_to_str:n {#2} } }
13582   { \prg_return_true: } { \prg_return_false: }
13583 }

```

(End of definition for `\str_if_in:NnTF` and `\str_if_in:nnTF`. These functions are documented on page 132.)

`\str_case:nn` The aim here is to allow the case statement to be evaluated using a known number of
`\str_case:Vn` expansion steps (two), and without needing to use an explicit “end of recursion” marker.
`\str_case:on` That is achieved by using the test input as the final case, as this is always true. The
`\str_case:en` trick is then to tidy up the output such that the appropriate case code plus either the
`\str_case:nV` true or false branch code is inserted.
`\str_case:nv`

```

13584 \cs_new:Npn \str_case:nn #1#2
13585 {
13586   \exp:w
13587   \__str_case:nnTF {#1} {#2} { } { }
13588 }
13589 \cs_new:Npn \str_case:nnT #1#2#3
13590 {
13591   \exp:w
13592   \__str_case:nnTF {#1} {#2} {#3} { }
13593 }
13594 \cs_new:Npn \str_case:nnF #1#2
13595 {
13596   \exp:w
13597   \__str_case:nnTF {#1} {#2} { }
13598 }
13599 \cs_new:Npn \str_case:nnTF #1#2
13600 {
13601   \exp:w
13602   \__str_case:nnTF {#1} {#2}
13603 }

```

`\str_case:VnTF`
`\str_case:onTF`
`\str_case:enTF`
`\str_case:nVTF`
`\str_case:Nn`
`\str_case:NnTF`
`\str_case_e:nn`
`\str_case_e:en`
`\str_case_e:nnTF`
`\str_case_e:enTF`
`__str_case:nnTF`
`__str_case_e:nnTF`
`__str_case:nw`
`__str_case_e:nw`
`__str_case_end:nw`

```

13604 \cs_new:Npn \__str_case:nnTF #1#2#3#4
13605   { \__str_case:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
13606 \cs_generate_variant:Nn \str_case:nn { V , o , e , nV , nv }
13607 \prg_generate_conditional_variant:Nnn \str_case:nn
13608   { V , o , e , nV , nv } { T , F , TF }
13609 \cs_new_eq:NN \str_case:Nn \str_case:Vn
13610 \cs_new_eq:NN \str_case:NnT \str_case:VnT
13611 \cs_new_eq:NN \str_case:NnF \str_case:VnF
13612 \cs_new_eq:NN \str_case:NnTF \str_case:VnTF
13613 \cs_new:Npn \__str_case:nw #1#2#3
13614   {
13615     \str_if_eq:nnTF {#1} {#2}
13616     { \__str_case_end:nw {#3} }
13617     { \__str_case:nw {#1} }
13618   }
13619 \cs_new:Npn \str_case_e:nn #1#2
13620   {
13621     \exp:w
13622     \__str_case_e:nnTF {#1} {#2} { } { }
13623   }
13624 \cs_new:Npn \str_case_e:nnT #1#2#3
13625   {
13626     \exp:w
13627     \__str_case_e:nnTF {#1} {#2} {#3} { }
13628   }
13629 \cs_new:Npn \str_case_e:nnF #1#2
13630   {
13631     \exp:w
13632     \__str_case_e:nnTF {#1} {#2} { }
13633   }
13634 \cs_new:Npn \str_case_e:nnTF #1#2
13635   {
13636     \exp:w
13637     \__str_case_e:nnTF {#1} {#2}
13638   }
13639 \cs_new:Npn \__str_case_e:nnTF #1#2#3#4
13640   { \__str_case_e:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
13641 \cs_generate_variant:Nn \str_case_e:nn { e }
13642 \prg_generate_conditional_variant:Nnn \str_case_e:nn { e } { T , F , TF }
13643 \cs_new:Npn \__str_case_e:nw #1#2#3
13644   {
13645     \str_if_eq:eeTF {#1} {#2}
13646     { \__str_case_end:nw {#3} }
13647     { \__str_case_e:nw {#1} }
13648   }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare \s__str_mark and the false branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first \s__str_mark and so #4 is the **false** code (the true code is mopped up by #3).

```

13649 \cs_new:Npn \__str_case_end:nw #1#2#3 \s__str_mark #4#5 \s__str_stop

```

```
13650 { \exp_end: #1 #4 }
```

(End of definition for `\str_case:nnTF` and others. These functions are documented on page 132.)

55.5 Mapping over strings

<pre>\str_map_function:NN \str_map_function:cN \str_map_function:nN \str_map_inline:Nn \str_map_inline:cN \str_map_inline:nN \str_map_variable:NNn \str_map_variable:cNn \str_map_variable:nNn \str_map_break: \str_map_break:n __str_map_function:w __str_map_function:nn __str_map_inline:NN __str_map_variable:NnN</pre>	<p>The inline and variable mappings are similar to the usual token list mappings but start out by turning the argument to an “other string”. Doing the same for the expandable function mapping would require <code>__kernel_str_to_other:n</code>, quadratic in the string length. To deal with spaces in that case, <code>__str_map_function:w</code> replaces the following space by a braced space and a further call to itself. These are received by <code>__str_map_function:nn</code>, which passes the space to <code>#1</code> and calls <code>__str_map_function:w</code> to deal with the next space. The space before the braced space allows to optimize the <code>\q__str_recursion_tail</code> test. Of course we need to include a trailing space (the question mark is needed to avoid losing the space when <code>TeX</code> tokenizes the line). At the cost of about three more auxiliaries this code could get a 9 times speed up by testing only every 9-th character for whether it is <code>\q__str_recursion_tail</code> (also by converting 9 spaces at a time in the <code>\str_map_function:nN</code> case).</p> <p>For the <code>map_variable</code> functions we use a string assignment to store each character because spaces are made catcode 12 before the loop.</p>
---	---

```
13651 \cs_new:Npn \str_map_function:nN #1#2
13652 {
13653   \exp_after:wN \__str_map_function:w
13654   \exp_after:wN \__str_map_function:nn \exp_after:wN #2
13655   \__kernel_tl_to_str:w {#1}
13656   \q__str_recursion_tail ? ~
13657   \prg_break_point:Nn \str_map_break: { }
13658 }
13659 \cs_new:Npn \str_map_function:NN
13660 { \exp_args:No \str_map_function:nN }
13661 \cs_new:Npn \__str_map_function:w #1 ~
13662 { #1 { ~ { ~ } \__str_map_function:w } }
13663 \cs_new:Npn \__str_map_function:nn #1#2
13664 {
13665   \if_meaning:w \q__str_recursion_tail #2
13666   \exp_after:wN \str_map_break:
13667   \fi:
13668   #1 #2 \__str_map_function:nn {#1}
13669 }
13670 \cs_generate_variant:Nn \str_map_function:NN { c }
13671 \cs_new_protected:Npn \str_map_inline:nn #1#2
13672 {
13673   \int_gincr:N \g__kernel_prg_map_int
13674   \cs_gset_protected:cpn
13675   { \__str_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
13676   \use:e
13677   {
13678     \exp_not:N \__str_map_inline:NN
13679     \exp_not:c { \__str_map_ \int_use:N \g__kernel_prg_map_int :w }
13680     \__kernel_str_to_other_fast:n {#1}
13681   }
13682   \q__str_recursion_tail
```

```

13683     \prg_break_point:Nn \str_map_break:
13684     { \int_gdecr:N \g__kernel_prg_map_int }
13685   }
13686 \cs_new_protected:Npn \str_map_inline:Nn
13687   { \exp_args:No \str_map_inline:nn }
13688 \cs_generate_variant:Nn \str_map_inline:Nn { c }
13689 \cs_new:Npn \__str_map_inline:NN #1#2
13690   {
13691     \__str_if_recursion_tail_break:NN #2 \str_map_break:
13692     \exp_args:No #1 { \token_to_str:N #2 }
13693     \__str_map_inline:NN #1
13694   }
13695 \cs_new_protected:Npn \str_map_variable:nNn #1#2#3
13696   {
13697     \use:e
13698     {
13699       \exp_not:n { \__str_map_variable:NnN #2 {#3} }
13700       \__kernel_str_to_other_fast:n {#1}
13701     }
13702     \q__str_recursion_tail
13703     \prg_break_point:Nn \str_map_break: { }
13704   }
13705 \cs_new_protected:Npn \str_map_variable:NNn
13706   { \exp_args:No \str_map_variable:nNn }
13707 \cs_new_protected:Npn \__str_map_variable:NnN #1#2#3
13708   {
13709     \__str_if_recursion_tail_break:NN #3 \str_map_break:
13710     \str_set:Nn #1 {#3}
13711     \use:n {#2}
13712     \__str_map_variable:NnN #1 {#2}
13713   }
13714 \cs_generate_variant:Nn \str_map_variable:NNn { c }
13715 \cs_new:Npn \str_map_break:
13716   { \prg_map_break:Nn \str_map_break: { } }
13717 \cs_new:Npn \str_map_break:n
13718   { \prg_map_break:Nn \str_map_break: }

```

(End of definition for `\str_map_function:NN` and others. These functions are documented on page 133.)

`\str_map_tokens:Nn` Uses an auxiliary of `\str_map_function:NN`.

```

\str_map_tokens:cn 13719 \cs_new:Npn \str_map_tokens:nn #1#2
\str_map_tokens:nn 13720   {
13721     \exp_args:Nno \use:nn
13722     { \__str_map_function:w \__str_map_function:nn {#2} }
13723     { \__kernel_tl_to_str:w {#1} }
13724     \q__str_recursion_tail ? ~
13725     \prg_break_point:Nn \str_map_break: { }
13726   }
13727 \cs_new:Npn \str_map_tokens:Nn { \exp_args:No \str_map_tokens:nn }
13728 \cs_generate_variant:Nn \str_map_tokens:NNn { c }

```

(End of definition for `\str_map_tokens:Nn` and `\str_map_tokens:nn`. These functions are documented on page 134.)

55.6 Accessing specific characters in a string

`__kernel_str_to_other:n` First apply `\tl_to_str:n`, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the `\s__str_mark` and `\s__str_stop` markers. `__str_to_other_loop:w` The end is detected when `__str_to_other_loop:w` finds one of the trailing A, distinguished from any contents of the initial token list by their category. Then `__str_to_other_end:w` is called, and finds the result between `\s__str_mark` and the first A (well, there is also the need to remove a space).

```

13729 \cs_new:Npn \__kernel_str_to_other:n #1
13730   {
13731     \exp_after:wN \__str_to_other_loop:w
13732     \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_mark \s__str_stop
13733   }
13734 \group_begin:
13735 \tex_lccode:D ‘\* = ‘\ %
13736 \tex_lccode:D ‘\A = ‘\A %
13737 \tex_lowercase:D
13738   {
13739     \group_end:
13740     \cs_new:Npn \__str_to_other_loop:w
13741       #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \s__str_stop
13742     {
13743       \if_meaning:w A #8
13744         \__str_to_other_end:w
13745       \fi:
13746       \__str_to_other_loop:w
13747       #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \s__str_stop
13748     }
13749     \cs_new:Npn \__str_to_other_end:w \fi: #1 \s__str_mark #2 * A #3 \s__str_stop
13750     { \fi: #2 }
13751   }

```

(End of definition for `__kernel_str_to_other:n`, `__str_to_other_loop:w`, and `__str_to_other_end:w`.)

`__kernel_str_to_other_fast:n` The difference with `__kernel_str_to_other:n` is that the converted part is left in the input stream, making these commands only restricted-expandable. `__kernel_str_to_other_fast_loop:w` `__str_to_other_fast_end:w`

```

13752 \cs_new:Npn \__kernel_str_to_other_fast:n #1
13753   {
13754     \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
13755     A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_stop
13756   }
13757 \group_begin:
13758 \tex_lccode:D ‘\* = ‘\ %
13759 \tex_lccode:D ‘\A = ‘\A %
13760 \tex_lowercase:D
13761   {
13762     \group_end:
13763     \cs_new:Npn \__str_to_other_fast_loop:w
13764       #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
13765     {
13766       \if_meaning:w A #9
13767         \__str_to_other_fast_end:w
13768       \fi:

```

```

13769         #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
13770         \_str_to_other_fast_loop:w *
13771     }
13772     \cs_new:Npn \_str_to_other_fast_end:w #1 * A #2 \s__str_stop {#1}
13773 }

```

(End of definition for `_kernel_str_to_other_fast:n`, `_kernel_str_to_other_fast_loop:w`, and `_str_to_other_fast_end:w`.)

`\str_item:Nn` The `\str_item:nn` hands its argument with spaces escaped to `_str_item:nn`, and `\str_item:cn` makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The `\str_item_ignore_spaces:nn` function does not escape spaces, which are thus ignored by `_str_item:nn` since everything else is done with undelimited arguments. Evaluate the `<index>` argument #2 and count characters in the string, passing those two numbers to `_str_item:w` for further analysis. If the `<index>` is negative, shift it by the `<count>` to know the how many character to discard, and if that is still negative give an empty result. If the `<index>` is larger than the `<count>`, give an empty result, and otherwise discard `<index> - 1` characters before returning the following one. The shift by `-1` is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the `<index>` is zero.

`\str_item_ignore_spaces:nn`
`_str_item:nn`
`_str_item:w`

```

13774 \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
13775 \cs_generate_variant:Nn \str_item:Nn { c }
13776 \cs_new:Npn \str_item:nn #1#2
13777 {
13778     \exp_args:Nf \tl_to_str:n
13779     {
13780         \exp_args:Nf \_str_item:nn
13781         { \_kernel_str_to_other:n {#1} } {#2}
13782     }
13783 }
13784 \cs_new:Npn \str_item_ignore_spaces:nn #1
13785 { \exp_args:No \_str_item:nn { \tl_to_str:n {#1} } }
13786 \cs_new:Npn \_str_item:nn #1#2
13787 {
13788     \exp_after:wN \_str_item:w
13789     \int_value:w \int_eval:n {#2} \exp_after:wN ;
13790     \int_value:w \_str_count:n {#1} ;
13791     #1 \s__str_stop
13792 }
13793 \cs_new:Npn \_str_item:w #1; #2;
13794 {
13795     \int_compare:nNnTF {#1} < 0
13796     {
13797         \int_compare:nNnTF {#1} < {-#2}
13798         { \_str_use_none_delimit_by_s_stop:w }
13799         {
13800             \exp_after:wN \_str_use_i_delimit_by_s_stop:nw
13801             \exp:w \exp_after:wN \_str_skip_exp_end:w
13802             \int_value:w \int_eval:n { #1 + #2 } ;
13803         }
13804     }
13805     {
13806         \int_compare:nNnTF {#1} > {#2}
13807         { \_str_use_none_delimit_by_s_stop:w }

```



```

13808     {
13809     \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
13810     \exp:w \__str_skip_exp_end:w #1 ; { }
13811     }
13812   }
13813 }

```

(End of definition for `\str_item:Nn` and others. These functions are documented on page 136.)

`__str_skip_exp_end:w` Removes `max(#1,0)` characters from the input stream, and then leaves `\exp_end:`. This should be expanded using `\exp:w`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `__str_skip_end:NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `__str_skip_end:NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\exp_end:` (see places where `__str_skip_exp_end:w` is called).

```

13814 \cs_new:Npn \__str_skip_exp_end:w #1;
13815   {
13816     \if_int_compare:w #1 > 8 \exp_stop_f:
13817     \exp_after:wN \__str_skip_loop:wNNNNNNNN
13818     \else:
13819     \exp_after:wN \__str_skip_end:w
13820     \int_value:w \int_eval:w
13821     \fi:
13822     #1 ;
13823   }
13824 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
13825   {
13826     \exp_after:wN \__str_skip_exp_end:w
13827     \int_value:w \int_eval:n { #1 - 8 } ;
13828   }
13829 \cs_new:Npn \__str_skip_end:w #1 ;
13830   {
13831     \exp_after:wN \__str_skip_end:NNNNNNNN
13832     \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
13833   }
13834 \cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }

```

(End of definition for `__str_skip_exp_end:w` and others.)

`\str_range:Nnn` Sanitize the string. Then evaluate the arguments. At this stage we also decrement the `<start index>`, since our goal is to know how many characters should be removed. `\str_range:nnn` Then limit the range to be non-negative and at most the length of the string (this avoids needing to check for the end of the string when grabbing characters), shifting negative numbers by the appropriate amount. Afterwards, skip characters, then keep some more, and finally drop the end of the string. `\str_range_ignore_spaces:nnn`

```

\__str_range:nnn
\__str_range:w
\__str_range:nw

```

```

13835 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
13836 \cs_generate_variant:Nn \str_range:Nnn { c }
13837 \cs_new:Npn \str_range:nnn #1#2#3
13838   {
13839     \exp_args:Nf \tl_to_str:n

```

```

13840     {
13841         \exp_args:Nf \__str_range:nnn
13842         { \__kernel_str_to_other:n {#1} } {#2} {#3}
13843     }
13844 }
13845 \cs_new:Npn \str_range_ignore_spaces:nnn #1
13846 { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
13847 \cs_new:Npn \__str_range:nnn #1#2#3
13848 {
13849     \exp_after:wN \__str_range:w
13850     \int_value:w \__str_count:n {#1} \exp_after:wN ;
13851     \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
13852     \int_value:w \int_eval:n {#3} ;
13853     #1 \s__str_stop
13854 }
13855 \cs_new:Npn \__str_range:w #1; #2; #3;
13856 {
13857     \exp_args:Nf \__str_range:nnw
13858     { \__str_range_normalize:nn {#2} {#1} }
13859     { \__str_range_normalize:nn {#3} {#1} }
13860 }
13861 \cs_new:Npn \__str_range:nnw #1#2
13862 {
13863     \exp_after:wN \__str_collect_delimit_by_q_stop:w
13864     \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
13865     \exp:w \__str_skip_exp_end:w #1 ;
13866 }

```

(End of definition for `\str_range:Nnn` and others. These functions are documented on page 137.)

`__str_range_normalize:nn` This function converts an `<index>` argument into an explicit position in the string (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the `<index>` #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

```

13867 \cs_new:Npn \__str_range_normalize:nn #1#2
13868 {
13869     \int_eval:n
13870     {
13871         \if_int_compare:w #1 < \c_zero_int
13872         \if_int_compare:w #1 < -#2 \exp_stop_f:
13873             0
13874         \else:
13875             #1 + #2 + 1
13876         \fi:
13877     \else:
13878         \if_int_compare:w #1 < #2 \exp_stop_f:
13879             #1
13880         \else:
13881             #2
13882         \fi:
13883     \fi:
13884     }
13885 }

```

(End of definition for `__str_range_normalize:nn`.)

`_str_collect_delimit_by_q_stop:w` Collects $\max(\#1,0)$ characters, and removes everything else until `\s__str_stop`. This is somewhat similar to `_str_skip_exp_end:w`, but accepts integer expression arguments. `_str_collect_loop:wn` This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `_str_collect_end:nnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by $\#1$ characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the `\or:` disappear.

```

13886 \cs_new:Npn \_str_collect_delimit_by_q_stop:w #1;
13887   { \_str_collect_loop:wn #1 ; { } }
13888 \cs_new:Npn \_str_collect_loop:wn #1 ;
13889   {
13890     \if_int_compare:w #1 > 7 \exp_stop_f:
13891     \exp_after:wN \_str_collect_loop:wnNNNNNNN
13892     \else:
13893     \exp_after:wN \_str_collect_end:wn
13894     \fi:
13895     #1 ;
13896   }
13897 \cs_new:Npn \_str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
13898   {
13899     \exp_after:wN \_str_collect_loop:wn
13900     \int_value:w \int_eval:n { #1 - 7 } ;
13901     { #2 #3#4#5#6#7#8#9 }
13902   }
13903 \cs_new:Npn \_str_collect_end:wn #1 ;
13904   {
13905     \exp_after:wN \_str_collect_end:nnnnnnnw
13906     \if_case:w \if_int_compare:w #1 > \c_zero_int
13907     #1 \else: 0 \fi: \exp_stop_f:
13908     \or: \or: \or: \or: \or: \or: \or: \fi:
13909   }
13910 \cs_new:Npn \_str_collect_end:nnnnnnnw #1#2#3#4#5#6#7#8 #9 \s__str_stop
13911   { #1#2#3#4#5#6#7#8 }

```

(End of definition for `_str_collect_delimit_by_q_stop:w` and others.)

55.7 Counting characters

`\str_count_spaces:N` To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing `X<number>`, and that `<number>` is added to the sum of 9 that precedes, to adjust the result.

```

13912 \cs_new:Npn \str_count_spaces:N
13913   { \exp_args:No \str_count_spaces:n }
13914 \cs_generate_variant:Nn \str_count_spaces:N { c }
13915 \cs_new:Npn \str_count_spaces:n #1
13916   {
13917     \int_eval:n
13918     {
13919       \exp_after:wN \_str_count_spaces_loop:w
13920       \tl_to_str:n {#1} ~
13921       X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
13922       \s__str_stop

```

```

13923     }
13924   }
13925   \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
13926   {
13927     \if_meaning:w X #9
13928     \__str_use_i_delimit_by_s_stop:nw
13929     \fi:
13930     9 + \__str_count_spaces_loop:w
13931   }

```

(End of definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `__str_count_spaces_loop:w`. These functions are documented on page 135.)

`\str_count:N` To count characters in a string we could first escape all spaces using `__kernel_str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, `\str_count:n` sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, `loop`, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The internal function `__str_count:n`, used in `\str_item:nn` and `\str_range:nnn`, is similar to `\str_count_ignore_spaces:n` but expects its argument to already be a string or a string with spaces escaped.

```

13932   \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
13933   \cs_generate_variant:Nn \str_count:N { c }
13934   \cs_new:Npn \str_count:n #1
13935   {
13936     \__str_count_aux:n
13937     {
13938       \str_count_spaces:n {#1}
13939       + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
13940     }
13941   }
13942   \cs_new:Npn \__str_count:n #1
13943   {
13944     \__str_count_aux:n
13945     { \__str_count_loop:NNNNNNNNN #1 }
13946   }
13947   \cs_new:Npn \str_count_ignore_spaces:n #1
13948   {
13949     \__str_count_aux:n
13950     { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
13951   }
13952   \cs_new:Npn \__str_count_aux:n #1
13953   {
13954     \int_eval:n
13955     {
13956       #1
13957       { X 8 } { X 7 } { X 6 }
13958       { X 5 } { X 4 } { X 3 }
13959       { X 2 } { X 1 } { X 0 }
13960     } \s__str_stop
13961   }
13962 }

```

```

13963 \cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
13964 {
13965   \if_meaning:w X #9
13966   \exp_after:wN \__str_use_none_delimit_by_s_stop:w
13967   \fi:
13968   9 + \__str_count_loop:NNNNNNNNN
13969 }

```

(End of definition for `\str_count:N` and others. These functions are documented on page 135.)

55.8 The first character in a string

`\str_head:N` The `_ignore_spaces` variant applies `\tl_to_str:n` then grabs the first item, thus skipping spaces. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`.
`\str_head:c` To circumvent the fact that \TeX skips spaces when grabbing undelimited macro parameters, `__str_head:w` takes an argument delimited by a space. If `#1` starts with a non-space character, `__str_use_i_delimit_by_s_stop:nw` leaves that in the input stream. On the other hand, if `#1` starts with a space, the `__str_head:w` takes an empty argument, and the single (initially braced) space in the definition of `__str_head:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `__str_use_i_delimit_by_s_stop:nw`.

```

13970 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
13971 \cs_generate_variant:Nn \str_head:N { c }
13972 \cs_new:Npn \str_head:n #1
13973 {
13974   \exp_after:wN \__str_head:w
13975   \tl_to_str:n {#1}
13976   { { } } ~ \s__str_stop
13977 }
13978 \cs_new:Npn \__str_head:w #1 ~ %
13979 { \__str_use_i_delimit_by_s_stop:nw #1 { ~ } }
13980 \cs_new:Npn \str_head_ignore_spaces:n #1
13981 {
13982   \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
13983   \tl_to_str:n {#1} { } \s__str_stop
13984 }

```

(End of definition for `\str_head:N` and others. These functions are documented on page 136.)

`\str_tail:N` Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N` `\if_charcode:w` `\scan_stop:.` This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:.` The auxiliary function then inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker `X` be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `__str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\s__str_mark`. One can check that an empty (or blank) string yields an empty tail.

```

13985 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
13986 \cs_generate_variant:Nn \str_tail:N { c }

```

```

13987 \cs_new:Npn \str_tail:n #1
13988   {
13989     \exp_after:wN \__str_tail_auxi:w
13990     \reverse_if:N \if_charcode:w
13991     \scan_stop: \tl_to_str:n {#1} X X \s__str_stop
13992   }
13993 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \s__str_stop { \fi: #1 }
13994 \cs_new:Npn \str_tail_ignore_spaces:n #1
13995   {
13996     \exp_after:wN \__str_tail_auxii:w
13997     \tl_to_str:n {#1} \s__str_mark \s__str_mark \s__str_stop
13998   }
13999 \cs_new:Npn \__str_tail_auxii:w #1 #2 \s__str_mark #3 \s__str_stop { #2 }

```

(End of definition for `\str_tail:N` and others. These functions are documented on page 136.)

55.9 String manipulation

```

\str_casefold:n Case changing for programmatic reasons is done by first detokenizing input then doing
\str_casefold:V a simple loop that only has to worry about spaces and everything else. The output is
\str_lowercase:n detokenized to allow data sharing with text-based case changing. Similarly, for 8-bit
\str_lowercase:f engines the multi-byte information is shared.
\str_uppercase:n
\str_uppercase:f
14000 \cs_new:Npn \str_casefold:n #1 { \__str_change_case:nn {#1} { casefold } }
14001 \cs_new:Npn \str_lowercase:n #1 { \__str_change_case:nn {#1} { lowercase } }
14002 \cs_new:Npn \str_uppercase:n #1 { \__str_change_case:nn {#1} { uppercase } }
14003 \cs_generate_variant:Nn \str_casefold:n { V }
14004 \cs_generate_variant:Nn \str_lowercase:n { f }
14005 \cs_generate_variant:Nn \str_uppercase:n { f }
14006 \cs_new:Npn \__str_change_case:nn #1
14007   {
14008     \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
14009     { \tl_to_str:n {#1} }
14010   }
14011 \cs_new:Npn \__str_change_case_aux:nn #1#2
14012   {
14013     \__str_change_case_loop:nw {#2} #1 \q__str_recursion_tail \q__str_recursion_stop
14014     \__str_change_case_result:n { }
14015   }
14016 \cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
14017   { #2 \__str_change_case_result:n { #3 #1 } }
14018 \cs_generate_variant:Nn \__str_change_case_output:nw { f }
14019 \cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2
14020   { \tl_to_str:n {#2} }
14021 \cs_new:Npn \__str_change_case_loop:nw #1#2 \q__str_recursion_stop
14022   {
14023     \tl_if_head_is_space:nTF {#2}
14024     { \__str_change_case_space:n }
14025     { \__str_change_case_char:nN }
14026     {#1} #2 \q__str_recursion_stop
14027   }
14028 \exp_last_unbraced:NNNN
14029 \cs_new:Npn \__str_change_case_space:n #1 \c_space_tl
14030   {

```

```

14031     \__str_change_case_output:nw { ~ }
14032     \__str_change_case_loop:nw {#1}
14033   }
14034 \cs_new:Npn \__str_change_case_char:nN #1#2
14035   {
14036     \__str_if_recursion_tail_stop_do:Nn #2
14037     { \__str_change_case_end:wn }
14038     \__str_change_case_codepoint:nN {#1} #2
14039   }
14040 \if_int_compare:w 0
14041   \cs_if_exist:NT \tex_XeTeXversion:D { 1 }
14042   \cs_if_exist:NT \tex luatexversion:D { 1 }
14043   > 0 \exp_stop_f:
14044   \cs_new:Npn \__str_change_case_codepoint:nN #1#2
14045     { \__str_change_case_char:fnn { \int_eval:n {'#2} } {#1} {#2} }
14046 \else:
14047   \cs_new:Npe \__str_change_case_codepoint:nN #1#2
14048     {
14049     \exp_not:N \int_compare:nNnTF {'#2} > { "80 }
14050     {
14051       \cs_if_exist:NTF \tex_pdftexversion:D
14052       { \exp_not:N \__str_change_case_char_auxi:nN }
14053       {
14054         \exp_not:N \int_compare:nNnTF {'#2} > { "FF }
14055         { \exp_not:N \__str_change_case_char_auxii:nN }
14056         { \exp_not:N \__str_change_case_char_auxi:nN }
14057       }
14058     }
14059     { \exp_not:N \__str_change_case_char_auxii:nN }
14060     {#1} #2
14061   }
14062 \cs_new:Npn \__str_change_case_char_auxi:nN #1#2
14063   {
14064     \int_compare:nNnTF {'#2} < { "E0 }
14065     { \__str_change_case_codepoint:nNN }
14066     {
14067       \int_compare:nNnTF {'#2} < { "F0 }
14068       { \__str_change_case_codepoint:nNNN }
14069       { \__str_change_case_codepoint:nNNNNN }
14070     }
14071     {#1} #2
14072   }
14073 \cs_new:Npn \__str_change_case_char_auxii:nN #1#2
14074   { \__str_change_case_char:fnn { \int_eval:n {'#2} } {#1} {#2} }
14075 \cs_new:Npn \__str_change_case_codepoint:nNN #1#2#3
14076   {
14077     \__str_change_case_char:fnn
14078     { \int_eval:n { ('#2 - "C0) * "40 + '#3 - "80 } }
14079     {#1} {#2#3}
14080   }
14081 \cs_new:Npn \__str_change_case_codepoint:nNNN #1#2#3#4
14082   {
14083     \__str_change_case_char:fnn
14084     {

```

```

14085         \int_eval:n
14086         { ('#2 - "E0) * "1000 + ('#3 - "80) * "40 + '#4 - "80 }
14087     }
14088     {#1} {#2#3#4}
14089 }
14090 \cs_new:Npn \__str_change_case_codepoint:nNNNN #1#2#3#4#5
14091 {
14092     \__str_change_case_char:fnn
14093     {
14094         \int_eval:n
14095         {
14096             ('#2 - "F0) * "40000
14097             + ('#3 - "80) * "1000
14098             + ('#4 - "80) * "40
14099             + '#5 - "80
14100         }
14101     }
14102     {#1} {#2#3#4#5}
14103 }
14104 \fi:
14105 \cs_new:Npn \__str_change_case_char:nnn #1#2#3
14106 {
14107     \__str_change_case_output:fw
14108     {
14109         \exp_args:Ne \__str_change_case_char_aux:nnn
14110         { \__kernel_codepoint_case:nn {#2} {#1} } {#1} {#3}
14111     }
14112     \__str_change_case_loop:nw {#2}
14113 }
14114 \cs_generate_variant:Nn \__str_change_case_char:nnn { f }
14115 \cs_new:Npn \__str_change_case_char_aux:nnn #1#2#3
14116 {
14117     \use:e { \__str_change_case_char:nnnnn #1 {#2} {#3} }
14118 }
14119 \cs_new:Npn \__str_change_case_char:nnnnn #1#2#3#4#5
14120 {
14121     \int_compare:nNnTF {#1} = {#4}
14122     { \tl_to_str:n {#5} }
14123     {
14124         \codepoint_str_generate:n {#1}
14125         \tl_if_blank:nF {#2}
14126         {
14127             \codepoint_str_generate:n {#2}
14128             \tl_if_blank:nF {#3}
14129             { \codepoint_str_generate:n {#3} }
14130         }
14131     }
14132 }

```

(End of definition for `\str_casefold:n` and others. These functions are documented on page 140.)

`\str_mdfive_hash:n`

`\str_mdfive_hash:e`

```

14133 \cs_new:Npn \str_mdfive_hash:n #1 { \tex_mdffivesum:D { \tl_to_str:n {#1} } }
14134 \cs_new:Npn \str_mdfive_hash:e #1 { \tex_mdffivesum:D {#1} }

```


(End of definition for `\str_mdfive_hash:n`. This function is documented on page 140.)

`\c_ampersand_str` For all of those strings, use `\cs_to_str:N` to get characters with the correct category
`\c_atsign_str` code without worries

```

14135 \str_const:Ne \c_ampersand_str { \cs_to_str:N \& }
14136 \str_const:Ne \c_atsign_str { \cs_to_str:N \@ }
14137 \str_const:Ne \c_backslash_str { \cs_to_str:N \\ }
14138 \str_const:Ne \c_left_brace_str { \cs_to_str:N \{ }
14139 \str_const:Ne \c_right_brace_str { \cs_to_str:N \} }
14140 \str_const:Ne \c_circumflex_str { \cs_to_str:N \^ }
14141 \str_const:Ne \c_colon_str { \cs_to_str:N \: }
14142 \str_const:Ne \c_dollar_str { \cs_to_str:N \$ }
14143 \str_const:Ne \c_hash_str { \cs_to_str:N \# }
14144 \str_const:Ne \c_percent_str { \cs_to_str:N \% }
14145 \str_const:Ne \c_tilde_str { \cs_to_str:N \~ }
14146 \str_const:Ne \c_underscore_str { \cs_to_str:N \_ }
14147 \str_const:Ne \c_zero_str { 0 }

```

(End of definition for `\c_ampersand_str` and others. These variables are documented on page 141.)

`\c_empty_str` An empty string is simply an empty token list.

```

14148 \cs_new_eq:NN \c_empty_str \c_empty_tl

```

(End of definition for `\c_empty_str`. This variable is documented on page 141.)

`\l_tmpa_str` Scratch strings.
`\l_tmpb_str`
`\g_tmpa_str`
`\g_tmpb_str`

```

14149 \str_new:N \l_tmpa_str
14150 \str_new:N \l_tmpb_str
14151 \str_new:N \g_tmpa_str
14152 \str_new:N \g_tmpb_str

```

(End of definition for `\l_tmpa_str` and others. These variables are documented on page 141.)

55.10 Viewing strings

`\str_show:n` Displays a string on the terminal.
`\str_show:N`
`\str_show:c`
`\str_log:n`
`\str_log:N`
`\str_log:c`

```

14153 \cs_new_eq:NN \str_show:n \tl_show:n
14154 \cs_new_protected:Npn \str_show:N #1
14155 {
14156   \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
14157   { \tl_show:N #1 }
14158 }
14159 \cs_generate_variant:Nn \str_show:N { c }
14160 \cs_new_eq:NN \str_log:n \tl_log:n
14161 \cs_new_protected:Npn \str_log:N #1
14162 {
14163   \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
14164   { \tl_log:N #1 }
14165 }
14166 \cs_generate_variant:Nn \str_log:N { c }

```

(End of definition for `\str_show:n` and others. These functions are documented on page 140.)

```

14167 </package>

```

Chapter 56

l3str-convert implementation

```
14168 (*package)
```

```
14169 (@@=str)
```

56.1 Helpers

56.1.1 Variables and constants

```
  \__str_tmp:w Internal scratch space for some functions.  
\l__str_internal_tl 14170 \cs_new_protected:Npn \__str_tmp:w { }  
14171 \tl_new:N \l__str_internal_tl
```

(End of definition for __str_tmp:w and \l__str_internal_tl.)

```
\g__str_result_tl The \g__str_result_tl variable is used to hold the result of various internal string  
operations (mostly conversions) which are typically performed in a group. The variable  
is global so that it remains defined outside the group, to be assigned to a user-provided  
variable.
```

```
14172 \tl_new:N \g__str_result_tl
```

(End of definition for \g__str_result_tl.)

```
\c__str_replacement_char_int When converting, invalid bytes are replaced by the Unicode replacement character  
"FFFD.
```

```
14173 \int_const:Nn \c__str_replacement_char_int { "FFFD }
```

(End of definition for \c__str_replacement_char_int.)

```
\c__str_max_byte_int The maximal byte number.  
14174 \int_const:Nn \c__str_max_byte_int { 255 }
```

(End of definition for \c__str_max_byte_int.)

```
\s__str Internal scan marks.
```

```
14175 \scan_new:N \s__str
```

(End of definition for \s__str.)

`\q__str_nil` Internal quarks.

```
14176 \quark_new:N \q__str_nil
```

(End of definition for \q__str_nil.)

`\g__str_alias_prop` To avoid needing one file per encoding/escaping alias, we keep track of those in a property list.

```
14177 \prop_new:N \g__str_alias_prop
14178 \prop_gput:Nnn \g__str_alias_prop { latin1 } { iso88591 }
14179 \prop_gput:Nnn \g__str_alias_prop { latin2 } { iso88592 }
14180 \prop_gput:Nnn \g__str_alias_prop { latin3 } { iso88593 }
14181 \prop_gput:Nnn \g__str_alias_prop { latin4 } { iso88594 }
14182 \prop_gput:Nnn \g__str_alias_prop { latin5 } { iso88599 }
14183 \prop_gput:Nnn \g__str_alias_prop { latin6 } { iso885910 }
14184 \prop_gput:Nnn \g__str_alias_prop { latin7 } { iso885913 }
14185 \prop_gput:Nnn \g__str_alias_prop { latin8 } { iso885914 }
14186 \prop_gput:Nnn \g__str_alias_prop { latin9 } { iso885915 }
14187 \prop_gput:Nnn \g__str_alias_prop { latin10 } { iso885916 }
14188 \prop_gput:Nnn \g__str_alias_prop { utf16le } { utf16 }
14189 \prop_gput:Nnn \g__str_alias_prop { utf16be } { utf16 }
14190 \prop_gput:Nnn \g__str_alias_prop { utf32le } { utf32 }
14191 \prop_gput:Nnn \g__str_alias_prop { utf32be } { utf32 }
14192 \prop_gput:Nnn \g__str_alias_prop { hexadecimal } { hex }
14193 \bool_lazy_any:nTF
14194 {
14195   \sys_if_engine luatex_p:
14196   \sys_if_engine xetex_p:
14197 }
14198 {
14199   \prop_gput:Nnn \g__str_alias_prop { default } { }
14200 }
14201 {
14202   \prop_gput:Nnn \g__str_alias_prop { default } { utf8 }
14203 }
```

(End of definition for \g__str_alias_prop.)

`\g__str_error_bool` In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.

```
14204 \bool_new:N \g__str_error_bool
```

(End of definition for \g__str_error_bool.)

`\l__str_byte_flag` `\l__str_error_flag` Conversions from one *<encoding>*/*<escaping>* pair to another are done within expanding assignments. Errors are signalled by raising the relevant flag.

```
14205 \flag_new:N \l__str_byte_flag
```

```
14206 \flag_new:N \l__str_error_flag
```

(End of definition for \l__str_byte_flag and \l__str_error_flag.)

56.2 String conditionals

```

\__str_if_contains_char:NnT      \__str_if_contains_char:nnTF {<token list>} <char>
\__str_if_contains_char:NnTF
\__str_if_contains_char:nnTF
  \__str_if_contains_char_aux:nn
  \__str_if_contains_char_auxi:nN
  \__str_if_contains_char_true:
14207 \prg_new_conditional:Npnn \__str_if_contains_char:Nn #1#2 { T , TF }
14208   {
14209     \exp_after:wN \__str_if_contains_char_aux:nn \exp_after:wN {#1} {#2}
14210     { \prg_break:n { ? \fi: } }
14211     \prg_break_point:
14212     \prg_return_false:
14213   }
14214 \cs_new:Npn \__str_if_contains_char_aux:nn #1#2
14215   { \__str_if_contains_char_auxi:nN {#2} #1 }
14216 \prg_new_conditional:Npnn \__str_if_contains_char:nn #1#2 { TF }
14217   {
14218     \__str_if_contains_char_auxi:nN {#2} #1 { \prg_break:n { ? \fi: } }
14219     \prg_break_point:
14220     \prg_return_false:
14221   }
14222 \cs_new:Npn \__str_if_contains_char_auxi:nN #1#2
14223   {
14224     \if_charcode:w #1 #2
14225     \exp_after:wN \__str_if_contains_char_true:
14226     \fi:
14227     \__str_if_contains_char_auxi:nN {#1}
14228   }
14229 \cs_new:Npn \__str_if_contains_char_true:
14230   { \prg_break:n { \prg_return_true: \use_none:n } }

```

(End of definition for `__str_if_contains_char:NnT` and others.)

```

\__str_octal_use:NTF      \__str_octal_use:NTF <token> {<true code>} {<false code>}

```

If the `<token>` is an octal digit, it is left in the input stream, *followed* by the `<true code>`. Otherwise, the `<false code>` is left in the input stream.

T_EXhackers note: This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. T_EX dutifully detects octal digits for us: if #1 is an octal digit, then the right-hand side of the comparison is '1#1, greater than 1. Otherwise, the right-hand side stops as '1, and the conditional takes the false branch.

```

14231 \prg_new_conditional:Npnn \__str_octal_use:N #1 { TF }
14232   {
14233     \if_int_compare:w 1 < '1 \token_to_str:N #1 \exp_stop_f:
14234     #1 \prg_return_true:
14235     \else:
14236     \prg_return_false:
14237     \fi:
14238   }

```

(End of definition for `__str_octal_use:NTF`.)

`__str_hexadecimal_use:NTF` T_EX detects uppercase hexadecimal digits for us (see `__str_octal_use:NTF`), but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

14239 \prg_new_conditional:Npnn \__str_hexadecimal_use:N #1 { TF }
14240 {
14241   \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
14242     #1 \prg_return_true:
14243   \else:
14244     \if_case:w \int_eval:n { \exp_after:wN ' \token_to_str:N #1 - 'a }
14245       A
14246     \or: B
14247     \or: C
14248     \or: D
14249     \or: E
14250     \or: F
14251   \else:
14252     \prg_return_false:
14253     \exp_after:wN \use_none:n
14254   \fi:
14255   \prg_return_true:
14256 \fi:
14257 }

```

(End of definition for `__str_hexadecimal_use:NTF`.)

56.3 Conversions

56.3.1 Producing one byte or character

`\c__str_byte_0_tl` For each integer N in the range $[0, 255]$, we create a constant token list which holds three character tokens with category code other: the character with character code N , followed by the representation of N as two hexadecimal digits. The value -1 is given a default token list which ensures that later functions give an empty result for the input -1 .

```

14258 \group_begin:
14259   \__kernel_tl_set:Nx \l__str_internal_tl { \tl_to_str:n { 0123456789ABCDEF } }
14260   \tl_map_inline:Nn \l__str_internal_tl
14261     {
14262       \tl_map_inline:Nn \l__str_internal_tl
14263         {
14264           \tl_const:ce { c__str_byte_ \int_eval:n {"#1##1"} _tl }
14265           { \char_generate:nn { "#1##1" } { 12 } #1 ##1 }
14266         }
14267     }
14268 \group_end:
14269 \tl_const:cn { c__str_byte_-1_tl } { { } \use_none:n { } }

```

(End of definition for `\c__str_byte_0_tl` and others.)

`__str_output_byte:n` Those functions must be used carefully: feeding them a value outside the range $[-1, 255]$ will attempt to use the undefined token list variable `\c__str_byte_⟨number⟩_tl`. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value -1 produces an empty result in both cases.

```

14270 \cs_new:Npn \__str_output_byte:n #1
14271 { \__str_output_byte:w #1 \__str_output_end: }
14272 \cs_new:Npn \__str_output_byte:w
14273 {
14274   \exp_after:wN \exp_after:wN
14275   \exp_after:wN \use_i:nnn
14276   \cs:w c__str_byte_ \int_eval:w
14277 }
14278 \cs_new:Npn \__str_output_hexadecimal:n #1
14279 {
14280   \exp_after:wN \exp_after:wN
14281   \exp_after:wN \use_none:n
14282   \cs:w c__str_byte_ \int_eval:n {#1} _tl \cs_end:
14283 }
14284 \cs_new:Npn \__str_output_end:
14285 { \scan_stop: _tl \cs_end: }

```

(End of definition for `__str_output_byte:n` and others.)

`__str_output_byte_pair_be:n` Convert a number in the range [0,65535] to a pair of bytes, either big-endian or little-endian.

`__str_output_byte_pair_le:n`
`__str_output_byte_pair:nnN`

```

14286 \cs_new:Npn \__str_output_byte_pair_be:n #1
14287 {
14288   \exp_args:Nf \__str_output_byte_pair:nnN
14289   { \int_div_truncate:nn { #1 } { "100 } } {#1} \use:nn
14290 }
14291 \cs_new:Npn \__str_output_byte_pair_le:n #1
14292 {
14293   \exp_args:Nf \__str_output_byte_pair:nnN
14294   { \int_div_truncate:nn { #1 } { "100 } } {#1} \use_ii_i:nn
14295 }
14296 \cs_new:Npn \__str_output_byte_pair:nnN #1#2#3
14297 {
14298   #3
14299   { \__str_output_byte:n { #1 } }
14300   { \__str_output_byte:n { #2 - #1 * "100 } }
14301 }

```

(End of definition for `__str_output_byte_pair_be:n`, `__str_output_byte_pair_le:n`, and `__str_output_byte_pair:nnN`.)

56.3.2 Mapping functions for conversions

`__str_convert_gmap:N` This maps the function #1 over all characters in `\g__str_result_tl`, which should be a byte string in most cases, sometimes a native string.

`__str_convert_gmap_loop:NN`

```

14302 \cs_new_protected:Npn \__str_convert_gmap:N #1
14303 {
14304   \__kernel_tl_gset:Nx \g__str_result_tl
14305   {
14306     \exp_after:wN \__str_convert_gmap_loop:NN
14307     \exp_after:wN #1
14308     \g__str_result_tl { ? \prg_break: }
14309     \prg_break_point:
14310   }

```

```

14311 }
14312 \cs_new:Npn \__str_convert_gmap_loop:NN #1#2
14313 {
14314   \use_none:n #2
14315   #1#2
14316   \__str_convert_gmap_loop:NN #1
14317 }

```

(End of definition for __str_convert_gmap:N and __str_convert_gmap_loop:NN.)

__str_convert_gmap_internal:N This maps the function #1 over all character codes in \g__str_result_tl, which must
 __str_convert_gmap_internal_loop:Nw be in the internal representation.

```

14318 \cs_new_protected:Npn \__str_convert_gmap_internal:N #1
14319 {
14320   \__kernel_tl_gset:Nx \g__str_result_tl
14321   {
14322     \exp_after:wN \__str_convert_gmap_internal_loop:Nww
14323     \exp_after:wN #1
14324     \g__str_result_tl \s__str \s__str_stop \prg_break: \s__str
14325     \prg_break_point:
14326   }
14327 }
14328 \cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__str #3 \s__str
14329 {
14330   \__str_use_none_delimit_by_s_stop:w #3 \s__str_stop
14331   #1 {#3}
14332   \__str_convert_gmap_internal_loop:Nww #1
14333 }

```

(End of definition for __str_convert_gmap_internal:N and __str_convert_gmap_internal_loop:Nw.)

56.3.3 Error-reporting during conversion

__str_if_flag_error:Nne When converting using the function \str_set_convert:Nnnn, errors should be reported
 __str_if_flag_no_error:Nne to the user after each step in the conversion. Errors are signalled by raising some flag
 (typically @@_error), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions \str_set_convert:NnnnTF, errors should be suppressed. This is done by changing __str_if_flag_error:Nne into __str_if_flag_no_error:Nne locally.

```

14334 \cs_new_protected:Npn \__str_if_flag_error:Nne #1
14335 {
14336   \flag_if_raised:NTF #1
14337   { \msg_error:nne { str } }
14338   { \use_none:n }
14339 }
14340 \cs_new_protected:Npn \__str_if_flag_no_error:Nne #1#2#3
14341 { \flag_if_raised:NT #1 { \bool_gset_true:N \g__str_error_bool } }

```

(End of definition for __str_if_flag_error:Nne and __str_if_flag_no_error:Nne.)

__str_if_flag_times:NT At the end of each conversion step, we raise all relevant errors as one error message,
 built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints #2 followed by the number of occurrences of an error if it occurred, nothing otherwise.

```

14342 \cs_new:Npn \__str_if_flag_times:NT #1#2
14343   { \flag_if_raised:NT #1 { #2~(x \flag_height:N #1 ) } }

```

(End of definition for __str_if_flag_times:NT.)

56.3.4 Framework for conversions

Most functions in this module expect to be working with “native” strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of T_EX tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- “unescape” produces a string of bytes;
- “decode” takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

$$\langle \text{bytes} \rangle \backslash s_str \langle \text{Unicode code point} \rangle \backslash s_str$$

where we have collected the $\langle \text{bytes} \rangle$ which combined to form this particular Unicode character, and the $\langle \text{Unicode code point} \rangle$ is in the range $[0, "10FFFF]$.

- “encode” encodes the internal list of code points as a byte string in the new encoding;
- “escape” escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

```

\str_set_convert:Nnnn
\str_gset_convert:Nnnn
\str_set_convert:NnnnTF
\str_gset_convert:NnnnTF
\__str_convert:nNNnnn

```

The input string is stored in $\backslash g_str_result_tl$, then we: unescape and decode; encode and escape; exit the group and store the result in the user’s variable. The various conversion functions all act on $\backslash g_str_result_tl$. Errors are silenced for the conditional functions by redefining $\backslash __str_if_flag_error:Nne$ locally.

```

14344 \cs_new_protected:Npn \str_set_convert:Nnnn
14345   { \__str_convert:nNNnnn { } \tl_set_eq:NN }
14346 \cs_new_protected:Npn \str_gset_convert:Nnnn
14347   { \__str_convert:nNNnnn { } \tl_gset_eq:NN }
14348 \prg_new_protected_conditional:Npnn
14349   \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
14350   {
14351     \bool_gset_false:N \g__str_error_bool
14352     \__str_convert:nNNnnn
14353     { \cs_set_eq:NN \__str_if_flag_error:Nne \__str_if_flag_no_error:Nne }
14354     \tl_set_eq:NN #1 {#2} {#3} {#4}
14355     \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
14356   }
14357 \prg_new_protected_conditional:Npnn
14358   \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }

```



```

14359 {
14360   \bool_gset_false:N \g__str_error_bool
14361   \__str_convert:nNNnnn
14362   { \cs_set_eq:NN \__str_if_flag_error:Nne \__str_if_flag_no_error:Nne }
14363   \tl_gset_eq:NN #1 {#2} {#3} {#4}
14364   \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
14365 }
14366 \cs_new_protected:Npn \__str_convert:nNNnnn #1#2#3#4#5#6
14367 {
14368   \group_begin:
14369   #1
14370   \__kernel_tl_gset:Nx \g__str_result_tl { \__kernel_str_to_other_fast:n {#4} }
14371   \exp_after:wN \__str_convert:wwwnn
14372   \tl_to_str:n {#5} /// \s__str_stop
14373   { decode } { unescape }
14374   \prg_do_nothing:
14375   \__str_convert_decode_:
14376   \exp_after:wN \__str_convert:wwwnn
14377   \tl_to_str:n {#6} /// \s__str_stop
14378   { encode } { escape }
14379   \use_ii_i:nn
14380   \__str_convert_encode_:
14381   \__kernel_tl_gset:Nx \g__str_result_tl
14382   { \tl_to_str:V \g__str_result_tl }
14383   \group_end:
14384   #2 #3 \g__str_result_tl
14385 }

```

(End of definition for `\str_set_convert:Nnnn` and others. These functions are documented on page 144.)

`__str_convert:wwwnn` The task of `__str_convert:wwwnn` is to split $\langle\text{encoding}\rangle/\langle\text{escaping}\rangle$ pairs into their components, #1 and #2. Calls to `__str_convert:nnn` ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

- #1 is the encoding conversion function;
- #2 is the escaping function;
- #3 is the escaping name for use in an error message;
- #4 is `\prg_do_nothing:` for unescaping/decoding, and `\use_ii_i:nn` for encoding/escaping;
- #5 is the default encoding function (either “decode” or “encode”), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do #2#1 in this order, and in the encoding/escaping phase, the order should be reversed: #4#2#1 does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, #1, should be performed.

```

14386 \cs_new_protected:Npn \__str_convert:wwwnn
14387   #1 / #2 // #3 \s__str_stop #4#5
14388   {

```

```

14389     \__str_convert:nnn {enc} {#4} {#1}
14390     \__str_convert:nnn {esc} {#5} {#2}
14391     \exp_args:Ncc \__str_convert:NNnNN
14392     { \__str_convert_#4_#1: } { \__str_convert_#5_#2: } {#2}
14393   }
14394 \cs_new_protected:Npn \__str_convert:NNnNN #1#2#3#4#5
14395   {
14396     \if_meaning:w #1 #5
14397     \tl_if_empty:nF {#3}
14398     { \msg_error:nne { str } { native-escaping } {#3} }
14399     #1
14400     \else:
14401     #4 #2 #1
14402     \fi:
14403   }

```

(End of definition for `__str_convert:wwwnn` and `__str_convert:NNnNN`.)

`__str_convert:nnn` The arguments of `__str_convert:nnn` are: `enc` or `esc`, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three arguments, to `__str_convert:nynn`. The task is then to make sure that the conversion function `#3_#1` corresponding to the type `#3` and filtered name `#1` is defined, then set our initial conversion function `#3_#4` equal to that.

How do we get the `#3_#1` conversion to be defined if it isn't? Two main cases.

First, if `#1` is a key in `\g__str_alias_prop`, then the value `\l__str_internal_tl` tells us what file to load. Loading is skipped if the file was already read, *i.e.*, if the conversion command based on `\l__str_internal_tl` already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, `#1` may be absent from the property list. The `\cs_if_exist:cF` test is automatically false, and we search for a file defining the encoding or escaping `#1` (this should allow third-party `.def` files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on `\l__str_internal_tl` is defined, so we can set the `#3_#1` function equal to that. In some cases (*e.g.*, `utf16be`), the `#3_#1` function is actually defined within the file we just loaded, and it is different from the `\l__str_internal_tl`-based function: we mustn't clobber that different definition.

```

14404 \cs_new_protected:Npn \__str_convert:nnn #1#2#3
14405   {
14406     \cs_if_exist:cF { \__str_convert_#2_#3: }
14407     {
14408       \exp_args:Ne \__str_convert:nynn
14409       { \__str_convert_lowercase_alphanum:n {#3} }
14410       {#1} {#2} {#3}
14411     }
14412   }
14413 \cs_new_protected:Npn \__str_convert:nynn #1#2#3#4
14414   {
14415     \cs_if_exist:cF { \__str_convert_#3_#1: }
14416     {
14417       \prop_get:NnNF \g__str_alias_prop {#1} \l__str_internal_tl
14418       { \tl_set:Nn \l__str_internal_tl {#1} }

```

```

14419     \cs_if_exist:cF { __str_convert_#3_ \l__str_internal_tl : }
14420     {
14421         \file_if_exist:nTF { l3str-#2- \l__str_internal_tl .def }
14422         {
14423             \group_begin:
14424             \cctab_select:N \c_code_cctab
14425             \file_input:n { l3str-#2- \l__str_internal_tl .def }
14426             \group_end:
14427         }
14428         {
14429             \tl_clear:N \l__str_internal_tl
14430             \msg_error:nnee { str } { unknown-#2 } {#4} {#1}
14431         }
14432     }
14433     \cs_if_exist:cF { __str_convert_#3_#1: }
14434     {
14435         \cs_gset_eq:cc { __str_convert_#3_#1: }
14436         { __str_convert_#3_ \l__str_internal_tl : }
14437     }
14438 }
14439 \cs_gset_eq:cc { __str_convert_#3_#4: } { __str_convert_#3_#1: }
14440 }

```

(End of definition for __str_convert:nnn and __str_convert:nnnn.)

__str_convert_lowercase_alphanum:n
 __str_convert_lowercase_alphanum_loop:N

This function keeps only letters and digits, with upper case letters converted to lower case.

```

14441 \cs_new:Npn \__str_convert_lowercase_alphanum:n #1
14442 {
14443     \exp_after:wN \__str_convert_lowercase_alphanum_loop:N
14444     \tl_to_str:n {#1} { ? \prg_break: }
14445     \prg_break_point:
14446 }
14447 \cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
14448 {
14449     \use_none:n #1
14450     \if_int_compare:w '#1 > 'Z \exp_stop_f:
14451     \if_int_compare:w '#1 > 'z \exp_stop_f: \else:
14452         \if_int_compare:w '#1 < 'a \exp_stop_f: \else:
14453             #1
14454             \fi:
14455             \fi:
14456         \else:
14457             \if_int_compare:w '#1 < 'A \exp_stop_f:
14458             \if_int_compare:w 1 < 1#1 \exp_stop_f:
14459                 #1
14460                 \fi:
14461             \else:
14462                 \__str_output_byte:n { '#1 + 'a - 'A }
14463                 \fi:
14464             \fi:
14465             \__str_convert_lowercase_alphanum_loop:N
14466         }

```

(End of definition for `_str_convert_lowercase_alphanum:n` and `_str_convert_lowercase_alphanum_loop:N`.)

56.3.5 Byte unescape and escape

Strings of bytes may need to be stored in auxiliary files in safe “escaping” formats. Each such escaping is only loaded as needed. By default, on input any non-byte is filtered out, while the output simply consists in letting bytes through.

In the case of 8-bit engines, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag `\l__str_byte_flag`. Spaces have already been given the correct category code when this function is called.

`_str_filter_bytes:n`
`_str_filter_bytes_aux:N`

```

14467 \bool_lazy_any:nTF
14468 {
14469   \sys_if_engine luatex_p:
14470   \sys_if_engine xetex_p:
14471 }
14472 {
14473   \cs_new:Npn \_str_filter_bytes:n #1
14474     {
14475       \_str_filter_bytes_aux:N #1
14476       { ? \prg_break: }
14477       \prg_break_point:
14478     }
14479   \cs_new:Npn \_str_filter_bytes_aux:N #1
14480     {
14481       \use_none:n #1
14482       \if_int_compare:w '#1 < 256 \exp_stop_f:
14483         #1
14484       \else:
14485         \flag_raise:N \l__str_byte_flag
14486       \fi:
14487       \_str_filter_bytes_aux:N
14488     }
14489 }
14490 { \cs_new_eq:NN \_str_filter_bytes:n \use:n }

```

(End of definition for `_str_filter_bytes:n` and `_str_filter_bytes_aux:N`.)

`_str_convert_unescape_:` The simplest unescaping method removes non-bytes from `\g__str_result_tl`.

`_str_convert_unescape_bytes:`

```

14491 \bool_lazy_any:nTF
14492 {
14493   \sys_if_engine luatex_p:
14494   \sys_if_engine xetex_p:
14495 }
14496 {
14497   \cs_new_protected:Npn \_str_convert_unescape_:
14498     {
14499       \flag_clear:N \l__str_byte_flag
14500       \_kernel_tl_gset:Nx \g__str_result_tl
14501         { \exp_args:No \_str_filter_bytes:n \g__str_result_tl }
14502       \_str_if_flag_error:Nne \l__str_byte_flag { non-byte } { bytes }
14503     }
14504 }

```

```

14505 { \cs_new_protected:Npn \__str_convert_unescape_: { } }
14506 \cs_new_eq:NN \__str_convert_unescape_bytes: \__str_convert_unescape_:

```

(End of definition for __str_convert_unescape_: and __str_convert_unescape_bytes:.)

__str_convert_escape_: The simplest form of escape leaves the bytes from the previous step of the conversion
 __str_convert_escape_bytes: unchanged.

```

14507 \cs_new_protected:Npn \__str_convert_escape_: { }
14508 \cs_new_eq:NN \__str_convert_escape_bytes: \__str_convert_escape_:

```

(End of definition for __str_convert_escape_: and __str_convert_escape_bytes:.)

56.3.6 Native strings

__str_convert_decode_: Convert each character to its character code, one at a time.
 __str_decode_native_char:N

```

14509 \cs_new_protected:Npn \__str_convert_decode_:
14510 { \__str_convert_gmap:N \__str_decode_native_char:N }
14511 \cs_new:Npn \__str_decode_native_char:N #1
14512 { #1 \s__str \int_value:w '#1 \s__str }

```

(End of definition for __str_convert_decode_: and __str_decode_native_char:N.)

__str_convert_encode_: The conversion from an internal string to native character tokens basically maps \char_ -
 __str_encode_native_char:n generate:nn through the code-points, but in non-Unicode-aware engines we use a fall-
 back character ? rather than nothing when given a character code outside [0,255]. We
 detect the presence of bad characters using a flag and only produce a single error after
 the e-expanding assignment.

```

14513 \bool_lazy_any:nTF
14514 {
14515   \sys_if_engine luatex_p:
14516   \sys_if_engine xetex_p:
14517 }
14518 {
14519   \cs_new_protected:Npn \__str_convert_encode_:
14520   { \__str_convert_gmap_internal:N \__str_encode_native_char:n }
14521   \cs_new:Npn \__str_encode_native_char:n #1
14522   { \char_generate:nn {#1} {12} }
14523 }
14524 {
14525   \cs_new_protected:Npn \__str_convert_encode_:
14526   {
14527     \flag_clear:N \l__str_error_flag
14528     \__str_convert_gmap_internal:N \__str_encode_native_char:n
14529     \__str_if_flag_error:Nne \l__str_error_flag
14530     { native-overflow } { }
14531   }
14532   \cs_new:Npn \__str_encode_native_char:n #1
14533   {
14534     \if_int_compare:w #1 > \c__str_max_byte_int
14535     \flag_raise:N \l__str_error_flag
14536     ?
14537     \else:
14538     \char_generate:nn {#1} {12}
14539     \fi:

```

```

14540     }
14541 \msg_new:nnnn { str } { native-overflow }
14542     { Character-code-too-large-for-this-engine. }
14543     {
14544     This-engine-only-support-8-bit-characters:~
14545     valid-character-codes-are-in-the-range-[0,255].~
14546     To-manipulate-arbitrary-Unicode,~use-LuaTeX-or-XeTeX.
14547     }
14548 }

```

(End of definition for `__str_convert_encode_:` and `__str_encode_native_char:n`.)

56.3.7 `clist`

`__str_convert_decode_clist:` Convert each integer to the internal form. We first turn `\g__str_result_tl` into a `clist` variable, as this avoids problems with leading or trailing commas.

```

14549 \cs_new_protected:Npn \__str_convert_decode_clist:
14550 {
14551   \clist_gset:No \g__str_result_tl \g__str_result_tl
14552   \__kernel_tl_gset:Nx \g__str_result_tl
14553   {
14554     \exp_args:No \clist_map_function:nN
14555     \g__str_result_tl \__str_decode_clist_char:n
14556   }
14557 }
14558 \cs_new:Npn \__str_decode_clist_char:n #1
14559 { #1 \s__str \int_eval:n {#1} \s__str }

```

(End of definition for `__str_convert_decode_clist:` and `__str_decode_clist_char:n`.)

`__str_convert_encode_clist:` Convert the internal list of character codes to a comma-list of character codes. The first line produces a comma-list with a leading comma, removed in the next step (this also works in the empty case, since `\tl_tail:N` does not trigger an error in this case).

```

14560 \cs_new_protected:Npn \__str_convert_encode_clist:
14561 {
14562   \__str_convert_gmap_internal:N \__str_encode_clist_char:n
14563   \__kernel_tl_gset:Nx \g__str_result_tl { \tl_tail:N \g__str_result_tl }
14564 }
14565 \cs_new:Npn \__str_encode_clist_char:n #1 { , #1 }

```

(End of definition for `__str_convert_encode_clist:` and `__str_encode_clist_char:n`.)

56.3.8 8-bit encodings

It is not clear in what situations 8-bit encodings are used, hence it is not clear what should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings.

The data needed to support a given 8-bit encoding is stored in a file that consists of a single function call

```

\__str_declare_eight_bit_encoding:nnnn {<name>} {<modulo>} {<mapping>}
{<missing>}

```

This declares the encoding $\langle name \rangle$ to map bytes to Unicode characters according to the $\langle mapping \rangle$, and map those bytes which are not mentioned in the $\langle mapping \rangle$ either to the replacement character (if they appear in $\langle missing \rangle$), or to themselves. The $\langle mapping \rangle$ argument is a token list of pairs $\{\langle byte \rangle\} \{\langle Unicode \rangle\}$ expressed in uppercase hexadecimal notation. The $\langle missing \rangle$ argument is a token list of $\{\langle byte \rangle\}$. Every $\langle byte \rangle$ which does not appear in the $\langle mapping \rangle$ nor the $\langle missing \rangle$ lists maps to itself in Unicode, so for instance the `latin1` encoding has empty $\langle mapping \rangle$ and $\langle missing \rangle$ lists. The $\langle modulo \rangle$ is a (decimal) integer between 256 and 558 inclusive, modulo which all Unicode code points supported by the encodings must be different.

We use two integer arrays per encoding. When decoding we only use the `decode` integer array, with entry $n + 1$ (offset needed because integer array indices start at 1) equal to the Unicode code point that corresponds to the n -th byte in the encoding under consideration, or -1 if the given byte is invalid in this encoding. When encoding we use both arrays: upon seeing a code point n , we look up the entry $(1 \text{ plus } n \text{ modulo some number } M)$ in the `encode` array, which tells us the byte that might encode the given Unicode code point, then we check in the `decode` array that indeed this byte encodes the Unicode code point we want. Here, M is an encoding-dependent integer between 256 and 558 (it turns out), chosen so that among the Unicode code points that can be validly represented in the given encoding, no pair of code points have the same value modulo M .

Loop through both lists of bytes to fill in the `decode` integer array, then fill the `encode` array accordingly. For bytes that are invalid in the given encoding, store -1 in the `decode` array.

```

\__str_declare_eight_bit_encoding:nmmn
\__str_declare_eight_bit_aux:NNmmn
\__str_declare_eight_bit_loop:Nnn
\__str_declare_eight_bit_loop:Nn

```

```

14566 \cs_new_protected:Npn \__str_declare_eight_bit_encoding:nmmn #1
14567   {
14568     \tl_set:Nn \l__str_internal_tl {#1}
14569     \cs_new_protected:cpn { __str_convert_decode_#1: }
14570       { \__str_convert_decode_eight_bit:n {#1} }
14571     \cs_new_protected:cpn { __str_convert_encode_#1: }
14572       { \__str_convert_encode_eight_bit:n {#1} }
14573     \exp_args:Ncc \__str_declare_eight_bit_aux:NNmmn
14574       { g__str_decode_#1_intarray } { g__str_encode_#1_intarray }
14575   }
14576 \cs_new_protected:Npn \__str_declare_eight_bit_aux:NNmmn #1#2#3#4#5
14577   {
14578     \intarray_new:Nn #1 { 256 }
14579     \int_step_inline:nnn { 0 } { 255 }
14580       { \intarray_gset:Nnn #1 { 1 + ##1 } {##1} }
14581     \__str_declare_eight_bit_loop:Nnn #1
14582     #4 { \s__str_stop \prg_break: } { }
14583     \prg_break_point:
14584     \__str_declare_eight_bit_loop:Nn #1
14585     #5 { \s__str_stop \prg_break: }
14586     \prg_break_point:
14587     \intarray_new:Nn #2 {#3}
14588     \int_step_inline:nnn { 0 } { 255 }
14589     {
14590       \int_compare:nNf { \intarray_item:Nn #1 { 1 + ##1 } } = { -1 }
14591       {
14592         \intarray_gset:Nnn #2
14593         {
14594           1 +

```

```

14595         \int_mod:nn { \intarray_item:Nn #1 { 1 + ##1 } }
14596         { \intarray_count:N #2 }
14597     }
14598     {##1}
14599 }
14600 }
14601 }
14602 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nnn #1#2#3
14603 {
14604     \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
14605     \intarray_gset:Nnn #1 { 1 + "#2 } { "#3 }
14606     \__str_declare_eight_bit_loop:Nnn #1
14607 }
14608 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nn #1#2
14609 {
14610     \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
14611     \intarray_gset:Nnn #1 { 1 + "#2 } { -1 }
14612     \__str_declare_eight_bit_loop:Nn #1
14613 }

```

(End of definition for `__str_declare_eight_bit_encoding:n` and others.)

```

\__str_convert_decode_eight_bit:n
  \__str_decode_eight_bit_aux:n
  \__str_decode_eight_bit_aux:Nn

```

The map from bytes to Unicode code points is in the `decode` array corresponding to the given encoding. Define `__str_tmp:w` and pass it successively all bytes in the string. It produces an internal representation with suitable `\s__str` inserted, and the corresponding code point is obtained by looking it up in the integer array. If the entry is `-1` then issue a replacement character and raise the flag indicating that there was an error.

```

14614 \cs_new_protected:Npn \__str_convert_decode_eight_bit:n #1
14615 {
14616     \cs_set:Npe \__str_tmp:w
14617     {
14618         \exp_not:N \__str_decode_eight_bit_aux:Nn
14619         \exp_not:c { g__str_decode_#1_intarray }
14620     }
14621     \flag_clear:N \l__str_error_flag
14622     \__str_convert_gmap:N \__str_tmp:w
14623     \__str_if_flag_error:Nne \l__str_error_flag { decode-8-bit } {#1}
14624 }
14625 \cs_new:Npn \__str_decode_eight_bit_aux:Nn #1#2
14626 {
14627     #2 \s__str
14628     \exp_args:Nf \__str_decode_eight_bit_aux:n
14629     { \intarray_item:Nn #1 { 1 + '#2 } }
14630     \s__str
14631 }
14632 \cs_new:Npn \__str_decode_eight_bit_aux:n #1
14633 {
14634     \if_int_compare:w #1 < \c_zero_int
14635         \flag_raise:N \l__str_error_flag
14636         \int_value:w \c__str_replacement_char_int
14637     \else:
14638         #1
14639     \fi:
14640 }

```


(End of definition for `__str_convert_encode_eight_bit:n`, `__str_decode_eight_bit_aux:n`, and `__str_decode_eight_bit_aux:Nn`.)

`__str_convert_encode_eight_bit:n`
`__str_encode_eight_bit_aux:nnN`
`__str_encode_eight_bit_aux:NNn`

It is not practical to make an integer array with indices in the full Unicode range, so we work modulo some number, which is simply the size of the `encode` integer array for the given encoding. This gives us a candidate byte for representing a given Unicode code point. Of course taking the modulo leads to collisions so we check in the `decode` array that the byte we got is indeed correct. Otherwise the Unicode code point we started from is simply not representable in the given encoding.

```

14641 \int_new:N \l__str_modulo_int
14642 \cs_new_protected:Npn \__str_convert_encode_eight_bit:n #1
14643 {
14644   \cs_set:Npe \__str_tmp:w
14645   {
14646     \exp_not:N \__str_encode_eight_bit_aux:NNn
14647     \exp_not:c { g__str_encode_#1_intarray }
14648     \exp_not:c { g__str_decode_#1_intarray }
14649   }
14650   \flag_clear:N \l__str_error_flag
14651   \__str_convert_gmap_internal:N \__str_tmp:w
14652   \__str_if_flag_error:Nne \l__str_error_flag { encode-8-bit } {#1}
14653 }
14654 \cs_new:Npn \__str_encode_eight_bit_aux:NNn #1#2#3
14655 {
14656   \exp_args:Nf \__str_encode_eight_bit_aux:nnN
14657   {
14658     \intarray_item:Nn #1
14659     { 1 + \int_mod:nn {#3} { \intarray_count:N #1 } }
14660   }
14661   {#3}
14662   #2
14663 }
14664 \cs_new:Npn \__str_encode_eight_bit_aux:nnN #1#2#3
14665 {
14666   \int_compare:nNnTF { \intarray_item:Nn #3 { 1 + #1 } } = {#2}
14667   { \__str_output_byte:n {#1} }
14668   { \flag_raise:N \l__str_error_flag }
14669 }

```

(End of definition for `__str_convert_encode_eight_bit:n`, `__str_encode_eight_bit_aux:nnN`, and `__str_encode_eight_bit_aux:NNn`.)

56.4 Messages

General messages, and messages for the encodings and escapings loaded by default (“native”, and “bytes”).

```

14670 \msg_new:nnn { str } { unknown-esc }
14671 { Escaping-scheme~'#1'~(filtered:~'#2')~unknown. }
14672 \msg_new:nnn { str } { unknown-enc }
14673 { Encoding-scheme~'#1'~(filtered:~'#2')~unknown. }
14674 \msg_new:nnnn { str } { native-escaping }
14675 { The~'native'~encoding~scheme~does~not~support~any~escaping. }
14676 {

```

```

14677 Since~native~strings~do~not~consist~in~bytes,~
14678 none~of~the~escaping~methods~make~sense.~
14679 The~specified~escaping,~'#1',~will~be~ignored.
14680 }
14681 \msg_new:nnn { str } { file-not-found }
14682 { File~'l3str-#1.def'~not~found. }

```

Message used when the “bytes” unescaping fails because the string given to `\str_set_convert:Nnnn` contains a non-byte. This cannot happen for the -8-bit engines. Messages used for other escapings and encodings are defined in each definition file.

```

14683 \bool_lazy_any:nT
14684 {
14685   \sys_if_engine luatex_p:
14686   \sys_if_engine xetex_p:
14687 }
14688 {
14689   \msg_new:nnnn { str } { non-byte }
14690   { String~invalid~in~escaping~'#1':~it~may~only~contain~bytes. }
14691   {
14692     Some~characters~in~the~string~you~asked~to~convert~are~not~
14693     8-bit~characters.~Perhaps~the~string~is~a~'native'~Unicode~string?~
14694     If~it~is,~try~using\\
14695     \\
14696     \iow_indent:n
14697     {
14698       \iow_char:N\\str_set_convert:Nnnn \\
14699       \\ \ <str-var>~\{~<string>~\}~\{~<native-encoding>~\}~\{~<target-encoding>~\}
14700     }
14701   }
14702 }

```

Those messages are used when converting to and from 8-bit encodings.

```

14703 \msg_new:nnnn { str } { decode-8-bit }
14704 { Invalid~string~in~encoding~'#1'. }
14705 {
14706   LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
14707   any~character~in~the~encoding~'#1'.
14708 }
14709 \msg_new:nnnn { str } { encode-8-bit }
14710 { Unicode~string~cannot~be~converted~to~encoding~'#1'. }
14711 {
14712   The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
14713   LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
14714   string~contains~a~character~that~'#1'~does~not~support.
14715 }

```

56.5 Escaping definitions

Several of those encodings are defined by the pdf file format. The following byte storage methods are defined:

- `bytes` (default), non-bytes are filtered out, and bytes are left untouched (this is defined by default);

- `hex` or `hexadecimal`, as per the pdfTeX primitive `\pdfescapehex`
- `name`, as per the pdfTeX primitive `\pdfescapename`
- `string`, as per the pdfTeX primitive `\pdfescapestring`
- `url`, as per the percent encoding of urls.

56.5.1 Unescape methods

`__str_convert_unescape_hex:` Take chars two by two, and interpret each pair as the hexadecimal code for a byte.
`__str_unescape_hex_auxi:N` Anything else than hexadecimal digits is ignored, raising the flag. A string which contains
`__str_unescape_hex_auxii:N` an odd number of hexadecimal digits gets 0 appended to it: this is equivalent to appending a 0 in all cases, and dropping it if it is alone.

```

14716 \cs_new_protected:Npn \__str_convert_unescape_hex:
14717   {
14718     \group_begin:
14719     \flag_clear:N \l__str_error_flag
14720     \int_set:Nn \tex_escapechar:D { 92 }
14721     \__kernel_tl_gset:Nx \g__str_result_tl
14722     {
14723       \__str_output_byte:w "
14724       \exp_last_unbraced:Nf \__str_unescape_hex_auxi:N
14725       { \tl_to_str:N \g__str_result_tl }
14726       0 { ? 0 - 1 \prg_break: }
14727       \prg_break_point:
14728       \__str_output_end:
14729     }
14730     \__str_if_flag_error:Nne \l__str_error_flag { unescape-hex } { }
14731   \group_end:
14732 }
14733 \cs_new:Npn \__str_unescape_hex_auxi:N #1
14734   {
14735     \use_none:n #1
14736     \__str_hexadecimal_use:NTF #1
14737     { \__str_unescape_hex_auxii:N }
14738     {
14739       \flag_raise:N \l__str_error_flag
14740       \__str_unescape_hex_auxi:N
14741     }
14742   }
14743 \cs_new:Npn \__str_unescape_hex_auxii:N #1
14744   {
14745     \use_none:n #1
14746     \__str_hexadecimal_use:NTF #1
14747     {
14748       \__str_output_end:
14749       \__str_output_byte:w " \__str_unescape_hex_auxi:N
14750     }
14751     {
14752       \flag_raise:N \l__str_error_flag
14753       \__str_unescape_hex_auxii:N
14754     }
14755   }

```

```

14756 \msg_new:nnnn { str } { unescape-hex }
14757 { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
14758 {
14759   Some~characters~in~the~string~you~asked~to~convert~are~not~
14760   hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
14761 }

```

(End of definition for `__str_convert_unescape_hex:`, `__str_unescape_hex_auxi:N`, and `__str_unescape_hex_auxii:N`.)

```

__str_convert_unescape_name:
__str_unescape_name_loop:wNN
__str_convert_unescape_url:
__str_unescape_url_loop:wNN

```

The `__str_convert_unescape_name:` function replaces each occurrence of # followed by two hexadecimal digits in `\g__str_result_tl` by the corresponding byte. The `url` function is identical, with escape character % instead of #. Thus we define the two together. The arguments of `__str_tmp:w` are the character code of # or % in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

The looping auxiliary #3 finds the next escape character, reads the following two characters, and tests them. The test `__str_hexadecimal_use:NTF` leaves the uppercase digit in the input stream, hence we surround the test with `__str_output_byte:w "` and `__str_output_end:.` If both characters are hexadecimal digits, they should be removed before looping: this is done by `\use_i:nnn`. If one of the characters is not a hexadecimal digit, then feed "#1 to `__str_output_byte:w` to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove `\use_i:nnn`).

```

14762 \cs_set_protected:Npn \__str_tmp:w #1#2#3
14763 {
14764   \cs_new_protected:cpn { __str_convert_unescape_#2: }
14765   {
14766     \group_begin:
14767     \flag_clear:N \l__str_byte_flag
14768     \flag_clear:N \l__str_error_flag
14769     \int_set:Nn \tex_escapechar:D { 92 }
14770     \__kernel_tl_gset:Nx \g__str_result_tl
14771     {
14772       \exp_after:wN #3 \g__str_result_tl
14773       #1 ? { ? \prg_break: }
14774       \prg_break_point:
14775     }
14776     \__str_if_flag_error:Nne \l__str_byte_flag { non-byte } { #2 }
14777     \__str_if_flag_error:Nne \l__str_error_flag { unescape-#2 } { }
14778   \group_end:
14779 }
14780 \cs_new:Npn #3 ##1#1##2##3
14781 {
14782   \__str_filter_bytes:n {##1}
14783   \use_none:n ##3
14784   \__str_output_byte:w "
14785   \__str_hexadecimal_use:NTF ##2
14786   {
14787     \__str_hexadecimal_use:NTF ##3
14788     { }
14789     {
14790       \flag_raise:N \l__str_error_flag

```

```

14791         * 0 + '#1 \use_i:nn
14792     }
14793 }
14794 {
14795     \flag_raise:N \l__str_error_flag
14796     0 + '#1 \use_i:nn
14797 }
14798 \__str_output_end:
14799 \use_i:nnn #3 ##2##3
14800 }
14801 \msg_new:nnnn { str } { unescape-#2 }
14802 { String~invalid~in~escaping~'#2'. }
14803 {
14804     LaTeX~came~across~the~escape~character~'#1'~not~followed~by~
14805     two~hexadecimal~digits.~This~is~invalid~in~the~escaping~'#2'.
14806 }
14807 }
14808 \exp_after:wN \__str_tmp:w \c_hash_str { name }
14809 \__str_unescape_name_loop:wNN
14810 \exp_after:wN \__str_tmp:w \c_percent_str { url }
14811 \__str_unescape_url_loop:wNN

```

(End of definition for `__str_convert_unescape_name:` and others.)

```

\__str_convert_unescape_string:
\__str_unescape_string_newlines:wN
\__str_unescape_string_loop:wNNN
\__str_unescape_string_repeat:NNNNN

```

The `string` escaping is somewhat similar to the `name` and `url` escapings, with escape character `\`. The first step is to convert all three line endings, `^^J`, `^^M`, and `^^M^^J` to the common `^^J`, as per the PDF specification. This step cannot raise the flag.

Then the following escape sequences are decoded.

```

\ n Line feed (10)
\ r Carriage return (13)
\ t Horizontal tab (9)
\ b Backspace (8)
\ f Form feed (12)
\ ( Left parenthesis
\ ) Right parenthesis
\\ Backslash

```

`\ddd` (backslash followed by 1 to 3 octal digits) Byte `ddd` (octal), subtracting 256 in case of overflow.

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.

```

14812 \group_begin:
14813 \char_set_catcode_other:N ^^J
14814 \char_set_catcode_other:N ^^M
14815 \cs_set_protected:Npn \__str_tmp:w #1
14816 {
14817     \cs_new_protected:Npn \__str_convert_unescape_string:

```

```

14818 {
14819   \group_begin:
14820     \flag_clear:N \l__str_byte_flag
14821     \flag_clear:N \l__str_error_flag
14822     \int_set:Nn \tex_escapechar:D { 92 }
14823     \__kernel_tl_gset:Nx \g__str_result_tl
14824       {
14825         \exp_after:wN \__str_unescape_string_newlines:wN
14826         \g__str_result_tl \prg_break: ^M ?
14827         \prg_break_point:
14828       }
14829     \__kernel_tl_gset:Nx \g__str_result_tl
14830       {
14831         \exp_after:wN \__str_unescape_string_loop:wNNN
14832         \g__str_result_tl #1 ?? { ? \prg_break: }
14833         \prg_break_point:
14834       }
14835     \__str_if_flag_error:Nne \l__str_byte_flag { non-byte } { string }
14836     \__str_if_flag_error:Nne \l__str_error_flag { unescape-string } { }
14837   \group_end:
14838 }
14839 }
14840 \exp_args:No \__str_tmp:w { \c_backslash_str }
14841 \exp_last_unbraced:NNNNo
14842 \cs_new:Npn \__str_unescape_string_loop:wNNN #1 \c_backslash_str #2#3#4
14843 {
14844   \__str_filter_bytes:n {#1}
14845   \use_none:n #4
14846   \__str_output_byte:w '
14847   \__str_octal_use:NTF #2
14848   {
14849     \__str_octal_use:NTF #3
14850     {
14851       \__str_octal_use:NTF #4
14852       {
14853         \if_int_compare:w #2 > 3 \exp_stop_f:
14854         - 256
14855         \fi:
14856         \__str_unescape_string_repeat:NNNNNN
14857       }
14858       { \__str_unescape_string_repeat:NNNNNN ? }
14859     }
14860     { \__str_unescape_string_repeat:NNNNNN ?? }
14861   }
14862   {
14863     \str_case_e:nnF {#2}
14864     {
14865       { \c_backslash_str } { 134 }
14866       { ( ) } { 50 }
14867       { ) } { 51 }
14868       { r } { 15 }
14869       { f } { 14 }
14870       { n } { 12 }
14871       { t } { 11 }

```

```

14872         { b } { 10 }
14873         { ^^J } { 0 - 1 }
14874     }
14875     {
14876         \flag_raise:N \l__str_error_flag
14877         0 - 1 \use_i:nn
14878     }
14879 }
14880 \__str_output_end:
14881 \use_i:nn \__str_unescape_string_loop:wNNN #2#3#4
14882 }
14883 \cs_new:Npn \__str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
14884 { \__str_output_end: \__str_unescape_string_loop:wNNN }
14885 \cs_new:Npn \__str_unescape_string_newlines:wN #1 ^^M #2
14886 {
14887     #1
14888     \if_charcode:w ^^J #2 \else: ^^J \fi:
14889     \__str_unescape_string_newlines:wN #2
14890 }
14891 \msg_new:nmmn { str } { unescape-string }
14892 { String-invalid~in~escaping~'string'. }
14893 {
14894     LaTeX~came~across~an~escape~character~'\c_backslash_str'~
14895     not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'(',~')',~
14896     '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
14897     of~a~line.
14898 }
14899 \group_end:

```

(End of definition for `__str_convert_unescape_string:` and others.)

56.5.2 Escape methods

Currently, none of the escape methods can lead to errors, assuming that their input is made out of bytes.

```

\__str_convert_escape_hex: Loop and convert each byte to hexadecimal.
\__str_escape_hex_char:N
14900 \cs_new_protected:Npn \__str_convert_escape_hex:
14901 { \__str_convert_gmap:N \__str_escape_hex_char:N }
14902 \cs_new:Npn \__str_escape_hex_char:N #1
14903 { \__str_output_hexadecimal:n { '#1' } }

```

(End of definition for `__str_convert_escape_hex:` and `__str_escape_hex_char:N`.)

```

\__str_convert_escape_name: For each byte, test whether it should be output as is, or be “hash-encoded”. Roughly,
\__str_escape_name_char:n bytes outside the range ["2A, "7E] are hash-encoded. We keep two lists of exceptions:
\__str_if_escape_name:nTF characters in \c__str_escape_name_not_str are not hash-encoded, and characters in
\c__str_escape_name_str the \c__str_escape_name_str are encoded.
\c__str_escape_name_not_str
14904 \str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } %$
14905 \str_const:Nn \c__str_escape_name_str { {}/<>[] }
14906 \cs_new_protected:Npn \__str_convert_escape_name:
14907 { \__str_convert_gmap:N \__str_escape_name_char:n }
14908 \cs_new:Npn \__str_escape_name_char:n #1
14909 {

```

```

14910     \__str_if_escape_name:nTF {#1} {#1}
14911     { \c_hash_str \__str_output_hexadecimal:n {'#1} }
14912   }
14913 \prg_new_conditional:Npnn \__str_if_escape_name:n #1 { TF }
14914   {
14915     \if_int_compare:w '#1 < "2A \exp_stop_f:
14916     \__str_if_contains_char:NnTF \c__str_escape_name_not_str {#1}
14917     \prg_return_true: \prg_return_false:
14918   \else:
14919     \if_int_compare:w '#1 > "7E \exp_stop_f:
14920     \prg_return_false:
14921   \else:
14922     \__str_if_contains_char:NnTF \c__str_escape_name_str {#1}
14923     \prg_return_false: \prg_return_true:
14924   \fi:
14925 \fi:
14926   }

```

(End of definition for __str_convert_escape_name: and others.)

__str_convert_escape_string: Any character below (and including) space, and any character above (and including) del, are converted to octal. One backslash is added before each parenthesis and backslash.

```

\__str_escape_string_char:N
\__str_if_escape_string:N
\c__str_escape_string_str
14927 \str_const:Ne \c__str_escape_string_str
14928   { \c_backslash_str ( ) }
14929 \cs_new_protected:Npn \__str_convert_escape_string:
14930   { \__str_convert_gmap:N \__str_escape_string_char:N }
14931 \cs_new:Npn \__str_escape_string_char:N #1
14932   {
14933     \__str_if_escape_string:NTF #1
14934     {
14935       \__str_if_contains_char:NnT
14936       \c__str_escape_string_str {#1}
14937       { \c_backslash_str }
14938     #1
14939   }
14940   {
14941     \c_backslash_str
14942     \int_div_truncate:nn {'#1} {64}
14943     \int_mod:nn { \int_div_truncate:nn {'#1} { 8 } } { 8 }
14944     \int_mod:nn {'#1} { 8 }
14945   }
14946   }
14947 \prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
14948   {
14949     \if_int_compare:w '#1 < "27 \exp_stop_f:
14950     \prg_return_false:
14951   \else:
14952     \if_int_compare:w '#1 > "7A \exp_stop_f:
14953     \prg_return_false:
14954   \else:
14955     \prg_return_true:
14956   \fi:
14957 \fi:
14958   }

```


(End of definition for `__str_convert_escape_string:` and others.)

`__str_convert_escape_url:` This function is similar to `__str_convert_escape_name:`, escaping different characters.

```
\__str_escape_url_char:n 14959 \cs_new_protected:Npn \__str_convert_escape_url:
\__str_if_escape_url:nTF 14960 { \__str_convert_gmap:N \__str_escape_url_char:n }
14961 \cs_new:Npn \__str_escape_url_char:n #1
14962 {
14963   \__str_if_escape_url:nTF {#1} {#1}
14964   { \c_percent_str \__str_output_hexadecimal:n { '#1' } }
14965 }
14966 \prg_new_conditional:Npnn \__str_if_escape_url:n #1 { TF }
14967 {
14968   \if_int_compare:w '#1 < "30 \exp_stop_f:
14969   \__str_if_contains_char:nnTF { "-. } {#1}
14970   \prg_return_true: \prg_return_false:
14971   \else:
14972   \if_int_compare:w '#1 > "7E \exp_stop_f:
14973   \prg_return_false:
14974   \else:
14975   \__str_if_contains_char:nnTF { : ; = ? @ [ ] } {#1}
14976   \prg_return_false: \prg_return_true:
14977   \fi:
14978   \fi:
14979 }
```

(End of definition for `__str_convert_escape_url:`, `__str_escape_url_char:n`, and `__str_if_escape_url:nTF`.)

56.6 Encoding definitions

The `native` encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

- UTF-8;
- UTF-16, big-, little-endian, or with byte order mark;
- UTF-32, big-, little-endian, or with byte order mark;
- the ISO 8859 code pages, numbered from 1 to 16, skipping the inexistent ISO 8859-12.

56.6.1 utf-8 support

```
\__str_convert_encode_utf8: Loop through the internal string, and convert each character to its UTF-8 representation.
\__str_encode_utf_viii_char:n The representation is built from the right-most (least significant) byte to the left-most
\__str_encode_utf_viii_loop:wwnnw (most significant) byte. Continuation bytes are in the range [128, 191], taking 64 different
values, hence we roughly want to express the character code in base 64, shifting the
first digit in the representation by some number depending on how many continuation
bytes there are. In the range [0,127], output the corresponding byte directly. In the
range [128,2047], output the remainder modulo 64, plus 128 as a continuation byte, then
output the quotient (which is in the range [0,31]), shifted by 192. In the next range,
[2048,65535], split the character code into residue and quotient modulo 64, output the
```

residue as a first continuation byte, then repeat; this leaves us with a quotient in the range [0,15], which we output shifted by 224. The last range, [65536,1114111], follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? `__str_encode_utf_vii_loop:wwnw` takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in the ranges [0,127], [192,223], [224,239], and [240,247] (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient #1 is less than the limit #3 for that range, output the leading byte (#1 shifted by #4) and stop. Otherwise, we need one more step: use the quotient of #1 by 64, and #1 as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder #2 - 64#1 + 128. The bizarre construction `- 1 + 0 *` removes the spurious initial continuation byte (better methods welcome).

```

14980 \cs_new_protected:cpn { __str_convert_encode_utf8: }
14981   { \__str_convert_gmap_internal:N \__str_encode_utf_viii_char:n }
14982 \cs_new:Npn \__str_encode_utf_viii_char:n #1
14983   {
14984     \__str_encode_utf_viii_loop:wwnw #1 ; - 1 + 0 * ;
14985     { 128 } { 0 }
14986     { 32 } { 192 }
14987     { 16 } { 224 }
14988     { 8 } { 240 }
14989     \s__str_stop
14990   }
14991 \cs_new:Npn \__str_encode_utf_viii_loop:wwnw #1; #2; #3#4 #5 \s__str_stop
14992   {
14993     \if_int_compare:w #1 < #3 \exp_stop_f:
14994       \__str_output_byte:n { #1 + #4 }
14995       \exp_after:wN \__str_use_none_delimit_by_s_stop:w
14996     \fi:
14997     \exp_after:wN \__str_encode_utf_viii_loop:wwnw
14998       \int_value:w \int_div_truncate:nn {#1} {64} ; #1 ;
14999     #5 \s__str_stop
15000     \__str_output_byte:n { #2 - 64 * ( #1 - 2 ) }
15001   }

```

(End of definition for `__str_convert_encode_utf8:`, `__str_encode_utf_viii_char:n`, and `__str_encode_utf_viii_loop:wwnw`.)

`__str_missing` `__str_extra` `__str_overlong` `__str_overflow` When decoding a string that is purportedly in the UTF-8 encoding, four different errors can occur, signalled by a specific flag for each (we define those flags using `\flag_clear_new:N` rather than `\flag_new:N`, because they are shared with other encoding definition files).

- “Missing continuation byte”: a leading byte is not followed by the right number of continuation bytes.
- “Extra continuation byte”: a continuation byte appears where it was not expected, *i.e.*, not after an appropriate leading byte.

- “Overlong”: a Unicode character is expressed using more bytes than necessary, for instance, "C0"80 for the code point 0, instead of a single null byte.
- “Overflow”: this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one L^AT_EX3 error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct UTF-8 string should look like, then add error-specific information.

```

15002 \flag_clear_new:N \l__str_missing_flag
15003 \flag_clear_new:N \l__str_extra_flag
15004 \flag_clear_new:N \l__str_overlong_flag
15005 \flag_clear_new:N \l__str_overflow_flag
15006 \msg_new:nnnn { str } { utf8-decode }
15007 {
15008     Invalid-UTF-8-string:
15009     \exp_last_unbraced:Nf \use_none:n
15010     {
15011         \__str_if_flag_times:NT \l__str_missing_flag { ,~missing~continuation~byte }
15012         \__str_if_flag_times:NT \l__str_extra_flag { ,~extra~continuation~byte }
15013         \__str_if_flag_times:NT \l__str_overlong_flag { ,~overlong~form }
15014         \__str_if_flag_times:NT \l__str_overflow_flag { ,~code~point~too~large }
15015     }
15016     .
15017 }
15018 {
15019     In-the-UTF-8-encoding,~each-Unicode-character~consists-in-
15020     1-to-4-bytes,~with-the-following-bit-pattern: \
15021     \iow_indent:n
15022     {
15023         Code-point~\ \ \ \ <~128:~0xxxxxxx \
15024         Code-point~\ \ \ \ <~2048:~110xxxxx~10xxxxxx \
15025         Code-point~\ \ \ \ <~65536:~1110xxxx~10xxxxxx~10xxxxxx \
15026         Code-point~ \ \ \ \ <~1114112:~11110xxx~10xxxxxx~10xxxxxx~10xxxxxx \
15027     }
15028     Bytes~of~the~form~10xxxxxx~are~called~continuation~bytes.
15029     \flag_if_raised:NT \l__str_missing_flag
15030     {
15031         \\\
15032         A~leading~byte~(in~the~range~[192,255])~was~not~followed~by~
15033         the~appropriate~number~of~continuation~bytes.
15034     }
15035     \flag_if_raised:NT \l__str_extra_flag
15036     {
15037         \\\
15038         LaTeX~came~across~a~continuation~byte~when~it~was~not~expected.
15039     }
15040     \flag_if_raised:NT \l__str_overlong_flag
15041     {
15042         \\\
15043         Every~Unicode~code~point~must~be~expressed~in~the~shortest~
15044         possible~form.~For~instance,~'0xC0'~'0x83'~is~not~a~valid~
15045         representation~for~the~code~point~3.

```

```

15046     }
15047     \flag_if_raised:NT \l__str_overflow_flag
15048     {
15049         \\\
15050         Unicode~limits~code~points~to~the~range~[0,1114111].
15051     }
15052 }
15053 \prop_gput:Nnn \g_msg_module_name_prop { str } { LaTeX }
15054 \prop_gput:Nnn \g_msg_module_type_prop { str } { }

```

(End of definition for `__str_missing` and others.)

`__str_convert_decode_utf8:` Decoding is significantly harder than encoding. As before, lower some flags, which are tested at the end (in bulk, to trigger at most one L^AT_EX₃ error, as explained above).
`__str_decode_utf_viii_start:N` We expect successive multi-byte sequences of the form *⟨start byte⟩* *⟨continuation bytes⟩*. The `_start` auxiliary tests the first byte:
`__str_decode_utf_viii_continuation:wwN`
`__str_decode_utf_viii_aux:wNnnwN`
`__str_decode_utf_viii_overflow:w`
`__str_decode_utf_viii_end:`

- [0, "7F]: the byte stands alone, and is converted to its own character code;
- ["80, "BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character "FFFD;
- ["C0, "FF]: this byte should be followed by some continuation byte(s).

In the first two cases, `\use_none_delimit_by_q_stop:w` removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the `_continuation` auxiliary. We expect `#3` to be in the range ["80, "BF]. The test for this goes as follows: if the character code is less than "80, we compare it to `-"C0`, yielding `false`; otherwise to `"C0`, yielding `true` in the range ["80, "BF] and `false` otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement character, and continue parsing with the `_start` auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the `_aux` function.

The `_aux` function tests whether we should look for more continuation bytes or not. If the number it receives as `#1` is less than the maximum `#4` for the current range, then we are done: check for an overlong representation by comparing `#1` with the maximum `#3` for the previous range. Otherwise, we call the `_continuation` auxiliary again, after shifting the “current code point” by `#4` (maximum from the range we just checked).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the `_start` auxiliary leaves its first argument in the input stream: the end-marker begins with `\prg_break:`, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the `\use_none:n #3` construction removes the first token from the end-marker, and leaves the `_end` auxiliary, which raises the appropriate error flag before ending the mapping.

```

15055 \cs_new_protected:cpn { __str_convert_decode_utf8: }
15056 {

```

```

15057 \flag_clear:N \l__str_error_flag
15058 \flag_clear:N \l__str_missing_flag
15059 \flag_clear:N \l__str_extra_flag
15060 \flag_clear:N \l__str_overlong_flag
15061 \flag_clear:N \l__str_overflow_flag
15062 \__kernel_tl_gset:Nx \g__str_result_tl
15063 {
15064   \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl
15065   { \prg_break: \__str_decode_utf_viii_end: }
15066   \prg_break_point:
15067 }
15068 \__str_if_flag_error:Nne \l__str_error_flag { utf8-decode } { }
15069 }
15070 \cs_new:Npn \__str_decode_utf_viii_start:N #1
15071 {
15072   #1
15073   \if_int_compare:w '#1 < "C0 \exp_stop_f:
15074     \s__str
15075     \if_int_compare:w '#1 < "80 \exp_stop_f:
15076       \int_value:w '#1
15077     \else:
15078       \flag_raise:N \l__str_extra_flag
15079       \flag_raise:N \l__str_error_flag
15080       \int_use:N \c__str_replacement_char_int
15081     \fi:
15082   \else:
15083     \exp_after:wN \__str_decode_utf_viii_continuation:wwN
15084     \int_value:w \int_eval:n { '#1 - "C0 } \exp_after:wN
15085     \fi:
15086     \s__str
15087     \__str_use_none_delimit_by_s_stop:w {"80} {"800} {"10000} {"110000} \s__str_stop
15088     \__str_decode_utf_viii_start:N
15089   }
15090 \cs_new:Npn \__str_decode_utf_viii_continuation:wwN
15091   #1 \s__str #2 \__str_decode_utf_viii_start:N #3
15092   {
15093     \use_none:n #3
15094     \if_int_compare:w '#3 <
15095       \if_int_compare:w '#3 < "80 \exp_stop_f: - \fi:
15096       "C0 \exp_stop_f:
15097     #3
15098     \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
15099     \int_value:w \int_eval:n { #1 * "40 + '#3 - "80 } \exp_after:wN
15100   \else:
15101     \s__str
15102     \flag_raise:N \l__str_missing_flag
15103     \flag_raise:N \l__str_error_flag
15104     \int_use:N \c__str_replacement_char_int
15105   \fi:
15106   \s__str
15107   #2
15108   \__str_decode_utf_viii_start:N #3
15109 }
15110 \cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN

```

```

15111 #1 \s__str #2#3#4 #5 \__str_decode_utf_viii_start:N #6
15112 {
15113 \if_int_compare:w #1 < #4 \exp_stop_f:
15114 \s__str
15115 \if_int_compare:w #1 < #3 \exp_stop_f:
15116 \flag_raise:N \l__str_overlong_flag
15117 \flag_raise:N \l__str_error_flag
15118 \int_use:N \c__str_replacement_char_int
15119 \else:
15120 #1
15121 \fi:
15122 \else:
15123 \if_meaning:w \s__str_stop #5
15124 \__str_decode_utf_viii_overflow:w #1
15125 \fi:
15126 \exp_after:wN \__str_decode_utf_viii_continuation:wwN
15127 \int_value:w \int_eval:n { #1 - #4 } \exp_after:wN
15128 \fi:
15129 \s__str
15130 #2 {#4} #5
15131 \__str_decode_utf_viii_start:N
15132 }
15133 \cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
15134 {
15135 \fi: \fi:
15136 \flag_raise:N \l__str_overflow_flag
15137 \flag_raise:N \l__str_error_flag
15138 \int_use:N \c__str_replacement_char_int
15139 }
15140 \cs_new:Npn \__str_decode_utf_viii_end:
15141 {
15142 \s__str
15143 \flag_raise:N \l__str_missing_flag
15144 \flag_raise:N \l__str_error_flag
15145 \int_use:N \c__str_replacement_char_int \s__str
15146 \prg_break:
15147 }

```

(End of definition for __str_convert_decode_utf8: and others.)

56.6.2 utf-16 support

The definitions are done in a category code regime where the bytes 254 and 255 used by the byte order mark have catcode 12.

```

15148 \group_begin:
15149 \char_set_catcode_other:N ^^fe
15150 \char_set_catcode_other:N ^^ff

```

`__str_convert_encode_utf16:` When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviours depending on the character code.

`__str_encode_utf_xvi_aux:N`
`__str_encode_utf_xvi_char:n`

- [0, "D7FF]: converted to two bytes;

- ["D800, "DFFF] are used as surrogates: they cannot be converted and are replaced by the replacement character;
- ["E000, "FFFF]: converted to two bytes;
- ["10000, "10FFFF]: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 – "10000/"400.

For the duration of this operation, `__str_tmp:w` is defined as a function to convert a number in the range [0, "FFFF] to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of #1 by "100, followed by #1 to `__str_encode_utf_xvi_be:nn` or its `le` analog: those compute the remainder, and output two bytes for the quotient and remainder.

```

15151 \cs_new_protected:cpn { __str_convert_encode_utf16: }
15152 {
15153   \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
15154   \tl_gput_left:Ne \g__str_result_tl { ^^fe ^^ff }
15155 }
15156 \cs_new_protected:cpn { __str_convert_encode_utf16be: }
15157 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }
15158 \cs_new_protected:cpn { __str_convert_encode_utf16le: }
15159 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
15160 \cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
15161 {
15162   \flag_clear:N \l__str_error_flag
15163   \cs_set_eq:NN \__str_tmp:w #1
15164   \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
15165   \__str_if_flag_error:Nne \l__str_error_flag { utf16-encode } { }
15166 }
15167 \cs_new:Npn \__str_encode_utf_xvi_char:n #1
15168 {
15169   \if_int_compare:w #1 < "D800 \exp_stop_f:
15170     \__str_tmp:w {#1}
15171   \else:
15172     \if_int_compare:w #1 < "10000 \exp_stop_f:
15173       \if_int_compare:w #1 < "E000 \exp_stop_f:
15174         \flag_raise:N \l__str_error_flag
15175         \__str_tmp:w { \c__str_replacement_char_int }
15176       \else:
15177         \__str_tmp:w {#1}
15178       \fi:
15179     \else:
15180       \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
15181       \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
15182     \fi:
15183   \fi:
15184 }

```

(End of definition for `__str_convert_encode_utf16:` and others.)

`__str_missing` When encoding a Unicode string to UTF-16, only one error can occur: code points in
`__str_extra` the range ["D800, "DFFF], corresponding to surrogates, cannot be encoded. We use the
`__str_end` all-purpose flag `@@_error` to signal that error.

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

```

15185 \flag_clear_new:N \l__str_missing_flag
15186 \flag_clear_new:N \l__str_extra_flag
15187 \flag_clear_new:N \l__str_end_flag
15188 \msg_new:nnnn { str } { utf16-encode }
15189 { Unicode-string-cannot-be-expressed-in-UTF-16:-surrogate. }
15190 {
15191   Surrogate-code-points-(in-the-range-[U+D800,-U+DFFF])~
15192   can-be-expressed-in-the-UTF-8-and-UTF-32-encodings,~
15193   but-not-in-the-UTF-16-encoding.
15194 }
15195 \msg_new:nnnn { str } { utf16-decode }
15196 {
15197   Invalid-UTF-16-string:
15198   \exp_last_unbraced:Nf \use_none:n
15199   {
15200     \__str_if_flag_times:NT \l__str_missing_flag { ,~missing~trail~surrogate }
15201     \__str_if_flag_times:NT \l__str_extra_flag { ,~extra~trail~surrogate }
15202     \__str_if_flag_times:NT \l__str_end_flag { ,~odd~number~of~bytes }
15203   }
15204 .
15205 }
15206 {
15207   In-the-UTF-16-encoding,~each~Unicode~character~is~encoded~as~
15208   2-or-4-bytes: \\
15209   \iow_indent:n
15210   {
15211     Code-point-in-[U+0000,-U+D7FF]:~two-bytes \\
15212     Code-point-in-[U+D800,-U+DFFF]:~illegal \\
15213     Code-point-in-[U+E000,-U+FFFF]:~two-bytes \\
15214     Code-point-in-[U+10000,-U+10FFFF]:~
15215     a~lead~surrogate~and~a~trail~surrogate \\
15216   }
15217   Lead-surrogates-are-pairs-of-bytes-in-the-range-[0xD800,-0xDBFF],~
15218   and~trail~surrogates-are-in-the-range-[0xDC00,-0xDFFF].
15219   \flag_if_raised:NT \l__str_missing_flag
15220   {
15221     \\ \\
15222     A~lead~surrogate~was~not~followed~by~a~trail~surrogate.
15223   }
15224   \flag_if_raised:NT \l__str_extra_flag
15225   {
15226     \\ \\
15227     LaTeX~came~across~a~trail~surrogate~when~it~was~not~expected.
15228   }
15229   \flag_if_raised:NT \l__str_end_flag
15230   {
15231     \\ \\
15232     The~string~contained~an~odd~number~of~bytes.~This~is~invalid:~
15233     the~basic~code~unit~for~UTF-16~is~16~bits~(2~bytes).
15234   }
15235 }

```


(End of definition for `__str_missing`, `__str_extra`, and `__str_end`.)

`__str_convert_decode_utf16:` As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are "FE and "FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianness. The three endianness cases are based on a common auxiliary whose first argument is 1 for big-endian and 2 for little-endian, and whose second argument, delimited by the scan mark `\s__str_stop`, is expanded once (the string may be long; passing `\g__str_result_tl` as an argument before expansion is cheaper).

The `__str_decode_utf_xvi:Nw` function defines `__str_tmp:w` to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using `__str_decode_utf_xvi_pair:NN` described below.

```
15236 \cs_new_protected:cpn { __str_convert_decode_utf16be: }
15237   { __str_decode_utf_xvi:Nw 1 \g__str_result_tl \s__str_stop }
15238 \cs_new_protected:cpn { __str_convert_decode_utf16le: }
15239   { __str_decode_utf_xvi:Nw 2 \g__str_result_tl \s__str_stop }
15240 \cs_new_protected:cpn { __str_convert_decode_utf16: }
15241   {
15242     \exp_after:wN __str_decode_utf_xvi_bom:NN
15243     \g__str_result_tl \s__str_stop \s__str_stop \s__str_stop
15244   }
15245 \cs_new_protected:Npn __str_decode_utf_xvi_bom:NN #1#2
15246   {
15247     \str_if_eq:nnTF { #1#2 } { ^^ff ^^fe }
15248     { __str_decode_utf_xvi:Nw 2 }
15249     {
15250       \str_if_eq:nnTF { #1#2 } { ^^fe ^^ff }
15251       { __str_decode_utf_xvi:Nw 1 }
15252       { __str_decode_utf_xvi:Nw 1 #1#2 }
15253     }
15254   }
15255 \cs_new_protected:Npn __str_decode_utf_xvi:Nw #1#2 \s__str_stop
15256   {
15257     \flag_clear:N \l__str_error_flag
15258     \flag_clear:N \l__str_missing_flag
15259     \flag_clear:N \l__str_extra_flag
15260     \flag_clear:N \l__str_end_flag
15261     \cs_set:Npn __str_tmp:w ##1 ##2 { ' ## #1 }
15262     \__kernel_tl_gset:Nx \g__str_result_tl
15263     {
15264       \exp_after:wN __str_decode_utf_xvi_pair:NN
15265       #2 \q__str_nil \q__str_nil
15266     } \prg_break_point:
15267   }
15268   \__str_if_flag_error:Nne \l__str_error_flag { utf16-decode } { }
15269 }
```

(End of definition for `__str_convert_decode_utf16:` and others.)

`__str_decode_utf_xvi_pair:NN` Bytes are read two at a time. At this stage, `__str_tmp:w #1#2` expands to the character code of the most significant byte, and we distinguish cases depending on which range it lies in:
`__str_decode_utf_xvi_quad:NNwNN`
`__str_decode_utf_xvi_pair_end:Nw`
`__str_decode_utf_xvi_error:nNN`
`__str_decode_utf_xvi_extra:NNw`

- ["D8, "DB] signals a lead surrogate, and the integer expression yields 1 (ε -TeX rounds ties away from zero);
- ["DC, "DF] signals a trail surrogate, unexpected here, and the integer expression yields 2;
- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the `\if_case:w` construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the `_pair` auxiliary.

The case of a lead surrogate is treated by the `_quad` auxiliary, whose arguments `#1`, `#2`, `#4` and `#5` are the four bytes. We expect the most significant byte of `#4#5` to be in the range ["DC, "DF] (trail surrogate). The test is similar to the test used for continuation bytes in the UTF-8 decoding functions. In the case where `#4#5` is indeed a trail surrogate, leave `#1#2#4#5 \s__str <code point> \s__str`, and remove the pair `#4#5` before looping with `__str_decode_utf_xvi_pair:NN`. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that "D7F7*"400 = "D800*"400+"DC00-"10000.

Every time we read a pair of bytes, we test for the end-marker `\q__str_nil`. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```

15270 \cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
15271 {
15272   \if_meaning:w \q__str_nil #2
15273   \__str_decode_utf_xvi_pair_end:Nw #1
15274   \fi:
15275   \if_case:w
15276     \int_eval:n { ( \__str_tmp:w #1#2 - "D6 ) / 4 } \scan_stop:
15277   \or: \exp_after:wN \__str_decode_utf_xvi_quad:NNwNN
15278   \or: \exp_after:wN \__str_decode_utf_xvi_extra:NNw
15279   \fi:
15280   #1#2 \s__str
15281   \int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__str
15282   \__str_decode_utf_xvi_pair:NN
15283 }
15284 \cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN
15285 #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
15286 {
15287   \if_meaning:w \q__str_nil #5
15288   \__str_decode_utf_xvi_error:nNN { missing } #1#2
15289   \__str_decode_utf_xvi_pair_end:Nw #4
15290   \fi:
15291   \if_int_compare:w
15292     \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
15293     0 = 1
15294     \else:
15295       \__str_tmp:w #4#5 < "E0
15296     \fi:
15297     \exp_stop_f:
15298     #1 #2 #4 #5 \s__str
15299     \int_eval:n
15300     {

```

```

15301         ( "100 * \_str_tmp:w #1#2 + \_str_tmp:w #2#1 - "D7F7 ) * "400
15302         + "100 * \_str_tmp:w #4#5 + \_str_tmp:w #5#4
15303     }
15304     \s__str
15305     \exp_after:wN \use_i:nnn
15306     \else:
15307     \_str_decode_utf_xvi_error:nNN { missing } #1#2
15308     \fi:
15309     \_str_decode_utf_xvi_pair:NN #4#5
15310 }
15311 \cs_new:Npn \_str_decode_utf_xvi_pair_end:Nw #1 \fi:
15312 {
15313     \fi:
15314     \if_meaning:w \q__str_nil #1
15315     \else:
15316     \_str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
15317     \fi:
15318     \prg_break:
15319 }
15320 \cs_new:Npn \_str_decode_utf_xvi_extra:NNw #1#2 \s__str #3 \s__str
15321 { \_str_decode_utf_xvi_error:nNN { extra } #1#2 }
15322 \cs_new:Npn \_str_decode_utf_xvi_error:nNN #1#2#3
15323 {
15324     \flag_raise:N \l__str_error_flag
15325     \flag_raise:c { l__str_#1_flag }
15326     #2 #3 \s__str
15327     \int_use:N \c__str_replacement_char_int \s__str
15328 }

```

(End of definition for `_str_decode_utf_xvi_pair:NN` and others.)

Restore the original catcodes of bytes 254 and 255.

```
15329 \group_end:
```

56.6.3 utf-32 support

The definitions are done in a category code regime where the bytes 0, 254 and 255 used by the byte order mark have catcode “other”.

```

15330 \group_begin:
15331 \char_set_catcode_other:N ^^00
15332 \char_set_catcode_other:N ^^fe
15333 \char_set_catcode_other:N ^^ff

```

`_str_convert_encode_utf32:` Convert each integer in the comma-list `\g__str_result_tl` to a sequence of four bytes. The functions for big-endian and little-endian encodings are very similar, but the `_str_output_byte:n` instructions are reversed.

```

\_str_convert_encode_utf32be:
  \_str_convert_encode_utf32le:
\_str_encode_utf_xxxii_be:n
  \_str_encode_utf_xxxii_be_aux:nn
\_str_encode_utf_xxxii_le:n
  \_str_encode_utf_xxxii_le_aux:nn
15334 \cs_new_protected:cpn { __str_convert_encode_utf32: }
15335 {
15336     \_str_convert_gmap_internal:N \_str_encode_utf_xxxii_be:n
15337     \tl_gput_left:Ne \g__str_result_tl { ^^00 ^^00 ^^fe ^^ff }
15338 }
15339 \cs_new_protected:cpn { __str_convert_encode_utf32be: }
15340 { \_str_convert_gmap_internal:N \_str_encode_utf_xxxii_be:n }
15341 \cs_new_protected:cpn { __str_convert_encode_utf32le: }

```

```

15342 { \_str_convert_gmap_internal:N \_str_encode_utf_xxxii_le:n }
15343 \cs_new:Npn \_str_encode_utf_xxxii_be:n #1
15344 {
15345   \exp_args:Nf \_str_encode_utf_xxxii_be_aux:nn
15346   { \int_div_truncate:nn {#1} { "100 } } {#1}
15347 }
15348 \cs_new:Npn \_str_encode_utf_xxxii_be_aux:nn #1#2
15349 {
15350   ^^00
15351   \_str_output_byte_pair_be:n {#1}
15352   \_str_output_byte:n { #2 - #1 * "100 }
15353 }
15354 \cs_new:Npn \_str_encode_utf_xxxii_le:n #1
15355 {
15356   \exp_args:Nf \_str_encode_utf_xxxii_le_aux:nn
15357   { \int_div_truncate:nn {#1} { "100 } } {#1}
15358 }
15359 \cs_new:Npn \_str_encode_utf_xxxii_le_aux:nn #1#2
15360 {
15361   \_str_output_byte:n { #2 - #1 * "100 }
15362   \_str_output_byte_pair_le:n {#1}
15363   ^^00
15364 }

```

(End of definition for `_str_convert_encode_utf32:` and others.)

`__str_overflow` There can be no error when encoding in UTF-32. When decoding, the string may not
`__str_end` have length $4n$, or it may contain code points larger than "10FFFF". The latter case often happens if the encoding was in fact not UTF-32, because most arbitrary strings are not valid in UTF-32.

```

15365 \flag_clear_new:N \l__str_overflow_flag
15366 \flag_clear_new:N \l__str_end_flag
15367 \msg_new:nnnn { str } { utf32-decode }
15368 {
15369   Invalid~UTF-32~string:
15370   \exp_last_unbraced:Nf \use_none:n
15371   {
15372     \_str_if_flag_times:NT \l__str_overflow_flag { ,~code-point-too-large }
15373     \_str_if_flag_times:NT \l__str_end_flag      { ,~truncated-string }
15374   }
15375   .
15376 }
15377 {
15378   In-the-UTF-32-encoding,~every~Unicode~character~
15379   (in-the-range~[U+0000,~U+10FFFF])~is-encoded-as~4-bytes.
15380   \flag_if_raised:NT \l__str_overflow_flag
15381   {
15382     \\\
15383     LaTeX~came~across~a~code~point~larger~than~1114111,~
15384     the~maximum~code~point~defined~by~Unicode.~
15385     Perhaps~the~string~was~not~encoded~in~the~UTF-32~encoding?
15386   }
15387   \flag_if_raised:NT \l__str_end_flag
15388   {

```

```

15389         \\\
15390         The-length-of-the-string-is-not-a-multiple-of-4.-
15391         Perhaps-the-string-was-truncated?
15392     }
15393 }

```

(End of definition for `__str_overflow` and `__str_end`.)

`__str_convert_decode_utf32`: The structure is similar to UTF-16 decoding functions. If the endianness is not given, test the first 4 bytes of the string (possibly `\s__str_stop` if the string is too short) for the presence of a byte-order mark. If there is a byte-order mark, use that endianness, and remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The `__str_decode_utf_xxxii:Nw` auxiliary receives 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets `__str_tmp:w` to expand to the character code of either of its two arguments depending on endianness, then triggers the `_loop` auxiliary inside an e-expanding assignment to `\g__str_result_t1`.

The `_loop` auxiliary first checks for the end-of-string marker `\s__str_stop`, calling the `_end` auxiliary if appropriate. Otherwise, leave the *(4 bytes)* `\s__str` behind, then check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first `\s__str_stop`. Break the map.

```

15394 \cs_new_protected:cpn { __str_convert_decode_utf32be: }
15395 { \__str_decode_utf_xxxii:Nw 1 \g__str_result_t1 \s__str_stop }
15396 \cs_new_protected:cpn { __str_convert_decode_utf32le: }
15397 { \__str_decode_utf_xxxii:Nw 2 \g__str_result_t1 \s__str_stop }
15398 \cs_new_protected:cpn { __str_convert_decode_utf32: }
15399 {
15400     \exp_after:wN \__str_decode_utf_xxxii_bom:NNNN \g__str_result_t1
15401     \s__str_stop \s__str_stop \s__str_stop \s__str_stop \s__str_stop
15402 }
15403 \cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4
15404 {
15405     \str_if_eq:nnTF { #1#2#3#4 } { ^^ff ^^fe ^^00 ^^00 }
15406     { \__str_decode_utf_xxxii:Nw 2 }
15407     {
15408         \str_if_eq:nnTF { #1#2#3#4 } { ^^00 ^^00 ^^fe ^^ff }
15409         { \__str_decode_utf_xxxii:Nw 1 }
15410         { \__str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
15411     }
15412 }
15413 \cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__str_stop
15414 {
15415     \flag_clear:N \l__str_overflow_flag
15416     \flag_clear:N \l__str_end_flag
15417     \flag_clear:N \l__str_error_flag
15418     \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
15419     \__kernel_t1_gset:Nx \g__str_result_t1
15420     {
15421         \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
15422         #2 \s__str_stop \s__str_stop \s__str_stop \s__str_stop
15423     }

```

```

15424     }
15425     \__str_if_flag_error:Nne \l__str_error_flag { utf32-decode } { }
15426   }
15427   \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
15428   {
15429     \if_meaning:w \s__str_stop #4
15430     \exp_after:wN \__str_decode_utf_xxxii_end:w
15431     \fi:
15432     #1#2#3#4 \s__str
15433     \if_int_compare:w \__str_tmp:w #1#4 > \c_zero_int
15434       \flag_raise:N \l__str_overflow_flag
15435       \flag_raise:N \l__str_error_flag
15436       \int_use:N \c__str_replacement_char_int
15437     \else:
15438       \if_int_compare:w \__str_tmp:w #2#3 > 16 \exp_stop_f:
15439         \flag_raise:N \l__str_overflow_flag
15440         \flag_raise:N \l__str_error_flag
15441         \int_use:N \c__str_replacement_char_int
15442       \else:
15443         \int_eval:n
15444         { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }
15445       \fi:
15446     \fi:
15447     \s__str
15448     \__str_decode_utf_xxxii_loop:NNNN
15449   }
15450   \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__str_stop
15451   {
15452     \tl_if_empty:nF {#1}
15453     {
15454       \flag_raise:N \l__str_end_flag
15455       \flag_raise:N \l__str_error_flag
15456       #1 \s__str
15457       \int_use:N \c__str_replacement_char_int \s__str
15458     }
15459     \prg_break:
15460   }

```

(End of definition for `__str_convert_decode_utf32:` and others.)

Restore the original catcodes of bytes 0, 254 and 255.

```
15461 \group_end:
```

56.7 PDF names and strings by expansion

```

\str_convert_pdfname:n
\__str_convert_pdfname:n
  \_str_convert_pdfname_bytes:n
  \_str_convert_pdfname_bytes_aux:n
\__str_convert_pdfname_bytes_aux:nm

```

To convert to PDF names by expansion, we work purely on UTF-8 input. The first step is to make a string with “other” spaces, after which we use a simple token-by-token approach. In Unicode engines, we break down everything before one-byte codepoints, but for 8-bit engines there is no need to worry. Actual escaping is covered by the same code as used in the non-expandable route.

```

15462 \cs_new:Npn \str_convert_pdfname:n #1
15463   {
15464     \exp_args:Ne \tl_to_str:n

```

```

15465     { \str_map_function:nN {#1} \__str_convert_pdfname:n }
15466   }
15467 \bool_lazy_or:nnTF
15468 { \sys_if_engine luatex_p: }
15469 { \sys_if_engine xetex_p: }
15470 {
15471   \cs_new:Npn \__str_convert_pdfname:n #1
15472     {
15473       \int_compare:nNnTF { '#1 } > { "7F }
15474         { \__str_convert_pdfname_bytes:n {#1} }
15475         { \__str_escape_name_char:n {#1} }
15476     }
15477   \cs_new:Npn \__str_convert_pdfname_bytes:n #1
15478     {
15479       \exp_args:Ne \__str_convert_pdfname_bytes_aux:n
15480         { \__kernel_codepoint_to_bytes:n {'#1} }
15481     }
15482   \cs_new:Npn \__str_convert_pdfname_bytes_aux:n #1
15483     { \__str_convert_pdfname_bytes_aux:nnnn #1 }
15484   \cs_new:Npe \__str_convert_pdfname_bytes_aux:nnnn #1#2#3#4
15485     {
15486       \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#1}
15487       \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#2}
15488       \exp_not:N \tl_if_blank:nF {#3}
15489       {
15490         \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#3}
15491         \exp_not:N \tl_if_blank:nF {#4}
15492         {
15493           \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#4}
15494         }
15495       }
15496     }
15497   }
15498   { \cs_new_eq:NN \__str_convert_pdfname:n \__str_escape_name_char:n }

```

(End of definition for `\str_convert_pdfname:n` and others. This function is documented on page 144.)

```

15499 </package>

```

56.7.1 iso 8859 support

The iso-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the ISO-8859 family, we keep track only of differences, and of unassigned bytes.

```

15500 (*iso88591)
15501 \__str_declare_eight_bit_encoding:nnnn { iso88591 } { 256 }
15502 {
15503 }
15504 {
15505 }
15506 </iso88591>
15507 (*iso88592)
15508 \__str_declare_eight_bit_encoding:nnnn { iso88592 } { 399 }
15509 {

```

15510 { A1 } { 0104 }
15511 { A2 } { 02D8 }
15512 { A3 } { 0141 }
15513 { A5 } { 013D }
15514 { A6 } { 015A }
15515 { A9 } { 0160 }
15516 { AA } { 015E }
15517 { AB } { 0164 }
15518 { AC } { 0179 }
15519 { AE } { 017D }
15520 { AF } { 017B }
15521 { B1 } { 0105 }
15522 { B2 } { 02DB }
15523 { B3 } { 0142 }
15524 { B5 } { 013E }
15525 { B6 } { 015B }
15526 { B7 } { 02C7 }
15527 { B9 } { 0161 }
15528 { BA } { 015F }
15529 { BB } { 0165 }
15530 { BC } { 017A }
15531 { BD } { 02DD }
15532 { BE } { 017E }
15533 { BF } { 017C }
15534 { C0 } { 0154 }
15535 { C3 } { 0102 }
15536 { C5 } { 0139 }
15537 { C6 } { 0106 }
15538 { C8 } { 010C }
15539 { CA } { 0118 }
15540 { CC } { 011A }
15541 { CF } { 010E }
15542 { D0 } { 0110 }
15543 { D1 } { 0143 }
15544 { D2 } { 0147 }
15545 { D5 } { 0150 }
15546 { D8 } { 0158 }
15547 { D9 } { 016E }
15548 { DB } { 0170 }
15549 { DE } { 0162 }
15550 { E0 } { 0155 }
15551 { E3 } { 0103 }
15552 { E5 } { 013A }
15553 { E6 } { 0107 }
15554 { E8 } { 010D }
15555 { EA } { 0119 }
15556 { EC } { 011B }
15557 { EF } { 010F }
15558 { F0 } { 0111 }
15559 { F1 } { 0144 }
15560 { F2 } { 0148 }
15561 { F5 } { 0151 }
15562 { F8 } { 0159 }
15563 { F9 } { 016F }


```

15564     { FB } { 0171 }
15565     { FE } { 0163 }
15566     { FF } { 02D9 }
15567   }
15568   {
15569   }
15570 </iso88592>
15571 <iso88593>
15572 \_str_declare_eight_bit_encoding:nnnn { iso88593 } { 384 }
15573   {
15574     { A1 } { 0126 }
15575     { A2 } { 02D8 }
15576     { A6 } { 0124 }
15577     { A9 } { 0130 }
15578     { AA } { 015E }
15579     { AB } { 011E }
15580     { AC } { 0134 }
15581     { AF } { 017B }
15582     { B1 } { 0127 }
15583     { B6 } { 0125 }
15584     { B9 } { 0131 }
15585     { BA } { 015F }
15586     { BB } { 011F }
15587     { BC } { 0135 }
15588     { BF } { 017C }
15589     { C5 } { 010A }
15590     { C6 } { 0108 }
15591     { D5 } { 0120 }
15592     { D8 } { 011C }
15593     { DD } { 016C }
15594     { DE } { 015C }
15595     { E5 } { 010B }
15596     { E6 } { 0109 }
15597     { F5 } { 0121 }
15598     { F8 } { 011D }
15599     { FD } { 016D }
15600     { FE } { 015D }
15601     { FF } { 02D9 }
15602   }
15603   {
15604     { A5 }
15605     { AE }
15606     { BE }
15607     { C3 }
15608     { D0 }
15609     { E3 }
15610     { F0 }
15611   }
15612 </iso88593>
15613 <iso88594>
15614 \_str_declare_eight_bit_encoding:nnnn { iso88594 } { 383 }
15615   {
15616     { A1 } { 0104 }

```

```

15617 { A2 } { 0138 }
15618 { A3 } { 0156 }
15619 { A5 } { 0128 }
15620 { A6 } { 013B }
15621 { A9 } { 0160 }
15622 { AA } { 0112 }
15623 { AB } { 0122 }
15624 { AC } { 0166 }
15625 { AE } { 017D }
15626 { B1 } { 0105 }
15627 { B2 } { 02DB }
15628 { B3 } { 0157 }
15629 { B5 } { 0129 }
15630 { B6 } { 013C }
15631 { B7 } { 02C7 }
15632 { B9 } { 0161 }
15633 { BA } { 0113 }
15634 { BB } { 0123 }
15635 { BC } { 0167 }
15636 { BD } { 014A }
15637 { BE } { 017E }
15638 { BF } { 014B }
15639 { CO } { 0100 }
15640 { C7 } { 012E }
15641 { C8 } { 010C }
15642 { CA } { 0118 }
15643 { CC } { 0116 }
15644 { CF } { 012A }
15645 { DO } { 0110 }
15646 { D1 } { 0145 }
15647 { D2 } { 014C }
15648 { D3 } { 0136 }
15649 { D9 } { 0172 }
15650 { DD } { 0168 }
15651 { DE } { 016A }
15652 { EO } { 0101 }
15653 { E7 } { 012F }
15654 { E8 } { 010D }
15655 { EA } { 0119 }
15656 { EC } { 0117 }
15657 { EF } { 012B }
15658 { FO } { 0111 }
15659 { F1 } { 0146 }
15660 { F2 } { 014D }
15661 { F3 } { 0137 }
15662 { F9 } { 0173 }
15663 { FD } { 0169 }
15664 { FE } { 016B }
15665 { FF } { 02D9 }
15666 }
15667 {
15668 }
15669 </iso88594>
15670 <*iso88595>

```

```
15671 \__str_declare_eight_bit_encoding:nmmn { iso88595 } { 374 }
15672 {
15673   { A1 } { 0401 }
15674   { A2 } { 0402 }
15675   { A3 } { 0403 }
15676   { A4 } { 0404 }
15677   { A5 } { 0405 }
15678   { A6 } { 0406 }
15679   { A7 } { 0407 }
15680   { A8 } { 0408 }
15681   { A9 } { 0409 }
15682   { AA } { 040A }
15683   { AB } { 040B }
15684   { AC } { 040C }
15685   { AE } { 040E }
15686   { AF } { 040F }
15687   { B0 } { 0410 }
15688   { B1 } { 0411 }
15689   { B2 } { 0412 }
15690   { B3 } { 0413 }
15691   { B4 } { 0414 }
15692   { B5 } { 0415 }
15693   { B6 } { 0416 }
15694   { B7 } { 0417 }
15695   { B8 } { 0418 }
15696   { B9 } { 0419 }
15697   { BA } { 041A }
15698   { BB } { 041B }
15699   { BC } { 041C }
15700   { BD } { 041D }
15701   { BE } { 041E }
15702   { BF } { 041F }
15703   { C0 } { 0420 }
15704   { C1 } { 0421 }
15705   { C2 } { 0422 }
15706   { C3 } { 0423 }
15707   { C4 } { 0424 }
15708   { C5 } { 0425 }
15709   { C6 } { 0426 }
15710   { C7 } { 0427 }
15711   { C8 } { 0428 }
15712   { C9 } { 0429 }
15713   { CA } { 042A }
15714   { CB } { 042B }
15715   { CC } { 042C }
15716   { CD } { 042D }
15717   { CE } { 042E }
15718   { CF } { 042F }
15719   { D0 } { 0430 }
15720   { D1 } { 0431 }
15721   { D2 } { 0432 }
15722   { D3 } { 0433 }
15723   { D4 } { 0434 }
15724   { D5 } { 0435 }
```

```

15725     { D6 } { 0436 }
15726     { D7 } { 0437 }
15727     { D8 } { 0438 }
15728     { D9 } { 0439 }
15729     { DA } { 043A }
15730     { DB } { 043B }
15731     { DC } { 043C }
15732     { DD } { 043D }
15733     { DE } { 043E }
15734     { DF } { 043F }
15735     { E0 } { 0440 }
15736     { E1 } { 0441 }
15737     { E2 } { 0442 }
15738     { E3 } { 0443 }
15739     { E4 } { 0444 }
15740     { E5 } { 0445 }
15741     { E6 } { 0446 }
15742     { E7 } { 0447 }
15743     { E8 } { 0448 }
15744     { E9 } { 0449 }
15745     { EA } { 044A }
15746     { EB } { 044B }
15747     { EC } { 044C }
15748     { ED } { 044D }
15749     { EE } { 044E }
15750     { EF } { 044F }
15751     { F0 } { 2116 }
15752     { F1 } { 0451 }
15753     { F2 } { 0452 }
15754     { F3 } { 0453 }
15755     { F4 } { 0454 }
15756     { F5 } { 0455 }
15757     { F6 } { 0456 }
15758     { F7 } { 0457 }
15759     { F8 } { 0458 }
15760     { F9 } { 0459 }
15761     { FA } { 045A }
15762     { FB } { 045B }
15763     { FC } { 045C }
15764     { FD } { 00A7 }
15765     { FE } { 045E }
15766     { FF } { 045F }
15767     }
15768     {
15769     }
15770     </iso88595>
15771     <*iso88596>
15772     \_str_declare_eight_bit_encoding:nmnn { iso88596 } { 344 }
15773     {
15774         { AC } { 060C }
15775         { BB } { 061B }
15776         { BF } { 061F }
15777         { C1 } { 0621 }
15778         { C2 } { 0622 }

```

15779 { C3 } { 0623 }
15780 { C4 } { 0624 }
15781 { C5 } { 0625 }
15782 { C6 } { 0626 }
15783 { C7 } { 0627 }
15784 { C8 } { 0628 }
15785 { C9 } { 0629 }
15786 { CA } { 062A }
15787 { CB } { 062B }
15788 { CC } { 062C }
15789 { CD } { 062D }
15790 { CE } { 062E }
15791 { CF } { 062F }
15792 { D0 } { 0630 }
15793 { D1 } { 0631 }
15794 { D2 } { 0632 }
15795 { D3 } { 0633 }
15796 { D4 } { 0634 }
15797 { D5 } { 0635 }
15798 { D6 } { 0636 }
15799 { D7 } { 0637 }
15800 { D8 } { 0638 }
15801 { D9 } { 0639 }
15802 { DA } { 063A }
15803 { E0 } { 0640 }
15804 { E1 } { 0641 }
15805 { E2 } { 0642 }
15806 { E3 } { 0643 }
15807 { E4 } { 0644 }
15808 { E5 } { 0645 }
15809 { E6 } { 0646 }
15810 { E7 } { 0647 }
15811 { E8 } { 0648 }
15812 { E9 } { 0649 }
15813 { EA } { 064A }
15814 { EB } { 064B }
15815 { EC } { 064C }
15816 { ED } { 064D }
15817 { EE } { 064E }
15818 { EF } { 064F }
15819 { FO } { 0650 }
15820 { F1 } { 0651 }
15821 { F2 } { 0652 }
15822 }
15823 {
15824 { A1 }
15825 { A2 }
15826 { A3 }
15827 { A5 }
15828 { A6 }
15829 { A7 }
15830 { A8 }
15831 { A9 }
15832 { AA }

```

15833     { AB }
15834     { AE }
15835     { AF }
15836     { B0 }
15837     { B1 }
15838     { B2 }
15839     { B3 }
15840     { B4 }
15841     { B5 }
15842     { B6 }
15843     { B7 }
15844     { B8 }
15845     { B9 }
15846     { BA }
15847     { BC }
15848     { BD }
15849     { BE }
15850     { C0 }
15851     { DB }
15852     { DC }
15853     { DD }
15854     { DE }
15855     { DF }
15856     }
15857     (</iso88596>)
15858     (*iso88597)
15859     \__str_declare_eight_bit_encoding:nnnn { iso88597 } { 498 }
15860     {
15861         { A1 } { 2018 }
15862         { A2 } { 2019 }
15863         { A4 } { 20AC }
15864         { A5 } { 20AF }
15865         { AA } { 037A }
15866         { AF } { 2015 }
15867         { B4 } { 0384 }
15868         { B5 } { 0385 }
15869         { B6 } { 0386 }
15870         { B8 } { 0388 }
15871         { B9 } { 0389 }
15872         { BA } { 038A }
15873         { BC } { 038C }
15874         { BE } { 038E }
15875         { BF } { 038F }
15876         { C0 } { 0390 }
15877         { C1 } { 0391 }
15878         { C2 } { 0392 }
15879         { C3 } { 0393 }
15880         { C4 } { 0394 }
15881         { C5 } { 0395 }
15882         { C6 } { 0396 }
15883         { C7 } { 0397 }
15884         { C8 } { 0398 }
15885         { C9 } { 0399 }
15886         { CA } { 039A }

```

15887 { CB } { 039B }
15888 { CC } { 039C }
15889 { CD } { 039D }
15890 { CE } { 039E }
15891 { CF } { 039F }
15892 { D0 } { 03A0 }
15893 { D1 } { 03A1 }
15894 { D3 } { 03A3 }
15895 { D4 } { 03A4 }
15896 { D5 } { 03A5 }
15897 { D6 } { 03A6 }
15898 { D7 } { 03A7 }
15899 { D8 } { 03A8 }
15900 { D9 } { 03A9 }
15901 { DA } { 03AA }
15902 { DB } { 03AB }
15903 { DC } { 03AC }
15904 { DD } { 03AD }
15905 { DE } { 03AE }
15906 { DF } { 03AF }
15907 { E0 } { 03B0 }
15908 { E1 } { 03B1 }
15909 { E2 } { 03B2 }
15910 { E3 } { 03B3 }
15911 { E4 } { 03B4 }
15912 { E5 } { 03B5 }
15913 { E6 } { 03B6 }
15914 { E7 } { 03B7 }
15915 { E8 } { 03B8 }
15916 { E9 } { 03B9 }
15917 { EA } { 03BA }
15918 { EB } { 03BB }
15919 { EC } { 03BC }
15920 { ED } { 03BD }
15921 { EE } { 03BE }
15922 { EF } { 03BF }
15923 { F0 } { 03C0 }
15924 { F1 } { 03C1 }
15925 { F2 } { 03C2 }
15926 { F3 } { 03C3 }
15927 { F4 } { 03C4 }
15928 { F5 } { 03C5 }
15929 { F6 } { 03C6 }
15930 { F7 } { 03C7 }
15931 { F8 } { 03C8 }
15932 { F9 } { 03C9 }
15933 { FA } { 03CA }
15934 { FB } { 03CB }
15935 { FC } { 03CC }
15936 { FD } { 03CD }
15937 { FE } { 03CE }
15938 }
15939 {
15940 { AE }

```

15941     { D2 }
15942   }
15943 </iso88597>
15944 <*iso88598>
15945 \_str_declare_eight_bit_encoding:nnnn { iso88598 } { 308 }
15946   {
15947     { AA } { 00D7 }
15948     { BA } { 00F7 }
15949     { DF } { 2017 }
15950     { E0 } { 05D0 }
15951     { E1 } { 05D1 }
15952     { E2 } { 05D2 }
15953     { E3 } { 05D3 }
15954     { E4 } { 05D4 }
15955     { E5 } { 05D5 }
15956     { E6 } { 05D6 }
15957     { E7 } { 05D7 }
15958     { E8 } { 05D8 }
15959     { E9 } { 05D9 }
15960     { EA } { 05DA }
15961     { EB } { 05DB }
15962     { EC } { 05DC }
15963     { ED } { 05DD }
15964     { EE } { 05DE }
15965     { EF } { 05DF }
15966     { FO } { 05E0 }
15967     { F1 } { 05E1 }
15968     { F2 } { 05E2 }
15969     { F3 } { 05E3 }
15970     { F4 } { 05E4 }
15971     { F5 } { 05E5 }
15972     { F6 } { 05E6 }
15973     { F7 } { 05E7 }
15974     { F8 } { 05E8 }
15975     { F9 } { 05E9 }
15976     { FA } { 05EA }
15977     { FD } { 200E }
15978     { FE } { 200F }
15979   }
15980   {
15981     { A1 }
15982     { BF }
15983     { CO }
15984     { C1 }
15985     { C2 }
15986     { C3 }
15987     { C4 }
15988     { C5 }
15989     { C6 }
15990     { C7 }
15991     { C8 }
15992     { C9 }
15993     { CA }
15994     { CB }

```



```

15995     { CC }
15996     { CD }
15997     { CE }
15998     { CF }
15999     { DO }
16000     { D1 }
16001     { D2 }
16002     { D3 }
16003     { D4 }
16004     { D5 }
16005     { D6 }
16006     { D7 }
16007     { D8 }
16008     { D9 }
16009     { DA }
16010     { DB }
16011     { DC }
16012     { DD }
16013     { DE }
16014     { FB }
16015     { FC }
16016     }
16017 </iso88598>
16018 (*iso88599)
16019 \_str_declare\_eight\_bit\_encoding:nmn { iso88599 } { 352 }
16020 {
16021     { DO } { 011E }
16022     { DD } { 0130 }
16023     { DE } { 015E }
16024     { FO } { 011F }
16025     { FD } { 0131 }
16026     { FE } { 015F }
16027 }
16028 {
16029 }
16030 </iso88599>
16031 (*iso885910)
16032 \_str_declare\_eight\_bit\_encoding:nmn { iso885910 } { 383 }
16033 {
16034     { A1 } { 0104 }
16035     { A2 } { 0112 }
16036     { A3 } { 0122 }
16037     { A4 } { 012A }
16038     { A5 } { 0128 }
16039     { A6 } { 0136 }
16040     { A8 } { 013B }
16041     { A9 } { 0110 }
16042     { AA } { 0160 }
16043     { AB } { 0166 }
16044     { AC } { 017D }
16045     { AE } { 016A }
16046     { AF } { 014A }
16047     { B1 } { 0105 }

```

```

16048     { B2 } { 0113 }
16049     { B3 } { 0123 }
16050     { B4 } { 012B }
16051     { B5 } { 0129 }
16052     { B6 } { 0137 }
16053     { B8 } { 013C }
16054     { B9 } { 0111 }
16055     { BA } { 0161 }
16056     { BB } { 0167 }
16057     { BC } { 017E }
16058     { BD } { 2015 }
16059     { BE } { 016B }
16060     { BF } { 014B }
16061     { C0 } { 0100 }
16062     { C7 } { 012E }
16063     { C8 } { 010C }
16064     { CA } { 0118 }
16065     { CC } { 0116 }
16066     { D1 } { 0145 }
16067     { D2 } { 014C }
16068     { D7 } { 0168 }
16069     { D9 } { 0172 }
16070     { E0 } { 0101 }
16071     { E7 } { 012F }
16072     { E8 } { 010D }
16073     { EA } { 0119 }
16074     { EC } { 0117 }
16075     { F1 } { 0146 }
16076     { F2 } { 014D }
16077     { F7 } { 0169 }
16078     { F9 } { 0173 }
16079     { FF } { 0138 }
16080     }
16081     {
16082     }
16083     </iso885910>
16084     <*iso885911>
16085     \_str_declare_eight_bit_encoding:nnnn { iso885911 } { 369 }
16086     {
16087         { A1 } { OE01 }
16088         { A2 } { OE02 }
16089         { A3 } { OE03 }
16090         { A4 } { OE04 }
16091         { A5 } { OE05 }
16092         { A6 } { OE06 }
16093         { A7 } { OE07 }
16094         { A8 } { OE08 }
16095         { A9 } { OE09 }
16096         { AA } { OE0A }
16097         { AB } { OE0B }
16098         { AC } { OE0C }
16099         { AD } { OE0D }
16100         { AE } { OE0E }
16101         { AF } { OE0F }

```

16102 { B0 } { OE10 }
16103 { B1 } { OE11 }
16104 { B2 } { OE12 }
16105 { B3 } { OE13 }
16106 { B4 } { OE14 }
16107 { B5 } { OE15 }
16108 { B6 } { OE16 }
16109 { B7 } { OE17 }
16110 { B8 } { OE18 }
16111 { B9 } { OE19 }
16112 { BA } { OE1A }
16113 { BB } { OE1B }
16114 { BC } { OE1C }
16115 { BD } { OE1D }
16116 { BE } { OE1E }
16117 { BF } { OE1F }
16118 { C0 } { OE20 }
16119 { C1 } { OE21 }
16120 { C2 } { OE22 }
16121 { C3 } { OE23 }
16122 { C4 } { OE24 }
16123 { C5 } { OE25 }
16124 { C6 } { OE26 }
16125 { C7 } { OE27 }
16126 { C8 } { OE28 }
16127 { C9 } { OE29 }
16128 { CA } { OE2A }
16129 { CB } { OE2B }
16130 { CC } { OE2C }
16131 { CD } { OE2D }
16132 { CE } { OE2E }
16133 { CF } { OE2F }
16134 { D0 } { OE30 }
16135 { D1 } { OE31 }
16136 { D2 } { OE32 }
16137 { D3 } { OE33 }
16138 { D4 } { OE34 }
16139 { D5 } { OE35 }
16140 { D6 } { OE36 }
16141 { D7 } { OE37 }
16142 { D8 } { OE38 }
16143 { D9 } { OE39 }
16144 { DA } { OE3A }
16145 { DF } { OE3F }
16146 { E0 } { OE40 }
16147 { E1 } { OE41 }
16148 { E2 } { OE42 }
16149 { E3 } { OE43 }
16150 { E4 } { OE44 }
16151 { E5 } { OE45 }
16152 { E6 } { OE46 }
16153 { E7 } { OE47 }
16154 { E8 } { OE48 }
16155 { E9 } { OE49 }

```

16156     { EA } { OE4A }
16157     { EB } { OE4B }
16158     { EC } { OE4C }
16159     { ED } { OE4D }
16160     { EE } { OE4E }
16161     { EF } { OE4F }
16162     { FO } { OE50 }
16163     { F1 } { OE51 }
16164     { F2 } { OE52 }
16165     { F3 } { OE53 }
16166     { F4 } { OE54 }
16167     { F5 } { OE55 }
16168     { F6 } { OE56 }
16169     { F7 } { OE57 }
16170     { F8 } { OE58 }
16171     { F9 } { OE59 }
16172     { FA } { OE5A }
16173     { FB } { OE5B }
16174     }
16175     {
16176         { DB }
16177         { DC }
16178         { DD }
16179         { DE }
16180     }
16181     </iso885911>
16182     (*iso885913)
16183     \__str_declare_eight_bit_encoding:nmn { iso885913 } { 399 }
16184     {
16185         { A1 } { 201D }
16186         { A5 } { 201E }
16187         { A8 } { 00D8 }
16188         { AA } { 0156 }
16189         { AF } { 00C6 }
16190         { B4 } { 201C }
16191         { B8 } { 00F8 }
16192         { BA } { 0157 }
16193         { BF } { 00E6 }
16194         { CO } { 0104 }
16195         { C1 } { 012E }
16196         { C2 } { 0100 }
16197         { C3 } { 0106 }
16198         { C6 } { 0118 }
16199         { C7 } { 0112 }
16200         { C8 } { 010C }
16201         { CA } { 0179 }
16202         { CB } { 0116 }
16203         { CC } { 0122 }
16204         { CD } { 0136 }
16205         { CE } { 012A }
16206         { CF } { 013B }
16207         { DO } { 0160 }
16208         { D1 } { 0143 }
16209         { D2 } { 0145 }

```

```

16210     { D4 } { 014C }
16211     { D8 } { 0172 }
16212     { D9 } { 0141 }
16213     { DA } { 015A }
16214     { DB } { 016A }
16215     { DD } { 017B }
16216     { DE } { 017D }
16217     { E0 } { 0105 }
16218     { E1 } { 012F }
16219     { E2 } { 0101 }
16220     { E3 } { 0107 }
16221     { E6 } { 0119 }
16222     { E7 } { 0113 }
16223     { E8 } { 010D }
16224     { EA } { 017A }
16225     { EB } { 0117 }
16226     { EC } { 0123 }
16227     { ED } { 0137 }
16228     { EE } { 012B }
16229     { EF } { 013C }
16230     { FO } { 0161 }
16231     { F1 } { 0144 }
16232     { F2 } { 0146 }
16233     { F4 } { 014D }
16234     { F8 } { 0173 }
16235     { F9 } { 0142 }
16236     { FA } { 015B }
16237     { FB } { 016B }
16238     { FD } { 017C }
16239     { FE } { 017E }
16240     { FF } { 2019 }
16241     }
16242     {
16243     }
16244     </iso885913>
16245     <*iso885914>
16246     \__str_declare_eight_bit_encoding:nmmn { iso885914 } { 529 }
16247     {
16248         { A1 } { 1E02 }
16249         { A2 } { 1E03 }
16250         { A4 } { 010A }
16251         { A5 } { 010B }
16252         { A6 } { 1E0A }
16253         { A8 } { 1E80 }
16254         { AA } { 1E82 }
16255         { AB } { 1E0B }
16256         { AC } { 1EF2 }
16257         { AF } { 0178 }
16258         { B0 } { 1E1E }
16259         { B1 } { 1E1F }
16260         { B2 } { 0120 }
16261         { B3 } { 0121 }
16262         { B4 } { 1E40 }
16263         { B5 } { 1E41 }

```

```

16264     { B7 } { 1E56 }
16265     { B8 } { 1E81 }
16266     { B9 } { 1E57 }
16267     { BA } { 1E83 }
16268     { BB } { 1E60 }
16269     { BC } { 1EF3 }
16270     { BD } { 1E84 }
16271     { BE } { 1E85 }
16272     { BF } { 1E61 }
16273     { DO } { 0174 }
16274     { D7 } { 1E6A }
16275     { DE } { 0176 }
16276     { FO } { 0175 }
16277     { F7 } { 1E6B }
16278     { FE } { 0177 }
16279   }
16280   {
16281   }
16282 </iso885914>
16283 <iso885915>
16284 \_str_declare_eight_bit_encoding:nnnn { iso885915 } { 383 }
16285   {
16286     { A4 } { 20AC }
16287     { A6 } { 0160 }
16288     { A8 } { 0161 }
16289     { B4 } { 017D }
16290     { B8 } { 017E }
16291     { BC } { 0152 }
16292     { BD } { 0153 }
16293     { BE } { 0178 }
16294   }
16295   {
16296   }
16297 </iso885915>
16298 <iso885916>
16299 \_str_declare_eight_bit_encoding:nnnn { iso885916 } { 558 }
16300   {
16301     { A1 } { 0104 }
16302     { A2 } { 0105 }
16303     { A3 } { 0141 }
16304     { A4 } { 20AC }
16305     { A5 } { 201E }
16306     { A6 } { 0160 }
16307     { A8 } { 0161 }
16308     { AA } { 0218 }
16309     { AC } { 0179 }
16310     { AE } { 017A }
16311     { AF } { 017B }
16312     { B2 } { 010C }
16313     { B3 } { 0142 }
16314     { B4 } { 017D }
16315     { B5 } { 201D }
16316     { B8 } { 017E }

```

```
16317 { B9 } { 010D }
16318 { BA } { 0219 }
16319 { BC } { 0152 }
16320 { BD } { 0153 }
16321 { BE } { 0178 }
16322 { BF } { 017C }
16323 { C3 } { 0102 }
16324 { C5 } { 0106 }
16325 { D0 } { 0110 }
16326 { D1 } { 0143 }
16327 { D5 } { 0150 }
16328 { D7 } { 015A }
16329 { D8 } { 0170 }
16330 { DD } { 0118 }
16331 { DE } { 021A }
16332 { E3 } { 0103 }
16333 { E5 } { 0107 }
16334 { F0 } { 0111 }
16335 { F1 } { 0144 }
16336 { F5 } { 0151 }
16337 { F7 } { 015B }
16338 { F8 } { 0171 }
16339 { FD } { 0119 }
16340 { FE } { 021B }
16341 }
16342 {
16343 }
16344 </iso885916>
```

Chapter 57

l3quark implementation

The following test files are used for this code: *m3quark001.lvt*.

```
16345 (*package)
```

57.1 Quarks

```
16346 (@@=quark)
```

`\quark_new:N` Allocate a new quark.

```
16347 \cs_new_protected:Npn \quark_new:N #1
16348   {
16349     \__kernel_chk_if_free_cs:N #1
16350     \cs_gset_nopar:Npn #1 {#1}
16351   }
```

(End of definition for `\quark_new:N`. This function is documented on page 147.)

`\q_nil` Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

```
\q_mark
\q_no_value
\q_stop
16352 \quark_new:N \q_nil
16353 \quark_new:N \q_mark
16354 \quark_new:N \q_no_value
16355 \quark_new:N \q_stop
```

(End of definition for `\q_nil` and others. These variables are documented on page 147.)

`\q_recursion_tail` Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```
16356 \quark_new:N \q_recursion_tail
16357 \quark_new:N \q_recursion_stop
```

(End of definition for `\q_recursion_tail` and `\q_recursion_stop`. These variables are documented on page 148.)

`\s__quark` Private scan mark used in l3quark. We don’t have l3scan yet, so we declare the scan mark here and add it to the scan mark pool later.

```
16358 \cs_new_eq:NN \s__quark \scan_stop:
```


(End of definition for `\s__quark`.)

`\q__quark_nil` Private quark use for some tests.

```
16359 \quark_new:N \q__quark_nil
```

(End of definition for `\q__quark_nil`.)

`\quark_if_recursion_tail_stop:N`
`\quark_if_recursion_tail_stop_do:Nn`

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
16360 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
16361 {
16362   \if_meaning:w \q_recursion_tail #1
16363   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
16364   \fi:
16365 }
16366 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
16367 {
16368   \if_meaning:w \q_recursion_tail #1
16369   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
16370   \else:
16371   \exp_after:wN \use_none:n
16372   \fi:
16373 }
```

(End of definition for `\quark_if_recursion_tail_stop:N` and `\quark_if_recursion_tail_stop_do:Nn`. These functions are documented on page 148.)

`\quark_if_recursion_tail_stop:n`
`\quark_if_recursion_tail_stop:o`
`\quark_if_recursion_tail_stop_do:nn`

See `\quark_if_nil:nTF` for the details. Expanding `__quark_if_recursion_tail:w` once in front of the tokens chosen here gives an empty result if and only if #1 is exactly `\q_recursion_tail`.

`\quark_if_recursion_tail_stop_do:n`
`__quark_if_recursion_tail:w`

```
16374 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
16375 {
16376   \tl_if_empty:oTF
16377   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
16378   { \use_none_delimit_by_q_recursion_stop:w }
16379   { }
16380 }
16381 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
16382 {
16383   \tl_if_empty:oTF
16384   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
16385   { \use_i_delimit_by_q_recursion_stop:nw }
16386   { \use_none:n }
16387 }
16388 \cs_new:Npn \__quark_if_recursion_tail:w
16389   #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
16390 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
16391 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(End of definition for `\quark_if_recursion_tail_stop:n`, `\quark_if_recursion_tail_stop_do:nn`, and `__quark_if_recursion_tail:w`. These functions are documented on page 148.)

`\quark_if_recursion_tail_break:NN` Analogues of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using #2.

```

16392 \cs_new:Npn \quark_if_recursion_tail_break:NN #1#2
16393 {
16394   \if_meaning:w \q_recursion_tail #1
16395   \exp_after:wN #2
16396   \fi:
16397 }
16398 \cs_new:Npn \quark_if_recursion_tail_break:nN #1#2
16399 {
16400   \tl_if_empty:oT
16401   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
16402   {#2}
16403 }

```

(End of definition for `\quark_if_recursion_tail_break:NN` and `\quark_if_recursion_tail_break:nN`. These functions are documented on page 149.)

`\quark_if_nil_p:N` Here we test if we found a special quark as the first argument. We better start with `\quark_if_nil:N` as the first argument since the whole thing may otherwise loop if #1 is wrongly given a string like `aabc` instead of a single token.⁹

```

\quark_if_no_value_p:N
\quark_if_no_value_p:c
\quark_if_no_value:NTF
\quark_if_no_value:cTF
16404 \prg_new_conditional:Npnn \quark_if_nil:N #1 { p , T , F , TF }
16405 {
16406   \if_meaning:w \q_nil #1
16407   \prg_return_true:
16408   \else:
16409   \prg_return_false:
16410   \fi:
16411 }
16412 \prg_new_conditional:Npnn \quark_if_no_value:N #1 { p , T , F , TF }
16413 {
16414   \if_meaning:w \q_no_value #1
16415   \prg_return_true:
16416   \else:
16417   \prg_return_false:
16418   \fi:
16419 }
16420 \prg_generate_conditional_variant:Nnn \quark_if_no_value:N
16421 { c } { p , T , F , TF }

```

(End of definition for `\quark_if_nil:N` and `\quark_if_no_value:N`. These functions are documented on page 147.)

`\quark_if_nil_p:n` Let us explain `\quark_if_nil:n`. Expanding `__quark_if_nil:w` once is safe thanks to the trailing `\q_nil ???`. The result of expanding once is empty if and only if both delimited arguments #1 and #2 are empty and #3 is delimited by the last tokens `?!`.
`\quark_if_nil_p:V` Thanks to the leading `{}`, the argument #1 is empty if and only if the argument of `\quark_if_nil:n` starts with `\q_nil`. The argument #2 is empty if and only if this `\q_nil` is followed immediately by `?` or by `{}`?, coming either from the trailing tokens in the definition of `\quark_if_nil:n`, or from its argument. In the first case, `__quark_if_nil:w` is followed by `{}\q_nil {}? !\q_nil ???`, hence #3 is delimited by the final `?!`, and the test returns true as wanted. In the second case, the result is not empty since

⁹It may still loop in special circumstances however!

the first ?! in the definition of `\quark_if_nil:n stop #3`. The auxiliary here is the same as `__tl_if_empty_if:o`, with the same comments applying.

```

16422 \prg_new_conditional:Npnn \quark_if_nil:n #1 { p , T , F , TF }
16423 {
16424   \__quark_if_empty_if:o
16425   { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
16426   \prg_return_true:
16427   \else:
16428   \prg_return_false:
16429   \fi:
16430 }
16431 \cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
16432 \prg_new_conditional:Npnn \quark_if_no_value:n #1 { p , T , F , TF }
16433 {
16434   \__quark_if_empty_if:o
16435   { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
16436   \prg_return_true:
16437   \else:
16438   \prg_return_false:
16439   \fi:
16440 }
16441 \cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
16442 \prg_generate_conditional_variant:Nnn \quark_if_nil:n
16443 { V , o } { p , TF , T , F }
16444 \cs_new:Npn \__quark_if_empty_if:o #1
16445 {
16446   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
16447   \__kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
16448 }

```

(End of definition for `\quark_if_nil:nTF` and others. These functions are documented on page 147.)

`__kernel_quark_new_test:N` The function `__kernel_quark_new_test:N` defines `#1` in a similar way as `\quark_if_recursion_tail_...` functions (as described below), using `\q_<namespace>_recursion_tail` as the test quark and `\q_<namespace>_recursion_stop` as the delimiter quark, where the `<namespace>` is determined as the first `_`-delimited part in `#1`.

There are six possible function types which this function can define, and which is defined depends on the signature of the function being defined:

```

:n gives an analogue of \quark_if_recursion_tail_stop:n
:nn gives an analogue of \quark_if_recursion_tail_stop_do:nn
:nN gives an analogue of \quark_if_recursion_tail_break:nN
:N gives an analogue of \quark_if_recursion_tail_stop:N
:Nn gives an analogue of \quark_if_recursion_tail_stop_do:Nn
:NN gives an analogue of \quark_if_recursion_tail_break:NN

```

Any other signature causes an error, as does a function without signature.

`_kernel_quark_new_conditional:Nn`

Similar to `_kernel_quark_new_test:N`, but defines quark branching conditionals like `\quark_if_nil:nTF` that test for the quark `\q_\namespace_\name`. The *namespace* and *name* are determined from the conditional #1, which must take the rather rigid form `_\namespace_quark_if_\name:\arg spec`. There are only two cases for the *arg spec* here:

:n gives an analogue of `\quark_if_nil:nTF`

:N gives an analogue of `\quark_if_nil:NTF`

Any other signature causes an error, as does a function without signature. We use low-level emptiness tests as `|3t|` is not available yet when these functions are used; thankfully we only care about whether strings are empty so a simple `\if_meaning:w \q_nil` (*string*) `\q_nil` suffices.

```
16449 \cs_new_protected:Npn \_kernel\_quark\_new\_test:N #1
16450 { \_quark\_new\_test\_aux:Ne #1 { \_quark\_module\_name:N #1 } }
\_quark\_new\_test:NNNn 16451 \cs_new_protected:Npn \_quark\_new\_test\_aux:Nn #1 #2
\_quark\_new\_test:Nccn 16452 {
\_quark\_new\_test\_aux:nnNNnnnn 16453   \if\_meaning:w \q\_nil #2 \q\_nil
\_quark\_new\_conditional:Nnnn 16454     \msg\_error:nne { quark } { invalid-function }
\_quark\_new\_conditional:Neen 16455     { \token\_to\_str:N #1 }
16456   \else:
16457     \_quark\_new\_test:Nccn #1
16458     { q\_#2\_recursion\_tail } { q\_#2\_recursion\_stop } { \_#2 }
16459   \fi:
16460 }
16461 \cs_generate_variant:Nn \_quark\_new\_test\_aux:Nn { Ne }
16462 \cs_new_protected:Npn \_quark\_new\_test:NNNn #1
16463 {
16464   \exp\_last\_unbraced:Nf \_quark\_new\_test\_aux:nnNNnnnn
16465   { \cs\_split\_function:N #1 }
16466   #1 { test }
16467 }
16468 \cs_generate_variant:Nn \_quark\_new\_test:NNNn { Ncc }
16469 \cs_new_protected:Npn \_kernel\_quark\_new\_conditional:Nn #1
16470 {
16471   \_quark\_new\_conditional:Neen #1
16472   { \_quark\_quark\_conditional\_name:N #1 }
16473   { \_quark\_module\_name:N #1 }
16474 }
16475 \cs_new_protected:Npn \_quark\_new\_conditional:Nnnn #1#2#3#4
16476 {
16477   \if\_meaning:w \q\_nil #2 \q\_nil
16478   \msg\_error:nne { quark } { invalid-function }
16479   { \token\_to\_str:N #1 }
16480   \else:
16481   \if\_meaning:w \q\_nil #3 \q\_nil
16482   \msg\_error:nne { quark } { invalid-function }
16483   { \token\_to\_str:N #1 }
16484   \else:
16485   \exp\_last\_unbraced:Nf \_quark\_new\_test\_aux:nnNNnnnn
16486   { \cs\_split\_function:N #1 }
```

```

16487         #1 { conditional }
16488         {#2} {#3} {#4}
16489     \fi:
16490 \fi:
16491 }
16492 \cs_generate_variant:Nn \__quark_new_conditional:Nnnn { Nee }
16493 \cs_new_protected:Npn \__quark_new_test_aux:nNNnnnn #1 #2 #3 #4 #5
16494 {
16495     \cs_if_exist_use:cTF { __quark_new_#5_#2:Nnnn } { #4 }
16496     {
16497         \msg_error:nnee { quark } { invalid-function }
16498         { \token_to_str:N #4 } {#2}
16499         \use_none:nnn
16500     }
16501 }

```

(End of definition for __kernel_quark_new_test:N and others.)

__quark_new_test_n:Nnnn These macros implement the six possibilities mentioned above, passing the right arguments to __quark_new_test_aux_do:nNNnnnnNNn, which defines some auxiliaries, and then to __quark_new_test_define_tl:nNnNNn (:n(n) variants) or to __quark_new_test_define_ifx:nNnNNn (:N(n)) which define the main conditionals.

```

16502 \cs_new_protected:Npn \__quark_new_test_n:Nnnn #1 #2 #3 #4
16503 {
16504     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
16505     \__quark_new_test_define_tl:nNnNNn #1 { }
16506 }
16507 \cs_new_protected:Npn \__quark_new_test_nn:Nnnn #1 #2 #3 #4
16508 {
16509     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16510     \__quark_new_test_define_tl:nNnNNn #1 { \use_none:n }
16511 }
16512 \cs_new_protected:Npn \__quark_new_test_nN:Nnnn #1 #2 #3 #4
16513 {
16514     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16515     \__quark_new_test_define_break_tl:nNNNNn #1 { }
16516 }
16517 \cs_new_protected:Npn \__quark_new_test_N:Nnnn #1 #2 #3 #4
16518 {
16519     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
16520     \__quark_new_test_define_ifx:nNnNNn #1 { }
16521 }
16522 \cs_new_protected:Npn \__quark_new_test_Nn:Nnnn #1 #2 #3 #4
16523 {
16524     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16525     \__quark_new_test_define_ifx:nNnNNn #1
16526     { \else: \exp_after:wN \use_none:n }
16527 }
16528 \cs_new_protected:Npn \__quark_new_test_NN:Nnnn #1 #2 #3 #4
16529 {
16530     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16531     \__quark_new_test_define_break_ifx:nNNNNn #1 { }
16532 }

```

(End of definition for __quark_new_test_n:Nnnn and others.)

_quark_new_test_aux_do:nNNnnnnNNn
 _quark_test_define_aux:NNNNnnNNn

_quark_new_test_aux_do:nNNnnnnNNn makes the control sequence names which will be used by _quark_test_define_aux:NNNNnnNNn, and then later by _quark_new_test_define_tl:nNnNNn or _quark_new_test_define_ifx:nNnNNn. The control sequences defined here are analogous to _quark_if_recursion_tail:w and to \use_-(none|i)_delimit_by_q_recursion_stop:(|n)w.

The name is composed by the name-space and the name of the quarks. Suppose _kernel_quark_new_test:N was used with:

```
\_kernel_quark_new_test:N \_test_quark_tail:n
```

then the first auxiliary will be _test_quark_recursion_tail:w, and the second one will be _test_use_none_delimit_by_q_recursion_stop:w.

Note that the actual quarks are *not* defined here. They should be defined separately using \quark_new:N.

```
16533 \cs_new_protected:Npn \_quark_new_test_aux_do:nNNnnnnNNn #1 #2 #3 #4 #5
16534 {
16535   \exp_args:Ncc \_quark_test_define_aux:NNNNnnNNn
16536   { #1 \_quark_recursion_tail:w }
16537   { #1 \use_#4_delimit_by_q_recursion_stop: #5 w }
16538   #2 #3
16539 }
16540 \cs_new_protected:Npn \_quark_test_define_aux:NNNNnnNNn #1 #2 #3 #4 #5 #6 #7
16541 {
16542   \cs_gset:Npn #1 ##1 #3 ##2 ? ##3 ?! { ##1 ##2 }
16543   \cs_gset:Npn #2 ##1 #6 #4 {#5}
16544   #7 {##1} #1 #2 #3
16545 }
```

(End of definition for _quark_new_test_aux_do:nNNnnnnNNn and _quark_test_define_aux:NNNNnnNNn.)

_quark_new_test_define_tl:nNnNNn
 _quark_new_test_define_ifx:nNnNNn
 _quark_new_test_define_break_tl:nNNNNn
 _quark_new_test_define_break_ifx:nNNNNn

Finally, these two macros define the main conditional function using what's been set up before.

```
16546 \cs_new_protected:Npn \_quark_new_test_define_tl:nNnNNn #1 #2 #3 #4 #5 #6
16547 {
16548   \cs_new:Npn #5 #1
16549   {
16550     \tl_if_empty:oTF
16551     { #2 {} ##1 {} ?! #4 ??! }
16552     {#3} {#6}
16553   }
16554 }
16555 \cs_new_protected:Npn \_quark_new_test_define_ifx:nNnNNn #1 #2 #3 #4 #5 #6
16556 {
16557   \cs_new:Npn #5 #1
16558   {
16559     \if_meaning:w #4 ##1
16560     \exp_after:wN #3
16561     #6
16562     \fi:
16563   }
16564 }
16565 \cs_new_protected:Npn \_quark_new_test_define_break_tl:nNNNNn #1 #2 #3
16566 { \_quark_new_test_define_tl:nNnNNn {##1##2} #2 {##2} }
16567 \cs_new_protected:Npn \_quark_new_test_define_break_ifx:nNNNNn #1 #2 #3
```

```
16568 { \_quark_new_test_define_ifx:nNnNNn {##1##2} #2 {##2} }
```

(End of definition for _quark_new_test_define_tl:nNnNNn and others.)

```
\_quark_new_conditional_n:Nnnn
\_quark_new_conditional_N:Nnnn
\_quark_new_conditional_n_aux:NNNn
\_quark_new_conditional_N_aux:NNNn
```

These macros implement the two possibilities for branching quark conditionals. To avoid constructing without defining the _<type>_if_quark_<name>:w helper, N-type function accepts a \prg_do_nothing: as a placeholder.

```
16569 \cs_new_protected:Npn \_quark_new_conditional_n:Nnnn #1 #2 #3
16570 {
16571   \exp_args:Ncc \_quark_new_conditional_n_aux:NNNn
16572   { \_ #3 _if_quark_ #2 :w } { q\_ #3 _ #2 } #1
16573 }
16574 \cs_new_protected:Npn \_quark_new_conditional_N:Nnnn #1 #2 #3
16575 {
16576   \exp_args:NNc \_quark_new_conditional_N_aux:NNNn
16577   \prg_do_nothing: { q\_ #3 _ #2 } #1
16578 }
16579 \cs_new_protected:Npn \_quark_new_conditional_n_aux:NNNn #1 #2 #3 #4
16580 {
16581   \cs_gset:Npn #1 ##1 #2 ##2 ? ##3 ?! { ##1##2 }
16582   \prg_new_conditional:Npnn #3 ##1 {#4}
16583   {
16584     \_quark_if_empty_if:o { #1 {} ##1 {} ?! #2 ??! }
16585     \prg_return_true:
16586     \else:
16587       \prg_return_false:
16588     \fi:
16589   }
16590 }
16591 \cs_new_protected:Npn \_quark_new_conditional_N_aux:NNNn #1 #2 #3 #4
16592 {
16593   \prg_new_conditional:Npnn #3 ##1 {#4}
16594   {
16595     \if_meaning:w #2 ##1
16596     \prg_return_true:
16597     \else:
16598       \prg_return_false:
16599     \fi:
16600   }
16601 }
```

(End of definition for _quark_new_conditional_n:Nnnn and others.)

```
\_quark_module_name:N
\_quark_module_name:w
\_quark_module_name_loop:w
\_quark_module_name_end:w
```

_quark_module_name:N takes a control sequence and returns its <module> name, determined as the first non-empty non-single-character word, separated by _ or :. These rules give the correct result for public functions \<module>_..., private functions _<module>_..., and variables such as \1_<module>_.... If no valid module is found the result is an empty string. The approach is to first cut off everything after the (first) : if any is present, then repeatedly grab _-delimited words until finding one of length at least 2 (we use low-level tests as |3t| is not fully available when _kernel_quark_new_test:N is first used. If no <module> is found (such as in \::n) we get the trailing marker \use_none:n {}, which expands to nothing.

```
16602 \cs_set:Npn \_quark_tmp:w #1#2
16603 {
```

```

16604 \cs_new:Npn \__quark_module_name:N ##1
16605 {
16606   \exp_last_unbraced:Nf \__quark_module_name:w
16607   { \cs_to_str:N ##1 } #1 \s__quark
16608 }
16609 \cs_new:Npn \__quark_module_name:w ##1 #1 ##2 \s__quark
16610 { \__quark_module_name_loop:w ##1 #2 \use_none:n { } #2 \s__quark }
16611 \cs_new:Npn \__quark_module_name_loop:w ##1 #2
16612 {
16613   \use_i_ii:nnn \if_meaning:w \prg_do_nothing:
16614   ##1 \prg_do_nothing: \prg_do_nothing:
16615   \exp_after:wN \__quark_module_name_loop:w
16616   \else:
16617     \__quark_module_name_end:w ##1
16618   \fi:
16619 }
16620 \cs_new:Npn \__quark_module_name_end:w
16621 ##1 \fi: ##2 \s__quark { \fi: ##1 }
16622 }
16623 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _ }

```

(End of definition for `__quark_module_name:N` and others.)

`__quark_quark_conditional_name:N`
`__quark_quark_conditional_name:w`

`__quark_quark_conditional_name:N` determines the quark name that the quark conditional function `##1` queries, as the part of the function name between `_quark_if_` and the trailing `:`. Again we define it through `__quark_tmp:w`, which receives `:` as `#1` and `_quark_if_` as `#2`. The auxiliary `__quark_quark_conditional_name:w` returns the part between the first `_quark_if_` and the next `:`, and we apply this auxiliary to the function name followed by `:` (in case the function name is lacking a signature), and `_quark_if_:` so that `__quark_quark_conditional_name:N` returns an empty string if `_quark_if_` is not present.

```

16624 \cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
16625 {
16626   \cs_new:Npn \__quark_quark_conditional_name:N ##1
16627   {
16628     \exp_last_unbraced:Nf \__quark_quark_conditional_name:w
16629     { \cs_to_str:N ##1 } #1 #2 #1 \s__quark
16630   }
16631   \cs_new:Npn \__quark_quark_conditional_name:w
16632   ##1 #2 ##2 #1 ##3 \s__quark {##2}
16633 }
16634 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark

```

(End of definition for `__quark_quark_conditional_name:N` and `__quark_quark_conditional_name:w`.)

57.2 Scan marks

16635 `<@@=scan>`

`\scan_new:N` Check whether the variable is already a scan mark, then declare it to be equal to `\scan_stop:` globally.

```

16636 \cs_new_protected:Npn \scan_new:N #1
16637 {

```



```

16638 \tl_if_in:NnTF \g__scan_marks_tl { #1 }
16639 {
16640   \msg_error:nne { scanmark } { already-defined }
16641   { \token_to_str:N #1 }
16642 }
16643 {
16644   \tl_gput_right:Nn \g__scan_marks_tl {#1}
16645   \cs_new_eq:NN #1 \scan_stop:
16646 }
16647 }

```

(End of definition for `\scan_new:N`. This function is documented on page 150.)

`\s_stop` We only declare one scan mark here, more can be defined by specific modules. Can't
`\g__scan_marks_tl` use `\scan_new:N` yet because `l3tl` isn't loaded, so define `\s_stop` by hand and add it to
`\g__scan_marks_tl`. We also add the scan marks declared earlier to the pool here. Since
it lives in a different namespace, a little DocStrip cheating is necessary.

```

16648 \cs_new_eq:NN \s_stop \scan_stop:
16649 \cs_gset_nopar:Npn \g__scan_marks_tl
16650 {
16651   \s_stop
16652   <@@=quark>
16653   \s__quark
16654   <@@=cs>
16655   \s__cs_mark
16656   \s__cs_stop
16657   <@@=scan>
16658 }

```

(End of definition for `\s_stop` and `\g__scan_marks_tl`. This variable is documented on page 150.)

`\use_none_delimit_by_s_stop:w` Similar to `\use_none_delimit_by_q_stop:w`.

```

16659 \cs_new:Npn \use_none_delimit_by_s_stop:w #1 \s_stop { }

```

(End of definition for `\use_none_delimit_by_s_stop:w`. This function is documented on page 150.)

```

16660 </package>

```

Chapter 58

l3seq implementation

The following test files are used for this code: *m3seq002,m3seq003*.

```
16661 (*package)
16662 (@@=seq)
```

A sequence is a control sequence whose top-level expansion is of the form “`\s__seq __seq_item:n {<item1>} ... __seq_item:n {<itemn>}`”, with a leading scan mark followed by *n* items of the same form. An earlier implementation used the structure “`\seq_elt:w <item1> \seq_elt_end: ... \seq_elt:w <itemn> \seq_elt_end:`”. This allowed rapid searching using a delimited function, but was not suitable for items containing `{, }` and `#` tokens, and also lead to the loss of surrounding braces around items

```
\__seq_item:n * \__seq_item:n {<item>}
```

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

```
\__seq_push_item_def:n \__seq_push_item_def:n {<code>}
```

```
\__seq_push_item_def:e
```

Saves the definition of `__seq_item:n` and redefines it to accept one parameter and expand to `<code>`. This function should always be balanced by use of `__seq_pop_item_def:.`

```
\__seq_pop_item_def: \__seq_pop_item_def:
```

Restores the definition of `__seq_item:n` most recently saved by `__seq_push_item_def:n`. This function should always be used in a balanced pair with `__seq_push_item_def:n`.

```
\s__seq This private scan mark.
```

```
16663 \scan_new:N \s__seq
```

(End of definition for `\s__seq`.)

```
\s__seq_mark Private scan marks.
```

```
\s__seq_stop 16664 \scan_new:N \s__seq_mark
```

```
16665 \scan_new:N \s__seq_stop
```

(End of definition for `\s__seq_mark` and `\s__seq_stop`.)

`__seq_item:n` The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
16666 \cs_new:Npn \__seq_item:n
16667   {
16668     \msg_expandable_error:nn { seq } { misused }
16669     \use_none:n
16670   }
```

(End of definition for __seq_item:n.)

`\l__seq_internal_a_tl` Scratch space for various internal uses.

```
\l__seq_internal_b_tl
16671 \tl_new:N \l__seq_internal_a_tl
16672 \tl_new:N \l__seq_internal_b_tl
```

(End of definition for \l__seq_internal_a_tl and \l__seq_internal_b_tl.)

`__seq_tmp:w` Scratch function for internal use.

```
16673 \cs_new_eq:NN \__seq_tmp:w ?
```

(End of definition for __seq_tmp:w.)

`\c_empty_seq` A sequence with no item, following the structure mentioned above.

```
16674 \tl_const:Nn \c_empty_seq { \s__seq }
```

(End of definition for \c_empty_seq. This variable is documented on page 163.)

58.1 Allocation and initialisation

`\seq_new:N` Sequences are initialized to `\c_empty_seq`.

```
\seq_new:c
16675 \cs_new_protected:Npn \seq_new:N #1
16676   {
16677     \__kernel_chk_if_free_cs:N #1
16678     \cs_gset_eq:NN #1 \c_empty_seq
16679   }
16680 \cs_generate_variant:Nn \seq_new:N { c }
```

(End of definition for \seq_new:N. This function is documented on page 151.)

`\seq_clear:N` Clearing a sequence is similar to setting it equal to the empty one.

```
\seq_clear:c
16681 \cs_new_protected:Npn \seq_clear:N #1
\seq_gclear:N
16682   { \seq_set_eq:NN #1 \c_empty_seq }
\seq_gclear:c
16683 \cs_generate_variant:Nn \seq_clear:N { c }
16684 \cs_new_protected:Npn \seq_gclear:N #1
16685   { \seq_gset_eq:NN #1 \c_empty_seq }
16686 \cs_generate_variant:Nn \seq_gclear:N { c }
```

(End of definition for \seq_clear:N and \seq_gclear:N. These functions are documented on page 151.)

`\seq_clear_new:N` Once again we copy code from the token list functions.

```
\seq_clear_new:c
16687 \cs_new_protected:Npn \seq_clear_new:N #1
\seq_gclear_new:N
16688   { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\seq_gclear_new:c
16689 \cs_generate_variant:Nn \seq_clear_new:N { c }
16690 \cs_new_protected:Npn \seq_gclear_new:N #1
16691   { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
16692 \cs_generate_variant:Nn \seq_gclear_new:N { c }
```

(End of definition for `\seq_clear_new:N` and `\seq_gclear_new:N`. These functions are documented on page 151.)

`\seq_set_eq:NN` Copying a sequence is the same as copying the underlying token list.

```

\seq_set_eq:cN 16693 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
\seq_set_eq:Nc 16694 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
\seq_set_eq:cc 16695 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
\seq_gset_eq:NN 16696 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
\seq_gset_eq:cN 16697 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\seq_gset_eq:Nc 16698 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\seq_gset_eq:cN 16699 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\seq_gset_eq:cc 16700 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc

```

(End of definition for `\seq_set_eq:NN` and `\seq_gset_eq:NN`. These functions are documented on page 151.)

`\seq_set_from_clist:NN` Setting a sequence from a comma-separated list is done using a simple mapping.

```

\seq_set_from_clist:cN 16701 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
\seq_set_from_clist:Nc 16702 {
\seq_set_from_clist:cc 16703   \__kernel_tl_set:Nx #1
\seq_set_from_clist:Nn 16704   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
\seq_set_from_clist:cn 16705 }
\seq_gset_from_clist:NN 16706 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
\seq_gset_from_clist:cN 16707 {
\seq_gset_from_clist:Nc 16708   \__kernel_tl_set:Nx #1
\seq_gset_from_clist:Nc 16709   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
\seq_gset_from_clist:cc 16710 }
\seq_gset_from_clist:Nn 16711 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
\seq_gset_from_clist:cn 16712 {
16713   \__kernel_tl_gset:Nx #1
16714   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
16715 }
16716 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
16717 {
16718   \__kernel_tl_gset:Nx #1
16719   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
16720 }
16721 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
16722 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
16723 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
16724 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
16725 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
16726 \cs_generate_variant:Nn \seq_gset_from_clist:Nn { c }

```

(End of definition for `\seq_set_from_clist:NN` and others. These functions are documented on page 152.)

`\seq_const_from_clist:Nn` Almost identical to `\seq_set_from_clist:Nn`.

```

\seq_const_from_clist:cn 16727 \cs_new_protected:Npn \seq_const_from_clist:Nn #1#2
16728 {
16729   \tl_const:Ne #1
16730   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
16731 }
16732 \cs_generate_variant:Nn \seq_const_from_clist:Nn { c }

```

(End of definition for `\seq_const_from_clist:Nn`. This function is documented on page 152.)

```

\seq_set_split:Nnn When the separator is empty, everything is very simple, just map \__seq_wrap_item:n
\seq_set_split:NVn through the items of the last argument. For non-trivial separators, the goal is to split a
\seq_set_split:NnV given token list at the marker, strip spaces from each item, and remove one set of outer
\seq_set_split:NVV braces if after removing leading and trailing spaces the item is enclosed within braces. Af-
\seq_set_split:Nne ter \tl_replace_all:Nnn, the token list \l__seq_internal_a_tl is a repetition of the
\seq_set_split:Nee pattern \__seq_set_split:Nw \prg_do_nothing: <item with spaces> \__seq_set_-
\seq_set_split:Nnx split_end:. Then, e-expansion causes \__seq_set_split:Nw to trim spaces, and leaves
\seq_set_split:Nxx its result as \__seq_set_split:w <trimmed item> \__seq_set_split_end:. This is
\seq_gset_split:Nnn then converted to the l3seq internal structure by another e-expansion. In the first step,
\seq_gset_split:NVn we insert \prg_do_nothing: to avoid losing braces too early: that would cause space
\seq_gset_split:NnV trimming to act within those lost braces. The second step is solely there to strip braces
\seq_gset_split:NVV which are outermost after space trimming.
\seq_gset_split:Nne
\seq_gset_split:Nee
\seq_gset_split:Nnx
\seq_gset_split:Nxx
\seq_set_split_keep_spaces:Nnn
\seq_set_split_keep_spaces:NnV
\seq_gset_split_keep_spaces:Nnn
\seq_gset_split_keep_spaces:NnV
\__seq_set_split:NNnn
  \__seq_set_split:Nw
  \__seq_set_split:w
\__seq_set_split_end:
16733 \cs_new_protected:Npn \seq_set_split:Nnn
16734   { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \tl_trim_spaces:n }
16735 \cs_new_protected:Npn \seq_gset_split:Nnn
16736   { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \tl_trim_spaces:n }
16737 \cs_new_protected:Npn \seq_set_split_keep_spaces:Nnn
16738   { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \exp_not:n }
16739 \cs_new_protected:Npn \seq_gset_split_keep_spaces:Nnn
16740   { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \exp_not:n }
16741 \cs_new_protected:Npn \__seq_set_split:NNNnn #1#2#3#4#5
16742   {
16743     \tl_if_empty:nTF {#4}
16744     {
16745       \tl_set:Nn \l__seq_internal_a_tl
16746         { \tl_map_function:nN {#5} \__seq_wrap_item:n }
16747     }
16748     {
16749       \tl_set:Nn \l__seq_internal_a_tl
16750       {
16751         \__seq_set_split:Nw #2 \prg_do_nothing:
16752         #5
16753         \__seq_set_split_end:
16754       }
16755       \tl_replace_all:Nnn \l__seq_internal_a_tl {#4}
16756       {
16757         \__seq_set_split_end:
16758         \__seq_set_split:Nw #2 \prg_do_nothing:
16759       }
16760       \__kernel_tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
16761     }
16762     #1 #3 { \s__seq \l__seq_internal_a_tl }
16763   }
16764 \cs_new:Npn \__seq_set_split:Nw #1#2 \__seq_set_split_end:
16765   {
16766     \exp_not:N \__seq_set_split:w
16767     \exp_args:No #1 {#2}
16768     \exp_not:N \__seq_set_split_end:
16769   }
16770 \cs_new:Npn \__seq_set_split:w #1 \__seq_set_split_end:
16771   { \__seq_wrap_item:n {#1} }

```

```

16772 \cs_generate_variant:Nn \seq_set_split:Nnn { NV , NnV , NVV , Nne , Nee }
16773 \cs_generate_variant:Nn \seq_set_split:Nnn { Nnx , Nxx }
16774 \cs_generate_variant:Nn \seq_gset_split:Nnn { NV , NnV , NVV , Nne , Nee }
16775 \cs_generate_variant:Nn \seq_gset_split:Nnn { Nnx , Nxx }
16776 \cs_generate_variant:Nn \seq_set_split_keep_spaces:Nnn { NnV }
16777 \cs_generate_variant:Nn \seq_gset_split_keep_spaces:Nnn { NnV }

```

(End of definition for `\seq_set_split:Nnn` and others. These functions are documented on page 152.)

`\seq_set_filter:NNn` Similar to `\seq_map_inline:Nn`, without a `\prg_break_point:` because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The `__seq_wrap_item:n` function inserts the relevant `__seq_item:n` without expansion in the input stream, hence in the e-expanding assignment.

```

16778 \cs_new_protected:Npn \seq_set_filter:NNn
16779   { \__seq_set_filter:NNNn \__kernel_tl_set:Nx }
16780 \cs_new_protected:Npn \seq_gset_filter:NNn
16781   { \__seq_set_filter:NNNn \__kernel_tl_gset:Nx }
16782 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
16783   {
16784     \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
16785     #1 #2 { #3 }
16786     \__seq_pop_item_def:
16787   }

```

(End of definition for `\seq_set_filter:NNn`, `\seq_gset_filter:NNn`, and `__seq_set_filter:NNNn`. These functions are documented on page 153.)

`\seq_concat:NNN` When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops f-expansion.

```

16788 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
16789   { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
16790 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
16791   { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
16792 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
16793 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End of definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 153.)

`\seq_if_exist_p:N` Copies of the cs functions defined in l3basics.

```

16794 \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
16795   { TF , T , F , p }
16796 \prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c
16797   { TF , T , F , p }

```

(End of definition for `\seq_if_exist:NTF`. This function is documented on page 153.)

58.2 Appending data to either end

`\seq_put_left:Nn` When adding to the left of a sequence, remove `\s__seq`. This is done by `__seq_put_left_aux:w`, which also stops f-expansion.

```

16798 \cs_new_protected:Npn \seq_put_left:Nn #1#2
16799   {
16800     \__kernel_tl_set:Nx #1

```

```

\seq_gput_left:Nn
\seq_gput_left:Nv
\seq_gput_left:Nv

```

```

16801     {
16802     \exp_not:n { \s__seq \__seq_item:n {#2} }
16803     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
16804     }
16805   }
16806 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
16807   {
16808     \__kernel_tl_gset:Nx #1
16809     {
16810       \exp_not:n { \s__seq \__seq_item:n {#2} }
16811       \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
16812     }
16813   }
16814 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
16815 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , Ne , No , Nx }
16816 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , ce , co , cx }
16817 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , Ne , No , Nx }
16818 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , ce , co , cx }

```

(End of definition for `\seq_put_left:Nn`, `\seq_gput_left:Nn`, and `__seq_put_left_aux:w`. These functions are documented on page 153.)

`\seq_put_right:Nn` Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `__seq_item:n`.

```

\seq_put_right:NV
\seq_put_right:Nv
\seq_put_right:Ne
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:cv
\seq_put_right:cx
\seq_put_right:co
\seq_put_right:cx
16819 \cs_new_protected:Npn \seq_put_right:Nn #1#2
16820   { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
16821 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
16822   { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
16823 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , Ne , No , Nx }
16824 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , ce , co , cx }
16825 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , Ne , No , Nx }
16826 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , ce , co , cx }

```

(End of definition for `\seq_put_right:Nn` and `\seq_gput_right:Nn`. These functions are documented on page 153.)

`\seq_gput_right:Nn`

```

\seq_gput_right:NV
\seq_gput_right:Nv
\seq_gput_right:Ne
\__seq_wrap_item:n
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:cv
\seq_gput_right:cx

```

58.3 Modifying sequences

This function converts its argument to a proper sequence item in an e-expansion context.

```

16827 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }

```

(End of definition for `__seq_wrap_item:n`.)

An internal sequence for the removal routines.

```

16828 \seq_new:N \l__seq_remove_seq

```

(End of definition for `\l__seq_remove_seq`.)

`\seq_remove_duplicates:N` Removing duplicates means making a new list then copying it.

```

\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
\__seq_remove_duplicates:NN
16829 \cs_new_protected:Npn \seq_remove_duplicates:N
16830   { \__seq_remove_duplicates:NN \seq_set_eq:NN }
16831 \cs_new_protected:Npn \seq_gremove_duplicates:N
16832   { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
16833 \cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2

```

```

16834 {
16835   \seq_clear:N \l__seq_remove_seq
16836   \seq_map_inline:Nn #2
16837   {
16838     \seq_if_in:NnF \l__seq_remove_seq {##1}
16839     { \seq_put_right:Nn \l__seq_remove_seq {##1} }
16840   }
16841   #1 #2 \l__seq_remove_seq
16842 }
16843 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
16844 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }

```

(End of definition for `\seq_remove_duplicates:N`, `\seq_gremove_duplicates:N`, and `__seq_remove_duplicates:NN`. These functions are documented on page 156.)

```

\seq_remove_all:Nn The idea of the code here is to avoid a relatively expensive addition of items one at
\seq_remove_all:NV a time to an intermediate sequence. The approach taken is therefore similar to that
\seq_remove_all:Ne in \__seq_pop_right:NNN, using a “flexible” e-type expansion to do most of the work.
\seq_remove_all:Nx As \tl_if_eq:nnT is not expandable, a two-part strategy is needed. First, the e-type
\seq_remove_all:cn expansion uses \str_if_eq:nnT to find potential matches. If one is found, the expansion
\seq_remove_all:cV is halted and the necessary set up takes place to use the \tl_if_eq:NNT test. The e-type
\seq_remove_all:ce is started again, including all of the items copied already. This happens repeatedly until
\seq_remove_all:cx the entire sequence has been scanned. The code is set up to avoid needing an intermediate
\seq_gremove_all:Nn scratch list: the lead-off e-type expansion (#1 #2 {#2}) ensures that nothing is lost.
\seq_gremove_all:NV
\seq_gremove_all:Ne
\seq_gremove_all:Nx
\seq_gremove_all:cn
\seq_gremove_all:cV
\seq_gremove_all:ce
\seq_gremove_all:cx
\__seq_remove_all_aux:NNn
16845 \cs_new_protected:Npn \seq_remove_all:Nn
16846 { \__seq_remove_all_aux:NNn \__kernel_tl_set:Nx }
16847 \cs_new_protected:Npn \seq_gremove_all:Nn
16848 { \__seq_remove_all_aux:NNn \__kernel_tl_gset:Nx }
16849 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
16850 {
16851   \__seq_push_item_def:n
16852   {
16853     \str_if_eq:nnT {##1} {#3}
16854     {
16855       \if_false: { \fi: }
16856       \tl_set:Nn \l__seq_internal_b_tl {##1}
16857       #1 #2
16858       { \if_false: } \fi:
16859       \exp_not:o {#2}
16860       \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
16861       { \use_none:nn }
16862     }
16863     \__seq_wrap_item:n {##1}
16864   }
16865   \tl_set:Nn \l__seq_internal_a_tl {#3}
16866   #1 #2 {#2}
16867   \__seq_pop_item_def:
16868 }
16869 \cs_generate_variant:Nn \seq_remove_all:Nn { NV , Ne , c , cV , ce }
16870 \cs_generate_variant:Nn \seq_remove_all:Nn { Nx , cx }
16871 \cs_generate_variant:Nn \seq_gremove_all:Nn { NV , Ne , c , cV , ce }
16872 \cs_generate_variant:Nn \seq_gremove_all:Nn { Nx , cx }

```

(End of definition for `\seq_remove_all:Nn`, `\seq_gremove_all:Nn`, and `__seq_remove_all_aux:NNn`. These functions are documented on page 156.)

`_seq_int_eval:w` Useful to more quickly go through items.
16873 `\cs_new_eq:NN _seq_int_eval:w \tex_numexpr:D`

(End of definition for `_seq_int_eval:w`.)

`\seq_set_item:Nnn` The conditionals are distinguished from the `Nnn` versions by the last argument `\use_ii:nn` vs `\use_i:nn`.
`\seq_set_item:cnn`

```

16874 \cs_new_protected:Npn \seq_set_item:Nnn #1#2#3
16875 { \_seq_set_item:NnnNN #1 {#2} {#3} \_kernel_tl_set:Nx \use_i:nn }
16876 \cs_new_protected:Npn \seq_gset_item:Nnn #1#2#3
16877 { \_seq_set_item:NnnNN #1 {#2} {#3} \_kernel_tl_gset:Nx \use_i:nn }
16878 \cs_generate_variant:Nn \seq_set_item:Nnn { c }
16879 \cs_generate_variant:Nn \seq_gset_item:Nnn { c }
16880 \prg_new_protected_conditional:Npnn \seq_set_item:Nnn #1#2#3 { TF , T , F }
16881 { \_seq_set_item:NnnNN #1 {#2} {#3} \_kernel_tl_set:Nx \use_ii:nn }
16882 \prg_new_protected_conditional:Npnn \seq_gset_item:Nnn #1#2#3 { TF , T , F }
16883 { \_seq_set_item:NnnNN #1 {#2} {#3} \_kernel_tl_gset:Nx \use_ii:nn }
16884 \prg_generate_conditional_variant:Nnn \seq_set_item:Nnn { c } { TF , T , F }
16885 \prg_generate_conditional_variant:Nnn \seq_gset_item:Nnn { c } { TF , T , F }

```

Save the item to be stored and evaluate the position and the sequence length only once. Then depending on the sign of the position, check that it is not bigger than the length (in absolute value) nor zero.

```

16886 \cs_new_protected:Npn \_seq_set_item:NnnNN #1#2#3
16887 {
16888   \tl_set:Nn \l__seq_internal_a_tl { \_seq_item:n {#3} }
16889   \exp_args:Nff \_seq_set_item:nnNNNN
16890     { \int_eval:n {#2} } { \seq_count:N #1 } #1 \use_none:nn
16891 }
16892 \cs_new_protected:Npn \_seq_set_item:nnNNNN #1#2
16893 {
16894   \int_compare:nNnTF {#1} > 0
16895     { \int_compare:nNnF {#1} > {#2} { \_seq_set_item:nNnnNNN { #1 - 1 } } }
16896     {
16897       \int_compare:nNnF {#1} < {-#2}
16898         {
16899           \int_compare:nNnF {#1} = 0
16900             { \_seq_set_item:nNnnNNN { #2 + #1 } }
16901         }
16902     }
16903   \_seq_set_item_false:nnNNNN {#1} {#2}
16904 }

```

If the position is not ok, `_seq_set_item_false:nnNNNN` calls an error or returns false (depending on the `\use_i:nn` vs `\use_ii:nn` argument mentioned above).

```

16905 \cs_new_protected:Npn \_seq_set_item_false:nnNNNN #1#2#3#4#5#6
16906 {
16907   #6
16908   {
16909     \msg_error:nneee { seq } { item-too-large }
16910     { \token_to_str:N #3 } {#2} {#1}
16911   }
16912   { \prg_return_false: }
16913 }

```

If the position is ok, `__seq_set_item:nNnnNNNN` makes the assignment and returns `true` (in the case of conditionals). Here `#1` is an integer expression (position minus one), it needs to be evaluated. The sequence `#5` starts with `\s__seq` (even if empty), which stops the integer expression and is absorbed by it. The `\if_meaning:w` test is slightly faster than an integer test (but only works when testing against zero, hence the offset we chose in the position). When we are done skipping items, insert the saved item `\l__seq_internal_a_tl`. For put functions the last argument of `__seq_set_item_end:w` is `\use_none:nn` and it absorbs the item `#2` that we are removing: this is only useful for the pop functions.

```

16914 \cs_new_protected:Npn \__seq_set_item:nNnnNNNN #1#2#3#4#5#6#7#8
16915 {
16916   #7 #5
16917   {
16918     \s__seq
16919     \exp_after:wN \__seq_set_item:wn
16920     \int_value:w \__seq_int_eval:w #1
16921     #5 \s__seq_stop #6
16922   }
16923   #8 { } { \prg_return_true: }
16924 }
16925 \cs_new:Npn \__seq_set_item:wn #1 \__seq_item:n #2
16926 {
16927   \if_meaning:w 0 #1 \__seq_set_item_end:w \fi:
16928   \exp_not:n { \__seq_item:n {#2} }
16929   \exp_after:wN \__seq_set_item:wn
16930   \int_value:w \__seq_int_eval:w #1 - 1 \s__seq
16931 }
16932 \cs_new:Npn \__seq_set_item_end:w #1 \exp_not:n #2 #3 \s__seq #4 \s__seq_stop #5
16933 {
16934   #1
16935   \exp_not:o \l__seq_internal_a_tl
16936   \exp_not:n {#4}
16937   #5 #2
16938 }

```

(End of definition for `\seq_set_item:NnnTF` and others. These functions are documented on page 156.)

```

\seq_reverse:N Previously, \seq_reverse:N was coded by collecting the items in reverse order after an
\seq_reverse:c \exp_stop_f: marker.
\seq_greverse:N
\seq_greverse:c \cs_new_protected:Npn \seq_reverse:N #1
__seq_reverse:NN {
__seq_reverse_item:nwn \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw
\tl_set:Nf #2 { #2 \exp_stop_f: }
}
\cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f:
{
#2 \exp_stop_f:
\@@_item:n {#1}
}

```

At first, this seems optimal, since we can forget about each item as soon as it is placed after `\exp_stop_f:`. Unfortunately, \TeX 's usual tail recursion does not take place in

this case: since the following `__seq_reverse_item:nw` only reads tokens until `\exp_stop_f:`, and never reads the `\@@_item:n {#1}` left by the previous call, `TEX` cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the `__seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. `TEX` can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```

16939 \cs_new_protected:Npn \seq_reverse:N
16940   { \__seq_reverse:NN \__kernel_tl_set:Nx }
16941 \cs_new_protected:Npn \seq_greverse:N
16942   { \__seq_reverse:NN \__kernel_tl_gset:Nx }
16943 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
16944   {
16945     \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
16946     \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
16947     #1 #2 { #2 \exp_not:n { } }
16948     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
16949   }
16950 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
16951   {
16952     #2
16953     \exp_not:n { \__seq_item:n {#1} #3 }
16954   }
16955 \cs_generate_variant:Nn \seq_reverse:N { c }
16956 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End of definition for `\seq_reverse:N` and others. These functions are documented on page 157.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

(End of definition for `\seq_sort:Nn` and `\seq_gsort:Nn`. These functions are documented on page 157.)

`\seq_gsort:Nn`

`\seq_gsort:cn`

58.4 Sequence conditionals

`\seq_if_empty_p:N`

Similar to token lists, we compare with the empty sequence.

`\seq_if_empty_p:c`

```

16957 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }

```

`\seq_if_empty:NTF`

```

16958   {

```

`\seq_if_empty:cTF`

```

16959     \if_meaning:w #1 \c_empty_seq

```

```

16960     \prg_return_true:

```

```

16961     \else:

```

```

16962     \prg_return_false:

```

```

16963     \fi:

```

```

16964   }

```

```

16965 \prg_generate_conditional_variant:Nnn \seq_if_empty:N

```

```

16966   { c } { p , T , F , TF }

```

(End of definition for `\seq_if_empty:NTF`. This function is documented on page 157.)

`\seq_shuffle:N`

We apply the Fisher–Yates shuffle, storing items in `\toks` registers. We use the primitive

`\seq_shuffle:c`

`\tex_uniformdeviate:D` for speed reasons. Its non-uniformity is of order its argument

`\seq_gshuffle:N`

divided by 2^{28} , not too bad for small lists. For sequences with more than 13 elements

`\seq_gshuffle:c`

`__seq_shuffle:NN`

`__seq_shuffle_item:n`

`\g__seq_internal_seq`

there are more possible permutations than possible seeds ($13! > 2^{28}$) so the question of uniformity is somewhat moot. The integer variables are declared in `l3int`: load-order issues.

```

16967 \seq_new:N \g__seq_internal_seq
16968 \cs_new_protected:Npn \seq_shuffle:N { \__seq_shuffle:NN \seq_set_eq:NN }
16969 \cs_new_protected:Npn \seq_gshuffle:N { \__seq_shuffle:NN \seq_gset_eq:NN }
16970 \cs_new_protected:Npn \__seq_shuffle:NN #1#2
16971 {
16972   \int_compare:nNnTF { \seq_count:N #2 } > \c_max_register_int
16973   {
16974     \msg_error:nne { seq } { shuffle-too-large }
16975     { \token_to_str:N #2 }
16976   }
16977   {
16978     \group_begin:
16979     \int_zero:N \l__seq_internal_a_int
16980     \__seq_push_item_def:
16981     \cs_gset_eq:NN \__seq_item:n \__seq_shuffle_item:n
16982     #2
16983     \__seq_pop_item_def:
16984     \seq_gclear:N \g__seq_internal_seq
16985     \int_step_inline:nn \l__seq_internal_a_int
16986     {
16987       \seq_gput_right:Ne \g__seq_internal_seq
16988       { \tex_the:D \tex_toks:D ##1 }
16989     }
16990     \group_end:
16991     #1 #2 \g__seq_internal_seq
16992     \seq_gclear:N \g__seq_internal_seq
16993   }
16994 }
16995 \cs_new_protected:Npn \__seq_shuffle_item:n
16996 {
16997   \int_incr:N \l__seq_internal_a_int
16998   \int_set:Nn \l__seq_internal_b_int
16999   { 1 + \tex_uniformdeviate:D \l__seq_internal_a_int }
17000   \tex_toks:D \l__seq_internal_a_int
17001   = \tex_toks:D \l__seq_internal_b_int
17002   \tex_toks:D \l__seq_internal_b_int
17003 }
17004 \cs_generate_variant:Nn \seq_shuffle:N { c }
17005 \cs_generate_variant:Nn \seq_gshuffle:N { c }

```

(End of definition for `\seq_shuffle:N` and others. These functions are documented on page 157.)

`\seq_if_in:NnTF` The approach here is to define `__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop breaks, returning `\prg_return_false:`. Everything is inside a group so that `__seq_item:n` is preserved in nested situations.

```

17006 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
17007 { T , F , TF }
17008 {
17009   \group_begin:

```

`\seq_if_in:NvTF`
`\seq_if_in:NvTF`
`\seq_if_in:NeTF`
`\seq_if_in:NoTF`
`\seq_if_in:NxTF`
`\seq_if_in:cnTF`
`\seq_if_in:cVTF`
`\seq_if_in:cvTF`
`\seq_if_in:ceTF`
`\seq_if_in:coTF`
`\seq_if_in:cxTF`
`__seq_if_in:`

```

17010 \tl_set:Nn \l__seq_internal_a_tl {#2}
17011 \cs_set_protected:Npn \__seq_item:n ##1
17012 {
17013   \tl_set:Nn \l__seq_internal_b_tl {##1}
17014   \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
17015     \exp_after:wN \__seq_if_in:
17016   \fi:
17017 }
17018 #1
17019 \group_end:
17020 \prg_return_false:
17021 \prg_break_point:
17022 }
17023 \cs_new:Npn \__seq_if_in:
17024 { \prg_break:n { \group_end: \prg_return_true: } }
17025 \prg_generate_conditional_variant:Nnn \seq_if_in:Nn
17026 { NV , Nv , Ne , No , Nx , c , cV , cv , ce , co , cx } { T , F , TF }

```

(End of definition for `\seq_if_in:NnTF` and `__seq_if_in:.` This function is documented on page 157.)

58.5 Recovering data from sequences

```

\__seq_pop:NNNN
\__seq_pop_TF:NNNN

```

The two pop functions share their emptiness tests. We also use a common emptiness test for all branching get and pop functions.

```

17027 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
17028 {
17029   \if_meaning:w #3 \c_empty_seq
17030     \tl_set:Nn #4 { \q_no_value }
17031   \else:
17032     #1#2#3#4
17033   \fi:
17034 }
17035 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
17036 {
17037   \if_meaning:w #3 \c_empty_seq
17038     % \tl_set:Nn #4 { \q_no_value }
17039     \prg_return_false:
17040   \else:
17041     #1#2#3#4
17042     \prg_return_true:
17043   \fi:
17044 }

```

(End of definition for `__seq_pop:NNNN` and `__seq_pop_TF:NNNN`.)

```

\seq_get_left:NN
\seq_get_left:cN
\__seq_get_left:wnw

```

Getting an item from the left of a sequence is pretty easy: just trim off the first item after `__seq_item:n` at the start. We append a `\q_no_value` item to cover the case of an empty sequence

```

17045 \cs_new_protected:Npn \seq_get_left:NN #1#2
17046 {
17047   \__kernel_tl_set:Nx #2
17048   {
17049     \exp_after:wN \__seq_get_left:wnw

```

```

17050         #1 \_seq_item:n { \q_no_value } \s__seq_stop
17051     }
17052 }
17053 \cs_new:Npn \_seq_get_left:wnw #1 \_seq_item:n #2#3 \s__seq_stop
17054 { \exp_not:n {#2} }
17055 \cs_generate_variant:Nn \seq_get_left:NN { c }

```

(End of definition for `\seq_get_left:NN` and `_seq_get_left:wnw`. This function is documented on page 153.)

```

\seq_pop_left:NN The approach to popping an item is pretty similar to that to get an item, with the only
\seq_pop_left:cN difference being that the sequence itself has to be redefined. This makes it more sensible
\seq_gpop_left:NN to use an auxiliary function for the local and global cases.
\seq_gpop_left:cN
\_seq_pop_left:NNN
\_seq_pop_left:wnwNNN
17056 \cs_new_protected:Npn \seq_pop_left:NN
17057 { \_seq_pop:NNNN \_seq_pop_left:NNN \tl_set:Nn }
17058 \cs_new_protected:Npn \seq_gpop_left:NN
17059 { \_seq_pop:NNNN \_seq_pop_left:NNN \tl_gset:Nn }
17060 \cs_new_protected:Npn \_seq_pop_left:NNN #1#2#3
17061 { \exp_after:wN \_seq_pop_left:wnwNNN #2 \s__seq_stop #1#2#3 }
17062 \cs_new_protected:Npn \_seq_pop_left:wnwNNN
17063 #1 \_seq_item:n #2#3 \s__seq_stop #4#5#6
17064 {
17065     #4 #5 { #1 #3 }
17066     \tl_set:Nn #6 {#2}
17067 }
17068 \cs_generate_variant:Nn \seq_pop_left:NN { c }
17069 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End of definition for `\seq_pop_left:NN` and others. These functions are documented on page 154.)

```

\seq_get_right:NN First remove \s__seq and prepend \q_no_value. The first argument of \_seq_get_
\seq_get_right:cN right_loop:nw is the last item found, and the second argument is empty until the end
\_seq_get_right_loop:nw of the loop, where it is code that applies \exp_not:n to the last item and ends the loop.
\_seq_get_right_end:NnN
17070 \cs_new_protected:Npn \seq_get_right:NN #1#2
17071 {
17072     \_kernel_tl_set:Nx #2
17073     {
17074         \exp_after:wN \use_i_ii:nnn
17075         \exp_after:wN \_seq_get_right_loop:nw
17076         \exp_after:wN \q_no_value
17077         #1
17078         \_seq_get_right_end:NnN \_seq_item:n
17079     }
17080 }
17081 \cs_new:Npn \_seq_get_right_loop:nw #1#2 \_seq_item:n
17082 {
17083     #2 \use_none:n {#1}
17084     \_seq_get_right_loop:nw
17085 }
17086 \cs_new:Npn \_seq_get_right_end:NnN #1#2#3 { \exp_not:n {#2} }
17087 \cs_generate_variant:Nn \seq_get_right:NN { c }

```

(End of definition for `\seq_get_right:NN`, `_seq_get_right_loop:nw`, and `_seq_get_right_end:NnN`. This function is documented on page 154.)

`\seq_pop_right:NN` The approach to popping from the right is a bit more involved, but does use some
`\seq_pop_right:cN` of the same ideas as getting from the right. What is needed is a “flexible length”
`\seq_gpop_right:NN` way to set a token list variable. This is supplied by the `{ \if_false: } \fi:`
`\seq_gpop_right:cN` `... \if_false: { \fi: }` construct. Using an e-type expansion and a “non-expanding”
`__seq_pop_right:NNN` definition for `__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items are
`__seq_pop_right_loop:nn` stored back in the sequence. That needs a loop of unknown length, hence using the
 strange `\if_false:` way of including braces. When the last item of the sequence is
 reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted
 in front of the final entry. This therefore does the pop assignment. One more iteration
 is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```

17088 \cs_new_protected:Npn \seq_pop_right:NN
17089   { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx }
17090 \cs_new_protected:Npn \seq_gpop_right:NN
17091   { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx }
17092 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
17093   {
17094     \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
17095     \cs_set_eq:NN \__seq_item:n \scan_stop:
17096     #1 #2
17097     { \if_false: } \fi: \s__seq
17098       \exp_after:wN \use_i:nnn
17099       \exp_after:wN \__seq_pop_right_loop:nn
17100       #2
17101       {
17102         \if_false: { \fi: }
17103         \__kernel_tl_set:Nx #3
17104       }
17105       { } \use_none:nn
17106     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
17107   }
17108 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
17109   {
17110     #2 { \exp_not:n {#1} }
17111     \__seq_pop_right_loop:nn
17112   }
17113 \cs_generate_variant:Nn \seq_pop_right:NN { c }
17114 \cs_generate_variant:Nn \seq_gpop_right:NN { c }
  
```

(End of definition for `\seq_pop_right:NN` and others. These functions are documented on page 154.)

`\seq_get_left:NNTF` Getting from the left or right with a check on the results. The first argument to `__seq_`
`\seq_get_left:cNTF` `pop_TF:NNNN` is left unused.
`\seq_get_right:NNTF`
`\seq_get_right:cNTF`

```

17115 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
17116   { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
17117 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
17118   { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
17119 \prg_generate_conditional_variant:Nnn \seq_get_left:NN
17120   { c } { T , F , TF }
17121 \prg_generate_conditional_variant:Nnn \seq_get_right:NN
17122   { c } { T , F , TF }
  
```

(End of definition for `\seq_get_left:NNTF` and `\seq_get_right:NNTF`. These functions are documented on page 155.)

`\seq_pop_left:NNTF` More or less the same for popping.

```

\seq_pop_left:cNTF 17123 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2
\seq_gpop_left:NNTF 17124 { T , F , TF }
\seq_gpop_left:cNTF 17125 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
\seq_pop_right:NNTF 17126 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2
\seq_pop_right:cNTF 17127 { T , F , TF }
\seq_gpop_right:NNTF 17128 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
\seq_gpop_right:cNTF 17129 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2
17130 { T , F , TF }
17131 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx #1 #2 }
17132 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2
17133 { T , F , TF }
17134 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx #1 #2 }
17135 \prg_generate_conditional_variant:Nnn \seq_pop_left:NN { c }
17136 { T , F , TF }
17137 \prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c }
17138 { T , F , TF }
17139 \prg_generate_conditional_variant:Nnn \seq_pop_right:NN { c }
17140 { T , F , TF }
17141 \prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
17142 { T , F , TF }

```

(End of definition for `\seq_pop_left:NNTF` and others. These functions are documented on page 155.)

`\seq_item:Nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the argument delimited by `__seq_item:n` is `\prg_break:` instead of being empty, terminating the loop and returning nothing at all.

```

\seq_item:NV 17143 \cs_new:Npn \seq_item:Nn #1
\seq_item:Ne 17144 { \exp_after:wN \__seq_item:wNn #1 \s__seq_stop #1 }
\seq_item:cn 17145 \cs_new:Npn \__seq_item:wNn \s__seq #1 \s__seq_stop #2#3
\seq_item:cV 17146 {
\seq_item:ce 17147   \exp_args:Nf \__seq_item:nwn
\__seq_item:wNn 17148   { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
\__seq_item:nN 17149   #1
\__seq_item:nwn 17150   \prg_break: \__seq_item:n { }
17151   \prg_break_point:
17152 }
17153 \cs_new:Npn \__seq_item:nN #1#2
17154 {
17155   \int_compare:nNnTF {#1} < 0
17156   { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
17157   {#1}
17158 }
17159 \cs_new:Npn \__seq_item:nwn #1#2 \__seq_item:n #3
17160 {
17161   #2
17162   \int_compare:nNnTF {#1} = 1
17163   { \prg_break:n { \exp_not:n {#3} } }
17164   { \exp_args:Nf \__seq_item:nwn { \int_eval:n { #1 - 1 } } }
17165 }
17166 \cs_generate_variant:Nn \seq_item:Nn { NV , Ne , c , cV , ce }

```

(End of definition for `\seq_item:Nn` and others. This function is documented on page 154.)

`\seq_rand_item:N` Importantly, `\seq_item:Nn` only evaluates its argument once.

```
\seq_rand_item:c 17167 \cs_new:Npn \seq_rand_item:N #1
17168 {
17169   \seq_if_empty:NF #1
17170   { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }
17171 }
17172 \cs_generate_variant:Nn \seq_rand_item:N { c }
```

(End of definition for `\seq_rand_item:N`. This function is documented on page 155.)

58.6 Mapping over sequences

`\seq_map_break:` To break a function, the special token `\prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```
17173 \cs_new:Npn \seq_map_break:
17174 { \prg_map_break:Nn \seq_map_break: { } }
17175 \cs_new:Npn \seq_map_break:n
17176 { \prg_map_break:Nn \seq_map_break: }
```

(End of definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 159.)

`\seq_map_function:NN` The idea here is to apply the code of #2 to each item in the sequence without altering the definition of `__seq_item:n`. The even-numbered arguments of `__seq_map_function:Nw` delimited by `__seq_item:n` are almost always empty, except at the end of the loop where it is `\prg_break:.` This allows to break the loop without needing to do a (relatively-expensive) quark test.

```
17177 \cs_new:Npn \seq_map_function:NN #1#2
17178 {
17179   \exp_after:wN \use_i_ii:nnn
17180   \exp_after:wN \__seq_map_function:Nw
17181   \exp_after:wN #2
17182   #1
17183   \prg_break:
17184   \__seq_item:n { } \__seq_item:n { } \__seq_item:n { } \__seq_item:n { }
17185   \prg_break_point:
17186   \prg_break_point:Nn \seq_map_break: { }
17187 }
17188 \cs_new:Npn \__seq_map_function:Nw #1
17189 #2 \__seq_item:n #3
17190 #4 \__seq_item:n #5
17191 #6 \__seq_item:n #7
17192 #8 \__seq_item:n #9
17193 {
17194 #2 #1 {#3}
17195 #4 #1 {#5}
17196 #6 #1 {#7}
17197 #8 #1 {#9}
17198 \__seq_map_function:Nw #1
17199 }
17200 \cs_generate_variant:Nn \seq_map_function:NN { c }
```

(End of definition for `\seq_map_function:Nn` and `__seq_map_function:Nw`. This function is documented on page 157.)

`__seq_push_item_def:n` The definition of `__seq_item:n` needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```

\__seq_push_item_def:e
\__seq_push_item_def:
\__seq_pop_item_def:
17201 \cs_new_protected:Npn \__seq_push_item_def:n
17202   {
17203     \__seq_push_item_def:
17204     \cs_gset:Npn \__seq_item:n ##1
17205   }
17206 \cs_new_protected:Npn \__seq_push_item_def:e
17207   {
17208     \__seq_push_item_def:
17209     \cs_gset:Npe \__seq_item:n ##1
17210   }
17211 \cs_new_protected:Npn \__seq_push_item_def:
17212   {
17213     \int_gincr:N \g__kernel_prg_map_int
17214     \cs_gset_eq:cN { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
17215     \__seq_item:n
17216   }
17217 \cs_new_protected:Npn \__seq_pop_item_def:
17218   {
17219     \cs_gset_eq:Nc \__seq_item:n
17220     { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
17221     \int_gdecr:N \g__kernel_prg_map_int
17222   }

```

(End of definition for `__seq_push_item_def:n`, `__seq_push_item_def:e`, and `__seq_pop_item_def:`.)

`\seq_map_inline:Nn` The idea here is that `__seq_item:n` is already “applied” to each item in a sequence, and so an in-line mapping is just a case of redefining `__seq_item:n`.

```

\seq_map_inline:cn
17223 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
17224   {
17225     \__seq_push_item_def:n {#2}
17226     #1
17227     \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
17228   }
17229 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End of definition for `\seq_map_inline:Nn`. This function is documented on page 158.)

`\seq_map_tokens:Nn` This is based on the function mapping but using the same tricks as described for `\prop_map_tokens:Nn`. The idea is to remove the leading `\s__seq` and apply the tokens such that they are safe with the break points, hence the `\use:n`.

```

\seq_map_tokens:cn
\__seq_map_tokens:nw
17230 \cs_new:Npn \seq_map_tokens:Nn #1#2
17231   {
17232     \exp_last_unbraced:Nno
17233     \use_i:nn { \__seq_map_tokens:nw {#2} } #1
17234     \prg_break:
17235     \__seq_item:n { } \__seq_item:n { } \__seq_item:n { } \__seq_item:n { }
17236     \prg_break_point:
17237     \prg_break_point:Nn \seq_map_break: { }

```

```

17238 }
17239 \cs_generate_variant:Nn \seq_map_tokens:Nn { c }
17240 \cs_new:Npn \__seq_map_tokens:nw #1
17241   #2 \__seq_item:n #3
17242   #4 \__seq_item:n #5
17243   #6 \__seq_item:n #7
17244   #8 \__seq_item:n #9
17245 {
17246   #2 \use:n {#1} {#3}
17247   #4 \use:n {#1} {#5}
17248   #6 \use:n {#1} {#7}
17249   #8 \use:n {#1} {#9}
17250   \__seq_map_tokens:nw {#1}
17251 }

```

(End of definition for `\seq_map_tokens:Nn` and `__seq_map_tokens:nw`. This function is documented on page 158.)

`\seq_map_variable:NNn` This is just a specialised version of the in-line mapping function, using an e-type expansion for the code set up so that the number of # tokens required is as expected.

```

\seq_map_variable:Ncn
\seq_map_variable:cNn
\seq_map_variable:ccn
17252 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
17253 {
17254   \__seq_push_item_def:e
17255   {
17256     \tl_set:Nn \exp_not:N #2 {##1}
17257     \exp_not:n {#3}
17258   }
17259   #1
17260   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
17261 }
17262 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
17263 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End of definition for `\seq_map_variable:NNn`. This function is documented on page 158.)

`\seq_map_indexed_function:NN` Similar to `\seq_map_function:NN` but we keep track of the item index as a ;-delimited argument of `__seq_map_indexed:Nw`.

```

\seq_map_indexed_inline:Nn
\__seq_map_indexed:nNN
\__seq_map_indexed:Nw
17264 \cs_new:Npn \seq_map_indexed_function:NN #1#2
17265 {
17266   \__seq_map_indexed:NN #1#2
17267   \prg_break_point:Nn \seq_map_break: { }
17268 }
17269 \cs_new_protected:Npn \seq_map_indexed_inline:Nn #1#2
17270 {
17271   \int_gincr:N \g__kernel_prg_map_int
17272   \cs_gset_protected:cpn
17273     { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w } ##1##2 {#2}
17274   \exp_args:Nnc \__seq_map_indexed:NN #1
17275     { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
17276   \prg_break_point:Nn \seq_map_break:
17277     { \int_gdecr:N \g__kernel_prg_map_int }
17278 }
17279 \cs_new:Npn \__seq_map_indexed:NN #1#2
17280 {

```

```

17281 \exp_after:wN \__seq_map_indexed:Nw
17282 \exp_after:wN #2
17283 \int_value:w 1
17284 \exp_after:wN \use_i:nn
17285 \exp_after:wN ;
17286 #1
17287 \prg_break: \__seq_item:n { } \prg_break_point:
17288 }
17289 \cs_new:Npn \__seq_map_indexed:Nw #1#2 ; #3 \__seq_item:n #4
17290 {
17291 #3
17292 #1 {#2} {#4}
17293 \exp_after:wN \__seq_map_indexed:Nw
17294 \exp_after:wN #1
17295 \int_value:w \int_eval:w 1 + #2 ;
17296 }

```

(End of definition for `\seq_map_indexed_function:NN` and others. These functions are documented on page 158.)

```

\seq_map_pairwise_function:NNN
\seq_map_pairwise_function:NcN
\seq_map_pairwise_function:cNN
\seq_map_pairwise_function:ccN
\__seq_map_pairwise_function:wNN
\__seq_map_pairwise_function:wNw
\__seq_map_pairwise_function:Nnnwnn

```

The idea is to first expand both sequences, adding the usual `{ ? \prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of #2 and #5, which for items in both sequences are `\s__seq __seq_item:n`. The function to be mapped are then be applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```

17297 \cs_new:Npn \seq_map_pairwise_function:NNN #1#2#3
17298 { \exp_after:wN \__seq_map_pairwise_function:wNN #2 \s__seq_stop #1 #3 }
17299 \cs_new:Npn \__seq_map_pairwise_function:wNN \s__seq #1 \s__seq_stop #2#3
17300 {
17301 \exp_after:wN \__seq_map_pairwise_function:wNw #2 \s__seq_stop #3
17302 #1 { ? \prg_break: } { }
17303 \prg_break_point:
17304 }
17305 \cs_new:Npn \__seq_map_pairwise_function:wNw \s__seq #1 \s__seq_stop #2
17306 {
17307 \__seq_map_pairwise_function:Nnnwnn #2
17308 #1 { ? \prg_break: } { }
17309 \s__seq_stop
17310 }
17311 \cs_new:Npn \__seq_map_pairwise_function:Nnnwnn #1#2#3#4 \s__seq_stop #5#6
17312 {
17313 \use_none:n #2
17314 \use_none:n #5
17315 #1 {#3} {#6}
17316 \__seq_map_pairwise_function:Nnnwnn #1 #4 \s__seq_stop
17317 }
17318 \cs_generate_variant:Nn \seq_map_pairwise_function:NNN { Nc , c , cc }

```

(End of definition for `\seq_map_pairwise_function:NNN` and others. This function is documented on page 158.)

```

\seq_set_map_e:NNN
\seq_gset_map_e:NNN
\__seq_set_map_e:NNNn

```

Very similar to `\seq_set_filter:NNn`. We could actually merge the two within a single function, but it would have weird semantics.

```

17319 \cs_new_protected:Npn \seq_set_map_e:NNn
17320 { \__seq_set_map_e:NNNn \__kernel_tl_set:Nx }
17321 \cs_new_protected:Npn \seq_gset_map_e:NNn
17322 { \__seq_set_map_e:NNNn \__kernel_tl_gset:Nx }
17323 \cs_new_protected:Npn \__seq_set_map_e:NNNn #1#2#3#4
17324 {
17325     \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
17326     #1 #2 { #3 }
17327     \__seq_pop_item_def:
17328 }

```

(End of definition for `\seq_set_map_e:NNn`, `\seq_gset_map_e:NNn`, and `__seq_set_map_e:NNNn`. These functions are documented on page 160.)

`\seq_set_map:NNn` Similar to `\seq_set_map_e:NNn`, but prevents expansion of the `<inline function>`.
`\seq_gset_map:NNn`
`__seq_set_map:NNNn`

```

17329 \cs_new_protected:Npn \seq_set_map:NNn
17330 { \__seq_set_map:NNNn \__kernel_tl_set:Nx }
17331 \cs_new_protected:Npn \seq_gset_map:NNn
17332 { \__seq_set_map:NNNn \__kernel_tl_gset:Nx }
17333 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
17334 {
17335     \__seq_push_item_def:n { \exp_not:n { \__seq_item:n {#4} } }
17336     #1 #2 { #3 }
17337     \__seq_pop_item_def:
17338 }

```

(End of definition for `\seq_set_map:NNn`, `\seq_gset_map:NNn`, and `__seq_set_map:NNNn`. These functions are documented on page 159.)

`\seq_count:N` Since counting the items in a sequence is quite common, we optimize it by grabbing
`\seq_count:c` 8 items at a time and correspondingly adding 8 to an integer expression. At the end of
`__seq_count:w` the loop, #9 is `__seq_count_end:w` instead of being empty. It removes 8+ and instead
`__seq_count_end:w` places the number of `__seq_item:n` that `__seq_count:w` grabbed before reaching the
end of the sequence.

```

17339 \cs_new:Npn \seq_count:N #1
17340 {
17341     \int_eval:n
17342     {
17343         \exp_after:wN \use_i:nn
17344         \exp_after:wN \__seq_count:w
17345         #1
17346         \__seq_count_end:w \__seq_item:n 7
17347         \__seq_count_end:w \__seq_item:n 6
17348         \__seq_count_end:w \__seq_item:n 5
17349         \__seq_count_end:w \__seq_item:n 4
17350         \__seq_count_end:w \__seq_item:n 3
17351         \__seq_count_end:w \__seq_item:n 2
17352         \__seq_count_end:w \__seq_item:n 1
17353         \__seq_count_end:w \__seq_item:n 0
17354         \prg_break_point:
17355     }
17356 }
17357 \cs_new:Npn \__seq_count:w
17358 #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4 \__seq_item:n

```

```

17359     #5 \__seq_item:n #6 \__seq_item:n #7 \__seq_item:n #8 #9 \__seq_item:n
17360     { #9 8 + \__seq_count:w }
17361 \cs_new:Npn \__seq_count_end:w 8 + \__seq_count:w #1#2 \prg_break_point: {#1}
17362 \cs_generate_variant:Nn \seq_count:N { c }

```

(End of definition for `\seq_count:N`, `__seq_count:w`, and `__seq_count_end:w`. This function is documented on page 160.)

58.7 Using sequences

```

\seq_use:Nnnn See \clist_use:Nnnn for a general explanation. The main difference is that we use
\seq_use:cnnn \__seq_item:n as a delimiter rather than commas. We also need to add \__seq_item:n
\__seq_use:NNnNnn at various places, and \s__seq.
\__seq_use_setup:w 17363 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
\__seq_use:nwwwnwn 17364 {
\__seq_use:nwwn 17365     \seq_if_exist:NTF #1
\seq_use:Nn 17366     {
\seq_use:cn 17367         \int_case:nnF { \seq_count:N #1 }
17368             {
17369                 { 0 } { }
17370                 { 1 } { \exp_after:wN \__seq_use:NNnNnn #1 ? { } { } }
17371                 { 2 } { \exp_after:wN \__seq_use:NNnNnn #1 {#2} }
17372             }
17373             {
17374                 \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
17375                 \s__seq_mark { \__seq_use:nwwwnwn {#3} }
17376                 \s__seq_mark { \__seq_use:nwwn {#4} }
17377                 \s__seq_stop { }
17378             }
17379         }
17380     {
17381         \msg_expandable_error:nnn
17382         { kernel } { bad-variable } {#1}
17383     }
17384 }
17385 \cs_generate_variant:Nn \seq_use:Nnnn { c }
17386 \cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
17387 \cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwnwn { } }
17388 \cs_new:Npn \__seq_use:nwwwnwn
17389     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
17390     \s__seq_mark #6#7 \s__seq_stop #8
17391     {
17392     #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
17393     \s__seq_mark {#6} #7 \s__seq_stop { #8 #1 #2 }
17394     }
17395 \cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \s__seq_stop #4
17396     { \exp_not:n { #4 #1 #2 } }
17397 \cs_new:Npn \seq_use:Nn #1#2
17398     { \seq_use:Nnnn #1 {#2} {#2} {#2} }
17399 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End of definition for `\seq_use:Nnnn` and others. These functions are documented on page 160.)

58.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

```

\seq_push:Nn      Pushing to a sequence is the same as adding on the left.
\seq_push:NV      17400 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\seq_push:Nv      17401 \cs_generate_variant:Nn \seq_push:Nn { NV , Nv , Ne , c , cV , cv , ce }
\seq_push:Ne      17402 \cs_generate_variant:Nn \seq_push:Nn { No , Nx , co , cx }
\seq_push:No      17403 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\seq_push:Nx      17404 \cs_generate_variant:Nn \seq_gpush:Nn { NV , Nv , Ne , c , cV , cv , ce }
\seq_push:cn      17405 \cs_generate_variant:Nn \seq_gpush:Nn { No , Nx , co , cx }
\seq_push:cV      (End of definition for \seq_push:Nn and \seq_gpush:Nn. These functions are documented on page 162.)
\seq_push:cv
\seqqppsh:NN      In most cases, getting items from the stack does not need to specify that this is from the
\seqqppsh:cN      left. So alias are provided.
\seqqppsh:NN      17406 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\seqqppop:cN      17407 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\seqqppsh:NV      17408 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\seqqppop:NN      17409 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
\seq_gpush:Ne      17410 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
\seq_gpush:No      17411 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN
\seq_gpush:Nx      (End of definition for \seq_get:NN, \seq_pop:NN, and \seq_gpop:NN. These functions are documented
\seq_gpush:cn      on page 161.)
\seq_gpush:cV
\seq_get:NNTF      More copies.
\seq_gpush:cV      17412 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_get:NNTF      17413 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:NNTF      17414 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpush:NNTF      17415 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
\seq_gpop:NNTF      17416 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
\seq_gpop:cNTF      17417 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

```

(End of definition for \seq_get:NNTF, \seq_pop:NNTF, and \seq_gpop:NNTF. These functions are documented on page 161.)

58.9 Viewing sequences

```

\seq_show:N      Apply the general \__kernel_chk_tl_type:NnnT.
\seq_show:c      17418 \cs_new_protected:Npn \seq_show:N { \__seq_show:NN \msg_show:nneeee }
\seq_log:N       17419 \cs_generate_variant:Nn \seq_show:N { c }
\seq_log:c       17420 \cs_new_protected:Npn \seq_log:N { \__seq_show:NN \msg_log:nneeee }
\__seq_show:NN   17421 \cs_generate_variant:Nn \seq_log:N { c }
\__seq_show_validate:nn 17422 \cs_new_protected:Npn \__seq_show:NN #1#2
17423 {
17424   \__kernel_chk_tl_type:NnnT #2 { seq }
17425   {
17426     \s__seq
17427     \exp_after:wN \use_i:nn \exp_after:wN \__seq_show_validate:nn #2
17428     \q_recursion_tail \q_recursion_tail \q_recursion_stop
17429   }
17430   {
17431     #1 { seq } { show }

```

```

17432         { \token_to_str:N #2 }
17433         { \seq_map_function:NN #2 \msg_show_item:n }
17434         { } { }
17435     }
17436 }
17437 \cs_new:Npn \__seq_show_validate:nn #1#2
17438 {
17439     \quark_if_recursion_tail_stop:n {#2}
17440     \__seq_wrap_item:n {#2}
17441     \__seq_show_validate:nn
17442 }

```

(End of definition for `\seq_show:N` and others. These functions are documented on page 164.)

58.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.

```

\l_tmpa_seq 17443 \seq_new:N \l_tmpa_seq
\l_tmpb_seq 17444 \seq_new:N \l_tmpb_seq
\g_tmpa_seq 17445 \seq_new:N \g_tmpa_seq
\g_tmpb_seq 17446 \seq_new:N \g_tmpb_seq

```

(End of definition for `\l_tmpa_seq` and others. These variables are documented on page 164.)

```

17447 </package>

```


Chapter 59

l3int implementation

```
17448 (*package)
```

```
17449 (@@=int)
```

The following test files are used for this code: m3int001,m3int002,m3int03.

`\c_max_register_int` Done in l3basics.

(End of definition for \c_max_register_int. This variable is documented on page 178.)

`__int_to_roman:w` Done in l3basics.

`\if_int_compare:w`

(End of definition for __int_to_roman:w and \if_int_compare:w. This function is documented on page 179.)

`\or:` Done in l3basics.

(End of definition for \or:. This function is documented on page 179.)

`\int_value:w` Here are the remaining primitives for number comparisons and expressions.

`__int_eval:w`

```
17450 \cs_new_eq:NN \int_value:w \tex_number:D
```

`__int_eval_end:`

```
17451 \cs_new_eq:NN \__int_eval:w \tex_numexpr:D
```

`\if_int_odd:w`

```
17452 \cs_new_eq:NN \__int_eval_end: \tex_relax:D
```

`\if_case:w`

```
17453 \cs_new_eq:NN \if_int_odd:w \tex_ifodd:D
```

```
17454 \cs_new_eq:NN \if_case:w \tex_ifcase:D
```

(End of definition for \int_value:w and others. These functions are documented on page 179.)

`\s__int_mark` Scan marks used throughout the module.

`\s__int_stop`

```
17455 \scan_new:N \s__int_mark
```

```
17456 \scan_new:N \s__int_stop
```

(End of definition for \s__int_mark and \s__int_stop.)

`__int_use_none_delimit_by_s_stop:w` Function to gobble until a scan mark.

```
17457 \cs_new:Npn \__int_use_none_delimit_by_s_stop:w #1 \s__int_stop { }
```

(End of definition for __int_use_none_delimit_by_s_stop:w.)

`\q__int_recursion_tail` Quarks for recursion.

`\q__int_recursion_stop`

```
17458 \quark_new:N \q__int_recursion_tail
```

```
17459 \quark_new:N \q__int_recursion_stop
```

(End of definition for `\q__int_recursion_tail` and `\q__int_recursion_stop`.)

`__int_if_recursion_tail_stop_do:Nn`
`__int_if_recursion_tail_stop:N`

Functions to query quarks.

```
17460 \__kernel_quark_new_test:N \__int_if_recursion_tail_stop_do:Nn
17461 \__kernel_quark_new_test:N \__int_if_recursion_tail_stop:N
```

(End of definition for `__int_if_recursion_tail_stop_do:Nn` and `__int_if_recursion_tail_stop:N`.)

59.1 Integer expressions

`\int_eval:n` Wrapper for `__int_eval:w`: can be used in an integer expression or directly in the input stream. It is very slightly faster to use `\the` rather than `\number` to turn the expression to a number. When debugging, we introduce parentheses to catch early termination (see `l3debug`).

```
17462 \cs_new:Npn \int_eval:n #1
17463 { \tex_the:D \__int_eval:w #1 \__int_eval_end: }
17464 \cs_new:Npn \int_eval:w { \tex_the:D \__int_eval:w }
```

(End of definition for `\int_eval:n` and `\int_eval:w`. These functions are documented on page 167.)

`\int_sign:n` See `\int_abs:n`. Evaluate the expression once (and when debugging is enabled, check that the expression is well-formed), then test the first character to determine the sign. This is wrapped in `\int_value:w ... \exp_stop_f:` to ensure a fixed number of expansions and to avoid dealing with closing the conditionals.

`__int_sign:Nw`

```
17465 \cs_new:Npn \int_sign:n #1
17466 {
17467   \int_value:w \exp_after:wN \__int_sign:Nw
17468   \int_value:w \__int_eval:w #1 \__int_eval_end: ;
17469   \exp_stop_f:
17470 }
17471 \cs_new:Npn \__int_sign:Nw #1#2 ;
17472 {
17473   \if_meaning:w 0 #1
17474   0
17475   \else:
17476   \if_meaning:w - #1 - \fi: 1
17477   \fi:
17478 }
```

(End of definition for `\int_sign:n` and `__int_sign:Nw`. This function is documented on page 168.)

`\int_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

`__int_abs:N`

`\int_max:nn`
`\int_min:nn`

`__int_maxmin:wwN`

```
17479 \cs_new:Npn \int_abs:n #1
17480 {
17481   \int_value:w \exp_after:wN \__int_abs:N
17482   \int_value:w \__int_eval:w #1 \__int_eval_end:
17483   \exp_stop_f:
17484 }
17485 \cs_new:Npn \__int_abs:N #1
17486 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
17487 \cs_new:Npn \int_max:nn #1#2
17488 {
```

```

17489     \int_value:w \exp_after:wN \__int_maxmin:wwN
17490     \int_value:w \__int_eval:w #1 \exp_after:wN ;
17491     \int_value:w \__int_eval:w #2 ;
17492     >
17493     \exp_stop_f:
17494   }
17495 \cs_new:Npn \int_min:nn #1#2
17496 {
17497     \int_value:w \exp_after:wN \__int_maxmin:wwN
17498     \int_value:w \__int_eval:w #1 \exp_after:wN ;
17499     \int_value:w \__int_eval:w #2 ;
17500     <
17501     \exp_stop_f:
17502   }
17503 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
17504 {
17505     \if_int_compare:w #1 #3 #2 ~
17506     #1
17507     \else:
17508     #2
17509     \fi:
17510 }

```

(End of definition for `\int_abs:n` and others. These functions are documented on page 168.)

`\int_div_truncate:nn` As `__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (e.g., `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(|#3#4| - 1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε -TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```

17511 \cs_new:Npn \int_div_truncate:nn #1#2
17512 {
17513     \int_value:w \__int_eval:w
17514     \exp_after:wN \__int_div_truncate:NwNw
17515     \int_value:w \__int_eval:w #1 \exp_after:wN ;
17516     \int_value:w \__int_eval:w #2 ;
17517     \__int_eval_end:
17518   }
17519 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
17520 {
17521     \if_meaning:w 0 #1
17522     0
17523     \else:
17524     (
17525     #1#2
17526     \if_meaning:w - #1 + \else: - \fi:
17527     ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
17528     )
17529     \fi:
17530     / #3#4

```

```
17531 }
```

For the sake of completeness:

```
17532 \cs_new:Npn \int_div_round:nn #1#2
17533 { \int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }
```

Finally there's the modulus operation.

```
17534 \cs_new:Npn \int_mod:nn #1#2
17535 {
17536   \int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
17537   \int_value:w \__int_eval:w #1 \exp_after:wN ;
17538   \int_value:w \__int_eval:w #2 ;
17539   \__int_eval_end:
17540 }
17541 \cs_new:Npn \__int_mod:ww #1; #2;
17542 { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(End of definition for `\int_div_truncate:nn` and others. These functions are documented on page 168.)

`__kernel_int_add:nnn` Equivalent to `\int_eval:n {#1+#2+#3}` except that overflow only occurs if the final result overflows $[-2^{31} + 1, 2^{31} - 1]$. The idea is to choose the order in which the three numbers are added together. If #1 and #2 have opposite signs (one is in $[-2^{31} + 1, -1]$ and the other in $[0, 2^{31} - 1]$) then #1+#2 cannot overflow so we compute the result as #1+#2+#3. If they have the same sign, then either #3 has the same sign and the order does not matter, or #3 has the opposite sign and any order in which #3 is not last will work. We use #1+#3+#2.

```
17543 \cs_new:Npn \__kernel_int_add:nnn #1#2#3
17544 {
17545   \int_value:w \__int_eval:w #1
17546   \if_int_compare:w #2 < \c_zero_int \exp_after:wN \reverse_if:N \fi:
17547   \if_int_compare:w #1 < \c_zero_int + #2 + #3 \else: + #3 + #2 \fi:
17548   \__int_eval_end:
17549 }
```

(End of definition for `__kernel_int_add:nnn`.)

59.2 Creating and initialising integers

`\int_new:N` Two ways to do this: one for the format and one for the L^AT_EX 2_ε package. In plain T_EX, `\int_new:c` `\newcount` (and other allocators) are `\outer:` to allow the code here to work in “generic” mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```
17550 \cs_new_protected:Npn \int_new:N #1
17551 {
17552   \__kernel_chk_if_free_cs:N #1
17553   \cs:w newcount \cs_end: #1
17554 }
17555 \cs_generate_variant:Nn \int_new:N { c }
```

(End of definition for `\int_new:N`. This function is documented on page 168.)

`\int_const:Nn` As stated, most constants can be defined as `\chardef` or `\mathchardef` but that's engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward. We cannot use `\int_gset:Nn` because (when `check-declarations` is enabled) this runs some checks that constants would fail.

```

\c__int_max_constdef_int
17556 \cs_new_protected:Npn \int_const:Nn #1#2
17557   { \__int_const:eN { \int_eval:n {#2} } #1 }
17558 \cs_generate_variant:Nn \int_const:Nn { c }
17559 \cs_new_protected:Npn \__int_const:nN #1#2
17560   {
17561     \int_compare:nNnTF {#1} < \c_zero_int
17562     {
17563       \int_new:N #2
17564       \tex_global:D
17565     }
17566     {
17567       \int_compare:nNnTF {#1} > \c__int_max_constdef_int
17568       {
17569         \int_new:N #2
17570         \tex_global:D
17571       }
17572       {
17573         \__kernel_chk_if_free_cs:N #2
17574         \tex_global:D \__int_constdef:Nw
17575       }
17576     }
17577     #2 = \__int_eval:w #1 \__int_eval_end:
17578   }
17579 \cs_generate_variant:Nn \__int_const:nN { e }
17580 \if_int_odd:w 0
17581   \cs_if_exist:NT \tex_luatexversion:D { 1 }
17582   \cs_if_exist:NT \tex_omathchardef:D { 1 }
17583   \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
17584   \cs_if_exist:NTF \tex_omathchardef:D
17585     { \cs_new_eq:NN \__int_constdef:Nw \tex_omathchardef:D }
17586     { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
17587   \tex_global:D \__int_constdef:Nw \c__int_max_constdef_int 1114111 ~
17588 \else:
17589   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
17590   \tex_global:D \__int_constdef:Nw \c__int_max_constdef_int 32767 ~
17591 \fi:

```

(End of definition for `\int_const:Nn` and others. This function is documented on page 168.)

`\int_zero:N` Functions that reset an *integer* register to zero.

```

\int_zero:c 17592 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero_int }
\int_gzero:N 17593 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero_int }
\int_zero:c 17594 \cs_generate_variant:Nn \int_zero:N { c }
\int_gzero:c 17595 \cs_generate_variant:Nn \int_gzero:N { c }

```

(End of definition for `\int_zero:N` and `\int_gzero:N`. These functions are documented on page 169.)

`\int_zero_new:N` Create a register if needed, otherwise clear it.

```

\int_zero_new:c 17596 \cs_new_protected:Npn \int_zero_new:N #1
\int_gzero_new:N
\int_gzero_new:c

```

```

17597 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
17598 \cs_new_protected:Npn \int_gzero_new:N #1
17599 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
17600 \cs_generate_variant:Nn \int_zero_new:N { c }
17601 \cs_generate_variant:Nn \int_gzero_new:N { c }

```

(End of definition for `\int_zero_new:N` and `\int_gzero_new:N`. These functions are documented on page 169.)

`\int_set_eq:NN` Setting equal means using one integer inside the set function of another. Check that assigned integer is local/global. No need to check that the other one is defined as `TeX` does it for us.

```

17602 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
17603 \cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }
17604 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
17605 \cs_generate_variant:Nn \int_gset_eq:NN { c , Nc , cc }

```

(End of definition for `\int_set_eq:NN` and `\int_gset_eq:NN`. These functions are documented on page 169.)

`\int_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

17606 \prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N
17607 { TF , T , F , p }
17608 \prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c
17609 { TF , T , F , p }

```

(End of definition for `\int_if_exist:NTF`. This function is documented on page 169.)

59.3 Setting and incrementing integers

`\int_add:Nn` Adding and subtracting to and from a counter. Including here the optional `by` would slow down these operations by a few percent.

```

17610 \cs_new_protected:Npn \int_add:Nn #1#2
17611 { \tex_advance:D #1 \__int_eval:w #2 \__int_eval_end: }
17612 \cs_new_protected:Npn \int_sub:Nn #1#2
17613 { \tex_advance:D #1 - \__int_eval:w #2 \__int_eval_end: }
17614 \cs_new_protected:Npn \int_gadd:Nn #1#2
17615 { \tex_global:D \tex_advance:D #1 \__int_eval:w #2 \__int_eval_end: }
17616 \cs_new_protected:Npn \int_gsub:Nn #1#2
17617 { \tex_global:D \tex_advance:D #1 - \__int_eval:w #2 \__int_eval_end: }
17618 \cs_generate_variant:Nn \int_add:Nn { c }
17619 \cs_generate_variant:Nn \int_gadd:Nn { c }
17620 \cs_generate_variant:Nn \int_sub:Nn { c }
17621 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End of definition for `\int_add:Nn` and others. These functions are documented on page 169.)

`\int_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

```

17622 \cs_new_protected:Npn \int_incr:N #1
17623 { \tex_advance:D #1 \c_one_int }
17624 \cs_new_protected:Npn \int_decr:N #1
17625 { \tex_advance:D #1 - \c_one_int }
17626 \cs_new_protected:Npn \int_gincr:N #1
17627 { \tex_global:D \tex_advance:D #1 \c_one_int }

```

```

17628 \cs_new_protected:Npn \int_gdecr:N #1
17629   { \tex_global:D \tex_advance:D #1 - \c_one_int }
17630 \cs_generate_variant:Nn \int_incr:N { c }
17631 \cs_generate_variant:Nn \int_decr:N { c }
17632 \cs_generate_variant:Nn \int_gincr:N { c }
17633 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End of definition for `\int_incr:N` and others. These functions are documented on page 169.)

`\int_set:Nn` As integers are register-based T_EX issues an error if they are not defined. While the = sign is optional, this version with = is slightly quicker than without, while adding the optional space after = slows things down minutely.

```

\int_set:cn
\int_gset:Nn
\int_gset:cn
17634 \cs_new_protected:Npn \int_set:Nn #1#2
17635   { #1 = \__int_eval:w #2 \__int_eval_end: }
17636 \cs_new_protected:Npn \int_gset:Nn #1#2
17637   { \tex_global:D #1 = \__int_eval:w #2 \__int_eval_end: }
17638 \cs_generate_variant:Nn \int_set:Nn { c }
17639 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End of definition for `\int_set:Nn` and `\int_gset:Nn`. These functions are documented on page 169.)

59.4 Using integers

`\int_use:N` Here is how counters are accessed. We hand-code the c variant for some speed gain.

```

\int_use:c
17640 \cs_new_eq:NN \int_use:N \tex_the:D
17641 \cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End of definition for `\int_use:N`. This function is documented on page 170.)

59.5 Integer expression conditionals

`__int_compare_error:` Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as != and >= are supported. `__int_compare_error:Nw` The tests first evaluate their left-hand side, with a trailing `__int_compare_error:.` This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts = (and itself) after triggering the relevant T_EX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `__int_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```

17642 \cs_new_protected:Npn \__int_compare_error:
17643   {
17644     \if_int_compare:w \c_zero_int \c_zero_int \fi:
17645     =
17646     \__int_compare_error:
17647   }
17648 \cs_new:Npn \__int_compare_error:Nw
17649   #1#2 \s__int_stop
17650   {
17651     { }
17652     \c_zero_int \fi:
17653     \msg_expandable_error:nnn
17654     { kernel } { unknown-comparison } {#1}

```

```

17655     \prg_return_false:
17656   }

```

(End of definition for `__int_compare_error:` and `__int_compare_error:Nw`.)

```

\int_compare_p:n Comparison tests using a simple syntax where only one set of braces is required, additional
\int_compare:nTF operators such as != and >= are supported, and multiple comparisons can be performed
  \__int_compare:w at once, for instance 0 < 5 <= 1. The idea is to loop through the argument, finding one
  \__int_compare:Nw operand at a time, and comparing it to the previous one. The looping auxiliary \__int_
  \__int_compare:NNw compare:Nw reads one <operand> and one <comparison> symbol, and leaves roughly
  \__int_compare:nnN <operand> \prg_return_false: \fi:
\__int_compare_end_=:NNw \reverse_if:N \if_int_compare:w <operand> <comparison>
  \__int_compare_=:NNw \__int_compare:Nw
  \__int_compare_<:NNw
  \__int_compare_>:NNw
  \__int_compare_=:NNw
  \__int_compare_!=:NNw
  \__int_compare_<=:NNw
  \__int_compare_>=:NNw

```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the *comparisons* is false, the true branch of the T_EX conditional is taken (because of `\reverse_if:N`), immediately returning false as the result of the test. There is no T_EX conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `\int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let T_EX evaluate this left hand side of the (in)equality using `__int_eval:w`. Since the relation symbols `<`, `>`, `=` and `!` are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, `__int_compare_error:` is expanded, inserting `=` and itself after an error. In all cases, `__int_compare:w` receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\s__int_stop` used to grab the entire argument when necessary.

```

17657 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
17658   {
17659     \exp_after:wN \__int_compare:w
17660     \int_value:w \__int_eval:w #1 \__int_compare_error:
17661   }
17662 \cs_new:Npn \__int_compare:w #1 \__int_compare_error:
17663   {
17664     \exp_after:wN \if_false: \int_value:w
17665     \__int_compare:Nw #1 e { = nd_ } \s__int_stop
17666   }

```

The goal here is to find an *operand* and a *comparison*. The *operand* is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `__int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `__int_compare:NNw` to determine the relation symbol, building a control sequence from it (`\token_to_str:N` gives better errors if #1 is not a character). All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by T_EX into `\scan_stop:`, ignored thanks to `\unexpanded`, and `__int_compare_error:Nw` raises an error.

```

17667 \cs_new:Npn \__int_compare:Nw #1#2 \s__int_stop
17668   {

```



```

17669 \exp_after:wN \__int_compare:NNw
17670 \__int_to_roman:w - 0 #2 \s__int_mark
17671 #1#2 \s__int_stop
17672 }
17673 \cs_new:Npn \__int_compare:NNw #1#2#3 \s__int_mark
17674 {
17675 \__kernel_exp_not:w
17676 \use:c
17677 {
17678 \__int_compare_ \token_to_str:N #1
17679 \if_meaning:w = #2 = \fi:
17680 :NNw
17681 }
17682 \__int_compare_error:Nw #1
17683 }

```

When the last *operand* is seen, `__int_compare:NNw` receives `e` and `=nd_` as arguments, hence calling `__int_compare_end_=:NNw` to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls `__int_compare:nnN` where `#1` is `\if_int_compare:w` or `\reverse_if:N \if_int_compare:w`, `#2` is the *operand*, and `#3` is one of `<`, `=`, or `>`. As announced earlier, we leave the *operand* for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return `false`. Otherwise, we apply the conditional `#1` to the *operand* `#2` and the comparison `#3`, and call `__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```

17684 \cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \s__int_stop
17685 {
17686 {#3} \exp_stop_f:
17687 \prg_return_false: \else: \prg_return_true: \fi:
17688 }
17689 \cs_new:Npn \__int_compare:nnN #1#2#3
17690 {
17691 {#2} \exp_stop_f:
17692 \prg_return_false: \exp_after:wN \__int_use_none_delimit_by_s_stop:w
17693 \fi:
17694 #1 #2 #3 \exp_after:wN \__int_compare:Nw \int_value:w \__int_eval:w
17695 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `__int_compare_error:Nw` (*token*) responsible for error detection.

```

17696 \cs_new:cpn { __int_compare_=:NNw } #1#2#3 =
17697 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
17698 \cs_new:cpn { __int_compare_<:NNw } #1#2#3 <
17699 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
17700 \cs_new:cpn { __int_compare_>:NNw } #1#2#3 >
17701 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
17702 \cs_new:cpn { __int_compare_=:NNw } #1#2#3 ==
17703 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
17704 \cs_new:cpn { __int_compare_!:=:NNw } #1#2#3 !=
17705 { \__int_compare:nnN { \if_int_compare:w } {#3} = }
17706 \cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
17707 { \__int_compare:nnN { \if_int_compare:w } {#3} > }

```

```

17708 \cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
17709 { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End of definition for \int_compare:nTF and others. This function is documented on page 171.)

\int_compare_p:nNn More efficient but less natural in typing.

```

\int_compare:nNnTF
17710 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
17711 {
17712   \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
17713   \prg_return_true:
17714   \else:
17715   \prg_return_false:
17716   \fi:
17717 }

```

(End of definition for \int_compare:nNnTF. This function is documented on page 170.)

\int_if_zero_p:n

```

\int_if_zero:nTF
17718 \prg_new_conditional:Npnn \int_if_zero:n #1 { p , T , F , TF }
17719 {
17720   \if_int_compare:w \__int_eval:w #1 = \c_zero_int
17721   \prg_return_true:
17722   \else:
17723   \prg_return_false:
17724   \fi:
17725 }

```

(End of definition for \int_if_zero:nTF. This function is documented on page 172.)

\int_case:nn

\int_case:nnTF

For integer cases, the first task to fully expand the check condition. The over all idea is then much the same as for \str_case:nnTF as described in l3str.

```

\__int_case:nnTF
\__int_case:nw
\__int_case_end:nw
17726 \cs_new:Npn \int_case:nnTF #1
17727 {
17728   \exp:w
17729   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
17730 }
17731 \cs_new:Npn \int_case:nnT #1#2#3
17732 {
17733   \exp:w
17734   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
17735 }
17736 \cs_new:Npn \int_case:nnF #1#2
17737 {
17738   \exp:w
17739   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
17740 }
17741 \cs_new:Npn \int_case:nn #1#2
17742 {
17743   \exp:w
17744   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
17745 }
17746 \cs_new:Npn \__int_case:nnTF #1#2#3#4
17747 { \__int_case:nw {#1} #2 {#1} { } \s__int_mark {#3} \s__int_mark {#4} \s__int_stop }
17748 \cs_new:Npn \__int_case:nw #1#2#3
17749 {

```

```

17750 \int_compare:nNnTF {#1} = {#2}
17751   { \__int_case_end:nw {#3} }
17752   { \__int_case:nw {#1} }
17753 }
17754 \cs_new:Npn \__int_case_end:nw #1#2#3 \s__int_mark #4#5 \s__int_stop
17755 { \exp_end: #1 #4 }

```

(End of definition for `\int_case:nTF` and others. This function is documented on page 172.)

`\int_if_odd_p:n` A predicate function.

```

\int_if_odd:nTF
\int_if_even_p:n
\int_if_even:nTF
17756 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
17757 {
17758   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
17759   \prg_return_true:
17760   \else:
17761   \prg_return_false:
17762   \fi:
17763 }
17764 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
17765 {
17766   \reverse_if:N \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
17767   \prg_return_true:
17768   \else:
17769   \prg_return_false:
17770   \fi:
17771 }

```

(End of definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 172.)

59.6 Integer expression loops

`\int_while_do:nn` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```

\int_until_do:nn
\int_do_while:nn
\int_do_until:nn
17772 \cs_new:Npn \int_while_do:nn #1#2
17773 {
17774   \int_compare:nT {#1}
17775   {
17776     #2
17777     \int_while_do:nn {#1} {#2}
17778   }
17779 }
17780 \cs_new:Npn \int_until_do:nn #1#2
17781 {
17782   \int_compare:nF {#1}
17783   {
17784     #2
17785     \int_until_do:nn {#1} {#2}
17786   }
17787 }
17788 \cs_new:Npn \int_do_while:nn #1#2
17789 {
17790   #2

```

```

17791     \int_compare:nT {#1}
17792         { \int_do_while:nn {#1} {#2} }
17793     }
17794 \cs_new:Npn \int_do_until:nn #1#2
17795     {
17796     #2
17797     \int_compare:nF {#1}
17798         { \int_do_until:nn {#1} {#2} }
17799     }

```

(End of definition for `\int_while_do:nn` and others. These functions are documented on page 173.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

```

\int_until_do:nNnn 17800 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
\int_do_while:nNnn 17801     {
\int_do_until:nNnn 17802     \int_compare:nNnT {#1} #2 {#3}
17803         {
17804         #4
17805         \int_while_do:nNnn {#1} #2 {#3} {#4}
17806     }
17807     }
17808 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
17809     {
17810     \int_compare:nNnF {#1} #2 {#3}
17811         {
17812         #4
17813         \int_until_do:nNnn {#1} #2 {#3} {#4}
17814     }
17815     }
17816 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
17817     {
17818     #4
17819     \int_compare:nNnT {#1} #2 {#3}
17820         { \int_do_while:nNnn {#1} #2 {#3} {#4} }
17821     }
17822 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
17823     {
17824     #4
17825     \int_compare:nNnF {#1} #2 {#3}
17826         { \int_do_until:nNnn {#1} #2 {#3} {#4} }
17827     }

```

(End of definition for `\int_while_do:nNnn` and others. These functions are documented on page 173.)

59.7 Integer step functions

`\int_step_function:nnnN` Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

17828 \cs_new:Npn \int_step_function:nnnN #1#2#3
17829     {

```

```

17830     \exp_after:wN \_int_step:wwwN
17831     \int_value:w \_int_eval:w #1 \exp_after:wN ;
17832     \int_value:w \_int_eval:w #2 \exp_after:wN ;
17833     \int_value:w \_int_eval:w #3 ;
17834 }
17835 \cs_new:Npn \_int_step:wwwN #1; #2; #3; #4
17836 {
17837     \int_compare:nNnTF {#2} > \c_zero_int
17838     { \_int_step:NwnnN > }
17839     {
17840         \int_compare:nNnTF {#2} = \c_zero_int
17841         {
17842             \msg_expandable_error:nnn
17843             { kernel } { zero-step } {#4}
17844             \prg_break:
17845         }
17846         { \_int_step:NwnnN < }
17847     }
17848     #1 ; {#2} {#3} #4
17849     \prg_break_point:
17850 }
17851 \cs_new:Npn \_int_step:NwnnN #1#2 ; #3#4#5
17852 {
17853     \if_int_compare:w #2 #1 #4 \exp_stop_f:
17854     \prg_break:n
17855     \fi:
17856     #5 {#2}
17857     \exp_after:wN \_int_step:NwnnN
17858     \exp_after:wN #1
17859     \int_value:w \_int_eval:w #2 + #3 ; {#3} {#4} #5
17860 }
17861 \cs_new:Npn \int_step_function:nN
17862 { \int_step_function:nnnN { 1 } { 1 } }
17863 \cs_new:Npn \int_step_function:nnN #1
17864 { \int_step_function:nnnN {#1} { 1 } }

```

(End of definition for `\int_step_function:nnnN` and others. These functions are documented on page 174.)

```

\int_step_inline:nn
\int_step_inline:nnn
\int_step_inline:nnnn
\int_step_variable:nNn
\int_step_variable:nnNn
\int_step_variable:nnnNn
\_int_step:NNnnnn

```

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

17865 \cs_new_protected:Npn \int_step_inline:nn
17866 { \int_step_inline:nnnn { 1 } { 1 } }
17867 \cs_new_protected:Npn \int_step_inline:nnn #1
17868 { \int_step_inline:nnnn {#1} { 1 } }
17869 \cs_new_protected:Npn \int_step_inline:nnnn
17870 {
17871     \int_gincr:N \g__kernel_prg_map_int
17872     \exp_args:NNc \_int_step:NNnnnn
17873     \cs_gset_protected:Npn

```

```

17874     { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
17875   }
17876 \cs_new_protected:Npn \int_step_variable:nNn
17877   { \int_step_variable:nnnNn { 1 } { 1 } }
17878 \cs_new_protected:Npn \int_step_variable:nnNn #1
17879   { \int_step_variable:nnnNn {#1} { 1 } }
17880 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
17881   {
17882     \int_gincr:N \g__kernel_prg_map_int
17883     \exp_args:NNc \__int_step:NNnnnn
17884     \cs_gset_protected:Npe
17885     { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
17886     {#1}{#2}{#3}
17887     {
17888       \tl_set:Nn \exp_not:N #4 {##1}
17889       \exp_not:n {#5}
17890     }
17891   }
17892 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
17893   {
17894     #1 #2 ##1 {#6}
17895     \int_step_function:nnnN {#3} {#4} {#5} #2
17896     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
17897   }

```

(End of definition for `\int_step_inline:nn` and others. These functions are documented on page 174.)

59.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```

\int_to_arabic:v
17898 \cs_new_eq:NN \int_to_arabic:n \int_eval:n
17899 \cs_generate_variant:Nn \int_to_arabic:n { v }

```

(End of definition for `\int_to_arabic:n`. This function is documented on page 175.)

`\int_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an f-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```

\__int_to_symbols:nnnn
\__int_to_symbols:ennn
17900 \cs_new:Npn \int_to_symbols:nnn #1#2#3
17901   {
17902     \int_compare:nNnTF {#1} > {#2}
17903     {
17904       \__int_to_symbols:ennn
17905       {
17906         \int_case:nn
17907         { 1 + \int_mod:nn { #1 - 1 } {#2} }
17908         {#3}
17909       }
17910     }

```

```

17911     }
17912     { \int_case:nn {#1} {#3} }
17913   }
17914 \cs_new:Npn \__int_to_symbols:nmmm #1#2#3#4
17915   {
17916     \exp_args:Nf \int_to_symbols:nnn
17917     { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
17918     #1
17919   }
17920 \cs_generate_variant:Nn \__int_to_symbols:nmmm { e }

```

(End of definition for `\int_to_symbols:nnn` and `__int_to_symbols:nmmm`. This function is documented on page 175.)

`\int_to_alpha:n` These both use the above function with input functions that make sense for the alphabet in English.

`\int_to_Alpha:n`

```

17921 \cs_new:Npn \int_to_alpha:n #1
17922   {
17923     \int_to_symbols:nnn {#1} { 26 }
17924     {
17925       { 1 } { a }
17926       { 2 } { b }
17927       { 3 } { c }
17928       { 4 } { d }
17929       { 5 } { e }
17930       { 6 } { f }
17931       { 7 } { g }
17932       { 8 } { h }
17933       { 9 } { i }
17934       { 10 } { j }
17935       { 11 } { k }
17936       { 12 } { l }
17937       { 13 } { m }
17938       { 14 } { n }
17939       { 15 } { o }
17940       { 16 } { p }
17941       { 17 } { q }
17942       { 18 } { r }
17943       { 19 } { s }
17944       { 20 } { t }
17945       { 21 } { u }
17946       { 22 } { v }
17947       { 23 } { w }
17948       { 24 } { x }
17949       { 25 } { y }
17950       { 26 } { z }
17951     }
17952   }
17953 \cs_new:Npn \int_to_Alpha:n #1
17954   {
17955     \int_to_symbols:nnn {#1} { 26 }
17956     {
17957       { 1 } { A }
17958       { 2 } { B }

```

```

17959     { 3 } { C }
17960     { 4 } { D }
17961     { 5 } { E }
17962     { 6 } { F }
17963     { 7 } { G }
17964     { 8 } { H }
17965     { 9 } { I }
17966     { 10 } { J }
17967     { 11 } { K }
17968     { 12 } { L }
17969     { 13 } { M }
17970     { 14 } { N }
17971     { 15 } { O }
17972     { 16 } { P }
17973     { 17 } { Q }
17974     { 18 } { R }
17975     { 19 } { S }
17976     { 20 } { T }
17977     { 21 } { U }
17978     { 22 } { V }
17979     { 23 } { W }
17980     { 24 } { X }
17981     { 25 } { Y }
17982     { 26 } { Z }
17983   }
17984 }

```

(End of definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 175.)

```

\int_to_base:nn Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is
\int_to_Base:nn a complicated calculation, we shouldn't perform it twice. Then check the sign, store it,
  \__int_to_base:nn either - or \c_empty_tl, and feed the absolute value to the next auxiliary function.
  \__int_to_Base:nn
  \__int_to_base:nnN 17985 \cs_new:Npn \int_to_base:nn #1
  \__int_to_Base:nnN 17986 { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
  \__int_to_base:nnN 17987 \cs_new:Npn \int_to_Base:nn #1
  \__int_to_Base:nnN 17988 { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
  \__int_to_Base:nnN 17989 \cs_new:Npn \__int_to_base:nn #1#2
  \__int_to_letter:n 17990 {
  \__int_to_Letter:n 17991   \int_compare:nNnTF {#1} < 0
  17992     { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
  17993     { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
  17994   }
  17995 \cs_new:Npn \__int_to_Base:nn #1#2
  17996 {
  17997   \int_compare:nNnTF {#1} < 0
  17998     { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
  17999     { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
  18000 }

```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in #1 is checked to see if it is less than the new base (#2). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new

base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```

18001 \cs_new:Npn \__int_to_base:nnN #1#2#3
18002   {
18003     \int_compare:nNnTF {#1} < {#2}
18004       { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
18005       {
18006         \exp_args:Nf \__int_to_base:nnnN
18007           { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
18008           {#1}
18009           {#2}
18010           #3
18011       }
18012   }
18013 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
18014   {
18015     \exp_args:Nf \__int_to_base:nnN
18016       { \int_div_truncate:nn {#2} {#3} }
18017     {#3}
18018     #4
18019     #1
18020   }
18021 \cs_new:Npn \__int_to_Base:nnN #1#2#3
18022   {
18023     \int_compare:nNnTF {#1} < {#2}
18024       { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
18025       {
18026         \exp_args:Nf \__int_to_Base:nnnN
18027           { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
18028           {#1}
18029           {#2}
18030           #3
18031       }
18032   }
18033 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
18034   {
18035     \exp_args:Nf \__int_to_Base:nnN
18036       { \int_div_truncate:nn {#2} {#3} }
18037     {#3}
18038     #4
18039     #1
18040   }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```

18041 \cs_new:Npn \__int_to_letter:n #1
18042   {
18043     \exp_after:wN \exp_after:wN
18044     \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
18045     a

```

```

18046     \or: b
18047     \or: c
18048     \or: d
18049     \or: e
18050     \or: f
18051     \or: g
18052     \or: h
18053     \or: i
18054     \or: j
18055     \or: k
18056     \or: l
18057     \or: m
18058     \or: n
18059     \or: o
18060     \or: p
18061     \or: q
18062     \or: r
18063     \or: s
18064     \or: t
18065     \or: u
18066     \or: v
18067     \or: w
18068     \or: x
18069     \or: y
18070     \or: z
18071     \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
18072     \fi:
18073   }
18074 \cs_new:Npn \__int_to_Letter:n #1
18075 {
18076   \exp_after:wN \exp_after:wN
18077   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
18078     A
18079     \or: B
18080     \or: C
18081     \or: D
18082     \or: E
18083     \or: F
18084     \or: G
18085     \or: H
18086     \or: I
18087     \or: J
18088     \or: K
18089     \or: L
18090     \or: M
18091     \or: N
18092     \or: O
18093     \or: P
18094     \or: Q
18095     \or: R
18096     \or: S
18097     \or: T
18098     \or: U
18099     \or: V

```

```

18100     \or: W
18101     \or: X
18102     \or: Y
18103     \or: Z
18104     \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
18105     \fi:
18106   }

```

(End of definition for `\int_to_base:nn` and others. These functions are documented on page 176.)

`\int_to_bin:n` Wrappers around the generic function.

```

\int_to_hex:n 18107 \cs_new:Npn \int_to_bin:n #1
\int_to_Hex:n 18108   { \int_to_base:nn {#1} { 2 } }
\int_to_oct:n 18109 \cs_new:Npn \int_to_hex:n #1
18110   { \int_to_base:nn {#1} { 16 } }
18111 \cs_new:Npn \int_to_Hex:n #1
18112   { \int_to_Base:nn {#1} { 16 } }
18113 \cs_new:Npn \int_to_oct:n #1
18114   { \int_to_base:nn {#1} { 8 } }

```

(End of definition for `\int_to_bin:n` and others. These functions are documented on page 176.)

`\int_to_roman:n` The `__int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop is terminated by the conversion of the Q.

```

\__int_to_roman:N 18115 \cs_new:Npn \int_to_roman:n #1
\__int_to_roman:N 18116   {
\__int_to_roman_i:w 18117     \exp_after:wN \__int_to_roman:N
\__int_to_roman_v:w 18118     \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_roman_x:w 18119   }
\__int_to_roman_l:w 18120 \cs_new:Npn \__int_to_roman:N #1
\__int_to_roman_c:w 18121   {
\__int_to_roman_d:w 18122     \use:c { \__int_to_roman_ #1 :w }
\__int_to_roman_m:w 18123     \__int_to_roman:N
\__int_to_roman_Q:w 18124   }
\__int_to_Roman_i:w 18125 \cs_new:Npn \int_to_Roman:n #1
\__int_to_Roman_v:w 18126   {
\__int_to_Roman_x:w 18127     \exp_after:wN \__int_to_Roman_aux:N
\__int_to_Roman_l:w 18128     \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_Roman_c:w 18129   }
\__int_to_Roman_d:w 18130 \cs_new:Npn \__int_to_Roman_aux:N #1
\__int_to_Roman_m:w 18131   {
\__int_to_Roman_Q:w 18132     \use:c { \__int_to_Roman_ #1 :w }
18133     \__int_to_Roman_aux:N
18134   }
18135 \cs_new:Npn \__int_to_roman_i:w { i }
18136 \cs_new:Npn \__int_to_roman_v:w { v }
18137 \cs_new:Npn \__int_to_roman_x:w { x }
18138 \cs_new:Npn \__int_to_roman_l:w { l }
18139 \cs_new:Npn \__int_to_roman_c:w { c }
18140 \cs_new:Npn \__int_to_roman_d:w { d }
18141 \cs_new:Npn \__int_to_roman_m:w { m }
18142 \cs_new:Npn \__int_to_roman_Q:w #1 { }

```

```

18143 \cs_new:Npn \__int_to_Roman_i:w { I }
18144 \cs_new:Npn \__int_to_Roman_v:w { V }
18145 \cs_new:Npn \__int_to_Roman_x:w { X }
18146 \cs_new:Npn \__int_to_Roman_l:w { L }
18147 \cs_new:Npn \__int_to_Roman_c:w { C }
18148 \cs_new:Npn \__int_to_Roman_d:w { D }
18149 \cs_new:Npn \__int_to_Roman_m:w { M }
18150 \cs_new:Npn \__int_to_Roman_Q:w #1 { }

```

(End of definition for `\int_to_roman:n` and others. These functions are documented on page 176.)

59.9 Converting from other formats to integers

`__int_pass_signs:wn` Called as `__int_pass_signs:wn <signs and digits> \s__int_stop {<code>}`, this function leaves in the input stream any sign it finds, then inserts the `<code>` before the first non-sign token (and removes `\s__int_stop`). More precisely, it deletes any + and passes any - to the input stream, hence should be called in an integer expression.

```

18151 \cs_new:Npn \__int_pass_signs:wn #1
18152 {
18153   \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
18154   \exp_after:wN \__int_pass_signs:wn
18155   \else:
18156     \exp_after:wN \__int_pass_signs_end:wn
18157     \exp_after:wN #1
18158   \fi:
18159 }
18160 \cs_new:Npn \__int_pass_signs_end:wn #1 \s__int_stop #2 { #2 #1 }

```

(End of definition for `__int_pass_signs:wn` and `__int_pass_signs_end:wn`.)

`\int_from_alpha:n` First take care of signs then loop through the input using the recursion quarks. The `__int_from_alpha:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `__int_from_alpha:N` converts one letter to an expression which evaluates to the correct number.

```

18161 \cs_new:Npn \int_from_alpha:n #1
18162 {
18163   \int_eval:n
18164   {
18165     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
18166     \s__int_stop { \__int_from_alpha:nN { 0 } }
18167     \q__int_recursion_tail \q__int_recursion_stop
18168   }
18169 }
18170 \cs_new:Npn \__int_from_alpha:nN #1#2
18171 {
18172   \__int_if_recursion_tail_stop_do:Nn #2 {#1}
18173   \exp_args:Nf \__int_from_alpha:nN
18174   { \int_eval:n { #1 * 26 + \__int_from_alpha:N #2 } }
18175 }
18176 \cs_new:Npn \__int_from_alpha:N #1
18177 { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }

```

(End of definition for `\int_from_alpha:n`, `__int_from_alpha:nN`, and `__int_from_alpha:N`. This function is documented on page 176.)

`\int_from_base:nn` Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of `__int_from_base:nnN`. To convert a single character, `__int_from_base:N` checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use `\int_eval:n`, hence is not safe for general use.

```

18178 \cs_new:Npn \int_from_base:nn #1#2
18179 {
18180   \int_eval:n
18181   {
18182     \exp_after:wN \__int_pass_signs:wN \tl_to_str:n {#1}
18183     \s__int_stop { \__int_from_base:nnN { 0 } {#2} }
18184     \q__int_recursion_tail \q__int_recursion_stop
18185   }
18186 }
18187 \cs_new:Npn \__int_from_base:nnN #1#2#3
18188 {
18189   \__int_if_recursion_tail_stop_do:Nn #3 {#1}
18190   \exp_args:Nf \__int_from_base:nnN
18191   { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
18192   {#2}
18193 }
18194 \cs_new:Npn \__int_from_base:N #1
18195 {
18196   \int_compare:nNnTF { '#1 } < { 58 }
18197   {#1}
18198   { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
18199 }

```

(End of definition for `\int_from_base:nn`, `__int_from_base:nnN`, and `__int_from_base:N`. This function is documented on page 177.)

`\int_from_bin:n` Wrappers around the generic function.

```

\int_from_hex:n 18200 \cs_new:Npn \int_from_bin:n #1
\int_from_oct:n 18201 { \int_from_base:nn {#1} { 2 } }
18202 \cs_new:Npn \int_from_hex:n #1
18203 { \int_from_base:nn {#1} { 16 } }
18204 \cs_new:Npn \int_from_oct:n #1
18205 { \int_from_base:nn {#1} { 8 } }

```

(End of definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 176.)

`\c__int_from_roman_i_int` Constants used to convert from Roman numerals to integers.

```

\c__int_from_roman_v_int 18206 \int_const:cn { c__int_from_roman_i_int } { 1 }
\c__int_from_roman_x_int 18207 \int_const:cn { c__int_from_roman_v_int } { 5 }
\c__int_from_roman_l_int 18208 \int_const:cn { c__int_from_roman_x_int } { 10 }
\c__int_from_roman_c_int 18209 \int_const:cn { c__int_from_roman_l_int } { 50 }
\c__int_from_roman_d_int 18210 \int_const:cn { c__int_from_roman_c_int } { 100 }
\c__int_from_roman_m_int 18211 \int_const:cn { c__int_from_roman_d_int } { 500 }
\c__int_from_roman_I_int 18212 \int_const:cn { c__int_from_roman_m_int } { 1000 }
\c__int_from_roman_V_int 18213 \int_const:cn { c__int_from_roman_I_int } { 1 }
\c__int_from_roman_X_int 18214 \int_const:cn { c__int_from_roman_V_int } { 5 }
\c__int_from_roman_X_int 18215 \int_const:cn { c__int_from_roman_X_int } { 10 }
\c__int_from_roman_L_int 18216 \int_const:cn { c__int_from_roman_L_int } { 50 }
\c__int_from_roman_C_int
\c__int_from_roman_D_int
\c__int_from_roman_M_int

```

```

18217 \int_const:cn { c__int_from_roman_C_int } { 100 }
18218 \int_const:cn { c__int_from_roman_D_int } { 500 }
18219 \int_const:cn { c__int_from_roman_M_int } { 1000 }

```

(End of definition for `\c__int_from_roman_i_int` and others.)

`\int_from_roman:n` The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by `TEX`. If any unknown letter is found, skip to the closing parenthesis and insert `*0-1` afterwards, to replace the value by `-1`.

```

18220 \cs_new:Npn \int_from_roman:n #1
18221 {
18222   \int_eval:n
18223   {
18224     (
18225       0
18226       \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
18227       \q__int_recursion_tail \q__int_recursion_tail \q__int_recursion_stop
18228     )
18229   }
18230 }
18231 \cs_new:Npn \__int_from_roman:NN #1#2
18232 {
18233   \__int_if_recursion_tail_stop:N #1
18234   \int_if_exist:cF { c__int_from_roman_ #1 _int }
18235   { \__int_from_roman_error:w }
18236   \__int_if_recursion_tail_stop_do:Nn #2
18237   { + \use:c { c__int_from_roman_ #1 _int } }
18238   \int_if_exist:cF { c__int_from_roman_ #2 _int }
18239   { \__int_from_roman_error:w }
18240   \int_compare:nNnTF
18241   { \use:c { c__int_from_roman_ #1 _int } }
18242   <
18243   { \use:c { c__int_from_roman_ #2 _int } }
18244   {
18245     + \use:c { c__int_from_roman_ #2 _int }
18246     - \use:c { c__int_from_roman_ #1 _int }
18247     \__int_from_roman:NN
18248   }
18249   {
18250     + \use:c { c__int_from_roman_ #1 _int }
18251     \__int_from_roman:NN #2
18252   }
18253 }
18254 \cs_new:Npn \__int_from_roman_error:w #1 \q__int_recursion_stop #2
18255 { #2 * 0 - 1 }

```

(End of definition for `\int_from_roman:n`, `__int_from_roman:NN`, and `__int_from_roman_error:w`. This function is documented on page 177.)

59.10 Viewing integer

`\int_show:N` Diagnostics.
`\int_show:c`
`__int_show:nN`

```

18256 \cs_new_eq:NN \int_show:N \__kernel_register_show:N
18257 \cs_generate_variant:Nn \int_show:N { c }

```

(End of definition for `\int_show:N` and `__int_show:nN`. This function is documented on page 177.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```

18258 \cs_new_protected:Npn \int_show:n
18259 { \__kernel_msg_show_eval:Nn \int_eval:n }

```

(End of definition for `\int_show:n`. This function is documented on page 178.)

`\int_log:N` Diagnostics.

```

\int_log:c 18260 \cs_new_eq:NN \int_log:N \__kernel_register_log:N
18261 \cs_generate_variant:Nn \int_log:N { c }

```

(End of definition for `\int_log:N`. This function is documented on page 178.)

`\int_log:n` Similar to `\int_show:n`.

```

18262 \cs_new_protected:Npn \int_log:n
18263 { \__kernel_msg_log_eval:Nn \int_eval:n }

```

(End of definition for `\int_log:n`. This function is documented on page 178.)

59.11 Random integers

`\int_rand:nm` Defined in `l3fp-random`.

(End of definition for `\int_rand:nm`. This function is documented on page 177.)

59.12 Constant integers

`\c_zero_int` The zero is defined in `l3basics`.

```

\c_one_int 18264 \int_const:Nn \c_one_int { 1 }

```

(End of definition for `\c_zero_int` and `\c_one_int`. These variables are documented on page 178.)

`\c_max_int` The largest number allowed is $2^{31} - 1$

```

18265 \int_const:Nn \c_max_int { 2 147 483 647 }

```

(End of definition for `\c_max_int`. This variable is documented on page 178.)

`\c_max_char_int` The largest character code is 1114111 (hexadecimal 10FFFF) in XeTeX and LuaTeX and 255 in other engines. In many places pTeX and upTeX support larger character codes but for instance the values of `\lccode` are restricted to $[0, 255]$.

```

18266 \int_const:Nn \c_max_char_int
18267 {
18268   \if_int_odd:w 0
18269     \cs_if_exist:NT \tex luatexversion:D { 1 }
18270     \cs_if_exist:NT \tex XeTeXversion:D { 1 } ~
18271     "10FFFF
18272   \else:
18273     "FF
18274   \fi:
18275 }

```

(End of definition for `\c_max_char_int`. This variable is documented on page 178.)

59.13 Scratch integers

`\l_tmpa_int` We provide two local and two global scratch counters, maybe we need more or less.

`\l_tmpb_int` 18276 `\int_new:N \l_tmpa_int`

`\g_tmpa_int` 18277 `\int_new:N \l_tmpb_int`

`\g_tmpb_int` 18278 `\int_new:N \g_tmpa_int`

18279 `\int_new:N \g_tmpb_int`

(End of definition for `\l_tmpa_int` and others. These variables are documented on page 178.)

59.14 Integers for earlier modules

<@@=seq>

`\l__int_internal_a_int`

`\l__int_internal_b_int`

18280 `\int_new:N \l__int_internal_a_int`

18281 `\int_new:N \l__int_internal_b_int`

(End of definition for `\l__int_internal_a_int` and `\l__int_internal_b_int`.)

18282 `</package>`

Chapter 60

I3flag implementation

```
18283 (*package)
```

```
18284 (@@=flag)
```

The following test files are used for this code: *m3flag001*.

60.1 Protected flag commands

The height h of a flag (which is initially zero) is stored by setting control sequences of the form $\langle flag\ name \rangle \langle integer \rangle$ to \relax for $0 \leq \langle integer \rangle < h$. These control sequences are produced by $\cs:w \langle flag\ var \rangle \langle integer \rangle \cs_end:$, namely the $\langle flag\ var \rangle$ is actually a (protected) macro expanding to its own csname.

`\flag_new:N` Evaluate the csname of #1 for use in constructing the various indexed macros.

`\flag_new:c`

```
18285 \cs_new_protected:Npn \flag_new:N #1
18286   { \cs_new_protected:Npe #1 { \cs_to_str:N #1 } }
18287 \cs_generate_variant:Nn \flag_new:N { c }
```

(End of definition for `\flag_new:N`. This function is documented on page 181.)

`\l_tmpa_flag` Two flag variables for scratch use.

`\l_tmpb_flag`

```
18288 \flag_new:N \l_tmpa_flag
18289 \flag_new:N \l_tmpb_flag
```

(End of definition for `\l_tmpa_flag` and `\l_tmpb_flag`. These variables are documented on page 183.)

`\flag_clear:N` Undefine control sequences, starting from the 0 flag, upwards, until reaching an undefined control sequence. We don't use `\cs_undefine:c` because that would act globally.

`\flag_clear:c`

`__flag_clear:wN`

```
18290 \cs_new_protected:Npn \flag_clear:N #1
18291   {
18292     \__flag_clear:wN 0 ; #1
18293     \prg_break_point:
18294   }
18295 \cs_generate_variant:Nn \flag_clear:N { c }
18296 \cs_new_protected:Npn \__flag_clear:wN #1 ; #2
18297   {
18298     \if_cs_exist:w #2 #1 \cs_end: \else:
18299     \prg_break:n
18300   \fi:
```

```

18301 \cs_set_eq:cN { #2 #1 } \tex_undefined:D
18302 \exp_after:wN \_flag_clear:wN
18303 \int_value:w \int_eval:w \c_one_int + #1 ; #2
18304 }

```

(End of definition for `\flag_clear:N` and `_flag_clear:wN`. This function is documented on page 182.)

`\flag_clear_new:N` As for other datatypes, clear the `(flag var)` or create a new one, as appropriate.

```

\flag_clear_new:c
18305 \cs_new_protected:Npn \flag_clear_new:N #1
18306 { \flag_if_exist:NTF #1 { \flag_clear:N } { \flag_new:N } #1 }
18307 \cs_generate_variant:Nn \flag_clear_new:N { c }

```

(End of definition for `\flag_clear_new:N`. This function is documented on page 182.)

`\flag_show:N` Show the height (terminal or log file) using appropriate `l3msg` auxiliaries.

```

\flag_show:c
\flag_log:N
\flag_log:c
\_flag_show:NN
18308 \cs_new_protected:Npn \flag_show:N { \_flag_show:NN \tl_show:n }
18309 \cs_generate_variant:Nn \flag_show:N { c }
18310 \cs_new_protected:Npn \flag_log:N { \_flag_show:NN \tl_log:n }
18311 \cs_generate_variant:Nn \flag_log:N { c }
18312 \cs_new_protected:Npn \_flag_show:NN #1#2
18313 {
18314   \_kernel_chk_defined:NT #2
18315   { \exp_args:Ne #1 { \tl_to_str:n { #2 height } = \flag_height:N #2 } }
18316 }

```

(End of definition for `\flag_show:N`, `\flag_log:N`, and `_flag_show:NN`. These functions are documented on page 182.)

60.2 Expandable flag commands

`\flag_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\flag_if_exist_p:c
\flag_if_exist:NTF
\flag_if_exist:cTF
18317 \prg_new_eq_conditional:NNn \flag_if_exist:N \cs_if_exist:N
18318 { TF , T , F , p }
18319 \prg_new_eq_conditional:NNn \flag_if_exist:c \cs_if_exist:c
18320 { TF , T , F , p }

```

(End of definition for `\flag_if_exist:NTF`. This function is documented on page 182.)

`\flag_if_raised_p:N` Test if the flag has a non-zero height, by checking the 0 control sequence.

```

\flag_if_raised_p:c
\flag_if_raised:NTF
\flag_if_raised:cTF
18321 \prg_new_conditional:Npnn \flag_if_raised:N #1 { p , T , F , TF }
18322 {
18323   \if_cs_exist:w #1 0 \cs_end:
18324   \prg_return_true:
18325   \else:
18326   \prg_return_false:
18327   \fi:
18328 }
18329 \prg_generate_conditional_variant:Nnn \flag_if_raised:N
18330 { c } { p , T , F , TF }

```

(End of definition for `\flag_if_raised:NTF`. This function is documented on page 182.)

\flag_height:N Extract the value of the flag by going through all of the control sequences starting from 0.

\flag_height:c

```

18331 \cs_new:Npn \flag_height:N #1 { \__flag_height_loop:wN 0; #1 }
18332 \cs_new:Npn \__flag_height_loop:wN #1 ; #2
18333   {
18334     \if_cs_exist:w #2 #1 \cs_end: \else:
18335       \exp_after:wN \__flag_height_end:wN
18336     \fi:
18337     \exp_after:wN \__flag_height_loop:wN
18338     \int_value:w \int_eval:w \c_one_int + #1 ; #2
18339   }
18340 \cs_new:Npn \__flag_height_end:wN #1 + #2 ; #3 {#2}
18341 \cs_generate_variant:Nn \flag_height:N { c }

```

(End of definition for `\flag_height:N`, `__flag_height_loop:wN`, and `__flag_height_end:wN`. This function is documented on page 182.)

\flag_raise:N Change the appropriate control sequence to `\relax` by expanding a `\cs:w ... \cs_end:` construction, then pass it to `\use_none:n` to avoid leaving anything in the input stream.

\flag_raise:c

```

18342 \cs_new:Npn \flag_raise:N #1
18343   { \exp_after:wN \use_none:n \cs:w #1 \flag_height:N #1 \cs_end: }
18344 \cs_generate_variant:Nn \flag_raise:N { c }

```

(End of definition for `\flag_raise:N`. This function is documented on page 182.)

\flag_ensure_raised:N Pass the control sequence with name `<flag name>0` to `\use_none:n`. Constructing the control sequence ensures that it changes from being undefined (if it was so) to being `\relax`.

\flag_ensure_raised:c

```

18345 \cs_new:Npn \flag_ensure_raised:N #1
18346   { \exp_after:wN \use_none:n \cs:w #1 0 \cs_end: }
18347 \cs_generate_variant:Nn \flag_ensure_raised:N { c }

```

(End of definition for `\flag_ensure_raised:N`. This function is documented on page 182.)

60.3 Old n-type flag commands

Here we keep the old flag commands since our policy is to no longer delete deprecated functions. The idea is to simply map `<flag name>` to `\l_<flag name>_flag`. When the debugging code is activated, it checks existence of the N-type flag variables that result.

```

\flag_new:n
\flag_clear:n 18348 \cs_new_protected:Npn \flag_new:n #1 { \flag_new:c { l_#1_flag } }
\flag_clear_new:n 18349 \cs_new_protected:Npn \flag_clear:n #1 { \flag_clear:c { l_#1_flag } }
\flag_if_exist_p:n 18350 \cs_new_protected:Npn \flag_clear_new:n #1 { \flag_clear_new:c { l_#1_flag } }
\flag_if_exist:nTF 18351 \cs_new:Npn \flag_if_exist_p:n #1 { \flag_if_exist_p:c { l_#1_flag } }
\flag_if_raised_p:n 18352 \cs_new:Npn \flag_if_exist:nT #1 { \flag_if_exist:cT { l_#1_flag } }
\flag_if_raised:nTF 18353 \cs_new:Npn \flag_if_exist:nF #1 { \flag_if_exist:cF { l_#1_flag } }
\flag_height:n 18354 \cs_new:Npn \flag_if_exist:nTF #1 { \flag_if_exist:cTF { l_#1_flag } }
\flag_raise:n 18355 \cs_new:Npn \flag_if_raised_p:n #1 { \flag_if_raised_p:c { l_#1_flag } }
\flag_ensure_raised:n 18356 \cs_new:Npn \flag_if_raised:nT #1 { \flag_if_raised:cT { l_#1_flag } }
18357 \cs_new:Npn \flag_if_raised:nF #1 { \flag_if_raised:cF { l_#1_flag } }
18358 \cs_new:Npn \flag_if_raised:nTF #1 { \flag_if_raised:cTF { l_#1_flag } }
18359 \cs_new:Npn \flag_height:n #1 { \flag_height:c { l_#1_flag } }

```

```

18360 \cs_new:Npn \flag_raise:n #1 { \flag_raise:c { l_#1_flag } }
18361 \cs_new:Npn \flag_ensure_raised:n #1 { \flag_ensure_raised:c { l_#1_flag } }

```

(End of definition for \flag_new:n and others.)

```

\flag_show:n To avoid changing the output here we mostly keep the old code.
\flag_log:n
\__flag_show:Nn
18362 \cs_new_protected:Npn \flag_show:n { \__flag_show:Nn \tl_show:n }
18363 \cs_new_protected:Npn \flag_log:n { \__flag_show:Nn \tl_log:n }
18364 \cs_new_protected:Npn \__flag_show:Nn #1#2
18365 {
18366   \exp_args:Nc \__kernel_chk_defined:NT { l_#2_flag }
18367   {
18368     \exp_args:Ne #1
18369     { \tl_to_str:n { flag-#2~height } = \flag_height:n {#2} }
18370   }
18371 }

```

(End of definition for \flag_show:n, \flag_log:n, and __flag_show:Nn.)

```

18372 \endpackage

```

Chapter 61

l3clist implementation

The following test files are used for this code: *m3clist002*.

```
18373 (*package)
18374 (@@=clist)
```

`\c_empty_clist` An empty comma list is simply an empty token list.

```
18375 \cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(End of definition for `\c_empty_clist`. This variable is documented on page 194.)

`\l__clist_internal_clist` Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before `\clist_new:N`

```
18376 \tl_new:N \l__clist_internal_clist
```

(End of definition for `\l__clist_internal_clist`.)

`\s__clist_mark` Internal scan marks.

```
\s__clist_stop 18377 \scan_new:N \s__clist_mark
18378 \scan_new:N \s__clist_stop
```

(End of definition for `\s__clist_mark` and `\s__clist_stop`.)

`__clist_use_none_delimit_by_s_mark:w` Functions to gobble up to a scan mark.

```
\__clist_use_none_delimit_by_s_stop:w 18379 \cs_new:Npn \__clist_use_none_delimit_by_s_mark:w #1 \s__clist_mark { }
\__clist_use_i_delimit_by_s_stop:nw 18380 \cs_new:Npn \__clist_use_none_delimit_by_s_stop:w #1 \s__clist_stop { }
18381 \cs_new:Npn \__clist_use_i_delimit_by_s_stop:nw #1 #2 \s__clist_stop {#1}
```

(End of definition for `__clist_use_none_delimit_by_s_mark:w`, `__clist_use_none_delimit_by_s_stop:w`, and `__clist_use_i_delimit_by_s_stop:nw`.)

`__clist_tmp:w` A temporary function for various purposes.

```
18382 \cs_new_protected:Npn \__clist_tmp:w { }
```

(End of definition for `__clist_tmp:w`.)

61.1 Removing spaces around items

`__clist_trim_next:w` Called as `\exp:w __clist_trim_next:w \prg_do_nothing: <comma list> ...` it expands to `{<trimmed item>}` where the `<trimmed item>` is the first non-empty result from removing spaces from both ends of comma-delimited items in the `<comma list>`. The `\prg_do_nothing:` marker avoids losing braces. The test for blank items is a somewhat optimized `\tl_if_empty:oTF` construction; if blank, another item is sought, otherwise trim spaces.

```

18383 \cs_new:Npn \__clist_trim_next:w #1 ,
18384   {
18385     \tl_if_empty:oTF { \use_none:nn #1 ? }
18386     { \__clist_trim_next:w \prg_do_nothing: }
18387     { \tl_trim_spaces_apply:oN {#1} \exp_end: }
18388   }

```

(End of definition for __clist_trim_next:w.)

`__clist_sanitize:n` The auxiliary `__clist_sanitize:Nn` receives a delimiter (`\c_empty_tl` the first time, afterwards a comma) and that item as arguments. Unless we are done with the loop it calls `__clist_wrap_item:w` to unbrace the item (using a comma delimiter is safe since `#2` came from removing spaces from an argument delimited by a comma) and possibly re-brace it if needed.

```

18389 \cs_new:Npn \__clist_sanitize:n #1
18390   {
18391     \exp_after:wN \__clist_sanitize:Nn \exp_after:wN \c_empty_tl
18392     \exp:w \__clist_trim_next:w \prg_do_nothing:
18393     #1 , \s__clist_stop \prg_break: , \prg_break_point:
18394   }
18395 \cs_new:Npn \__clist_sanitize:Nn #1#2
18396   {
18397     \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
18398     #1 \__clist_wrap_item:w #2 ,
18399     \exp_after:wN \__clist_sanitize:Nn \exp_after:wN ,
18400     \exp:w \__clist_trim_next:w \prg_do_nothing:
18401   }

```

(End of definition for __clist_sanitize:n and __clist_sanitize:Nn.)

`__clist_if_wrap:nTF` True if the argument must be wrapped to avoid getting altered by some clist operations.
`__clist_if_wrap:w` That is the case whenever the argument

- starts or end with a space or contains a comma,
- is empty, or
- consists of a single braced group.

If the argument starts or ends with a space or contains a comma then one of the three arguments of `__clist_if_wrap:w` will have its end delimiter (partly) in one of the three copies of `#1` in `__clist_if_wrap:nTF`; this has a knock-on effect meaning that the result of the expansion is not empty; in that case, wrap. Otherwise, the argument is safe unless it starts with a brace group (or is empty) and it is empty or consists of a single n-type argument.

```

18402 \prg_new_conditional:Npnn \__clist_if_wrap:n #1 { TF }

```

```

18403 {
18404   \tl_if_empty:oTF
18405   {
18406     \__clist_if_wrap:w
18407     \s__clist_mark ? #1 ~ \s__clist_mark ? ~ #1
18408     \s__clist_mark , ~ \s__clist_mark #1 ,
18409   }
18410   {
18411     \tl_if_head_is_group:nTF { #1 { } }
18412     {
18413       \tl_if_empty:nTF {#1}
18414       { \prg_return_true: }
18415       {
18416         \tl_if_empty:oTF { \use_none:n #1}
18417         { \prg_return_true: }
18418         { \prg_return_false: }
18419       }
18420     }
18421     { \prg_return_false: }
18422   }
18423   { \prg_return_true: }
18424 }
18425 \cs_new:Npn \__clist_if_wrap:w #1 \s__clist_mark ? ~ #2 ~ \s__clist_mark #3 , { }

```

(End of definition for `__clist_if_wrap:nTF` and `__clist_if_wrap:w`.)

`__clist_wrap_item:w` Safe items are put in `\exp_not:n`, otherwise we put an extra set of braces.

```

18426 \cs_new:Npn \__clist_wrap_item:w #1 ,
18427   { \__clist_if_wrap:nTF {#1} { \exp_not:n { {#1} } } { \exp_not:n {#1} } }

```

(End of definition for `__clist_wrap_item:w`.)

61.2 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.

```

\clist_new:c 18428 \cs_new_eq:NN \clist_new:N \tl_new:N
18429 \cs_new_eq:NN \clist_new:c \tl_new:c

```

(End of definition for `\clist_new:N`. This function is documented on page 185.)

`\clist_const:Nn` Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

```

\clist_const:Nx 18430 \cs_new_protected:Npn \clist_const:Nn #1#2
\clist_const:cn 18431   { \tl_const:Ne #1 { \__clist_sanitize:n {#2} } }
\clist_const:ce 18432 \cs_generate_variant:Nn \clist_const:Nn { Ne , c , ce }
\clist_const:cx 18433 \cs_generate_variant:Nn \clist_const:Nn { Nx , cx }

```

(End of definition for `\clist_const:Nn`. This function is documented on page 185.)

`\clist_clear:N` Clearing comma lists is just the same as clearing token lists.

```

\clist_clear:c 18434 \cs_new_eq:NN \clist_clear:N \tl_clear:N
\clist_gclear:N 18435 \cs_new_eq:NN \clist_clear:c \tl_clear:c
\clist_gclear:c 18436 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
18437 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c

```

(End of definition for `\clist_clear:N` and `\clist_gclear:N`. These functions are documented on page 185.)

```

\clist_clear_new:N  Once again a copy from the token list functions.
\clist_clear_new:c 18438 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:N 18439 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
\clist_gclear_new:c 18440 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
18441 \cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c

```

(End of definition for `\clist_clear_new:N` and `\clist_gclear_new:N`. These functions are documented on page 185.)

```

\clist_set_eq:NN  Once again, these are simple copies from the token list functions.
\clist_set_eq:cN 18442 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:Nc 18443 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_set_eq:cc 18444 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:NN 18445 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:cN 18446 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:Nc 18447 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN 18448 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\clist_gset_eq:cc 18449 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc

```

(End of definition for `\clist_set_eq:NN` and `\clist_gset_eq:NN`. These functions are documented on page 185.)

```

\clist_set_from_seq:NN  Setting a comma list from a comma-separated list is done using a simple mapping. Safe
\clist_set_from_seq:cN  items are put in \exp_not:n, otherwise we put an extra set of braces. The first comma
\clist_set_from_seq:Nc  must be removed, except in the case of an empty comma-list.
\clist_set_from_seq:cc 18450 \cs_new_protected:Npn \clist_set_from_seq:NN
\clist_gset_from_seq:NN 18451 { \__clist_set_from_seq:NNNN \clist_clear:N \__kernel_tl_set:Nx }
\clist_gset_from_seq:cN 18452 \cs_new_protected:Npn \clist_gset_from_seq:NN
\clist_gset_from_seq:Nc 18453 { \__clist_set_from_seq:NNNN \clist_gclear:N \__kernel_tl_gset:Nx }
\clist_gset_from_seq:cc 18454 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
\__clist_set_from_seq:NNNN 18455 {
\__clist_set_from_seq:n 18456   \seq_if_empty:NTF #4
18457     { #1 #3 }
18458     {
18459       #2 #3
18460       {
18461         \exp_after:wN \use_none:n \exp:w \exp_end_continue_f:w
18462         \seq_map_function:NN #4 \__clist_set_from_seq:n
18463       }
18464     }
18465   }
18466 \cs_new:Npn \__clist_set_from_seq:n #1
18467 {
18468   ,
18469   \__clist_if_wrap:nTF {#1}
18470     { \exp_not:n { {#1} } }
18471     { \exp_not:n {#1} }
18472 }
18473 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
18474 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
18475 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
18476 \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }

```


(End of definition for `\clist_set_from_seq:NN` and others. These functions are documented on page 185.)

```

\clist_concat:NNN Concatenating comma lists is not quite as easy as it seems, as there needs to be the
\clist_concat:ccc correct addition of a comma to the output. So a little work to do.
\clist_gconcat:NNN
\clist_gconcat:ccc
__clist_concat:NNNN
18477 \cs_new_protected:Npn \clist_concat:NNN
18478 { \__clist_concat:NNNN \__kernel_tl_set:Nx }
18479 \cs_new_protected:Npn \clist_gconcat:NNN
18480 { \__clist_concat:NNNN \__kernel_tl_gset:Nx }
18481 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
18482 {
18483   #1 #2
18484   {
18485     \exp_not:o #3
18486     \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
18487     \exp_not:o #4
18488   }
18489 }
18490 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
18491 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }

```

(End of definition for `\clist_concat:NNN`, `\clist_gconcat:NNN`, and `__clist_concat:NNNN`. These functions are documented on page 186.)

```

\clist_if_exist_p:N Copies of the cs functions defined in l3basics.
\clist_if_exist_p:c
\clist_if_exist:NTF
\clist_if_exist:cTF
18492 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
18493 { TF , T , F , p }
18494 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
18495 { TF , T , F , p }

```

(End of definition for `\clist_if_exist:NTF`. This function is documented on page 186.)

61.3 Adding data to comma lists

```

\clist_set:Nn
\clist_set:NV
\clist_set:Ne
\clist_set:No
\clist_set:Nx
\clist_set:cn
\clist_set:cV
\clist_set:ce
\clist_set:co
\clist_set:cx
18496 \cs_new_protected:Npn \clist_set:Nn #1#2
18497 { \__kernel_tl_set:Nx #1 { \__clist_sanitize:n {#2} } }
18498 \cs_new_protected:Npn \clist_gset:Nn #1#2
18499 { \__kernel_tl_gset:Nx #1 { \__clist_sanitize:n {#2} } }
18500 \cs_generate_variant:Nn \clist_set:Nn { NV , Ne , c , cV , ce }
18501 \cs_generate_variant:Nn \clist_set:Nn { No , Nx , co , cx }
18502 \cs_generate_variant:Nn \clist_gset:Nn { NV , Ne , c , cV , ce }
18503 \cs_generate_variant:Nn \clist_gset:Nn { No , Nx , co , cx }

```

(End of definition for `\clist_set:Nn` and `\clist_gset:Nn`. These functions are documented on page 186.)

```

\clist_gset:Nn
\clist_put_left:Nn
\clist_gset:Ne
\clist_put_left:NV
\clist_gset:No
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_left:cV
\clist_put_left:ce
\clist_put_left:co
\clist_put_left:cx
\clist_gset:Nn
\clist_gset:NV
\clist_gset:Nv
\clist_gset:Ne
\clist_gput_left:Nn
\clist_gput_left:Nv
\clist_gput_left:Ne
\clist_gput_left:No

```

Everything is based on concatenation after storing in `\l__clist_internal_clist`. This avoids having to worry here about space-trimming and so on.

```

18504 \cs_new_protected:Npn \clist_put_left:Nn
18505 { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
18506 \cs_new_protected:Npn \clist_gput_left:Nn
18507 { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
18508 \cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4

```

```

18509 {
18510 #2 \l__clist_internal_clist {#4}
18511 #1 #3 \l__clist_internal_clist #3
18512 }
18513 \cs_generate_variant:Nn \clist_put_left:Nn { NV , Nv , Ne , c , cV , cv , ce }
18514 \cs_generate_variant:Nn \clist_put_left:Nn { No , Nx , co , cx }
18515 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , Nv , Ne , c , cV , cv , ce }
18516 \cs_generate_variant:Nn \clist_gput_left:Nn { No , Nx , co , cx }

```

(End of definition for `\clist_put_left:Nn`, `\clist_gput_left:Nn`, and `__clist_put_left:NNNn`. These functions are documented on page 186.)

```

\clist_put_right:Nn
\clist_put_right:NV 18517 \cs_new_protected:Npn \clist_put_right:Nn
\clist_put_right:Nv 18518 { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
\clist_put_right:Ne 18519 \cs_new_protected:Npn \clist_gput_right:Nn
\clist_put_right:No 18520 { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
\clist_put_right:Nx 18521 \cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
\clist_put_right:cn 18522 {
\clist_put_right:cV 18523 #2 \l__clist_internal_clist {#4}
\clist_put_right:cv 18524 #1 #3 #3 \l__clist_internal_clist
\clist_put_right:ce 18525 }
\clist_put_right:co 18526 \cs_generate_variant:Nn \clist_put_right:Nn
\clist_put_right:cx 18527 { NV , Nv , Ne , c , cV , cv , ce }
\clist_gput_right:Nn 18528 \cs_generate_variant:Nn \clist_put_right:Nn
\clist_gput_right:NV 18529 { No , Nx , co , cx }
\clist_gput_right:Nv 18530 \cs_generate_variant:Nn \clist_gput_right:Nn
\clist_gput_right:Ne 18531 { NV , Nv , Ne , c , cV , cv , ce }
\clist_gput_right:No 18532 \cs_generate_variant:Nn \clist_gput_right:Nn
\clist_gput_right:Nx 18533 { No , Nx , co , cx }
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:cv
\clist_gput_right:ce
\clist_gput_right:cx
\clist_gput_right:co
\__clist_put_right:NNNn
\__clist_get:wN

```

(End of definition for `\clist_put_right:Nn`, `\clist_gput_right:Nn`, and `__clist_put_right:NNNn`. These functions are documented on page 186.)

61.4 Comma lists as stacks

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma. No need to trim spaces as comma-list *variables* are assumed to have “cleaned-up” items. (Note that grabbing a comma-delimited item removes an outer pair of braces if present, exactly as needed to uncover the underlying item.)

```

18534 \cs_new_protected:Npn \clist_get:NN #1#2
18535 {
18536 \if_meaning:w #1 \c_empty_clist
18537 \tl_set:Nn #2 { \q_no_value }
18538 \else:
18539 \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
18540 \fi:
18541 }
18542 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \s__clist_stop #3
18543 { \tl_set:Nn #3 {#1} }
18544 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End of definition for `\clist_get:NN` and `__clist_get:wN`. This function is documented on page 192.)

`\clist_pop:NN` An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign to the variable. The second argument of `__clist_pop:wwNNN` is a comma list ending in a comma and `\s__clist_mark`, unless the original clist contained exactly one item: then the argument is just `\s__clist_mark`. The next auxiliary picks either `\exp_not:n` or `\use_none:n` as #2, ensuring that the result can safely be an empty comma list.

```

18545 \cs_new_protected:Npn \clist_pop:NN
18546   { \__clist_pop:NNN \__kernel_tl_set:Nx }
18547 \cs_new_protected:Npn \clist_gpop:NN
18548   { \__clist_pop:NNN \__kernel_tl_gset:Nx }
18549 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
18550   {
18551     \if_meaning:w #2 \c_empty_clist
18552       \tl_set:Nn #3 { \q_no_value }
18553     \else:
18554       \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
18555     \fi:
18556   }
18557 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \s__clist_stop #3#4#5
18558   {
18559     \tl_set:Nn #5 {#1}
18560     #3 #4
18561     {
18562       \__clist_pop:wN \prg_do_nothing:
18563       #2 \exp_not:o
18564       , \s__clist_mark \use_none:n
18565       \s__clist_stop
18566     }
18567   }
18568 \cs_new:Npn \__clist_pop:wN #1 , \s__clist_mark #2 #3 \s__clist_stop { #2 {#1} }
18569 \cs_generate_variant:Nn \clist_pop:NN { c }
18570 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End of definition for `\clist_pop:NN` and others. These functions are documented on page 192.)

`\clist_get:NNTF` The same, as branching code: very similar to the above.

```

18571 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
18572   {
18573     \if_meaning:w #1 \c_empty_clist
18574       \prg_return_false:
18575     \else:
18576       \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
18577       \prg_return_true:
18578     \fi:
18579   }
18580 \prg_generate_conditional_variant:Nmn \clist_get:NN { c } { T , F , TF }
18581 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
18582   { \__clist_pop_TF:NNN \__kernel_tl_set:Nx #1 #2 }
18583 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
18584   { \__clist_pop_TF:NNN \__kernel_tl_gset:Nx #1 #2 }
18585 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
18586   {
18587     \if_meaning:w #2 \c_empty_clist
18588       \prg_return_false:
18589     \else:

```

```

18590     \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
18591     \prg_return_true:
18592     \fi:
18593   }
18594 \prg_generate_conditional_variant:Nnn \clist_pop:NN { c } { T , F , TF }
18595 \prg_generate_conditional_variant:Nnn \clist_gpop:NN { c } { T , F , TF }

```

(End of definition for `\clist_get:NNTF` and others. These functions are documented on page 192.)

`\clist_push:Nn` Pushing to a comma list is the same as adding on the left.

```

\clist_push:NV 18596 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
\clist_push:No 18597 \cs_generate_variant:Nn \clist_push:Nn { NV , No , Nx , c , cV , co , cx }
\clist_push:Nx 18598 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\clist_push:cn 18599 \cs_generate_variant:Nn \clist_gpush:Nn { NV , No , Nx , c , cV , co , cx }

```

(End of definition for `\clist_push:Nn` and `\clist_gpush:Nn`. These functions are documented on page 193.)

`\clist_gpush:Nn`

`\clist_gpush:NV`

`\clist_gpush:No`

`\clist_gpush:Nx`

`\clist_gpush:cn`

`\clist_gpush:cV`

`\clist_gpush:co`

`\clist_gpush:cx`

61.5 Modifying comma lists

An internal comma list and a sequence for the removal routines.

```

\l__clist_internal_remove_clist 18600 \clist_new:N \l__clist_internal_remove_clist
\l__clist_internal_remove_seq 18601 \seq_new:N \l__clist_internal_remove_seq

```

(End of definition for `\l__clist_internal_remove_clist` and `\l__clist_internal_remove_seq`.)

`\clist_remove_duplicates:N`

`\clist_remove_duplicates:c`

`\clist_gremove_duplicates:N`

`\clist_gremove_duplicates:c`

`__clist_remove_duplicates:NN`

Removing duplicates means making a new list then copying it.

```

18602 \cs_new_protected:Npn \clist_remove_duplicates:N
18603 { \__clist_remove_duplicates:NN \clist_set_eq:NN }
18604 \cs_new_protected:Npn \clist_gremove_duplicates:N
18605 { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
18606 \cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
18607 {
18608   \clist_clear:N \l__clist_internal_remove_clist
18609   \clist_map_inline:Nn #2
18610   {
18611     \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
18612     {
18613       \tl_put_right:Ne \l__clist_internal_remove_clist
18614       {
18615         \clist_if_empty:NF \l__clist_internal_remove_clist { , }
18616         \__clist_if_wrap:nTF {##1} { \exp_not:n { {##1} } } { \exp_not:n {##1} }
18617       }
18618     }
18619   }
18620   #1 #2 \l__clist_internal_remove_clist
18621 }
18622 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
18623 \cs_generate_variant:Nn \clist_gremove_duplicates:N { c }

```

(End of definition for `\clist_remove_duplicates:N`, `\clist_gremove_duplicates:N`, and `__clist_remove_duplicates:NN`. These functions are documented on page 187.)

`\clist_remove_all:Nn` The method used here for safe items is very similar to `\tl_replace_all:Nnn`. However, if the item contains commas or leading/trailing spaces, or is empty, or consists of a single brace group, we know that it can only appear within braces so the code would fail; instead just convert to a sequence and do the removal with `l3seq` code (it involves somewhat elaborate code to do most of the work expandably but the final token list comparisons non-expandably).

`\clist_remove_all:cn` For “safe” items, build a function delimited by the `<item>` that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the `<item>`. The loop is controlled by the argument grabbed by `__clist_remove_all:w`: when the item was found, the `\s__clist_mark` delimiter used is the one inserted by `__clist_tmp:w`, and `__clist_use_none_delimit_by_s_stop:w` is deleted. At the end, the final `<item>` is grabbed, and the argument of `__clist_tmp:w` contains `\s__clist_mark`: in that case, `__clist_remove_all:w` removes the second `\s__clist_mark` (inserted by `__clist_tmp:w`), and lets `__clist_use_none_delimit_by_s_stop:w` act.

`\clist_remove_all:NV` No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn’t remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

`\clist_remove_all:cV`

`\clist_gremove_all:Nn`

`\clist_gremove_all:cn`

`\clist_gremove_all:NV`

`\clist_gremove_all:cV`

`__clist_remove_all:NNNn`

`__clist_remove_all:w`

`__clist_remove_all:`

```

18624 \cs_new_protected:Npn \clist_remove_all:Nn
18625   { \__clist_remove_all:NNNn \clist_set_from_seq:NN \__kernel_tl_set:Nx }
18626 \cs_new_protected:Npn \clist_gremove_all:Nn
18627   { \__clist_remove_all:NNNn \clist_gset_from_seq:NN \__kernel_tl_gset:Nx }
18628 \cs_new_protected:Npn \__clist_remove_all:NNNn #1#2#3#4
18629   {
18630     \__clist_if_wrap:nTF {#4}
18631     {
18632       \seq_set_from_clist:NN \l__clist_internal_remove_seq #3
18633       \seq_remove_all:Nn \l__clist_internal_remove_seq {#4}
18634       #1 #3 \l__clist_internal_remove_seq
18635     }
18636     {
18637       \cs_set:Npn \__clist_tmp:w ##1 , #4 ,
18638         {
18639           ##1
18640           , \s__clist_mark , \__clist_use_none_delimit_by_s_stop:w ,
18641           \__clist_remove_all:
18642         }
18643       #2 #3
18644       {
18645         \exp_after:wN \__clist_remove_all:
18646         #3 , \s__clist_mark , #4 , \s__clist_stop
18647       }
18648       \clist_if_empty:NF #3
18649       {
18650         #2 #3
18651         {
18652           \exp_args:No \exp_not:o
18653           { \exp_after:wN \use_none:n #3 }
18654         }
18655       }

```

```

18656     }
18657   }
18658   \cs_new:Npn \__clist_remove_all:
18659     { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
18660   \cs_new:Npn \__clist_remove_all:w #1 , \s__clist_mark , #2 , { \exp_not:n {#1} }
18661   \cs_generate_variant:Nn \clist_remove_all:Nn { c , NV , cV }
18662   \cs_generate_variant:Nn \clist_gremove_all:Nn { c , NV , cV }

```

(End of definition for `\clist_remove_all:Nn` and others. These functions are documented on page 187.)

`\clist_reverse:N` Use `\clist_reverse:n` in an e-expanding assignment. The extra work that `\clist_reverse:n` does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

```

\clist_reverse:c
\clist_greverse:N
\clist_greverse:c
18663 \cs_new_protected:Npn \clist_reverse:N #1
18664   { \__kernel_tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
18665 \cs_new_protected:Npn \clist_greverse:N #1
18666   { \__kernel_tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
18667 \cs_generate_variant:Nn \clist_reverse:N { c }
18668 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End of definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 187.)

`\clist_reverse:n` The reversed token list is built one item at a time, and stored between `\s__clist_stop` and `\s__clist_mark`, in the form of ? followed by zero or more instances of “`<item>`,”. We start from a comma list “`<item1>, …, <itemn>`”. During the loop, the auxiliary `__clist_reverse:wwNww` receives “`?<itemi>`” as #1, “`<itemi+1>, …, <itemn>`” as #2, `__clist_reverse:wwNww` as #3, what remains until `\s__clist_stop` as #4, and “`<itemi-1>, …, <item1>`,” as #5. The auxiliary moves #1 just before #5, with a comma, and calls itself (#3). After the last item is moved, `__clist_reverse:wwNww` receives “`\s__clist_mark __clist_reverse:wwNww !`” as its argument #1, thus `__clist_reverse_end:ww` as its argument #3. This second auxiliary cleans up until the marker !, removes the trailing comma (introduced when the first item was moved after `\s__clist_stop`), and leaves its argument #1 within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```

18669 \cs_new:Npn \clist_reverse:n #1
18670   {
18671     \__clist_reverse:wwNww ? #1 ,
18672     \s__clist_mark \__clist_reverse:wwNww ! ,
18673     \s__clist_mark \__clist_reverse_end:ww
18674     \s__clist_stop ? \s__clist_mark
18675   }
18676 \cs_new:Npn \__clist_reverse:wwNww
18677   #1 , #2 \s__clist_mark #3 #4 \s__clist_stop ? #5 \s__clist_mark
18678   { #3 ? #2 \s__clist_mark #3 #4 \s__clist_stop #1 , #5 \s__clist_mark }
18679 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \s__clist_mark
18680   { \exp_not:o { \use_none:n #2 } }

```

(End of definition for `\clist_reverse:n`, `__clist_reverse:wwNww`, and `__clist_reverse_end:ww`. This function is documented on page 187.)

`\clist_sort:Nn` Implemented in `l3sort`.

`\clist_sort:cn`
`\clist_gsort:Nn` (End of definition for `\clist_sort:Nn` and `\clist_gsort:Nn`. These functions are documented on page 188.)
`\clist_gsort:cn`

61.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.
`\clist_if_empty_p:c` 18681 `\prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N`
`\clist_if_empty:NTF` 18682 `{ p , T , F , TF }`
`\clist_if_empty:cTF` 18683 `\prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c`
18684 `{ p , T , F , TF }`

(End of definition for `\clist_if_empty:NTF`. This function is documented on page 188.)

`\clist_if_empty_p:n` As usual, we insert a token (here ?) before grabbing any argument: this avoids losing
`\clist_if_empty:nTF` braces. The argument of `\tl_if_empty:oTF` is empty if #1 is ? followed by blank spaces
`__clist_if_empty_n:w` (besides, this particular variant of the emptiness test is optimized). If the item of the
`__clist_if_empty_n:wNw` comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second
auxiliary grabs `\prg_return_false:` as #2, unless every item in the comma list was blank
and the loop actually got broken by the trailing `\s__clist_mark \prg_return_false:`
item.

```
18685 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
18686 {
18687   \__clist_if_empty_n:w ? #1
18688   , \s__clist_mark \prg_return_false:
18689   , \s__clist_mark \prg_return_true:
18690   \s__clist_stop
18691 }
18692 \cs_new:Npn \__clist_if_empty_n:w #1 ,
18693 {
18694   \tl_if_empty:oTF { \use_none:nn #1 ? }
18695   { \__clist_if_empty_n:w ? }
18696   { \__clist_if_empty_n:wNw }
18697 }
18698 \cs_new:Npn \__clist_if_empty_n:wNw #1 \s__clist_mark #2#3 \s__clist_stop {#2}
```

(End of definition for `\clist_if_empty:nTF`, `__clist_if_empty_n:w`, and `__clist_if_empty_n:wNw`. This function is documented on page 188.)

`\clist_if_in:NnTF` For “safe” items, we simply surround the comma list, and the item, with commas, then
`\clist_if_in:NvTF` use the same code as for `\tl_if_in:Nn`. For “unsafe” items we follow the same route as
`\clist_if_in:NoTF` `\seq_if_in:Nn`, mapping through the list a comparison function. If found, return true
`\clist_if_in:cnTF` and remove `\prg_return_false:`.
`\clist_if_in:cVTF` 18699 `\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }`
`\clist_if_in:coTF` 18700 `{`
`\clist_if_in:nnTF` 18701 `\exp_args:No __clist_if_in_return:nnN #1 {#2} #1`
`\clist_if_in:nVTF` 18702 `}`
`\clist_if_in:noTF` 18703 `\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }`
`__clist_if_in_return:nnN` 18704 `{`
18705 `\clist_set:Nn \l__clist_internal_clist {#1}`
18706 `\exp_args:No __clist_if_in_return:nnN \l__clist_internal_clist {#2}`
18707 `\l__clist_internal_clist`
18708 `}`
18709 `\cs_new_protected:Npn __clist_if_in_return:nnN #1#2#3`
18710 `{`
18711 `__clist_if_wrap:nTF {#2}`
18712 `{`
18713 `\cs_set:Npe __clist_tmp:w ##1`

```

18714     {
18715     \exp_not:N \tl_if_eq:nnT {##1}
18716     \exp_not:n
18717     {
18718     {#2}
18719     { \clist_map_break:n { \prg_return_true: \use_none:n } }
18720     }
18721     }
18722     \clist_map_function:NN #3 \__clist_tmp:w
18723     \prg_return_false:
18724     }
18725     {
18726     \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
18727     \tl_if_empty:oTF
18728     { \__clist_tmp:w ,#1, {} } ,#2, }
18729     { \prg_return_false: } { \prg_return_true: }
18730     }
18731     }
18732     \prg_generate_conditional_variant:Nnn \clist_if_in:Nn
18733     { NV , No , c , cV , co } { T , F , TF }
18734     \prg_generate_conditional_variant:Nnn \clist_if_in:nn
18735     { nV , no } { T , F , TF }

```

(End of definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `__clist_if_in_return:nnN`. These functions are documented on page 188.)

61.7 Mapping over comma lists

`\clist_map_function:NN` If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing eight comma-delimited items at a time. The end is marked by `\s__clist_stop`, which may not appear in any of the items. Once the last group of eight items has been reached, we go through them more slowly using `__clist_map_function_end:w`. The auxiliary function `__clist_map_function:Nw` is also used in some other clist mappings.

```

18736 \cs_new:Npn \clist_map_function:NN #1#2
18737 {
18738   \clist_if_empty:NF #1
18739   {
18740     \exp_after:wN \__clist_map_function:Nw \exp_after:wN #2 #1 ,
18741     \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18742     \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18743     \prg_break_point:Nn \clist_map_break: { }
18744   }
18745 }
18746 \cs_new:Npn \__clist_map_function:Nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
18747 {
18748   \__clist_use_none_delimit_by_s_stop:w
18749   #9 \__clist_map_function_end:w \s__clist_stop
18750   #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
18751   \__clist_map_function:Nw #1
18752 }
18753 \cs_new:Npn \__clist_map_function_end:w \s__clist_stop #1#2
18754 {

```



```

18755     \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
18756     #1 {#2}
18757     \__clist_map_function_end:w \s__clist_stop
18758   }
18759 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End of definition for `\clist_map_function:NN`, `__clist_map_function:Nw`, and `__clist_map_function_end:w`. This function is documented on page 189.)

`\clist_map_function:nN`
`\clist_map_function:eN`
`__clist_map_function_n:Nn`
`__clist_map_unbrace:wn`

The `n`-type mapping function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on `__clist_trim_next:w`. The auxiliary `__clist_map_function_n:Nn` receives as arguments the function, and the next non-empty item (after space trimming but before brace removal). One level of braces is removed by `__clist_map_unbrace:wn`.

```

18760 \cs_new:Npn \clist_map_function:nN #1#2
18761   {
18762     \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #2
18763     \exp:w \__clist_trim_next:w \prg_do_nothing: #1 ,
18764     \s__clist_stop \clist_map_break: ,
18765     \prg_break_point:Nn \clist_map_break: { }
18766   }
18767 \cs_generate_variant:Nn \clist_map_function:nN { e }
18768 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
18769   {
18770     \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
18771     \__clist_map_unbrace:wn #2 , #1
18772     \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #1
18773     \exp:w \__clist_trim_next:w \prg_do_nothing:
18774   }
18775 \cs_new:Npn \__clist_map_unbrace:wn #1, #2 { #2 {#1} }

```

(End of definition for `\clist_map_function:nN`, `__clist_map_function_n:Nn`, and `__clist_map_unbrace:wn`. This function is documented on page 189.)

`\clist_map_inline:Nn`
`\clist_map_inline:cn`
`\clist_map_inline:nm`

Inline mapping is done by creating a suitable function “on the fly”: this is done globally to avoid any issues with TeX’s groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the `n` version simply by storing the comma-list in a variable. We don’t need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```

18776 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
18777   {
18778     \clist_if_empty:NF #1
18779     {
18780       \int_gincr:N \g__kernel_prg_map_int
18781       \cs_gset_protected:cpn
18782         { __clist_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
18783       \exp_last_unbraced:Nco \__clist_map_function:Nw
18784         { __clist_map_ \int_use:N \g__kernel_prg_map_int :w }
18785         #1 ,
18786         \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18787         \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18788         \prg_break_point:Nn \clist_map_break:
18789         { \int_gdecr:N \g__kernel_prg_map_int }

```

```

18790     }
18791   }
18792 \cs_new_protected:Npn \clist_map_inline:nn #1
18793   {
18794     \clist_set:Nn \l__clist_internal_clist {#1}
18795     \clist_map_inline:Nn \l__clist_internal_clist
18796   }
18797 \cs_generate_variant:Nn \clist_map_inline:Nn { c }

```

(End of definition for `\clist_map_inline:Nn` and `\clist_map_inline:nn`. These functions are documented on page 189.)

`\clist_map_variable:NNn` The N-type version is a straightforward application of `\clist_map_tokens:Nn`, calling `__clist_map_variable:Nnn` for each item to assign the variable and run the user's code. The n-type version is *not* implemented in terms of the n-type function `\clist_map_tokens:Nn`, because here we are allowed to clean up the n-type comma list non-expandably.

```

18798 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
18799   { \clist_map_tokens:Nn #1 { \__clist_map_variable:Nnn #2 {#3} } }
18800 \cs_generate_variant:Nn \clist_map_variable:NNn { c }
18801 \cs_new_protected:Npn \__clist_map_variable:Nnn #1#2#3
18802   { \tl_set:Nn #1 {#3} #2 }
18803 \cs_new_protected:Npn \clist_map_variable:nNn #1
18804   {
18805     \clist_set:Nn \l__clist_internal_clist {#1}
18806     \clist_map_variable:NNn \l__clist_internal_clist
18807   }

```

(End of definition for `\clist_map_variable:NNn`, `\clist_map_variable:nNn`, and `__clist_map_variable:Nnn`. These functions are documented on page 189.)

`\clist_map_tokens:Nn` Essentially a copy of `\clist_map_function:NN` with braces added.

```

\clist_map_tokens:cn
\__clist_map_tokens:nw
\__clist_map_tokens_end:w
18808 \cs_new:Npn \clist_map_tokens:Nn #1#2
18809   {
18810     \clist_if_empty:NF #1
18811     {
18812       \exp_last_unbraced:Nno \__clist_map_tokens:nw {#2} #1 ,
18813       \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18814       \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18815       \prg_break_point:Nn \clist_map_break: { }
18816     }
18817   }
18818 \cs_new:Npn \__clist_map_tokens:nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
18819   {
18820     \__clist_use_none_delimit_by_s_stop:w
18821     #9 \__clist_map_tokens_end:w \s__clist_stop
18822     \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
18823     \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
18824     \__clist_map_tokens:nw {#1}
18825   }
18826 \cs_new:Npn \__clist_map_tokens_end:w \s__clist_stop \use:n #1#2
18827   {
18828     \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
18829     #1 {#2}

```

```

18830     \__clist_map_tokens_end:w \s__clist_stop
18831   }
18832 \cs_generate_variant:Nn \clist_map_tokens:Nn { c }

```

(End of definition for `\clist_map_tokens:Nn`, `__clist_map_tokens:nw`, and `__clist_map_tokens_end:w`. This function is documented on page 189.)

`\clist_map_tokens:nn` Similar to `\clist_map_function:nN` but with a different way of grabbing items because `__clist_map_tokens_n:nw` we cannot use `\exp_after:wN` to pass the `{code}`.

```

18833 \cs_new:Npn \clist_map_tokens:nn #1#2
18834   {
18835     \__clist_map_tokens_n:nw {#2}
18836     \prg_do_nothing: #1 , \s__clist_stop \clist_map_break: ,
18837     \prg_break_point:Nn \clist_map_break: { }
18838   }
18839 \cs_new:Npn \__clist_map_tokens_n:nw #1#2 ,
18840   {
18841     \tl_if_empty:oF { \use_none:nn #2 ? }
18842     {
18843       \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
18844       \tl_trim_spaces_apply:oN {#2} \use_ii_i:nn
18845       \__clist_map_unbrace:wn , {#1}
18846     }
18847     \__clist_map_tokens_n:nw {#1} \prg_do_nothing:
18848   }

```

(End of definition for `\clist_map_tokens:nn` and `__clist_map_tokens_n:nw`. This function is documented on page 189.)

`\clist_map_break:` The break statements use the general `\prg_map_break:Nn` mechanism.
`\clist_map_break:n`

```

18849 \cs_new:Npn \clist_map_break:
18850   { \prg_map_break:Nn \clist_map_break: { } }
18851 \cs_new:Npn \clist_map_break:n
18852   { \prg_map_break:Nn \clist_map_break: }

```

(End of definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page 189.)

`\clist_count:N` Counting the items in a comma list is done using the same approach as for other token
`\clist_count:c` count functions: turn each entry into a +1 then use integer evaluation to actually do the
`\clist_count:n` mathematics. In the case of an n-type comma-list, we could of course use `\clist_map_`
`\clist_count:e` `function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop
`__clist_count:n` manually, and skip blank items (but not `{}`), hence the extra spaces).
`__clist_count:w`

```

18853 \cs_new:Npn \clist_count:N #1
18854   {
18855     \int_eval:n
18856     {
18857       0
18858       \clist_map_function:NN #1 \__clist_count:n
18859     }
18860   }
18861 \cs_generate_variant:Nn \clist_count:N { c }
18862 \cs_new:Npn \__clist_count:n #1 { + 1 }
18863 \cs_set_protected:Npn \__clist_tmp:w #1
18864   {

```

```

18865 \cs_new:Npn \clist_count:n ##1
18866 {
18867   \int_eval:n
18868   {
18869     0
18870     \__clist_count:w #1
18871     ##1 , \s__clist_stop \prg_break: , \prg_break_point:
18872   }
18873 }
18874 \cs_new:Npn \__clist_count:w ##1 ,
18875 {
18876   \__clist_use_none_delimit_by_s_stop:w ##1 \s__clist_stop
18877   \tl_if_blank:nF {##1} { + 1 }
18878   \__clist_count:w #1
18879 }
18880 }
18881 \exp_args:No \__clist_tmp:w \c_space_tl
18882 \cs_generate_variant:Nn \clist_count:n { e }

```

(End of definition for `\clist_count:N` and others. These functions are documented on page 190.)

61.8 Using comma lists

```

\clist_use:Nnnn
\clist_use:cnnn
__clist_use:wwn
__clist_use:nwwwnwn
__clist_use:nwwn
\clist_use:Nn
\clist_use:cn

```

First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the *separator between two* in the middle.

Otherwise, `__clist_use:nwwwnwn` takes the following arguments; 1: a *separator*, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a *continuation* function (`use_ii` or `use_iii` with its *separator* argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\s__clist_stop`. The *separator* and the first of the three items are placed in the result, then we use the *continuation*, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\s__clist_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\s__clist_mark` is taken as a third item, and now the second `\s__clist_mark` serves as a delimiter to `use_ii`, switching to the other *continuation*, `use_iii`, which uses the *separator between final two*.

```

18883 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
18884 {
18885   \clist_if_exist:NTF #1
18886   {
18887     \int_case:nnF { \clist_count:N #1 }
18888     {
18889       { 0 } { }
18890       { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
18891       { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
18892     }
18893     {
18894       \exp_after:wN \__clist_use:nwwwnwn
18895       \exp_after:wN { \exp_after:wN } #1 ,
18896       \s__clist_mark , { \__clist_use:nwwwnwn {#3} }

```

```

18897     \s__clist_mark , { \__clist_use:nwwn {#4} }
18898     \s__clist_stop { }
18899   }
18900 }
18901 {
18902   \msg_expandable_error:nnn
18903   { kernel } { bad-variable } {#1}
18904 }
18905 }
18906 \cs_generate_variant:Nn \clist_use:Nnnn { c }
18907 \cs_new:Npn \__clist_use:wn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
18908 \cs_new:Npn \__clist_use:nwwwnwn
18909   #1#2 , #3 , #4 , #5 \s__clist_mark , #6#7 \s__clist_stop #8
18910   { #6 {#3} , {#4} , #5 \s__clist_mark , {#6} #7 \s__clist_stop { #8 #1 #2 } }
18911 \cs_new:Npn \__clist_use:nwwn #1#2 , #3 \s__clist_stop #4
18912   { \exp_not:n { #4 #1 #2 } }
18913 \cs_new:Npn \clist_use:Nn #1#2
18914   { \clist_use:Nnnn #1 {#2} {#2} {#2} }
18915 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End of definition for `\clist_use:Nnnn` and others. These functions are documented on page 191.)

`\clist_use:nmn` Items are grabbed by `__clist_use:Nw`, which detects blank items with a `\tl_if_empty:oTF` test (in which case it recurses). Non-blank items are either the end of the list, in which case the argument #1 of `__clist_use:Nw` is used to properly end the list, or are normal items, which must be trimmed and properly unbraced. As we find successive items, the long list of `__clist_use:Nw` calls gets shortened and we end up calling `__clist_use_more:w` once we have found 3 items. This auxiliary leaves the first-found item and the general separator, and calls `__clist_use:Nw` to find more items. A subtlety is that we use `__clist_use_end:w` both in the case of a two-item list and for the last two items of a general list: to get the correct separator, `__clist_use_more:w` replaces the separator-of-two by the last-separator when called, namely as soon as we have found three items.

```

18916 \cs_new:Npn \clist_use:nmn #1#2#3#4
18917   {
18918     \__clist_use:Nw \__clist_use_none_delimit_by_s_stop:w
18919     \__clist_use:Nw \__clist_use_one:w
18920     \__clist_use:Nw \__clist_use_end:w
18921     \__clist_use_more:w ;
18922     {#2} {#3} {#4} ;
18923     \prg_do_nothing: #1 , \s__clist_mark ,
18924     \s__clist_stop
18925   }
18926 \cs_new:Npn \__clist_use:Nw #1#2 ; #3 ; #4 ,
18927   {
18928     \tl_if_empty:oTF { \use_none:nn #4 ? }
18929     { \__clist_use:Nw #1#2 ; }
18930     {
18931       \__clist_use_none_delimit_by_s_mark:w #4 #1 \s__clist_mark
18932       \tl_trim_spaces_apply:oN {#4} \use_ii_i:nn
18933       \__clist_map_unbrace:wn , { #2 ; }
18934     }
18935     #3 ; \prg_do_nothing:

```

```

18936 }
18937 \cs_new:Npn \__clist_use_one:w \s__clist_mark #1 , #2#3#4 \s__clist_stop
18938 { \exp_not:n {#3} }
18939 \cs_new:Npn \__clist_use_end:w
18940 \s__clist_mark #1 , #2#3#4#5#6 \s__clist_stop
18941 { \exp_not:n { #4 #5 #3 } }
18942 \cs_new:Npn \__clist_use_more:w ; #1#2#3#4#5#6 ;
18943 {
18944 \exp_not:n { #3 #5 }
18945 \__clist_use:Nw \__clist_use_end:w \__clist_use_more:w ;
18946 {#1} {#2} {#6} {#5} {#6} ;
18947 }
18948 \cs_new:Npn \clist_use:nn #1#2 { \clist_use:nnnn {#1} {#2} {#2} {#2} }

```

(End of definition for `\clist_use:nnnn` and others. These functions are documented on page 192.)

61.9 Using a single item

```

\clist_item:Nn To avoid needing to test the end of the list at each step, we first compute the  $\langle length \rangle$ 
\clist_item:cn of the list. If the item number is 0, less than  $-\langle length \rangle$ , or more than  $\langle length \rangle$ , the
\__clist_item:nnnN result is empty. If it is negative, but not less than  $-\langle length \rangle$ , add  $\langle length \rangle + 1$  to the
\__clist_item:ffoN item number before performing the loop. The loop itself is very simple, return the item
\__clist_item:ffnN if the counter reached 1, otherwise, decrease the counter and repeat.
\__clist_item_N_loop:nw
18949 \cs_new:Npn \clist_item:Nn #1#2
18950 {
18951 \__clist_item:ffoN
18952 { \clist_count:N #1 }
18953 { \int_eval:n {#2} }
18954 #1
18955 \__clist_item_N_loop:nw
18956 }
18957 \cs_new:Npn \__clist_item:nnnN #1#2#3#4
18958 {
18959 \int_compare:nNnTF {#2} < 0
18960 {
18961 \int_compare:nNnTF {#2} < { - #1 }
18962 { \__clist_use_none_delimit_by_s_stop:w }
18963 { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
18964 }
18965 {
18966 \int_compare:nNnTF {#2} > {#1}
18967 { \__clist_use_none_delimit_by_s_stop:w }
18968 { #4 {#2} }
18969 }
18970 { } , #3 , \s__clist_stop
18971 }
18972 \cs_generate_variant:Nn \__clist_item:nnnN { ffo, ff }
18973 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
18974 {
18975 \int_compare:nNnTF {#1} = 0
18976 { \__clist_use_i_delimit_by_s_stop:nw { \exp_not:n {#2} } }
18977 { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
18978 }

```

```
18979 \cs_generate_variant:Nn \clist_item:Nn { c }
```

(End of definition for `\clist_item:Nn`, `__clist_item:nnnN`, and `__clist_item_N_loop:nw`. This function is documented on page 193.)

```
\clist_item:nn This starts in the same way as \clist_item:Nn by counting the items of the comma list.
\clist_item:en The final item should be space-trimmed before being brace-stripped, hence we insert a
\__clist_item_n:nw couple of odd-looking \prg_do_nothing: to avoid losing braces. Blank items are ignored.
\__clist_item_n_loop:nw 18980 \cs_new:Npn \clist_item:nn #1#2
\__clist_item_n_end:n 18981 {
\__clist_item_n_strip:n 18982 \__clist_item:ffnN
\__clist_item_n_strip:w 18983 { \clist_count:n {#1} }
18984 { \int_eval:n {#2} }
18985 {#1}
18986 \__clist_item_n:nw
18987 }
18988 \cs_generate_variant:Nn \clist_item:nn { e }
18989 \cs_new:Npn \__clist_item_n:nw #1
18990 { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
18991 \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
18992 {
18993 \exp_args:No \tl_if_blank:nTF {#2}
18994 { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
18995 {
18996 \int_compare:nNnTF {#1} = 0
18997 { \exp_args:No \__clist_item_n_end:n {#2} }
18998 {
18999 \exp_args:Nf \__clist_item_n_loop:nw
19000 { \int_eval:n { #1 - 1 } }
19001 \prg_do_nothing:
19002 }
19003 }
19004 }
19005 \cs_new:Npn \__clist_item_n_end:n #1 #2 \s__clist_stop
19006 { \tl_trim_spaces_apply:nN {#1} \__clist_item_n_strip:n }
19007 \cs_new:Npn \__clist_item_n_strip:n #1 { \__clist_item_n_strip:w #1 , }
19008 \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }
```

(End of definition for `\clist_item:nn` and others. This function is documented on page 193.)

```
\clist_rand_item:n The N-type function is not implemented through the n-type function for efficiency: for
\clist_rand_item:N instance comma-list variables do not require space-trimming of their items. Even testing
\clist_rand_item:c for emptiness of an n-type comma-list is slow, so we count items first and use that both
\__clist_rand_item:nn for the emptiness test and the pseudo-random integer. Importantly, \clist_item:Nn
and \clist_item:nn only evaluate their argument once.
```

```
19009 \cs_new:Npn \clist_rand_item:n #1
19010 { \exp_args:Nf \__clist_rand_item:nn { \clist_count:n {#1} } {#1} }
19011 \cs_new:Npn \__clist_rand_item:nn #1#2
19012 {
19013 \int_compare:nNnF {#1} = 0
19014 { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }
19015 }
19016 \cs_new:Npn \clist_rand_item:N #1
19017 {
```

```

19018 \clist_if_empty:NF #1
19019 { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
19020 }
19021 \cs_generate_variant:Nn \clist_rand_item:N { c }

```

(End of definition for `\clist_rand_item:n`, `\clist_rand_item:N`, and `__clist_rand_item:nn`. These functions are documented on page 193.)

61.10 Viewing comma lists

```

\clist_show:N Apply the general \__kernel_chk_tl_type:NnnT with \exp_not:o #2 serving as a
\clist_show:c dummy code to prevent a check performed by this auxiliary.
\clist_log:N
\clist_log:c
\__clist_show:NN
19022 \cs_new_protected:Npn \clist_show:N { \__clist_show:NN \msg_show:nneeee }
19023 \cs_generate_variant:Nn \clist_show:N { c }
19024 \cs_new_protected:Npn \clist_log:N { \__clist_show:NN \msg_log:nneeee }
19025 \cs_generate_variant:Nn \clist_log:N { c }
19026 \cs_new_protected:Npn \__clist_show:NN #1#2
19027 {
19028   \__kernel_chk_tl_type:NnnT #2 { clist } { \exp_not:o #2 }
19029   {
19030     \int_compare:nNnTF { \clist_count:N #2 }
19031     = { \exp_args:No \clist_count:n #2 }
19032     {
19033       #1 { clist } { show }
19034       { \token_to_str:N #2 }
19035       { \clist_map_function:NN #2 \msg_show_item:n }
19036       { } { }
19037     }
19038     {
19039       \msg_error:nnee { clist } { non-clist }
19040       { \token_to_str:N #2 } { \tl_to_str:N #2 }
19041     }
19042   }
19043 }

```

(End of definition for `\clist_show:N`, `\clist_log:N`, and `__clist_show:NN`. These functions are documented on page 193.)

```

\clist_show:n A variant of the above: no existence check, empty first argument for the message.
\clist_log:n
\__clist_show:Nn
19044 \cs_new_protected:Npn \clist_show:n { \__clist_show:Nn \msg_show:nneeee }
19045 \cs_new_protected:Npn \clist_log:n { \__clist_show:Nn \msg_log:nneeee }
19046 \cs_new_protected:Npn \__clist_show:Nn #1#2
19047 {
19048   #1 { clist } { show }
19049   { } { \clist_map_function:nN {#2} \msg_show_item:n } { } { }
19050 }

```

(End of definition for `\clist_show:n`, `\clist_log:n`, and `__clist_show:Nn`. These functions are documented on page 194.)

61.11 Scratch comma lists

```
\l_tmpa_clist Temporary comma list variables.  
\l_tmpb_clist 19051 \clist_new:N \l_tmpa_clist  
\g_tmpa_clist 19052 \clist_new:N \l_tmpb_clist  
\g_tmpb_clist 19053 \clist_new:N \g_tmpa_clist  
19054 \clist_new:N \g_tmpb_clist
```

(End of definition for \l_tmpa_clist and others. These variables are documented on page 194.)

```
19055 </package>
```

Chapter 62

l3token implementation

```
19056 (*package)
```

```
19057 (*tex)
```

```
19058 (@@=char)
```

62.1 Internal auxiliaries

`\s__char_stop` Internal scan mark.

```
19059 \scan_new:N \s__char_stop
```

(End of definition for \s__char_stop.)

`\q__char_no_value` Internal recursion quarks.

```
19060 \quark_new:N \q__char_no_value
```

(End of definition for \q__char_no_value.)

`__char_quark_if_no_value_p:N` Functions to query recursion quarks.

```
\__char_quark_if_no_value:NTF 19061 \__kernel_quark_new_conditional:Nn \__char_quark_if_no_value:N { TF }
```

(End of definition for __char_quark_if_no_value:NTF.)

62.2 Manipulating and interrogating character tokens

`\char_set_catcode:nm` Simple wrappers around the primitives.

```
19062 \cs_new_protected:Npn \char_set_catcode:nm #1#2
```

```
19063 { \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
```

```
19064 \cs_new:Npn \char_value_catcode:n #1
```

```
19065 { \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }
```

```
19066 \cs_new_protected:Npn \char_show_value_catcode:n #1
```

```
19067 { \exp_args:Nf \tl_show:n { \char_value_catcode:n {#1} } }
```

(End of definition for \char_set_catcode:nm, \char_value_catcode:n, and \char_show_value_catcode:n. These functions are documented on page 198.)

```

\char_set_catcode_escape:N
  \char_set_catcode_group_begin:N 19068 \cs_new_protected:Npn \char_set_catcode_escape:N #1
  \char_set_catcode_group_end:N    19069   { \char_set_catcode:nn { '#1 } { 0 } }
  \char_set_catcode_math_toggle:N  19070 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
  \char_set_catcode_alignment:N    19071   { \char_set_catcode:nn { '#1 } { 1 } }
\char_set_catcode_end_line:N
  \char_set_catcode_parameter:N    19072 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
  \char_set_catcode_math_superscript:N 19073   { \char_set_catcode:nn { '#1 } { 2 } }
  \char_set_catcode_math_subscript:N 19074 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
  \char_set_catcode_ignore:N       19075   { \char_set_catcode:nn { '#1 } { 3 } }
  \char_set_catcode_space:N        19076 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
  \char_set_catcode_letter:N       19077   { \char_set_catcode:nn { '#1 } { 4 } }
  \char_set_catcode_other:N        19078 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
  \char_set_catcode_active:N       19079   { \char_set_catcode:nn { '#1 } { 5 } }
\char_set_catcode_comment:N
  \char_set_catcode_invalid:N      19080 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
  \char_set_catcode_group_begin:N  19081   { \char_set_catcode:nn { '#1 } { 6 } }
  \char_set_catcode_group_end:N    19082 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
  \char_set_catcode_math_toggle:N  19083   { \char_set_catcode:nn { '#1 } { 7 } }
  \char_set_catcode_math_superscript:N 19084 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
  \char_set_catcode_math_subscript:N 19085   { \char_set_catcode:nn { '#1 } { 8 } }
  \char_set_catcode_math_subscript:N 19086 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
  \char_set_catcode_ignore:N       19087   { \char_set_catcode:nn { '#1 } { 9 } }
  \char_set_catcode_ignore:N       19088 \cs_new_protected:Npn \char_set_catcode_space:N #1
  \char_set_catcode_ignore:N       19089   { \char_set_catcode:nn { '#1 } { 10 } }
  \char_set_catcode_ignore:N       19090 \cs_new_protected:Npn \char_set_catcode_letter:N #1
  \char_set_catcode_ignore:N       19091   { \char_set_catcode:nn { '#1 } { 11 } }
  \char_set_catcode_ignore:N       19092 \cs_new_protected:Npn \char_set_catcode_other:N #1
  \char_set_catcode_ignore:N       19093   { \char_set_catcode:nn { '#1 } { 12 } }
  \char_set_catcode_ignore:N       19094 \cs_new_protected:Npn \char_set_catcode_active:N #1
  \char_set_catcode_ignore:N       19095   { \char_set_catcode:nn { '#1 } { 13 } }
  \char_set_catcode_ignore:N       19096 \cs_new_protected:Npn \char_set_catcode_comment:N #1
  \char_set_catcode_ignore:N       19097   { \char_set_catcode:nn { '#1 } { 14 } }
  \char_set_catcode_ignore:N       19098 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
  \char_set_catcode_ignore:N       19099   { \char_set_catcode:nn { '#1 } { 15 } }

```

(End of definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 197.)

```

\char_set_catcode_escape:n
  \char_set_catcode_group_begin:n  19100 \cs_new_protected:Npn \char_set_catcode_escape:n #1
  \char_set_catcode_group_end:n    19101   { \char_set_catcode:nn {#1} { 0 } }
  \char_set_catcode_math_toggle:n  19102 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
  \char_set_catcode_alignment:n    19103   { \char_set_catcode:nn {#1} { 1 } }
\char_set_catcode_end_line:n
  \char_set_catcode_parameter:n    19104 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
  \char_set_catcode_math_superscript:n 19105   { \char_set_catcode:nn {#1} { 2 } }
  \char_set_catcode_math_subscript:n 19106 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
  \char_set_catcode_ignore:n       19107   { \char_set_catcode:nn {#1} { 3 } }
  \char_set_catcode_space:n        19108 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
  \char_set_catcode_letter:n       19109   { \char_set_catcode:nn {#1} { 4 } }
  \char_set_catcode_other:n        19110 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
  \char_set_catcode_active:n       19111   { \char_set_catcode:nn {#1} { 5 } }
  \char_set_catcode_group_begin:n  19112 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
  \char_set_catcode_group_end:n    19113   { \char_set_catcode:nn {#1} { 6 } }
  \char_set_catcode_math_superscript:n 19114 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
  \char_set_catcode_math_toggle:n  19115   { \char_set_catcode:nn {#1} { 7 } }

```

```

19116 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
19117   { \char_set_catcode:nn {#1} { 8 } }
19118 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
19119   { \char_set_catcode:nn {#1} { 9 } }
19120 \cs_new_protected:Npn \char_set_catcode_space:n #1
19121   { \char_set_catcode:nn {#1} { 10 } }
19122 \cs_new_protected:Npn \char_set_catcode_letter:n #1
19123   { \char_set_catcode:nn {#1} { 11 } }
19124 \cs_new_protected:Npn \char_set_catcode_other:n #1
19125   { \char_set_catcode:nn {#1} { 12 } }
19126 \cs_new_protected:Npn \char_set_catcode_active:n #1
19127   { \char_set_catcode:nn {#1} { 13 } }
19128 \cs_new_protected:Npn \char_set_catcode_comment:n #1
19129   { \char_set_catcode:nn {#1} { 14 } }
19130 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
19131   { \char_set_catcode:nn {#1} { 15 } }

```

(End of definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 198.)

`\char_set_mathcode:nn` Pretty repetitive, but necessary!

```

\char_value_mathcode:n 19132 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
\char_show_value_mathcode:n 19133   { \tex_mathcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
\char_set_lccode:nn 19134 \cs_new:Npn \char_value_mathcode:n #1
\char_value_lccode:n 19135   { \tex_the:D \tex_mathcode:D \int_eval:n {#1} \exp_stop_f: }
\char_show_value_lccode:n 19136 \cs_new_protected:Npn \char_show_value_mathcode:n #1
\char_set_uccode:nn 19137   { \exp_args:Nf \tl_show:n { \char_value_mathcode:n {#1} } }
\char_value_uccode:n 19138 \cs_new_protected:Npn \char_set_lccode:nn #1#2
\char_show_value_uccode:n 19139   { \tex_lccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
\char_set_sfcode:nn 19140 \cs_new:Npn \char_value_lccode:n #1
\char_value_sfcode:n 19141   { \tex_the:D \tex_lccode:D \int_eval:n {#1} \exp_stop_f: }
\char_show_value_sfcode:n 19142 \cs_new_protected:Npn \char_show_value_lccode:n #1
19143   { \exp_args:Nf \tl_show:n { \char_value_lccode:n {#1} } }
19144 \cs_new_protected:Npn \char_set_uccode:nn #1#2
19145   { \tex_uccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
19146 \cs_new:Npn \char_value_uccode:n #1
19147   { \tex_the:D \tex_uccode:D \int_eval:n {#1} \exp_stop_f: }
19148 \cs_new_protected:Npn \char_show_value_uccode:n #1
19149   { \exp_args:Nf \tl_show:n { \char_value_uccode:n {#1} } }
19150 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
19151   { \tex_sfcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
19152 \cs_new:Npn \char_value_sfcode:n #1
19153   { \tex_the:D \tex_sfcode:D \int_eval:n {#1} \exp_stop_f: }
19154 \cs_new_protected:Npn \char_show_value_sfcode:n #1
19155   { \exp_args:Nf \tl_show:n { \char_value_sfcode:n {#1} } }

```

(End of definition for `\char_set_mathcode:nn` and others. These functions are documented on page 199.)

`\l_char_active_seq` Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for “special” characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`).

```

19156 \seq_new:N \l_char_special_seq
19157 \seq_set_split:Nnn \l_char_special_seq { }

```

```

19158 { \ \ " \# \$ \% \& \ \ ^ \_ \{ \} \~ }
19159 \seq_new:N \l_char_active_seq
19160 \seq_set_split:Nnn \l_char_active_seq { }
19161 { \ " \$ & ^ \_ \~ }

```

(End of definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 200.)

62.3 Creating character tokens

`\char_set_active_eq:NN` Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX's `\letcharcode` primitive.

```

\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc
\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc
19162 \group_begin:
19163   \char_set_catcode_active:N \^^@
19164   \cs_set_protected:Npn \__char_tmp:nN #1#2
19165     {
19166       \cs_new_protected:cpn { #1 :nN } ##1
19167         {
19168           \group_begin:
19169             \char_set_lccode:nn { '^^@ } { ##1 }
19170             \tex_lowercase:D { \group_end: #2 ^^@ }
19171           }
19172       \cs_new_protected:cpe { #1 :NN } ##1
19173         { \exp_not:c { #1 : nN } { '##1 } }
19174     }
19175   \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
19176   \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
19177 \group_end:
19178 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
19179 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
19180 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
19181 \cs_generate_variant:Nn \char_gset_active_eq:nN { nc }

```

(End of definition for `\char_set_active_eq:NN` and others. These functions are documented on page 196.)

`__char_int_to_roman:w` For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

```

19182 \cs_new_eq:NN \__char_int_to_roman:w \tex_romannumeral:D

```

(End of definition for `__char_int_to_roman:w`.)

`\char_generate:nn` The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (XeTeX, LuaTeX). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

```

\__char_generate_aux:nn
\__char_generate_aux:nnw
\__char_generate_auxii:nnw
  \l__char_tmp_tl
\__char_generate_invalid_catcode:
19183 \cs_new:Npn \char_generate:nn #1#2
19184   {
19185     \exp:w \exp_after:wN \__char_generate_aux:w
19186       \int_value:w \int_eval:n {#1} \exp_after:wN ;
19187     \int_value:w \int_eval:n {#2} ;
19188   }

```

Before doing any actual conversion, first some special case filtering. Spaces are out here as LuaTeX emulation only makes normal (charcode 32 spaces). However, \char" is filtered out separately as that can't be done with macro emulation either, so is treated separately. That done, hand off to the engine-dependent part.

```

19189 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
19190 {
19191   \if_int_odd:w 0
19192     \if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
19193     \if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
19194     \if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
19195     \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
19196     \msg_expandable_error:nn { char }
19197     { invalid-catcode }
19198   \else:
19199     \if_int_odd:w 0
19200       \if_int_compare:w #1 < \c_zero_int 1 \fi:
19201       \if_int_compare:w #1 > \c_max_char_int 1 \fi: \exp_stop_f:
19202       \msg_expandable_error:nn { char }
19203       { out-of-range }
19204     \else:
19205       \if_int_compare:w #2#1 = 100 \exp_stop_f:
19206       \msg_expandable_error:nn { char } { null-space }
19207     \else:
19208       \__char_generate_aux:nnw {#1} {#2}
19209     \fi:
19210   \fi:
19211 \fi:
19212 \exp_end:
19213 }
19214 \tl_new:N \l__char_tmp_tl

```

Engine-dependent definitions are now needed for the implementation. Recent (u)pTeX and the Unicode engines LuaTeX and XeTeX have engine-level support for expandable character creation. pdfTeX and older (u)pTeX releases do not. The branching here is low-level to avoid fixing the category code of the null character used in the false branch. The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much. Older versions of XeTeX cannot generate active characters so we filter that: at some future stage that may change: the slightly odd ordering of auxiliaries reflects that.

```

19215 \group_begin:
19216   \char_set_catcode_active:N \textasciitilde L
19217   \cs_set:Npn \textasciitilde L { }
19218   \if_cs_exist:N \tex_Ucharcat:D
19219     \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
19220     {
19221       #3
19222       \exp_after:wN \exp_end:
19223       \tex_Ucharcat:D #1 \exp_stop_f: #2 \exp_stop_f:
19224     }
19225   \else:

```

For engines where \Ucharcat isn't available or emulated, we have to work in macros, and cover only the 8-bit range. The first stage is to build up a \tl containing \char" with each category code that can be accessed in this way, with an error set up for the other

cases. This is all done such that it can be quickly accessed using a `\if_case:w` low-level conditional. The list is done in reverse as this puts the case of an active token *first*: that's needed to cover the possibility that it is `\outer`. Getting the braces into the list is done using some standard `\if_false:` manipulation, while all of the `\exp_not:N` are required as there is an expansion in the setup.

```

19226     \char_set_catcode_active:n { 0 }
19227     \tl_set:Nn \l__char_tmp_tl { \exp_not:N ^^@ \exp_not:N \or: }
19228     \char_set_catcode_other:n { 0 }
19229     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
19230     \char_set_catcode_letter:n { 0 }
19231     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }

```

For making spaces, there needs to be an o-type expansion of a `\use:n` (or some other tokenization) to avoid dropping the space.

```

19232     \tl_put_right:Nn \l__char_tmp_tl { \use:n { ~ } \exp_not:N \or: }
19233     \tl_put_right:Nn \l__char_tmp_tl { \exp_not:N \or: }
19234     \char_set_catcode_math_subscript:n { 0 }
19235     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
19236     \char_set_catcode_math_superscript:n { 0 }
19237     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
19238     \char_set_catcode_parameter:n { 0 }
19239     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
19240     \tl_put_right:Nn \l__char_tmp_tl { { \if_false: } \fi: \exp_not:N \or: }
19241     \char_set_catcode_alignment:n { 0 }
19242     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
19243     \char_set_catcode_math_toggle:n { 0 }
19244     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
19245     \char_set_catcode_group_end:n { 0 }
19246     \tl_put_right:Nn \l__char_tmp_tl { \if_false: { \fi: ^^@ \exp_not:N \or: } % }
19247     \char_set_catcode_group_begin:n { 0 } % {
19248     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: } }

```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert `^^@` in each case. The e-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list.

```

19249     \cs_set_protected:Npn \__char_tmp:n #1
19250     {
19251         \char_set_lccode:nn { 0 } {#1}
19252         \char_set_lccode:nn { 32 } {#1}
19253         \exp_args:Ne \tex_lowercase:D
19254         {
19255             \tl_const:Ne
19256             \exp_not:c { c__char_ \__char_int_to_roman:w #1 _tl }
19257             { \exp_not:o \l__char_tmp_tl }
19258         }
19259     }
19260     \int_step_function:nnN { 0 } { 255 } \__char_tmp:n

```

As \TeX is very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. \TeX is happy if the token is hidden between braces within `\if_false: ... \fi:`. The rather low-level approach here expands in one step to the \langle *target token* \rangle (`\or: ...`), then `\exp_after:wN` (*target token*) (`\or: ...`) expands in one step to \langle *target token* \rangle . This means that `\exp_not:N` is applied to a potentially-problematic active token.

```

19261 \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
19262 {
19263   #3
19264   \if_false: { \fi:
19265     \exp_after:wN \exp_after:wN \exp_after:wN \exp_end:
19266     \exp_after:wN \exp_after:wN
19267     \if_case:w \tex_numexpr:D 13 - #2
19268       \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
19269       \exp_after:wN \exp_after:wN \exp_after:wN \scan_stop:
19270       \exp_after:wN \exp_after:wN \exp_after:wN \exp_not:N
19271       \cs:w c__char_ \__char_int_to_roman:w #1 _tl \cs_end:
19272     }
19273     \fi:
19274   }
19275   \fi:
19276 \group_end:

```

(End of definition for `\char_generate:nn` and others. This function is documented on page 196.)

`\c_catcode_active_space_tl` While `\char_generate:nn` can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code would need to avoid all spaces, making it quite unreadable. Instead we use the primitive `\tex_lowercase:D` trick.

```

19277 \group_begin:
19278   \char_set_catcode_active:N *
19279   \char_set_lccode:nn { '*' } { '\ }
19280   \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
19281 \group_end:

```

(End of definition for `\c_catcode_active_space_tl`. This variable is documented on page 196.)

`\c_catcode_other_space_tl` Create a space with category code 12: an “other” space.

```

19282 \tl_const:Ne \c_catcode_other_space_tl { \char_generate:nn { '\ } { 12 } }

```

(End of definition for `\c_catcode_other_space_tl`. This function is documented on page 197.)

62.4 Generic tokens

```

19283 (@@=token)

```

`\s__token_mark` Internal scan marks.

```

\s__token_stop 19284 \scan_new:N \s__token_mark
19285 \scan_new:N \s__token_stop

```

(End of definition for `\s__token_mark` and `\s__token_stop`.)

`\token_to_meaning:N` These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

`\token_to_meaning:c`
`\token_to_str:N`
`\token_to_str:c`

(End of definition for `\token_to_meaning:N` and `\token_to_str:N`. These functions are documented on page 201.)

`\token_to_catcode:N` The macro works by comparing the input token with `\if_catcode:w` with all valid category codes. Since the most common tokens in an average argument list are of category 11 or 12 those are tested first. And since a space and braces are no ordinary N-type arguments, and only control sequences let to those categories can match them they are tested last.

```

19286 \cs_new:Npn \token_to_catcode:N
19287   { \int_value:w \group_align_safe_begin: \_token_to_catcode:N }
19288 \cs_new:Npn \_token_to_catcode:N #1
19289   {
19290     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
19291       11
19292     \else:
19293       \if_catcode:w \exp_not:N #1 \c_catcode_other_token
19294         12
19295       \else:
19296         \if_catcode:w \exp_not:N #1 \c_math_toggle_token
19297           3
19298         \else:
19299           \if_catcode:w \exp_not:N #1 \c_alignment_token
19300             4
19301           \else:
19302             \if_catcode:w \exp_not:N #1 ##
19303               6
19304             \else:
19305               \if_catcode:w \exp_not:N #1 \c_math_superscript_token
19306                 7
19307               \else:
19308                 \if_catcode:w \exp_not:N #1 \c_math_subscript_token
19309                   8
19310                 \else:
19311                   \if_catcode:w \exp_not:N #1 \c_group_begin_token
19312                     1
19313                   \else:
19314                     \if_catcode:w \exp_not:N #1 \c_group_end_token
19315                       2
19316                     \else:
19317                       \if_catcode:w \exp_not:N #1 \c_space_token
19318                         10
19319                       \else:
19320                         \token_if_cs:NTF #1 { 16 } { 13 }
19321                         \fi:
19322                         \fi:
19323                         \fi:
19324                         \fi:
19325                         \fi:
19326                         \fi:
19327                         \fi:
19328                         \fi:
19329                         \fi:
19330                       \fi:
19331                     \group_align_safe_end:
19332                   \exp_stop_f:
19333                 }

```

(End of definition for `\token_to_catcode:N` and `_token_to_catcode:N`. This function is documented on page 201.)

`\c_group_begin_token` We define these useful tokens. For the brace and space tokens things have to be done by hand: the formal argument spec. for `\cs_new_eq:NN` does not cover them so we do things by hand. (As currently coded it would *work* with `\cs_new_eq:NN` but that's not really a great idea to show off: we want people to stick to the defined interfaces and that includes us.) So that these few odd names go into the log when appropriate there is a need to hand-apply the `_kernel_chk_if_free_cs:N` check.

```

19334 \group_begin:
19335   \_kernel_chk_if_free_cs:N \c_group_begin_token
19336   \tex_global:D \tex_let:D \c_group_begin_token {
19337     \_kernel_chk_if_free_cs:N \c_group_end_token
19338     \tex_global:D \tex_let:D \c_group_end_token }
19339   \char_set_catcode_math_toggle:N \*
19340   \cs_new_eq:NN \c_math_toggle_token *
19341   \char_set_catcode_alignment:N \*
19342   \cs_new_eq:NN \c_alignment_token *
19343   \cs_new_eq:NN \c_parameter_token #
19344   \cs_new_eq:NN \c_math_superscript_token ^
19345   \char_set_catcode_math_subscript:N \*
19346   \cs_new_eq:NN \c_math_subscript_token *
19347   \_kernel_chk_if_free_cs:N \c_space_token
19348   \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
19349   \cs_new_eq:NN \c_catcode_letter_token a
19350   \cs_new_eq:NN \c_catcode_other_token 1
19351 \group_end:

```

(End of definition for `\c_group_begin_token` and others. These functions are documented on page 200.)

`\c_catcode_active_tl` Not an implicit token!

```

19352 \group_begin:
19353   \char_set_catcode_active:N \*
19354   \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
19355 \group_end:

```

(End of definition for `\c_catcode_active_tl`. This variable is documented on page 200.)

62.5 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

`\token_if_group_begin:NTF`

```

19356 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
19357 {
19358   \if_catcode:w \exp_not:N #1 \c_group_begin_token
19359   \prg_return_true: \else: \prg_return_false: \fi:
19360 }

```

(End of definition for `\token_if_group_begin:NTF`. This function is documented on page 201.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.

`\token_if_group_end:NTF`

```

19361 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
19362 {

```

```

19363   \if_catcode:w \exp_not:N #1 \c_group_end_token
19364   \prg_return_true: \else: \prg_return_false: \fi:
19365 }

```

(End of definition for `\token_if_group_end:NTF`. This function is documented on page 201.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.

`\token_if_math_toggle:NTF`

```

19366 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
19367 {
19368   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
19369   \prg_return_true: \else: \prg_return_false: \fi:
19370 }

```

(End of definition for `\token_if_math_toggle:NTF`. This function is documented on page 202.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.

`\token_if_alignment:NTF`

```

19371 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
19372 {
19373   \if_catcode:w \exp_not:N #1 \c_alignment_token
19374   \prg_return_true: \else: \prg_return_false: \fi:
19375 }

```

(End of definition for `\token_if_alignment:NTF`. This function is documented on page 202.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.
`\token_if_parameter:NTF` We have to trick TeX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they remain after the group.

```

19376 \group_begin:
19377 \cs_set_eq:NN \c_parameter_token \scan_stop:
19378 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
19379 {
19380   \if_catcode:w \exp_not:N #1 \c_parameter_token
19381   \prg_return_true: \else: \prg_return_false: \fi:
19382 }
19383 \group_end:

```

(End of definition for `\token_if_parameter:NTF`. This function is documented on page 202.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

`\token_if_math_superscript:NTF`

```

19384 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
19385 { p , T , F , TF }
19386 {
19387   \if_catcode:w \exp_not:N #1 \c_math_superscript_token
19388   \prg_return_true: \else: \prg_return_false: \fi:
19389 }

```

(End of definition for `\token_if_math_superscript:NTF`. This function is documented on page 202.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_`
`\token_if_math_subscript:NTF` token for this.

```
19390 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
19391 {
19392   \if_catcode:w \exp_not:N #1 \c_math_subscript_token
19393   \prg_return_true: \else: \prg_return_false: \fi:
19394 }
```

(End of definition for \token_if_math_subscript:NTF. This function is documented on page 202.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.
`\token_if_space:NTF`

```
19395 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
19396 {
19397   \if_catcode:w \exp_not:N #1 \c_space_token
19398   \prg_return_true: \else: \prg_return_false: \fi:
19399 }
```

(End of definition for \token_if_space:NTF. This function is documented on page 202.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.
`\token_if_letter:NTF`

```
19400 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
19401 {
19402   \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
19403   \prg_return_true: \else: \prg_return_false: \fi:
19404 }
```

(End of definition for \token_if_letter:NTF. This function is documented on page 202.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token`
`\token_if_other:NTF` for this.

```
19405 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
19406 {
19407   \if_catcode:w \exp_not:N #1 \c_catcode_other_token
19408   \prg_return_true: \else: \prg_return_false: \fi:
19409 }
```

(End of definition for \token_if_other:NTF. This function is documented on page 202.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_catcode_active_tl` for
`\token_if_active:NTF` this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to
`\exp_not:N *`, where `*` is active.

```
19410 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
19411 {
19412   \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
19413   \prg_return_true: \else: \prg_return_false: \fi:
19414 }
```

(End of definition for \token_if_active:NTF. This function is documented on page 202.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.
`\token_if_eq_meaning:NNTF`

```
19415 \prg_new_eq_conditional:NNn \token_if_eq_meaning:NN \cs_if_eq:NN
19416 { p , T , F , TF }
```

(End of definition for \token_if_eq_meaning:NNTF. This function is documented on page 203.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

```
\token_if_eq_catcode:NNTF
19417 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
19418 {
19419   \if_catcode:w \exp_not:N #1 \exp_not:N #2
19420   \prg_return_true: \else: \prg_return_false: \fi:
19421 }
```

(End of definition for `\token_if_eq_catcode:NNTF`. This function is documented on page 202.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

```
\token_if_eq_charcode:NNTF
19422 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
19423 {
19424   \if_charcode:w \exp_not:N #1 \exp_not:N #2
19425   \prg_return_true: \else: \prg_return_false: \fi:
19426 }
```

(End of definition for `\token_if_eq_charcode:NNTF`. This function is documented on page 203.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` always outputs something like
`\token_if_macro:NNTF` `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`.
`__token_if_macro_p:w` However, this can fail the five `\..mark` primitives, whose meaning has the form
`\..mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in L^AT_EX3) before the string `macro:.` We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m..`

Both `ma` and `:` must be of category code 12 (other), so are detokenized.

```
19427 \use:e
19428 {
19429   \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N #1
19430   { p , T , F , TF }
19431   {
19432     \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
19433     \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma : }
19434     \s__token_stop
19435   }
19436   \cs_new:Npn \exp_not:N \__token_if_macro_p:w
19437   #1 \tl_to_str:n { ma } #2 \c_colon_str #3 \s__token_stop
19438 }
19439 {
19440   \str_if_eq:nnTF { #2 } { cro }
19441   { \prg_return_true: }
19442   { \prg_return_false: }
19443 }
```

(End of definition for `\token_if_macro:NNTF` and `__token_if_macro_p:w`. This function is documented on page 203.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as `\token_if_letter:N` etc. We use `\scan_stop:` for this.

`\token_if_cs:NTF`

```

19444 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
19445   {
19446     \if_catcode:w \exp_not:N #1 \scan_stop:
19447     \prg_return_true: \else: \prg_return_false: \fi:
19448   }

```

(End of definition for `\token_if_cs:NTF`. This function is documented on page 203.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX temporarily converts `\exp_not:N` $\langle token \rangle$ into `\scan_stop:` if $\langle token \rangle$ is expandable. An undefined token is not considered as expandable. No problem nesting the conditionals, since the third #1 is only skipped if it is non-expandable (hence not part of T_EX's conditional apparatus).

`\token_if_expandable:NTF`

```

19449 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
19450   {
19451     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
19452     \prg_return_false:
19453   \else:
19454     \if_cs_exist:N #1
19455     \prg_return_true:
19456   \else:
19457     \prg_return_false:
19458   \fi:
19459 \fi:
19460 }

```

(End of definition for `\token_if_expandable:NTF`. This function is documented on page 203.)

`__token_delimit_by_char:w` These auxiliary functions are used below to define some conditionals which detect whether the `\meaning` of their argument begins with a particular string. Each auxiliary takes an argument delimited by a string, a second one delimited by `\s__token_stop`, and returns the first one and its delimiter. This result is eventually compared to another string. Note that the “font” auxiliary is delimited by a space followed by “font”. This avoids an unnecessary check for the `\font` primitive below.

`__token_delimit_by_count:w`
`__token_delimit_by_dimen:w`
`__token_delimit_by_ufont:w`
`__token_delimit_by_macro:w`
`__token_delimit_by_muskip:w`
`__token_delimit_by_skip:w`
`__token_delimit_by_toks:w`

```

19461 \group_begin:
19462 \cs_set_protected:Npn \__token_tmp:w #1
19463   {
19464     \use:e
19465     {
19466       \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
19467       ##1 \tl_to_str:n {#1} ##2 \s__token_stop
19468       { ##1 \tl_to_str:n {#1} }
19469     }
19470   }
19471 \__token_tmp:w { char" }
19472 \__token_tmp:w { count }
19473 \__token_tmp:w { dimen }
19474 \__token_tmp:w { ~ font }
19475 \__token_tmp:w { macro }
19476 \__token_tmp:w { muskip }
19477 \__token_tmp:w { skip }
19478 \__token_tmp:w { toks }
19479 \group_end:

```

(End of definition for `_token_delimit_by_char:w` and others.)

`\token_if_chardef_p:N` Each of these conditionals tests whether its argument's `\meaning` starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the `\meaning` to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using `\str_if_eq:eeTF` to the result of applying `\token_to_str:N` to a control sequence. Second, the `\meaning` of primitives such as `\dimen` or `\dimendef` starts in the same way as registers such as `\dimen123`, so they must be tested for.

Characters used as delimiters must have catcode 12 and are obtained through `\tl_to_str:n`. This requires doing all definitions within `e`-expansion. The temporary function `_token_tmp:w` used to define each conditional receives three arguments: the name of the conditional, the auxiliary's delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary's result. Note that the `\meaning` of a protected long macro starts with `\protected\long macro`, with no space after `\protected` but a space after `\long`, hence the mixture of `\token_to_str:N` and `\tl_to_str:n`.

For the first six conditionals, `\cs_if_exist:cT` turns out to be `false` (thanks to the leading space for `font`), and the code boils down to a string comparison between the result of the auxiliary on the `\meaning` of the conditional's argument `####1`, and `#3`. Both are evaluated at run-time, as this is important to get the correct escape character.

The other five conditionals have additional code that compares the argument `####1` to two `TEX` primitives which would wrongly be recognized as registers otherwise. Despite using `TEX`'s primitive conditional construction, this does not break when `####1` is itself a conditional, because branches of the conditionals are only skipped if `####1` is one of the two primitives that are tested for (which are not `TEX` conditionals).

```

19480 \group_begin:
19481 \cs_set_protected:Npn \_token_tmp:w #1#2#3
19482 {
19483   \use:e
19484   {
19485     \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ##1
19486     { p , T , F , TF }
19487     {
19488       \cs_if_exist:cT { tex_ #2 :D }
19489       {
19490         \exp_not:N \if_meaning:w ##1 \exp_not:c { tex_ #2 :D }
19491         \exp_not:N \prg_return_false:
19492         \exp_not:N \else:
19493         \exp_not:N \if_meaning:w ##1 \exp_not:c { tex_ #2 def:D }
19494         \exp_not:N \prg_return_false:
19495         \exp_not:N \else:
19496       }
19497       \exp_not:N \str_if_eq:eeTF
19498       {
19499         \exp_not:N \exp_after:wN
19500         \exp_not:c { \_token_delimit_by_ #2 :w }
19501         \exp_not:N \token_to_meaning:N ##1
19502         ? \tl_to_str:n {#2} \s_token_stop

```

```

19503     }
19504     { \exp_not:n {#3} }
19505     { \exp_not:N \prg_return_true: }
19506     { \exp_not:N \prg_return_false: }
19507     \cs_if_exist:cT { tex_ #2 :D }
19508     {
19509         \exp_not:N \fi:
19510         \exp_not:N \fi:
19511     }
19512 }
19513 }
19514 }
19515 \__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
19516 \__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
19517 \__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
19518 \__token_tmp:w { protected_macro } { macro }
19519     { \tl_to_str:n { \protected } macro }
19520 \__token_tmp:w { protected_long_macro } { macro }
19521     { \token_to_str:N \protected \tl_to_str:n { \long } macro }
19522 \__token_tmp:w { font_selection } { ~ font } { select ~ font }
19523 \__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
19524 \__token_tmp:w { int_register } { count } { \token_to_str:N \count }
19525 \__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
19526 \__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
19527 \__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
19528 \group_end:

```

(End of definition for `\token_if_chardef:NTF` and others. These functions are documented on page 203.)

`\token_if_primitive_p:N`

`\token_if_primitive:NTF`

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., the letter A), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\tex_undefined:D` is not a primitive, but its meaning is undefined, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\s__token_stop`.

The meaning of each one of the five `\...mark` primitives has the form `<letters>:<user material>`. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either `"`, or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than ‘A’ (this is not quite a test for “only letters”,

but is close enough to work in this context). If this first character is : then we have a primitive, or `\tex_undefined:D`, and if it is " or a digit, then the token is not a primitive.

For LuaTeX we use a different implementation which just looks at the command code for the token and compares it to a list of non-primitives. Again, `\nullfont` is a special case because it is the only primitive with the normally non-primitive `set_font` command code.

In LuaMetaTeX some of the command names are different, so we check for both versions. The first one is always the LuaTeX version.

```

19529 \sys_if_engine luatex:TF
19530 {
19531 </tex>
19532 <*lua>
19533 do
19534   local get_next = token.get_next
19535   local get_command = token.get_command
19536   local get_index = token.get_index
19537   local get_mode = token.get_mode or token.get_index
19538   local cmd = command_id
19539   local set_font = cmd'get_font'
19540   local biggest_char = token.biggest_char and token.biggest_char()
19541                       or status.getconstants().max_character_code
19542
19543   local mode_below_biggest_char = {}
19544   local index_not_nil = {}
19545   local mode_not_null = {}
19546   local non_primitive = {
19547     [cmd'left_brace'] = true,
19548     [cmd'right_brace'] = true,
19549     [cmd'math_shift'] = true,
19550     [cmd'mac_param' or cmd'parameter'] = mode_below_biggest_char,
19551     [cmd'sup_mark' or cmd'superscript'] = true,
19552     [cmd'sub_mark' or cmd'subscript'] = true,
19553     [cmd'endv' or cmd'ignore'] = true,
19554     [cmd'spacer'] = true,
19555     [cmd'letter'] = true,
19556     [cmd'other_char'] = true,
19557     [cmd'tab_mark' or cmd'alignment_tab'] = mode_below_biggest_char,
19558     [cmd'char_given'] = true,
19559     [cmd'math_given' or 'math_char_given'] = true,
19560     [cmd'xmath_given' or 'math_char_xgiven'] = true,
19561     [cmd'set_font'] = mode_not_null,
19562     [cmd'undefined_cs'] = true,
19563     [cmd'call'] = true,
19564     [cmd'long_call' or cmd'protected_call'] = true,
19565     [cmd'outer_call' or cmd'tolerant_call'] = true,
19566     [cmd'long_outer_call' or cmd'tolerant_protected_call'] = true,
19567     [cmd'assign_glue' or cmd'register_glue'] = index_not_nil,
19568     [cmd'assign_mu_glue' or cmd'register_mu_glue' or cmd'register_mu_glue'] = index_not_nil,
19569     [cmd'assign_toks' or cmd'register_toks'] = index_not_nil,
19570     [cmd'assign_int' or cmd'register_int' or cmd'register_integer'] = index_not_nil,
19571     [cmd'assign_attr' or cmd'register_attribute'] = true,
19572     [cmd'assign_dimen' or cmd'register_dimen' or cmd'register_dimension'] = index_not_nil,
19573   }

```

```

19574
19575 luacmd("__token_if_primitive_lua:N", function()
19576   local tok = get_next()
19577   local is_non_primitive = non_primitive[get_command(tok)]
19578   return put_next(
19579     is_non_primitive == true
19580     and false_tok
19581   or is_non_primitive == nil
19582     and true_tok
19583   or is_non_primitive == mode_not_null
19584     and (get_mode(tok) == 0 and true_tok or false_tok)
19585   or is_non_primitive == index_not_nil
19586     and (get_index(tok) and false_tok or true_tok)
19587   or is_non_primitive == mode_below_biggest_char
19588     and (get_mode(tok) > biggest_char and true_tok or false_tok))
19589   end, "global")
19590 end
19591  $\langle$ /lua $\rangle$ 
19592  $\langle$ *tex $\rangle$ 
19593   \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
19594   {
19595     \__token_if_primitive_lua:N #1
19596   }
19597 }
19598 {
19599   \tex_global:D \tex_chardef:D \c__token_A_int = 'A ~ %
19600   \use:e
19601   {
19602     \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N #1
19603     { p , T , F , TF }
19604     {
19605       \exp_not:N \token_if_macro:NTF #1
19606       \exp_not:N \prg_return_false:
19607       {
19608         \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
19609         \exp_not:N \token_to_meaning:N #1
19610         \tl_to_str:n { : : : } \s__token_stop #1
19611       }
19612     }
19613     \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
19614     #1#2 #3 \c_colon_str #4 \s__token_stop
19615     {
19616       \exp_not:N \tl_if_empty:oTF
19617       { \exp_not:N \__token_if_primitive_space:w #3 ~ }
19618       {
19619         \exp_not:N \__token_if_primitive_loop:N #3
19620         \c_colon_str \s__token_stop
19621       }
19622       { \exp_not:N \__token_if_primitive_nullfont:N }
19623     }
19624   }
19625   \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
19626   \cs_new:Npn \__token_if_primitive_nullfont:N #1
19627   {

```

```

19628     \if_meaning:w \tex_nullfont:D #1
19629     \prg_return_true:
19630   \else:
19631     \prg_return_false:
19632   \fi:
19633 }
19634 \cs_new:Npn \__token_if_primitive_loop:N #1
19635 {
19636   \if_int_compare:w '#1 < \c__token_A_int %
19637     \exp_after:wN \__token_if_primitive:Nw
19638     \exp_after:wN #1
19639   \else:
19640     \exp_after:wN \__token_if_primitive_loop:N
19641   \fi:
19642 }
19643 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \s_token_stop
19644 {
19645   \if:w : #1
19646     \exp_after:wN \__token_if_primitive_undefined:N
19647   \else:
19648     \prg_return_false:
19649     \exp_after:wN \use_none:n
19650   \fi:
19651 }
19652 \cs_new:Npn \__token_if_primitive_undefined:N #1
19653 {
19654   \if_cs_exist:N #1
19655     \prg_return_true:
19656   \else:
19657     \prg_return_false:
19658   \fi:
19659 }
19660 }

```

(End of definition for `\token_if_primitive:NTF` and others. This function is documented on page 204.)

```

\token_case_catcode:Nn
\token_case_catcode:NnTF
\token_case_charcode:Nn
\token_case_charcode:NnTF
\token_case_meaning:Nn
\token_case_meaning:NnTF
  \__token_case:NNnTF
  \__token_case:NNw
  \__token_case_end:nw

```

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit “end of recursion” marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the true or false branch code is inserted.

```

19661 \cs_new:Npn \token_case_catcode:Nn #1#2
19662   { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} { } { } }
19663 \cs_new:Npn \token_case_catcode:NnT #1#2#3
19664   { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} {#3} { } }
19665 \cs_new:Npn \token_case_catcode:NnF #1#2
19666   { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} { } }
19667 \cs_new:Npn \token_case_catcode:NnTF
19668   { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF }
19669 \cs_new:Npn \token_case_charcode:Nn #1#2
19670   { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } { } }
19671 \cs_new:Npn \token_case_charcode:NnT #1#2#3
19672   { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} {#3} { } }
19673 \cs_new:Npn \token_case_charcode:NnF #1#2

```

```

19674 { \exp:w \_token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } }
19675 \cs_new:Npn \token_case_charcode:NnTF
19676 { \exp:w \_token_case:NNnTF \token_if_eq_charcode:NNTF }
19677 \cs_new:Npn \token_case_meaning:Nn #1#2
19678 { \exp:w \_token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } { } }
19679 \cs_new:Npn \token_case_meaning:NnT #1#2#3
19680 { \exp:w \_token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} {#3} { } }
19681 \cs_new:Npn \token_case_meaning:NnF #1#2
19682 { \exp:w \_token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } }
19683 \cs_new:Npn \token_case_meaning:NnTF
19684 { \exp:w \_token_case:NNnTF \token_if_eq_meaning:NNTF }
19685 \cs_new:Npn \_token_case:NNnTF #1#2#3#4#5
19686 {
19687   \_token_case:NNw #1 #2 #3 #2 { }
19688   \s__token_mark {#4}
19689   \s__token_mark {#5}
19690   \s__token_stop
19691 }
19692 \cs_new:Npn \_token_case:NNw #1#2#3#4
19693 {
19694   #1 #2 #3
19695   { \_token_case_end:nw {#4} }
19696   { \_token_case:NNw #1 #2 }
19697 }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare `\s__token_mark` and the **false** branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first `\s__token_mark` and so #4 is the **false** code (the **true** code is mopped up by #3).

```

19698 \cs_new:Npn \_token_case_end:nw #1#2#3 \s__token_mark #4#5 \s__token_stop
19699 { \exp_end: #1 #4 }

```

(End of definition for `\token_case_catcode:NnTF` and others. These functions are documented on page 205.)

62.6 Peeking ahead at the next token

```

19700 <@@=peek>

```

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

\l_peek_token Storage tokens which are publicly documented: the token peeked.

\g_peek_token 19701 \cs_new_eq:NN \l_peek_token ?
19702 \cs_new_eq:NN \g_peek_token ?

(End of definition for \l_peek_token and \g_peek_token. These variables are documented on page 205.)

\l__peek_search_token The token to search for as an implicit token: *cf.* `\l__peek_search_tl`.

19703 \cs_new_eq:NN \l__peek_search_token ?

(End of definition for \l__peek_search_token.)

\l__peek_search_tl The token to search for as an explicit token: *cf.* `\l__peek_search_token`.

19704 \tl_new:N \l__peek_search_tl

(End of definition for \l__peek_search_tl.)

__peek_true:w Functions used by the branching and space-stripping code.

__peek_true_aux:w 19705 \cs_new:Npn __peek_true:w { }

__peek_false:w 19706 \cs_new:Npn __peek_true_aux:w { }

__peek_tmp:w 19707 \cs_new:Npn __peek_false:w { }
19708 \cs_new:Npn __peek_tmp:w { }

(End of definition for __peek_true:w and others.)

\s__peek_mark Internal scan marks.

\s__peek_stop 19709 \scan_new:N \s__peek_mark
19710 \scan_new:N \s__peek_stop

(End of definition for \s__peek_mark and \s__peek_stop.)

__peek_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.

19711 \cs_new:Npn __peek_use_none_delimit_by_s_stop:w #1 \s__peek_stop { }

(End of definition for __peek_use_none_delimit_by_s_stop:w.)

\peek_after:Nw Simple wrappers for `\futurelet`: no arguments absorbed here.

\peek_gafter:Nw 19712 \cs_new_protected:Npn \peek_after:Nw
19713 { \tex_futurelet:D \l_peek_token }
19714 \cs_new_protected:Npn \peek_gafter:Nw
19715 { \tex_global:D \tex_futurelet:D \g_peek_token }

(End of definition for \peek_after:Nw and \peek_gafter:Nw. These functions are documented on page 205.)

__peek_true_remove:w A function to remove the next token and then regain control.

19716 \cs_new_protected:Npn __peek_true_remove:w
19717 {
19718 \tex_afterassignment:D __peek_true_aux:w
19719 \cs_set_eq:NN __peek_tmp:w
19720 }

(End of definition for __peek_true_remove:w.)

`\peek_remove_spaces:n` Repeatedly use `__peek_true_remove:w` to remove a space and call `__peek_true_au-
__peek_remove_spaces:` `aux:w`.

```

19721 \cs_new_protected:Npn \peek_remove_spaces:n #1
19722 {
19723   \cs_set:Npe \__peek_false:w { \exp_not:n {#1} }
19724   \group_align_safe_begin:
19725   \cs_set:Npn \__peek_true_aux:w { \peek_after:Nw \__peek_remove_spaces: }
19726   \__peek_true_aux:w
19727 }
19728 \cs_new_protected:Npn \__peek_remove_spaces:
19729 {
19730   \if_meaning:w \l_peek_token \c_space_token
19731     \exp_after:wN \__peek_true_remove:w
19732   \else:
19733     \group_align_safe_end:
19734     \exp_after:wN \__peek_false:w
19735   \fi:
19736 }

```

(End of definition for `\peek_remove_spaces:n` and `__peek_remove_spaces:`. This function is documented on page 206.)

`\peek_remove_filler:n` Here we expand the input, removing spaces and `\scan_stop:` tokens until we reach a non-expandable token. At that stage we re-insert the payload. To deal with the problem of `&` tokens, we have to put the align-safe group in the correct place.

```

\__peek_remove_filler:w
\__peek_remove_filler:
  \__peek_remove_filler_expand:w
19737 \cs_new_protected:Npn \peek_remove_filler:n #1
19738 {
19739   \cs_set:Npn \__peek_true_aux:w { \__peek_remove_filler:w }
19740   \cs_set:Npe \__peek_false:w
19741   {
19742     \exp_not:N \group_align_safe_end:
19743     \exp_not:n {#1}
19744   }
19745   \group_align_safe_begin:
19746   \__peek_remove_filler:w
19747 }
19748 \cs_new_protected:Npn \__peek_remove_filler:w
19749 {
19750   \exp_after:wN \peek_after:Nw \exp_after:wN \__peek_remove_filler:
19751   \exp:w \exp_end_continue_f:w
19752 }

```

Here we can nest conditionals as `\l_peek_token` is only skipped over in the nested one if it's a space: no problems with conditionals or outer tokens.

```

19753 \cs_new_protected:Npn \__peek_remove_filler:
19754 {
19755   \if_catcode:w \exp_not:N \l_peek_token \c_space_token
19756     \exp_after:wN \__peek_true_remove:w
19757   \else:
19758     \if_meaning:w \l_peek_token \scan_stop:
19759       \exp_after:wN \exp_after:wN \exp_after:wN
19760       \__peek_true_remove:w
19761     \else:
19762       \exp_after:wN \exp_after:wN \exp_after:wN

```

```

19763         \__peek_remove_filler_expand:w
19764         \fi:
19765     \fi:
19766 }

```

To deal with undefined control sequences in the same way T_EX does, we need to check for expansion manually.

```

19767 \cs_new_protected:Npn \__peek_remove_filler_expand:w
19768 {
19769     \exp_after:wN \if_meaning:w \exp_not:N \l_peek_token \l_peek_token
19770     \exp_after:wN \__peek_false:w
19771     \else:
19772     \exp_after:wN \__peek_remove_filler:w
19773     \fi:
19774 }

```

(End of definition for `\peek_remove_filler:n` and others. This function is documented on page 207.)

`__peek_token_generic_aux:NNNTF`

The generic functions store the test token in both implicit and explicit modes, and the `true` and `false` code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself. Here, `#1` is `__peek_true_remove:w` when removing the token and `__peek_true_aux:w` otherwise.

```

19775 \cs_new_protected:Npn \__peek_token_generic_aux:NNNTF #1#2#3#4#5
19776 {
19777     \group_align_safe_begin:
19778     \cs_set_eq:NN \l_peek_search_token #3
19779     \tl_set:Nn \l_peek_search_tl {#3}
19780     \cs_set:Npe \__peek_true_aux:w
19781     {
19782         \exp_not:N \group_align_safe_end:
19783         \exp_not:n {#4}
19784     }
19785     \cs_set_eq:NN \__peek_true:w #1
19786     \cs_set:Npe \__peek_false:w
19787     {
19788         \exp_not:N \group_align_safe_end:
19789         \exp_not:n {#5}
19790     }
19791     \peek_after:Nw #2
19792 }

```

(End of definition for `__peek_token_generic_aux:NNNTF`.)

`__peek_token_generic:NNTF`

For token removal there needs to be a call to the auxiliary function which does the work.

`__peek_token_remove_generic:NNTF`

```

19793 \cs_new_protected:Npn \__peek_token_generic:NNTF
19794 { \__peek_token_generic_aux:NNNTF \__peek_true_aux:w }
19795 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
19796 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
19797 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3
19798 { \__peek_token_generic:NNTF #1 #2 { } {#3} }
19799 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF
19800 { \__peek_token_generic_aux:NNNTF \__peek_true_remove:w }
19801 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
19802 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }

```

```

19803 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3
19804 { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End of definition for __peek_token_generic:NNTF and __peek_token_remove_generic:NNTF.)

__peek_execute_branches_meaning: The meaning test is straight forward.

```

19805 \cs_new:Npn \__peek_execute_branches_meaning:
19806 {
19807   \if_meaning:w \l_peek_token \l_peek_search_token
19808   \exp_after:wN \__peek_true:w
19809   \else:
19810   \exp_after:wN \__peek_false:w
19811   \fi:
19812 }

```

(End of definition for __peek_execute_branches_meaning:.)

__peek_execute_branches_catcode: The catcode and charcode tests are very similar, and in order to use the same auxiliaries
 __peek_execute_branches_charcode: we do something a little bit odd, firing \if_catcode:w and \if_charcode:w before
 __peek_execute_branches_catcode_aux: finding the operands for those tests, which are only given in the auxii:N and auxiii:
 __peek_execute_branches_catcode_auxii:N auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:
 __peek_execute_branches_catcode_auxiii:

- control sequences which are not equal to a non-active character token (*e.g.*, macro, primitive);
- active characters which are not equal to a non-active character token (*e.g.*, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using T_EX's \futurelet, because we can only access the \meaning of tokens in that way. In those cases, detected thanks to a comparison with \scan_stop:, we grab the following token, and compare it explicitly with the explicit search token stored in \l_peek_search_tl. The \exp_not:N prevents outer macros (coming from non- \LaTeX 3 code) from blowing up. In the third case, \l_peek_token is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

19813 \cs_new:Npn \__peek_execute_branches_catcode:
19814 { \if_catcode:w \__peek_execute_branches_catcode_aux: }
19815 \cs_new:Npn \__peek_execute_branches_charcode:
19816 { \if_charcode:w \__peek_execute_branches_catcode_aux: }
19817 \cs_new:Npn \__peek_execute_branches_catcode_aux:
19818 {
19819   \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
19820   \exp_after:wN \exp_after:wN
19821   \exp_after:wN \__peek_execute_branches_catcode_auxii:N
19822   \exp_after:wN \exp_not:N
19823   \else:
19824   \exp_after:wN \__peek_execute_branches_catcode_auxiii:
19825   \fi:
19826 }
19827 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1

```



```

19828 {
19829     \exp_not:N #1
19830     \exp_after:wN \exp_not:N \l__peek_search_tl
19831     \exp_after:wN \__peek_true:w
19832     \else:
19833     \exp_after:wN \__peek_false:w
19834     \fi:
19835     #1
19836 }
19837 \cs_new:Npn \__peek_execute_branches_catcode_auxiii:
19838 {
19839     \exp_not:N \l_peek_token
19840     \exp_after:wN \exp_not:N \l__peek_search_tl
19841     \exp_after:wN \__peek_true:w
19842     \else:
19843     \exp_after:wN \__peek_false:w
19844     \fi:
19845 }

```

(End of definition for `__peek_execute_branches_catcode:` and others.)

`\peek_catcode:NTF` The public functions themselves cannot be defined using `\prg_new_conditional:Npnn`.
`\peek_catcode_remove:NTF` Instead, the TF, T, F variants are defined in terms of corresponding variants of
`\peek_charcode:NTF` `__peek_token_generic:NNTF` or `__peek_token_remove_generic:NNTF`, with first argu-
`\peek_charcode_remove:NTF` ment one of `__peek_execute_branches_catcode:`, `__peek_execute_branches_-`
`\peek_meaning:NTF` `charcode:`, or `__peek_execute_branches_meaning:`.
`\peek_meaning_remove:NTF`

```

19846 \tl_map_inline:nn { { catcode } { charcode } { meaning } }
19847 {
19848     \tl_map_inline:nn { { } { _remove } }
19849     {
19850         \tl_map_inline:nn { { TF } { T } { F } }
19851         {
19852             \cs_new_protected:cpe { peek_ #1 ##1 :N #####1 }
19853             {
19854                 \exp_not:c { __peek_token ##1 _generic:NN #####1 }
19855                 \exp_not:c { __peek_execute_branches_ #1 : }
19856             }
19857         }
19858     }
19859 }

```

(End of definition for `\peek_catcode:NTF` and others. These functions are documented on page 206.)

`\peek_N_type:TF` All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens,
`__peek_execute_branches_N_type:` space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be
`__peek_N_type:w` outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old
`__peek_N_type_aux:nnw` trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the
token is one of the first three kinds of non-N-type tokens (explicit or implicit), thus we call
`__peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting
performance too much for non-outer tokens. The first filter is to search for `outer` in
the `\meaning` of `\l_peek_token`. If that is absent, `__peek_use_none_delimit_by_-`
`s_stop:w` cleans up, and we call `__peek_true:w`. Otherwise, the token can be a non-
outer macro or a primitive mark whose parameter or replacement text contains `outer`, it
can be the primitive `\outer`, or it can be an outer token. Macros and marks would have

ma in the part before the first occurrence of outer; the meaning of \outer has nothing after outer, contrarily to outer macros; and that covers all cases, calling __peek_true:w or __peek_false:w as appropriate. Here, there is no <search token>, so we feed a dummy \scan_stop: to the __peek_token_generic:NNTF function.

```

19860 \group_begin:
19861   \cs_set_protected:Npn \__peek_tmp:w #1 \s__peek_stop
19862   {
19863     \cs_new_protected:Npn \__peek_execute_branches_N_type:
19864     {
19865       \if_int_odd:w
19866         \if_catcode:w \exp_not:N \l_peek_token { \c_zero_int \fi:
19867         \if_catcode:w \exp_not:N \l_peek_token } \c_zero_int \fi:
19868         \if_meaning:w \l_peek_token \c_space_token \c_zero_int \fi:
19869         \c_one_int
19870         \exp_after:wN \__peek_N_type:w
19871         \token_to_meaning:N \l_peek_token
19872         \s__peek_mark \__peek_N_type_aux:nw
19873         #1 \s__peek_mark \__peek_use_none_delimit_by_s_stop:w
19874         \s__peek_stop
19875         \exp_after:wN \__peek_true:w
19876       \else:
19877         \exp_after:wN \__peek_false:w
19878       \fi:
19879     }
19880     \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \s__peek_mark ##3
19881     { ##3 {##1} {##2} }
19882   }
19883   \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \s__peek_stop
19884 \group_end:
19885 \cs_new_protected:Npn \__peek_N_type_aux:nw #1 #2 #3 \fi:
19886 {
19887   \fi:
19888   \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
19889   { \__peek_true:w }
19890   { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
19891 }
19892 \cs_new_protected:Npn \peek_N_type:TF
19893 {
19894   \__peek_token_generic:NNTF
19895   \__peek_execute_branches_N_type: \scan_stop:
19896 }
19897 \cs_new_protected:Npn \peek_N_type:T
19898 { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
19899 \cs_new_protected:Npn \peek_N_type:F
19900 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }

```

(End of definition for \peek_N_type:TF and others. This function is documented on page 207.)

```

19901 </tex>
19902 </package>

```

Chapter 63

l3prop implementation

The following test files are used for this code: `m3prop001`, `m3prop002`, `m3prop003`, `m3prop004`, `m3show001`.

```
19903 (*package)
```

```
19904 (@@=prop)
```

With the (default) flat data storage, a property list is a macro whose top-level expansion is of the form

```
\s__prop \__prop_chk:w \__prop_pair:wn <key1> \s__prop {<value1>}  
...  
\__prop_pair:wn <keyn> \s__prop {<valuen>}
```

where `\s__prop` is a scan mark (equal to `\scan_stop:`), `__prop_chk:w` produces a suitable error if the property list is used directly in the input stream, and `__prop_pair:wn` can be used to map through the property list.

With the linked data storage, each property list entry `<keyi>`–`<valuei>` is stored into a token list `__prop <prefix> <keyi>`. The `<prefix>` is one or more characters (no spaces), constructed automatically only once, when the property list is initially declared. The control sequence name does not conform to standard naming for variables because (1) this is an internal control sequence, not really a `expl3` variable; (2) keeping track of the scope `l` or `g` throughout all functions would be a pretty big mess, especially if users accidentally mix local and global use (we would have to always check for such mistakes, rather than only checking when suitable debug options are set); (3) shorter control sequence names use less memory and are quicker in case of hash collisions, which may matter since we are using many control sequences.

We need to enable mapping through such a property list, but without storing a list of all entries anywhere: this is achieved by making each of these token lists also store a pointer to the next entry. To enable efficient deletion, the token lists also store a pointer to the previous entry. This means we have a doubly-linked list. To avoid having to special-case the two ends of the doubly-linked list when deleting entries, we include as a zeroth entry in the doubly-linked list the property list variable itself, and we include as an $(n + 1)$ -th entry in the doubly-linked list an end-pointer `__prop <prefix>` (no trailing space, so it differs from an empty key). The space before `<prefix>` ensures there is no collision with other `l3prop` internal functions, even if we have very many linked property lists being defined.

The property list variable itself is a token list of the form

```
\__prop_flatten:w \__prop <prefix> \s__prop {<prefix>} \__prop <prefix> <key1>
```

Here, `__prop_flatten:w` serves as an efficiently recognized marker, and when `f`-expanded it is tasked with fully unpacking the property list into the same form as the default data storage so as to ease conversion. The `<prefix>` is used when looking up an entry. The token list `__prop <prefix>` (see below) contains a pointer to the last key to help insert a new entry. The pointer to `<key1>` is needed to start a mapping. The token list labeled by `<keyi>` is of the form

```
\use_none:n \__prop <prefix> <keyi-1> \__prop_pair:wn <keyi> \s__-
prop {<valuei>} \__prop <prefix> <keyi+1>
```

where the pointer to `<keyi-1>` is needed when deleting the `<keyi>`. Expanding this will run `__prop_pair:wn` on the `<keyi>`-`<valuei>` pair (for speed, `<keyi>` is kept as explicit tokens rather than slowly extracting it from a control sequence name), then move on to the next key, thus mapping through the whole list. The mapping is ended upon expanding `__prop <prefix>`, which is the token list

```
\use_none:n \__prop <prefix> <keyn>
```

Let us think about deleting the `<keyi>`. We need to update the `<keyi-1>` and `<keyi+1>` to point to each other instead of `<keyi>`. To edit the corresponding token lists, it is important that `__prop <prefix> <keyi>` be at the “same place” in the token lists also in the boundary cases $i = 1$ or $i = n$, namely as the second token, or as the second argument after `\s__prop`.

63.1 Internal auxiliaries

`__prop_tmp:w` Scratch macro, defined as needed, for instance to save `__prop_pair:wn` when concatenating.

```
19905 \cs_new_eq:NN \__prop_tmp:w ?
```

(End of definition for __prop_tmp:w.)

`\l__prop_internal_tl` Token list used in various places: for the prefix; when converting from flat to linked props; and to store the new key–value pair inserted by `\prop_put:Nnn`.

```
19906 \tl_new:N \l__prop_internal_tl
```

(End of definition for \l__prop_internal_tl.)

`\s__prop_mark` Internal scan marks.

```
\s__prop_stop 19907 \scan_new:N \s__prop_mark
```

```
19908 \scan_new:N \s__prop_stop
```

(End of definition for \s__prop_mark and \s__prop_stop.)

`\q__prop_recursion_tail` Internal recursion quarks.

```
\q__prop_recursion_stop 19909 \quark_new:N \q__prop_recursion_tail
```

```
19910 \quark_new:N \q__prop_recursion_stop
```

(End of definition for \q__prop_recursion_tail and \q__prop_recursion_stop.)

`__prop_if_recursion_tail_stop:n` Functions to query recursion quarks.
`__prop_if_recursion_tail_stop:o` 19911 `__kernel_quark_new_test:N __prop_if_recursion_tail_stop:n`
19912 `\cs_generate_variant:Nn __prop_if_recursion_tail_stop:n { o }`
(End of definition for `__prop_if_recursion_tail_stop:n` and `__prop_if_recursion_tail_stop:o`.)

63.2 Structure of a property list

`\s__prop` A private scan mark is used as a marker after each key, and at the very beginning of the property list.

19913 `\scan_new:N \s__prop`

(End of definition for `\s__prop`.)

`__prop_chk:w` This removes the flat property list from the input stream and complains about a bad use
`__prop_chk_loop:nw` of a property list. Since property lists do not have an end-marker, we slowly peek ahead
`__prop_chk_get:nw` in a loop. Speed does not matter since this is for an error situation. While `__prop_`
`pair:wn` does not keep a fixed definition, it always includes the internal `\s__prop` in its
argument specification, so that there is no risk of accidentally picking up a public token
instead of `__prop_pair:wn` when doing a meaning test. We collect the keys and values
to produce a more useful error message.

19914 `\cs_new_protected:Npn __prop_chk:w { __prop_chk_loop:nw { } }`
19915 `\cs_new_protected:Npn __prop_chk_loop:nw #1`
19916 `{`
19917 `\peek_meaning:NTF __prop_pair:wn`
19918 `{ __prop_chk_get:nw {#1} }`
19919 `{ \msg_error:nne { prop } { misused } {#1} }`
19920 `}`
19921 `\cs_new_protected:Npn __prop_chk_get:nw #1 __prop_pair:wn #2 \s__prop #3`
19922 `{ __prop_chk_loop:nw { #1 , ~ {#2} = { \tl_to_str:n {#3} } }`

(End of definition for `__prop_chk:w`, `__prop_chk_loop:nw`, and `__prop_chk_get:nw`.)

`__prop_pair:wn` Used as `__prop_pair:wn <key> \s__prop {<item>}` for both storage types, this internal
token starts each key–value pair in the property list. This default definition is changed
globally by any mapping function, so there is not much point trying to make it an error.
Instead, the error is produced by `__prop_chk:w`.

19923 `\cs_new:Npn __prop_pair:wn #1 \s__prop #2 { }`

(End of definition for `__prop_pair:wn`.)

`__prop_flatten:w` We implement here the fact that f-expanding a linked property list gives a flat property
list. Leaving a linked property list in the input stream will turn it into a flat property
list so that the error implemented by `__prop_chk:w` will correctly be triggered.

19924 `\cs_new_protected:Npn __prop_flatten:w #1 \s__prop #2#3`
19925 `{ \use:e { __prop_flatten_aux:N #3 } }`

(End of definition for `__prop_flatten:w`.)

`__prop_flatten:N` The main function `__prop_flatten:N` receives a linked property list and flattens it.
`__prop_flatten_aux:w` The auxiliary `__prop_flatten_aux:N` receives a pointer to the first key and flattens
`__prop_flatten_aux:N` the linked property list into a flat property list. This is only restricted-expandable
`__prop_flatten_loop:w` as it involves mapping through all of the property list's entries starting from $\langle key_1 \rangle$.
The looping function `__prop_flatten_loop:w` removes `use_none:n` and a backwards
pointer #2, leaves the key-value pair for `use:e` to receive, and calls itself again after
expanding the next key's token list. Its argument #3 is empty, except at the end where
it is the `use_none:nnnn` appearing in the definition of `__prop_flatten_aux:N`, which
ends the loop.

```

19926 \cs_new:Npn __prop_flatten:N #1
19927   { \exp_after:wN __prop_flatten_aux:w #1 }
19928 \cs_new:Npn __prop_flatten_aux:w #1 \s__prop #2 { __prop_flatten_aux:N }
19929 \cs_new:Npn __prop_flatten_aux:N #1
19930   {
19931     \s__prop __prop_chk:w
19932     \exp_after:wN __prop_flatten_loop:w #1
19933     \use_none:nnnn __prop_pair:wn \s__prop { }
19934   }
19935 \cs_new:Npn __prop_flatten_loop:w #1#2#3 __prop_pair:wn #4 \s__prop #5
19936   {
19937     #3
19938     \exp_not:n { __prop_pair:wn #4 \s__prop {#5} }
19939     \exp_after:wN __prop_flatten_loop:w
19940   }

```

(End of definition for `__prop_flatten:N` and others.)

`g__prop_prefix_int` Used to assign prefixes for each linked property list. It is converted to base `c__prop_`
`c__prop_basis_int` `basis_int`, then each digit is converted to a character, starting at ! (the character after
space).

```

19941 \int_new:N g__prop_prefix_int
19942 \int_const:Nn c__prop_basis_int { \c_max_char_int - '\! }

```

(End of definition for `g__prop_prefix_int` and `c__prop_basis_int`.)

`__prop_next_prefix:` Store in `l__prop_internal_tl` the conversion of `g__prop_prefix_int` to characters,
`__prop_to_prefix:n` and increment this integer for use in the next linked property list. No need to optimize
since this is only used when declaring the property list the first time. The aim here is to
make this string as short as we can, given the range of distinct characters available. This
speeds up the work of `cs:w ... cs_end:` that looks up keys in the hash table.

```

19943 \cs_new_protected:Npn __prop_next_prefix:
19944   {
19945     \tl_set:Nc l__prop_internal_tl
19946       { __prop_to_prefix:n { g__prop_prefix_int } }
19947     \int_gincr:N g__prop_prefix_int
19948   }
19949 \cs_new:Npn __prop_to_prefix:n #1
19950   {
19951     \int_compare:nNnTF {#1} > c__prop_basis_int
19952     {
19953       \exp_args:Nf __prop_to_prefix:n
19954         { \int_div_truncate:nn {#1} c__prop_basis_int }
19955       \exp_args:Nf __prop_to_prefix:n

```

```

19956         { \int_mod:nn {#1} \c__prop_basis_int }
19957     }
19958     { \char_generate:nn { '\! + #1 } { 12 } }
19959 }

```

(End of definition for `__prop_next_prefix:` and `__prop_to_prefix:n`.)

`__prop_if_flat:NTF` `__prop_if_flat_aux:w` We could either test for the presence of `__prop_chk:w` (flat property list) or of `__prop_flatten:w` (linked property list). We make the second choice; this way props that are accidentally `\relax` are treated as they were before. The auxiliary receives `\use_i:nn` or `\use_ii:nn` as #3. As a transitional fix we avoid erroring in case the prop is undefined (the `\exp_after:wN` is omitted in that case, taking the flat branch).

```

19960 \cs_new:Npn \__prop_if_flat:NTF #1
19961 {
19962     \prop_if_exist:NT #1
19963     \exp_after:wN \__prop_if_flat_aux:w #1
19964     \s__prop_mark \use_ii:nn
19965     \__prop_flatten:w \s__prop_mark \use_i:nn \s__prop_stop
19966 }
19967 \cs_new:Npn \__prop_if_flat_aux:w
19968     #1 \__prop_flatten:w #2 \s__prop_mark #3 #4 \s__prop_stop {#3}

```

(End of definition for `__prop_if_flat:NTF` and `__prop_if_flat_aux:w`.)

63.3 Allocation and initialisation

`\c_empty_prop` An empty flat prop.

```

19969 \tl_const:Nn \c_empty_prop { \s__prop \__prop_chk:w }

```

(End of definition for `\c_empty_prop`. This variable is documented on page 222.)

`\prop_new:N` Flat property lists are initialized with the value `\c_empty_prop`.

```

\prop_new:c
19970 \cs_new_protected:Npn \prop_new:N #1
19971 {
19972     \__kernel_chk_if_free_cs:N #1
19973     \cs_gset_eq:NN #1 \c_empty_prop
19974 }
19975 \cs_generate_variant:Nn \prop_new:N { c }

```

(End of definition for `\prop_new:N`. This function is documented on page 214.)

`\prop_new_linked:N` `\prop_new_linked:c` `__prop_new_linked:N` The auxiliary is used in `\prop_make_linked:N`. For linked property lists, get a new prefix in `\l__prop_internal_tl`, then use it to set up the internal structure: the last token in #1 is usually a pointer to the first key, which is here the end-pointer. That end-pointer has a pointer to the previous key (usually the last key), which is the variable #1 itself that begins the doubly-linked list.

```

19976 \cs_new_protected:Npn \prop_new_linked:N #1
19977 {
19978     \__kernel_chk_if_free_cs:N #1
19979     \__prop_new_linked:N #1
19980 }
19981 \cs_new_protected:Npn \__prop_new_linked:N #1
19982 {

```

```

19983  \__prop_next_prefix:
19984  \cs_gset_nopar:Npe #1
19985  {
19986    \__prop_flatten:w
19987    \exp_not:c { __prop ~ \l__prop_internal_tl }
19988    \s__prop { \l__prop_internal_tl }
19989    \exp_not:c { __prop ~ \l__prop_internal_tl }
19990  }
19991  \cs_gset_nopar:cpe { __prop ~ \l__prop_internal_tl }
19992  {
19993    \exp_not:N \use_none:n
19994    \exp_not:N #1
19995  }
19996  }
19997  \cs_generate_variant:Nn \prop_new_linked:N { c }

```

(End of definition for `\prop_new_linked:N` and `__prop_new_linked:N`. This function is documented on page 214.)

`\prop_clear:N` Clearing a flat property list is like declaring it anew, simply setting it equal to `\c_empty_prop`. For linked property lists we must clear all of the variables storing individual keys, which requires a loop. At each step of the loop, `__prop_clear_loop:Nw` receives `\cs_(g)set_eq:NN`, `\use_none:n`, the backwards pointer, an empty #4 (except at the end of the loop), and the key–value pair #5=#6 which we disregard. The looping auxiliary undefines the previous key’s token list (this includes the main token list, but that is fine because it is restored at the end) and calls itself after expanding the next key’s token list. The loop ends when #4 is `\use_none:nnnn`. After the loop, `__prop_clear:wNNN` correctly sets up the main variable #6 and the end-pointer #1. Importantly, this is done using `\cs_(g)set_nopar:Npe` and `\exp_not:n` because the almost-equivalent `\tl_set:Nn` would complain in debug mode about the fact that the main variable is undefined at this stage. Importantly, `__prop_clear_entries:NN` is used in the implementation of `\prop_set_eq:NN`.

```

20000  \cs_new_protected:Npn \prop_clear:N
20001  { \__prop_clear:NNN \cs_set_eq:NN \cs_set_nopar:Npe }
20002  \cs_generate_variant:Nn \prop_clear:N { c }
20003  \cs_new_protected:Npn \prop_gclear:N
20004  { \__prop_clear:NNN \cs_gset_eq:NN \cs_gset_nopar:Npe }
20005  \cs_generate_variant:Nn \prop_gclear:N { c }
20006  \cs_new_protected:Npn \__prop_clear:NNN #1#2#3
20007  {
20008    \__prop_if_flat:NTF #3
20009    { #1 #3 \c_empty_prop }
20010    { \exp_after:wN \__prop_clear:wNNN #3 #1 #2 #3 }
20011  }
20012  \cs_new_protected:Npn \__prop_clear:wNNN
20013  \__prop_flatten:w #1 \s__prop #2#3#4#5#6
20014  {
20015    \__prop_clear_entries:NN #4 #3
20016    #5 #6 { \exp_not:n { \__prop_flatten:w #1 \s__prop {#2} #1 } }
20017    #5 #1 { \exp_not:n { \use_none:n #6 } }
20018  }
20019  \cs_new_protected:Npn \__prop_clear_entries:NN #1#2
20020  {

```



```

20019 \exp_after:wN \_prop_clear_loop:Nw \exp_after:wN #1 #2
20020 \use_none:nmmm \_prop_pair:wn \s__prop { }
20021 }
20022 \cs_new_protected:Npn \_prop_clear_loop:Nw
20023 #1#2#3#4 \_prop_pair:wn #5 \s__prop #6
20024 {
20025 #1 #3 \tex_undefined:D
20026 #4
20027 \exp_after:wN \_prop_clear_loop:Nw
20028 \exp_after:wN #1
20029 }

```

(End of definition for `\prop_clear:N` and others. These functions are documented on page 214.)

```

\prop_clear_new:N A simple variation of the token list functions.
\prop_clear_new:c 20030 \cs_new_protected:Npn \prop_clear_new:N #1
\prop_gclear_new:N 20031 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new:c 20032 \cs_generate_variant:Nn \prop_clear_new:N { c }
\prop_clear_new_linked:N 20033 \cs_new_protected:Npn \prop_gclear_new:N #1
\prop_clear_new_linked:c 20034 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new_linked:N 20035 \cs_generate_variant:Nn \prop_gclear_new:N { c }
\prop_gclear_new_linked:c 20036 \cs_new_protected:Npn \prop_clear_new_linked:N #1
20037 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new_linked:N #1 } }
20038 \cs_generate_variant:Nn \prop_clear_new_linked:N { c }
20039 \cs_new_protected:Npn \prop_gclear_new_linked:N #1
20040 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new_linked:N #1 } }
20041 \cs_generate_variant:Nn \prop_gclear_new_linked:N { c }

```

(End of definition for `\prop_clear_new:N` and others. These functions are documented on page 214.)

```

\prop_set_eq:NN If both variables are accidentally the same variable (or equal flat property lists, as it
\prop_set_eq:cN turns out) we do nothing, otherwise the following code would lose all entries. If the
\prop_set_eq:Nc target variable #3 is a flat prop, either copy directly or flatten before copying. If it is a
\prop_set_eq:cc linked prop, we must clear it, then go through the entries in #4 to add them to #3.
\prop_gset_eq:NN 20042 \cs_new_protected:Npn \prop_set_eq:NN
\prop_gset_eq:cN 20043 { \_prop_set_eq:NNNN \cs_set_eq:NN \cs_set_nopar:Npe }
\prop_gset_eq:Nc 20044 \cs_generate_variant:Nn \prop_set_eq:NN { Nc , cN , cc }
\prop_gset_eq:cc 20045 \cs_new_protected:Npn \prop_gset_eq:NN
\_prop_set_eq:NNNN 20046 { \_prop_set_eq:NNNN \cs_gset_eq:NN \cs_gset_nopar:Npe }
\_prop_set_eq:wNNNN 20047 \cs_generate_variant:Nn \prop_gset_eq:NN { Nc , cN , cc }
\_prop_set_eq:nNnNN 20048 \cs_new_protected:Npn \_prop_set_eq:NNNN #1#2#3#4
\_prop_set_eq_loop:NNnw 20049 {
\_prop_set_eq_end:w 20050 \cs_if_eq:NMF #3#4
20051 {
20052 \_prop_if_flat:NTF #3
20053 {
20054 \_prop_if_flat:NTF #4
20055 { #1 #3 #4 }
20056 { #2 #3 { \_prop_flatten:N #4 } }
20057 }
20058 { \exp_after:wN \_prop_set_eq:wNNNN #3 #1#2#3#4 }
20059 }
20060 }
20061 \cs_new_protected:Npn \_prop_set_eq:wNNNN

```

```

20062 \__prop_flatten:w #1 \s__prop #2#3#4#5#6#7
20063 {
20064 \__prop_clear_entries:NN #4 #3
20065 \exp_args:Nf \__prop_set_eq:nNnNN {#7} #1 {#2} #5 #6
20066 }

```

We have used that `f`-expanding either type of prop gives a flat prop. At this stage `__prop_set_eq:nNnNN` receives the second variable as a flat prop, the end-pointer, the prefix, the suitable `\cs_(g)set_nopar:Npe` assignment, and the first variable itself. Remove the leading `\s__prop` and `__prop_chk:w` with `\use_i:nnn`, then start the loop.

```

20067 \cs_new_protected:Npn \__prop_set_eq:nNnNN #1#2#3#4#5
20068 {
20069 \use_i:nnn
20070 {
20071 \__prop_set_eq_loop:NNnw #5 #4 {#3}
20072 \__prop_flatten:w #2 \s__prop {#3}
20073 }
20074 #1
20075 \use_none:n \__prop_pair:wn ? \s__prop
20076 }

```

The looping function receives the current pointer `#1` (initially the variable itself), the defining function `#2` and the prefix `#3`, then a partial definition `#4` (which in later stages includes the backwards pointer), followed by the current value as `\s__prop {#5}`. It seeks the next key `#7` to construct in `\l__prop_internal_tl` the next pointer `__prop <prefix> <next key>` (the argument `#6` is empty, except at the end of the loop, where it is `\use_none:n` in such a way as to delete the `<space>` and `<next key>`). Then the token list (current pointer) `#1` is set-up to contain the partial definition and current value, as well as the newly constructed next pointer. After a line responsible for correctly ending the loop with `__prop_set_eq_end:w`, we loop, setting up the next definition, which starts with `\use_none:n` and a backwards pointer to `#1` followed by the `<next key> #7` and so on.

```

20077 \cs_new_protected:Npn \__prop_set_eq_loop:NNnw
20078 #1#2#3#4 \s__prop #5#6 \__prop_pair:wn #7 \s__prop
20079 {
20080 \tl_set:Ne \l__prop_internal_tl { \exp_not:c { __prop ~ #3 #6 ~ #7 } }
20081 #2 #1 { \exp_not:n { #4 \s__prop {#5} } \exp_not:o \l__prop_internal_tl }
20082 \use_none:n #6 \__prop_set_eq_end:w
20083 \exp_after:wN \__prop_set_eq_loop:NNnw \l__prop_internal_tl #2 {#3}
20084 \use_none:n #1 \__prop_pair:wn #7 \s__prop
20085 }

```

The end-code picks up what is needed to correctly assign the last token list (the end pointer), which is simply `\use_none:n __prop_<prefix><space><keyn>`.

```

20086 \cs_new_protected:Npn \__prop_set_eq_end:w
20087 \exp_after:wN \__prop_set_eq_loop:NNnw #1#2#3
20088 \use_none:n #4#5 \s__prop
20089 {
20090 \exp_after:wN #2 \l__prop_internal_tl { \exp_not:n { \use_none:n #4 } }
20091 }

```

(End of definition for `\prop_set_eq:NN` and others. These functions are documented on page 214.)

`\prop_make_flat:N`
`\prop_make_flat:c_d __prop_make_flat:Nn` The only interesting case is when given a linked prop. Clear the linked property list using `__prop_clear:wNNN` with local assignments (it does not matter since we are at

the outermost group level, and `\cs_set_eq:NN` is very slightly faster than its global version. Then store the contents (expanded preventively by `\exp_args:NNf`) with an assignment `\cs_set_nopar:Npe` that does not perform `l3debug` checks.

```

20092 \cs_new_protected:Npn \prop_make_flat:N #1
20093 {
20094   \int_compare:nNnTF { \tex_currentgrouplevel:D } = 0
20095   {
20096     \__prop_if_flat:NTF #1 { }
20097     { \exp_args:NNf \__prop_make_flat:Nn #1 {#1} }
20098   }
20099   {
20100     \msg_error:nnee { prop } { inner-make }
20101     { \token_to_str:N \prop_make_flat:N } { \token_to_str:N #1 }
20102   }
20103 }
20104 \cs_generate_variant:Nn \prop_make_flat:N { c }
20105 \cs_new_protected:Npn \__prop_make_flat:Nn #1#2
20106 {
20107   \exp_after:wN \__prop_clear:wNnN #1 \cs_set_eq:NN \cs_set_nopar:Npe #1
20108   \cs_set_nopar:Npe #1 { \exp_not:n {#2} }
20109 }

```

(End of definition for `\prop_make_flat:N` and `\prop_make_flat:c __prop_make_flat:Nn`. This function is documented on page 215.)

`\prop_make_linked:N`
`\prop_make_linked:c`
`__prop_make_linked:Nn`

The only interesting case is when given a flat prop. We expand the contents for later use. Then `__prop_new_linked:N` disregards that previous value of `#1` and initializes the linked prop. We can then use an auxiliary `__prop_set_eq:wNnNn` underlying `\prop_set_eq:MN`, with the prop contents saved as `\l__prop_internal_tl`. That step is a bit unsafe, as `\l__prop_internal_tl` (really, a flat prop here) is used within `__prop_set_eq:wNnNn` itself, but it is in fact expanded early enough to be ok.

```

20110 \cs_new_protected:Npn \prop_make_linked:N #1
20111 {
20112   \int_compare:nNnTF { \tex_currentgrouplevel:D } = 0
20113   {
20114     \__prop_if_flat:NTF #1
20115     { \exp_args:NNo \__prop_make_linked:Nn #1 {#1} } { }
20116   }
20117   {
20118     \msg_error:nnee { prop } { inner-make }
20119     { \token_to_str:N \prop_make_linked:N } { \token_to_str:N #1 }
20120   }
20121 }
20122 \cs_generate_variant:Nn \prop_make_linked:N { c }
20123 \cs_new_protected:Npn \__prop_make_linked:Nn #1#2
20124 {
20125   \__prop_new_linked:N #1
20126   \tl_set:Nn \l__prop_internal_tl {#2}
20127   \exp_after:wN \__prop_set_eq:wNnNn #1
20128   \cs_set_eq:NN \cs_set_nopar:Npe #1 \l__prop_internal_tl
20129 }

```

(End of definition for `\prop_make_linked:N` and `__prop_make_linked:Nn`. This function is documented on page 215.)

`\l_tmpa_prop` We can now initialize the scratch variables.

```

\l_tmpb_prop 20130 \prop_new:N \l_tmpa_prop
\g_tmpa_prop 20131 \prop_new:N \l_tmpb_prop
\g_tmpb_prop 20132 \prop_new:N \g_tmpa_prop
20133 \prop_new:N \g_tmpb_prop

```

(End of definition for `\l_tmpa_prop` and others. These variables are documented on page 222.)

```

\prop_concat:NNN The basic strategy is to copy the first variable into the target, then loop through the
\prop_concat:ccc second variable, calling \prop_(g)put:Nnn on each item. To avoid running the !3debug
\prop_gconcat:NNN scope checks on each of these steps, we use the auxiliaries that underly \prop_set_eq:NN
\prop_gconcat:ccc and \prop_put:Nnn, whose syntax is a bit unwieldy. We work directly with the target
__prop_concat:NNNNN prop #3 as a scratch space, because copying over from a temporary variable to #3 would
__prop_concat:nNNN be slow in the linked case. If #5 is #3 itself we have to be careful not to lose the data, and
we even take the opportunity to skip the copying step completely. To keep the correct
version of the duplicate keys we use the code underlying \prop_put_if_not_in:Nnn,
which involves passing \use_none:nnn to the auxiliary instead of nothing. There is no
need to check for the case where #3 is equal to #4 because in that case \prop_(g)set_
eq:NN #3 #4 (or rather the underlying auxiliary) is correctly set up to do no needless
work.

```

```

20134 \cs_new_protected:Npn \prop_concat:NNN
20135 { \__prop_concat:NNNNN \cs_set_eq:NN \cs_set_nopar:Npe }
20136 \cs_generate_variant:Nn \prop_concat:NNN { ccc }
20137 \cs_new_protected:Npn \prop_gconcat:NNN
20138 { \__prop_concat:NNNNN \cs_gset_eq:NN \cs_gset_nopar:Npe }
20139 \cs_generate_variant:Nn \prop_gconcat:NNN { ccc }
20140 \cs_new_protected:Npn \__prop_concat:NNNNN #1#2#3#4#5
20141 {
20142   \cs_if_eq:NNTF #3 #5
20143   { \__prop_concat:nNNN \use_none:nnn #2 #3 #4 }
20144   {
20145     \__prop_set_eq:NNNN #1 #2 #3 #4
20146     \__prop_concat:nNNN { } #2 #3 #5
20147   }
20148 }
20149 \cs_new_protected:Npn \__prop_concat:nNNN #1#2#3#4
20150 {
20151   \cs_gset_eq:NN \__prop_tmp:w \__prop_pair:wn
20152   \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop
20153   { \__prop_put:nNNnn {#1} #2 #3 {##1} }
20154   \exp_last_unbraced:Nf \use_none:nn #4
20155   \cs_gset_eq:NN \__prop_pair:wn \__prop_tmp:w
20156 }

```

(End of definition for `\prop_concat:NNN` and others. These functions are documented on page 216.)

```

\prop_put_from_keyval:Nn The core is a call to \keyval_parse:nnn, with an error message \__prop_missing_eq:n
\prop_put_from_keyval:cn for entries without =, and a call to (essentially) \prop_(g)put:Nnn for valid key–value
\prop_gput_from_keyval:Nn pairs. To avoid repeated scope checks (and errors) when !3debug is active, we instead use
\prop_gput_from_keyval:cn the auxiliary underlying \prop_put:Nnn. Because blank keys are valid here, in contrast
__prop_from_keyval:nn to !3keys, we set and restore \!__kernel_keyval_allow_blank_keys_bool. The key–
__prop_from_keyval:Nnn value argument may be quite large so we avoid reading it until it is really necessary.
__prop_missing_eq:n 20157 \cs_new_protected:Npn \prop_put_from_keyval:Nn #1

```

```

20158 { \_prop_from_keyval:nn { \_prop_put:nNNnn { } \cs_set_nopar:Npe #1 } }
20159 \cs_generate_variant:Nn \prop_put_from_keyval:Nn { c }
20160 \cs_new_protected:Npn \prop_gput_from_keyval:Nn #1
20161 { \_prop_from_keyval:nn { \_prop_put:nNNnn { } \cs_gset_nopar:Npe #1 } }
20162 \cs_generate_variant:Nn \prop_gput_from_keyval:Nn { c }
20163 \cs_new_protected:Npn \_prop_from_keyval:nn
20164 {
20165   \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
20166     { \_prop_from_keyval:Nnn \c_true_bool }
20167     { \_prop_from_keyval:Nnn \c_false_bool }
20168 }
20169 \cs_new_protected:Npn \_prop_from_keyval:Nnn #1#2#3
20170 {
20171   \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool \c_true_bool
20172   \keyval_parse:nnn \_prop_missing_eq:n {#2} {#3}
20173   \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool #1
20174 }
20175 \cs_new_protected:Npn \_prop_missing_eq:n
20176 { \msg_error:nnn { prop } { prop-keyval } }

```

(End of definition for `\prop_put_from_keyval:Nn` and others. These functions are documented on page 217.)

```

\prop_set_from_keyval:Nn
\prop_set_from_keyval:cn
\prop_gset_from_keyval:Nn
\prop_gset_from_keyval:cn

```

Just empty the prop (with the auxiliary underlying `\prop_clear:N` to avoid l3debug problems) and push key–value entries using `\prop_(g)put_from_keyval:Nn`.

```

20177 \cs_new_protected:Npn \prop_set_from_keyval:Nn #1
20178 {
20179   \_prop_clear:NNN \cs_set_eq:NN \cs_set_nopar:Npe #1
20180   \prop_put_from_keyval:Nn #1
20181 }
20182 \cs_generate_variant:Nn \prop_set_from_keyval:Nn { c }
20183 \cs_new_protected:Npn \prop_gset_from_keyval:Nn #1
20184 {
20185   \_prop_clear:NNN \cs_gset_eq:NN \cs_gset_nopar:Npe #1
20186   \prop_gput_from_keyval:Nn #1
20187 }
20188 \cs_generate_variant:Nn \prop_gset_from_keyval:Nn { c }

```

(End of definition for `\prop_set_from_keyval:Nn` and `\prop_gset_from_keyval:Nn`. These functions are documented on page 215.)

```

\prop_const_from_keyval:Nn
\prop_const_from_keyval:cn
\prop_const_linked_from_keyval:Nn
\prop_const_linked_from_keyval:cn

```

For both flat and linked constant props, we create #1 then use the same auxiliary as for `\prop_gput_from_keyval:Nn`. It is most natural to use the already packaged `\prop_gput:Nnn`, but that would mean doing an assignment on a supposedly constant property list. To avoid errors when l3debug is activated, we use the auxiliary underlying `\prop_gput:Nnn`.

```

20189 \cs_new_protected:Npn \prop_const_from_keyval:Nn #1
20190 {
20191   \prop_new:N #1
20192   \_prop_from_keyval:nn { \_prop_put:nNNnn { } \cs_gset_nopar:Npe #1 }
20193 }
20194 \cs_generate_variant:Nn \prop_const_from_keyval:Nn { c }
20195 \cs_new_protected:Npn \prop_const_linked_from_keyval:Nn #1
20196 {

```

```

20197     \prop_new_linked:N #1
20198     \__prop_from_keyval:nn { \__prop_put:nNnn { } \cs_gset_nopar:Npe #1 }
20199   }
20200 \cs_generate_variant:Nn \prop_const_linked_from_keyval:Nn { c }

```

(End of definition for `\prop_const_from_keyval:Nn` and `\prop_const_linked_from_keyval:Nn`. These functions are documented on page 215.)

63.4 Accessing data in property lists

Accessing/deleting/adding entries is mostly done by `__prop_split:NnTFn`, which must be fast because it is used in many `!3prop` functions. Its syntax is as follows.

```

\__prop_split:NnTFn <property list> {<key>}
  {<true code>} {<false code>} {<link code>}

```

If the `<property list>` uses the linked data storage, then it runs the `<link code>`, otherwise it does as follows.

It splits the `<property list>` at the `<key>`, giving three token lists: the `<entries before>` the `<key>`, the `<value>` associated with the `<key>` and the `<entries after>` the `<key>`. Both the `<entries before>` and the `<entries after>` can be empty or consist of some number of consecutive entries `__prop_pair:wn <keyi> \s__prop {<valuei>}`. If the `<key>` is present in the `<property list>` then the `<true code>` is left in the input stream, with #1, #2, and #3 replaced by the `<entries before>`, `<value>`, and `<entries after>`. If the `<key>` is not present in the `<property list>` then the `<false code>` is left in the input stream. Only the `<true code>` is used in the replacement text of a macro defined internally, which requires ## doubling.

<pre> __prop_split:NnTFn __prop_split_aux:nNFn __prop_split_test:wn __prop_split_flat:w __prop_split_linked:w __prop_split_wrong:Nw </pre>	<p>The aim is to distinguish four cases: a flat prop that contains the given <code><key></code>, a flat prop that does not contain it, a linked prop, and an invalid prop. The last case includes those that are set to <code>\relax</code> by c-expansion, as well as unrelated token list variables since these unfortunately used to “work” in earlier implementations. In the first three cases we run the T, F, and n arguments, and in the last case we raise an error, set the variable to a known state (empty prop), and run the F code (some conditionals such as <code>\prop_pop:NnNTF</code> otherwise blow up pretty badly).</p>
--	---

The first distinction between these cases is done by `__prop_split_test:wn`, which looks for the argument after `\s__prop`. For a flat prop it will be `__prop_chk:w`, which leads to running `__prop_split_flat:w`, explained below. For a linked prop it is the prefix, consisting of characters, so we end up running `__prop_split_linked:w`, which cleans up and selects the aforementioned n argument. For invalid props, or rather, variables that do not contain `\s__prop`, the argument includes `\fi:`, and we end up calling `__prop_split_wrong:Nw`, which calls `\prop_show:N` to raise a detailed error stating how the variable is wrong.

Let us return to `__prop_split_flat:w`. This function is defined dynamically as

```

\cs_set:Npn \__prop_split_flat:w \__prop_split_linked:w #1
\__prop_pair:wn <key> \s__prop #2
#3 \s__prop_mark #4 #5 \s__prop_stop
{ #4 {<true code>} }

```

Its job is to seek the $\langle key \rangle$ in the property list (known to be flat at this stage) by using an argument #1 delimited essentially by that key. If indeed the variable contained the $\langle key \rangle$, then #1 is the $\langle extract_1 \rangle$ before the key–value pair, #2 is the $\langle value \rangle$ associated with the $\langle key \rangle$, #3 is the $\langle extract_2 \rangle$ after the key–value pair, #4 is $\backslash use_i:nnn$, and we run $\backslash use_i:nnn \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \} \{ \langle link\ code \rangle \}$, selecting the $\langle true\ code \rangle$. Otherwise, the whole property list together with $\backslash s_prop_mark \backslash use_i:nnn$ is taken in as #1, then #2 is some tokens $? \backslash fi: \backslash_prop_split_wrong:Nw \langle variable \rangle$ that were only useful in the case of invalid props, #3 is empty, and most importantly #4 is $\backslash use_ii:nnn$. This command selects the $\langle false\ code \rangle$.

Note that we define $\backslash_prop_split_flat:w$ in all cases even though it is only used in the flat case. Indeed, to avoid taking in the whole property list (which may be large) as an argument more than strictly necessary, we would have to keep the $\langle true\ code \rangle$ positioned before the expansion of the prop variable in order to use it in the definition. The only way to do that is to store it using an assignment so we might as well just perform the assignment that we can actually use in the flat case.

```

20201 \cs_new_protected:Npn \__prop_split:NnTFn #1#2
20202 {
20203   \exp_after:wN \__prop_split_aux:nNnTFn
20204   \exp_after:wN { \tl_to_str:n {#2} } #1
20205 }
20206 \cs_new_protected:Npn \__prop_split_aux:nNnTFn #1#2#3
20207 {
20208   \cs_set:Npn \__prop_split_flat:w \__prop_split_linked:w ##1
20209   \__prop_pair:wn #1 \s__prop ##2 ##3 \s__prop_mark ##4 ##5 \s__prop_stop
20210   { ##4 {#3} }
20211   \exp_after:wN \__prop_split_test:wn #2 \s__prop_mark \use_i:nnn
20212   \__prop_pair:wn #1 \s__prop { ? \fi: \__prop_split_wrong:Nw #2 }
20213   \s__prop_mark \use_ii:nnn
20214   \s__prop_stop
20215 }
20216 \cs_new:Npn \__prop_split_flat:w { }
20217 \cs_new_protected:Npn \__prop_split_test:wn #1 \s__prop #2
20218 {
20219   \if_meaning:w \__prop_chk:w #2 \exp_after:wN \__prop_split_flat:w \fi:
20220   \__prop_split_linked:w
20221 }
20222 \cs_new_protected:Npn \__prop_split_linked:w #1 \s__prop_stop #2#3 {#3}
20223 \cs_new_protected:Npn \__prop_split_wrong:Nw #1#2 \s__prop_stop #3#4
20224 {
20225   \prop_show:N #1
20226   \cs_gset_eq:NN #1 \c_empty_prop
20227   #3
20228 }

```

(End of definition for $\backslash_prop_split:NnTFn$ and others.)

$\backslash prop_get:NnN$ Here we implement both $\backslash prop_get:NnN$ and its branching version through $\backslash_prop_get:NnnTF$. It receives the prop and key, followed by an assignment used when the value is found, $\langle true\ code \rangle$ to run after the assignment, and some fall-back $\langle false\ code \rangle$ for absent values. It relies on $\backslash_prop_split:NnTFn$. For a flat prop, the first four arguments of $\backslash_prop_split:NnTFn$ are used, and run either the assignment #3{##3} and $\langle true\ code \rangle$ #4, or the $\langle false\ code \rangle$ #5.

$\backslash prop_get:NVN$

$\backslash prop_get:NvN$

$\backslash prop_get:NeN$

$\backslash prop_get:NoN$

$\backslash prop_get:NxN$

$\backslash prop_get:cnN$

$\backslash prop_get:cVN$

$\backslash prop_get:cvN$

$\backslash prop_get:ceN$

$\backslash prop_get:coN$

$\backslash prop_get:cxN$

$\backslash prop_get:cnc$

$\backslash prop_get:NnNTF$

$\backslash prop_get:NVNTF$

$\backslash prop_get:NvNTF$

$\backslash prop_get:NeNTF$

```

20229 \cs_new_protected:Npn \prop_get:NnN #1#2#3
20230 {
20231   \__prop_get:NnnTF #1 {#2}
20232   { \tl_set:Nn #3 } { } { \tl_set:Nn #3 { \q_no_value } }
20233 }
20234 \cs_generate_variant:Nn \prop_get:NnN { NV , Nv , Ne , c , cV , cv , ce }
20235 \cs_generate_variant:Nn \prop_get:NnN { No , Nx , co , cx }
20236 \cs_generate_variant:Nn \prop_get:NnN { cnc }
20237 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
20238 {
20239   \__prop_get:NnnTF #1 {#2}
20240   { \tl_set:Nn #3 } \prg_return_true: \prg_return_false:
20241 }
20242 \prg_generate_conditional_variant:Nnn \prop_get:NnN
20243 { NV , Nv , Ne , c , cV , cv , ce } { T , F , TF }
20244 \prg_generate_conditional_variant:Nnn \prop_get:NnN
20245 { No , Nx , co , cx } { T , F , TF }
20246 \prg_generate_conditional_variant:Nnn \prop_get:NnN
20247 { cnc } { T , F , TF }
20248 \cs_new_protected:Npn \__prop_get:NnnTF #1#2#3#4#5
20249 {
20250   \__prop_split:NnTFn #1 {#2}
20251   { #3 {##2} #4 }
20252   {#5}
20253   { \exp_after:wN \__prop_get_linked:w #1 {#2} {#3} {#4} {#5} }
20254 }

```

For a linked prop we must work a bit: `__prop_get_linked:w` is followed by the expansion of the prop, then by four brace groups: the key #4, the assignment code #5, `\true code` #6, and `\false code` #7. If the key is present, its value is stored in the token list `__prop_#2~#4`. If that token list exists, `__prop_get_linked_aux:w` gets called followed by the expansion of that token list and we grab as #2 the value associated to that key, which we feed to the assignment code and follow-up code. If the key is absent the token list can be `\undefined` or `\relax`. In both cases `__prop_get_linked_aux:w` finds an empty brace group as #2, `\use_none:n` as #4 and the `\false code` as #5. Note that we made `__prop_get_linked:w` and subsequent auxiliaries expandable, because they are also used in `\prop_item:Nn`.

```

20255 \cs_new:Npn \__prop_get_linked:w
20256   \__prop_flatten:w #1 \s__prop #2#3#4#5#6#7
20257 {
20258   \if_cs_exist:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
20259   \exp_after:wN \exp_after:wN \exp_after:wN \__prop_get_linked_aux:w
20260   \cs:w __prop ~ #2 ~ \tl_to_str:n {#4} \exp_after:wN \cs_end:
20261   \else:
20262     \exp_after:wN \__prop_get_linked_aux:w
20263   \fi:
20264   \s__prop_mark {#5} {#6}
20265   \s__prop { } \s__prop_mark \use_none:n {#7}
20266   \s__prop_stop
20267 }
20268 \cs_new:Npn \__prop_get_linked_aux:w
20269   #1 \s__prop #2 #3 \s__prop_mark #4 #5 #6 \s__prop_stop { #4 {#2} #5 }

```

(End of definition for `\prop_get:NnN` and others. These functions are documented on page 217.)


```

\prop_item:Nn Getting the value corresponding to a key in a flat property list in an expandable fashion
\prop_item:NV simply uses \prop_map_tokens:Nn to go through the property list. The auxiliary
\prop_item:No \__prop_item:nnn receives the search string #1, the key #2 and the value #3 and returns
\prop_item:Ne as appropriate.
\prop_item:cn 20270 \cs_new:Npn \prop_item:Nn #1#2
\prop_item:cV 20271 {
\prop_item:co 20272 \__prop_if_flat:NTF #1
\prop_item:ce 20273 {
\__prop_item:nnn 20274 \exp_args:NNo \prop_map_tokens:Nn #1
20275 {
20276 \exp_after:wN \__prop_item:nnn
20277 \exp_after:wN { \tl_to_str:n {#2} }
20278 }
20279 }
20280 { \exp_after:wN \__prop_get_linked:w #1 {#2} \use:n { } { } }
20281 }
20282 \cs_new:Npn \__prop_item:nnn #1#2#3
20283 {
20284 \str_if_eq:eeT {#1} {#2}
20285 { \prop_map_break:n { \exp_not:n {#3} } }
20286 }
20287 \cs_generate_variant:Nn \prop_item:Nn { NV , No , Ne , c , cV , co , ce }

```

(End of definition for `\prop_item:Nn` and `__prop_item:nnn`. This function is documented on page 218.)

63.5 Removing data from property lists

```

\__prop_pop:NnNnNnTF This auxiliary is used by both the \prop_pop family and the \prop_remove family of functions.
\__prop_pop_linked:wnNnNnTF It receives a <prop> and a {<key>}, three assignment functions (\tl_set:Nn \cs_set_eq:NN
\__prop_pop_linked:NNNn \cs_set_nopar:Npe or their global versions), then {<code>} {<true code>}
\__prop_pop_linked:w {<false code>}.
\__prop_pop_linked_prev:w
\__prop_pop_linked_next:w

```

For a flat prop, split it. If the `<key>` is there, reconstruct the rest of the prop from the two extracts `##2 ##4` and assign using `\tl_(g)set:Nn`, then run `<code> {<value>}` with the `<value>` found, and run the `<true code>`. If the `<key>` is absent, run the `<false code>`.

For a linked prop, the removal is done by `__prop_pop_linked:wnNnNnTF`, which removes the key–value pair from the doubly-linked list and runs its last three arguments `{<code>} {<true code>} {<false code>}` depending on whether the key–value is found, in the same way as for flat props.

```

20288 \cs_new_protected:Npn \__prop_pop:NnNnNnTF #1#2#3#4#5#6#7
20289 {
20290 \__prop_split:NnTFn #1 {#2}
20291 {
20292 #4 #1 { \exp_not:n { \s__prop \__prop_chk:w ##1 ##3 } }
20293 #5 {##2}
20294 #6
20295 }
20296 {#7}
20297 {
20298 \exp_after:wN \__prop_pop_linked:wnNnNnTF #1 {#2}
20299 #3 #4 {#5} {#6} {#7}
20300 }

```

```
20301 }
```

The next auxiliary `__prop_pop_linked:wnNNnTF`, together with the `NNNn` auxiliary, checks if the key is present in the *(linked prop)*, then the corresponding value (if present) is passed as a braced argument to the *(code)* and the *(true code)* or *(false code)* is run as appropriate. Before that, there are also three assignments: the token lists for the previous key and next key are made to point to each other, cf. `__prop_pop_linked:w`, and the token list for the given key is made undefined.

```
20302 \cs_new_protected:Npn \__prop_pop_linked:wnNNnTF
20303   \__prop_flatten:w #1 \s__prop #2#3#4#5#6#7
20304   {
20305     \if_cs_exist:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
20306     \exp_after:wN \__prop_pop_linked:NNNn
20307     \cs:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
20308     #5 #6 {#7}
20309     \else:
20310       \exp_after:wN \use_iii:nnn
20311       \fi:
20312       \use_i:nn
20313     }
20314 \cs_new_protected:Npn \__prop_pop_linked:NNNn #1#2#3#4
20315   {
20316     \if_meaning:w \scan_stop: #1
20317     \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
20318     \else:
20319     \exp_after:wN \__prop_pop_linked:w #1 #1 #2 #3 {#4}
20320     \fi:
20321   }
20322 \cs_new_protected:Npn \__prop_pop_linked:w
20323   \use_none:n #1#2 \s__prop #3#4#5#6#7#8
20324   {
20325     #6 #5 \tex_undefined:D
20326     #7 #1
20327     {
20328       \exp_after:wN \__prop_pop_linked_prev:w #1
20329       \exp_not:N #4
20330     }
20331     #7 #4
20332     {
20333       \exp_not:n { \use_none:n #1 }
20334       \exp_not:f { \exp_after:wN \__prop_pop_linked_next:w #4 }
20335     }
20336     #8 {#3}
20337   }
20338 \cs_new:Npn \__prop_pop_linked_prev:w #1 \s__prop #2#3
20339   { \exp_not:n { #1 \s__prop {#2} } }
20340 \cs_new:Npn \__prop_pop_linked_next:w \use_none:n #1 { \exp_stop_f: }
```

(End of definition for `__prop_pop:NnNNnTF` and others.)

`\prop_remove:Nn` Deleting from a property relies on `__prop_pop:NnNNnTF`. The three assignment functions are suitably local or global. The last three arguments are `\use_none:n` and two empty brace groups: if the key is found we get `\use_none:n {<key>} <empty>`, which ex-

`\prop_remove:NV`
`\prop_remove:Ne`
`\prop_remove:cn`
`\prop_remove:cV`
`\prop_remove:ce`

`\prop_gremove:Nn`
`\prop_gremove:NV`
`\prop_gremove:Ne`
`\prop_gremove:cn`
`\prop_gremove:cV`
`\prop_gremove:ce`

pands to nothing, and otherwise we just get `<empty>`. The auxiliary takes care of actually removing the entry from the prop.

```

20341 \cs_new_protected:Npn \prop_remove:Nn #1#2
20342 {
20343   \__prop_pop:NnNnTF #1 {#2}
20344   \cs_set_eq:NN \cs_set_nopar:Npe
20345   \use_none:n { } { }
20346 }
20347 \cs_new_protected:Npn \prop_gremove:Nn #1#2
20348 {
20349   \__prop_pop:NnNnTF #1 {#2}
20350   \cs_gset_eq:NN \cs_gset_nopar:Npe
20351   \use_none:n { } { }
20352 }
20353 \cs_generate_variant:Nn \prop_remove:Nn { NV , Ne , c , cV , ce }
20354 \cs_generate_variant:Nn \prop_gremove:Nn { NV , Ne , c , cV , ce }

```

(End of definition for `\prop_remove:Nn` and `\prop_gremove:Nn`. These functions are documented on page 218.)

```

\prop_pop:NnN Popping a value is almost the same, but the value found is kept. For the non-branching
\prop_pop:NVN version, we additionally set the target token list to \q_no_value, while for the branching
\prop_pop:NoN version we must produce \prg_return_true: or \prg_return_false:.
\prop_pop:cnN 20355 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
\prop_pop:cVN 20356 {
\prop_pop:coN 20357   \__prop_pop:NnNnTF #1 {#2}
\prop_gpop:NnN 20358   \cs_set_eq:NN \cs_set_nopar:Npe
\prop_gpop:NVN 20359   { \tl_set:Nn #3 } { } { \tl_set:Nn #3 { \q_no_value } }
\prop_gpop:NoN 20360 }
\prop_gpop:cnN 20361 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
\prop_gpop:cVN 20362 {
\prop_gpop:coN 20363   \__prop_pop:NnNnTF #1 {#2}
\prop_pop:NnNTF 20364   \cs_gset_eq:NN \cs_gset_nopar:Npe
\prop_pop:NVNTF 20365   { \tl_set:Nn #3 } { } { \tl_set:Nn #3 { \q_no_value } }
\prop_pop:NoNTF 20366 }
\prop_pop:cnNTF 20367 \cs_generate_variant:Nn \prop_pop:NnN { NV , No }
\prop_pop:cVNTF 20368 \cs_generate_variant:Nn \prop_pop:NnN { c , cV , co }
\prop_pop:coNTF 20369 \cs_generate_variant:Nn \prop_gpop:NnN { NV , No }
\prop_gpop:NnNTF 20370 \cs_generate_variant:Nn \prop_gpop:NnN { c , cV , co }
\prop_gpop:NVNTF 20371 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
\prop_gpop:NoNTF 20372 {
\prop_gpop:cnNTF 20373   \__prop_pop:NnNnTF #1 {#2}
\prop_gpop:cVNTF 20374   \cs_set_eq:NN \cs_set_nopar:Npe
\prop_gpop:coNTF 20375   { \tl_set:Nn #3 } \prg_return_true: \prg_return_false:
20376 }
\prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
20377 {
20378   \__prop_pop:NnNnTF #1 {#2}
20379   \cs_gset_eq:NN \cs_gset_nopar:Npe
20380   { \tl_set:Nn #3 } \prg_return_true: \prg_return_false:
20381 }
20382 \prg_generate_conditional_variant:Nnn \prop_pop:NnN
20383 { NV , No , c , cV , co } { T , F , TF }
20384 \prg_generate_conditional_variant:Nnn \prop_gpop:NnN
20385

```

20386 { NV , No , c , cV , co } { T , F , TF }

(End of definition for `\prop_pop:NnN` and others. These functions are documented on page 217.)

63.6 Adding data to property lists

`\prop_put:Nnn` All of the `\prop_(g)put(_if_new):Nnn` functions are based on the same auxiliary, which receives `<code>` and an “assignment”, followed by `<prop> {<key>} {<new value>}`. The assignment `\cs_(g)set_nopar:Npe` is the primitive assignment without any checking: in the case of linked props it is applied to individual pieces of the linked prop, which are typically not yet defined. Debugging the scope of the variable is done at a higher level by letting `!3debug` change `\prop_put:Nnn` and friends. This allows other `!3prop` commands to directly call the underlying auxiliary to skip this checking step and avoid getting multiple error messages for the same error. The `<code>` (empty for `put` and `\use_none:nnn` for `put_if_not_in`) is placed before the assignment in cases where the key is already present, in order to suppress the assignment in the `put_if_not_in` case.

```

20387 \cs_new_protected:Npn \prop_put:Nnn
20388   { \__prop_put:nNnn { } \cs_set_nopar:Npe }
20389 \cs_new_protected:Npn \prop_gput:Nnn
20390   { \__prop_put:nNnn { } \cs_gset_nopar:Npe }
20391 \cs_new_protected:Npn \prop_put_if_not_in:Nnn
20392   { \__prop_put:nNnn \use_none:nnn \cs_set_nopar:Npe }
20393 \cs_new_protected:Npn \prop_gput_if_not_in:Nnn
20394   { \__prop_put:nNnn \use_none:nnn \cs_gset_nopar:Npe }
20395 \cs_generate_variant:Nn \prop_put:Nnn
20396   {
20397     NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
20398     Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee
20399   }
20400 \cs_generate_variant:Nn \prop_put:Nnn
20401   { Nno , No , Noo , Nnx , NVx , NxV , Nxx }
20402 \cs_generate_variant:Nn \prop_put:Nnn
20403   {
20404     c , cnV , cnv , cne , cV , cVV , cVv , cVe ,
20405     cv , cvV , cvv , cve , ce , ceV , cev , cee
20406   }
20407 \cs_generate_variant:Nn \prop_put:Nnn
20408   { cno , co , coo , cnx , cVx , cxV , cxx }
20409 \cs_generate_variant:Nn \prop_gput:Nnn
20410   {
20411     NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
20412     Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee
20413   }
20414 \cs_generate_variant:Nn \prop_gput:Nnn
20415   { Nno , No , Noo , Nnx , NVx , NxV , Nxx }
20416 \cs_generate_variant:Nn \prop_gput:Nnn
20417   {
20418     c , cnV , cnv , cne , cV , cVV , cVv , cVe ,
20419     cv , cvV , cvv , cve , ce , ceV , cev , cee
20420   }
20421 \cs_generate_variant:Nn \prop_gput:Nnn
20422   { cno , co , coo , cnx , cVx , cxV , cxx }

```

`\prop_gput:Nnn`

`\prop_gput:NnV`

`\prop_gput:Nnv`

`\prop_gput:Nne`

`\prop_gput:NVn`

`\prop_gput:NVV`

`\prop_gput:NVv`

`\prop_gput:NVe`

```

20423 \cs_generate_variant:Nn \prop_put_if_not_in:Nnn
20424 {
20425     NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
20426     Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee ,
20427     c , cnV , cnv , cne , cV , cVV , cVv , cVe ,
20428     cv , cvV , cvv , cve , ce , ceV , cev , cee
20429 }
20430 \cs_generate_variant:Nn \prop_gput_if_not_in:Nnn
20431 {
20432     NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
20433     Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee ,
20434     c , cnV , cnv , cne , cV , cVV , cVv , cVe ,
20435     cv , cvV , cvv , cve , ce , ceV , cev , cee
20436 }

```

Since the true branch of `__prop_split:NnTFn` is used as the replacement text of an internal macro, and since the `<key>` and new `<value>` may contain arbitrary tokens, it is not safe to include them in the argument of `__prop_split:NnTFn`. We thus start by storing in `\l__prop_internal_tl` tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in `__prop_split:NnTFn`. For a flat prop, if the `<key>` was absent, append the new key–value to the list; otherwise concatenate the extracts `##2` and `##4` with the new key–value pair `\l__prop_internal_tl`. The updated entry is placed at the same spot as the original `<key>` in the property list, preserving the order of entries. For a linked prop, call `__prop_put_linked:wNnN`, which constructs the control sequence in which we will place the new value. If it matches `\scan_stop:` then the key was not yet there and we add it using `__prop_put_linked_new:w`, otherwise it was already there and we use `__prop_put_linked_old:w`.

```

20437 \cs_new_protected:Npn \__prop_put:nNnNn #1#2#3#4#5
20438 {
20439     \tl_set:Nn \l__prop_internal_tl
20440     {
20441         \exp_not:N \__prop_pair:wN \tl_to_str:n {#4}
20442         \s__prop { \exp_not:n {#5} }
20443     }
20444     \__prop_split:NnTFn #3 {#4}
20445     {
20446         #1 #2 #3
20447         {
20448             \s__prop \__prop_chk:w \exp_not:n {##1}
20449             \l__prop_internal_tl \exp_not:n {##3}
20450         }
20451     }
20452     { #2 #3 { \exp_not:o {#3} \l__prop_internal_tl } }
20453     { \exp_after:wN \__prop_put_linked:wNnN #3 {#4} {#1} #2 }
20454 }
20455 \cs_new_protected:Npn \__prop_put_linked:wNnN
20456     \__prop_flatten:w #1 \s__prop #2#3#4
20457 {
20458     \exp_after:wN \__prop_put_linked:NnN
20459     \cs:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
20460     #1
20461 }
20462 \cs_new_protected:Npn \__prop_put_linked:NnN #1#2#3#4
20463 {

```

```

20464 \if_meaning:w \scan_stop: #1
20465 \exp_after:wN \__prop_put_linked_new:w #2 #1 #2 #4
20466 \else:
20467 \exp_after:wN \__prop_put_linked_old:w #1 { #3 #4 #1 }
20468 \fi:
20469 }

```

To add a new entry, `__prop_put_linked_new:w` receives the expansion of the end-pointer, namely `\use_none:n` (*last key pointer*), followed by the new key pointer #2, the end pointer #3, and an assignment function #4. Set up the doubly-linked list in the order #1, #2, #3, placing the key-value pair `\l__prop_internal_tl` in #2. To replace an old entry, `__prop_put_linked_old:w` receives the expansion of that entry, and it reassigns it (#5) using the assignment #6, by simply replacing the payload #2 `\s__prop #3` by `\l__prop_internal_tl`.

```

20470 \cs_new_protected:Npn \__prop_put_linked_new:w
20471 \use_none:n #1#2#3#4
20472 {
20473 #4 #1
20474 {
20475 \exp_after:wN \__prop_pop_linked_prev:w #1
20476 \exp_not:N #2
20477 }
20478 #4 #2
20479 {
20480 \exp_not:n { \use_none:n #1 }
20481 \l__prop_internal_tl
20482 \exp_not:N #3
20483 }
20484 #4 #3 { \exp_not:n { \use_none:n #2 } }
20485 }
20486 \cs_new_protected:Npn \__prop_put_linked_old:w
20487 \use_none:n #1#2 \s__prop #3#4#5
20488 {
20489 #5
20490 {
20491 \exp_not:n { \use_none:n #1 }
20492 \l__prop_internal_tl
20493 \exp_not:N #4
20494 }
20495 }

```

(End of definition for `\prop_put:Nnn` and others. These functions are documented on page 216.)

63.7 Property list conditionals

```

\prop_if_exist_p:N Copies of the cs functions defined in l3basics.
\prop_if_exist_p:c 20496 \prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N
\prop_if_exist:NTF 20497 { TF , T , F , p }
\prop_if_exist:cTF 20498 \prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c
20499 { TF , T , F , p }

```

(End of definition for `\prop_if_exist:NTF`. This function is documented on page 218.)

`\prop_if_empty_p:N` A flat property list is empty if it matches `\c_empty_prop`. A linked property list is empty if its second token (the end pointer) and last token (the first key pointer) are equal. There cannot be false positives because the end pointer takes the form `\use_none:n <pointer>` while the other pointers have more elaborate structure. The subtle code branch here is when a non-empty flat property list is given: then `__prop_if_empty:w` reads the whole property list as #1 and #2, #3, #4 are 2, 3, 4, respectively.

```

20500 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
20501   {
20502     \if_meaning:w #1 \c_empty_prop
20503     \prg_return_true:
20504   \else:
20505     \exp_after:wN \__prop_if_empty_return:w #1
20506     \__prop_flatten:w 2 \s__prop 34 \s__prop_stop
20507   \fi:
20508 }
20509 \cs_new:Npn \__prop_if_empty_return:w
20510 #1 \__prop_flatten:w #2 \s__prop #3#4#5 \s__prop_stop
20511 {
20512   \if_meaning:w #2 #4
20513   \prg_return_true:
20514 \else:
20515   \prg_return_false:
20516 \fi:
20517 }
20518 \prg_generate_conditional_variant:Nnn \prop_if_empty:N
20519 { c } { p , T , F , TF }

```

(End of definition for `\prop_if_empty:N` and `__prop_if_empty_return:w`. This function is documented on page 219.)

`\prop_if_in_p:Nn` For a linked prop, use `__prop_get_linked:w` to look up whether the control sequence constructed from the prefix and the sought-after key exists; this auxiliary calls `\use_none:n <value>` `\prg_return_true:` if the key is found, and otherwise `\prg_return_false:`. For a flat prop, testing expandably if a key is there requires to go through the key–value pairs one by one. This is rather slow, and a faster test would be

```

\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
{
  \@@_split:NnTFn #1 {#2}
  { \prg_return_true: }
  { \prg_return_false: }
  { ... }
}

```

but `__prop_split:NnTFn` is non-expandable. Instead, we use `\prop_map_tokens:Nn` to compare the search key to each key in turn using `\str_if_eq:ee`, which is expandable.

```

20520 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
20521 {
20522   \__prop_if_flat:NTF #1
20523   {
20524     \exp_after:wN \prop_map_tokens:Nn \exp_after:wN #1
20525     {
20526       \exp_after:wN \__prop_if_in_flat:nnn

```

```

20527         \exp_after:wN { \tl_to_str:n {#2} }
20528     }
20529     \prg_return_false:
20530 }
20531 {
20532     \exp_after:wN \__prop_get_linked:w #1 {#2}
20533     \use_none:n \prg_return_true: \prg_return_false:
20534 }
20535 }
20536 \cs_new:Npn \__prop_if_in_flat:nnn #1#2#3
20537 {
20538     \str_if_eq:eeT {#1} {#2}
20539     { \prop_map_break:n { \use_i:nn \prg_return_true: } }
20540 }
20541 \prg_generate_conditional_variant:Nnn \prop_if_in:Nn
20542 { NV , Ne , No , c , cV , ce , co } { p , T , F , TF }

```

(End of definition for `\prop_if_in:NnTF` and `__prop_if_in_flat:nnn`. This function is documented on page 219.)

63.8 Mapping over property lists

`\prop_map_function:NN` We first f-expand to flatten #1 in case it was a linked list. The `\use_i:nnn` removes the leading `\s__prop` `__prop_chk:w` of the flattened prop. The even-numbered arguments of `__prop_map_function:Nw` are keys, hence have string catcodes, except at the end where they are `\fi: \prop_map_break:.` The `\fi:` ends the `\if_false: #<even> \fi:` construction and we jump out of the loop. No need for any quark test.

```

20543 \cs_new:Npn \prop_map_function:NN #1#2
20544 {
20545     \exp_last_unbraced:Nnf
20546     \use_i:nnn { \__prop_map_function:Nw #2 } #1
20547     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
20548     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
20549     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
20550     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
20551     \prg_break_point:Nn \prop_map_break: { }
20552 }
20553 \cs_new:Npn \__prop_map_function:Nw #1
20554     \__prop_pair:wn #2 \s__prop #3
20555     \__prop_pair:wn #4 \s__prop #5
20556     \__prop_pair:wn #6 \s__prop #7
20557     \__prop_pair:wn #8 \s__prop #9
20558 {
20559     \if_false: #2 \fi: #1 {#2} {#3}
20560     \if_false: #4 \fi: #1 {#4} {#5}
20561     \if_false: #6 \fi: #1 {#6} {#7}
20562     \if_false: #8 \fi: #1 {#8} {#9}
20563     \__prop_map_function:Nw #1
20564 }
20565 \cs_generate_variant:Nn \prop_map_function:NN { Nc , c , cc }

```

(End of definition for `\prop_map_function:NN` and `__prop_map_function:Nw`. This function is documented on page 220.)

`\prop_map_inline:Nn` Mapping in line requires a nesting level counter. Store the current definition of `__prop_pair:wn`, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form `__prop_pair:wn <key> \s__prop {<value>}`, there are a leading and a trailing tokens, but both are equal to `\scan_stop:`, hence have no effect in such inline mapping. Such `\scan_stop:` could have affected ligatures if they appeared during the mapping.

```

20566 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
20567   {
20568     \cs_gset_eq:cN
20569       { __prop_map_ \int_use:N \g__kernel_prg_map_int :wn } \__prop_pair:wn
20570     \int_gincr:N \g__kernel_prg_map_int
20571     \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
20572     \exp_last_unbraced:Nf \use_none:nn #1
20573     \prg_break_point:Nn \prop_map_break:
20574     {
20575       \int_gdecr:N \g__kernel_prg_map_int
20576       \cs_gset_eq:Nc \__prop_pair:wn
20577         { __prop_map_ \int_use:N \g__kernel_prg_map_int :wn }
20578     }
20579   }
20580 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End of definition for `\prop_map_inline:Nn`. This function is documented on page 220.)

`\prop_map_tokens:Nn` The mapping is very similar to `\prop_map_function:NN`. The odd construction `\use:n {#1}` allows #1 to contain any token without interfering with `\prop_map_break:.`
`\prop_map_tokens:cn` `\use:n {#1}` allows #1 to contain any token without interfering with `\prop_map_break:.`
`__prop_map_tokens:nw` The loop stops when the `<key>` between `__prop_pair:wn` and `\s__prop` is `\fi: \prop_map_break:` instead of being a string.

```

20581 \cs_new:Npn \prop_map_tokens:Nn #1#2
20582   {
20583     \exp_last_unbraced:Nnf
20584       \use_i:nnn { \__prop_map_tokens:nw {#2} } #1
20585     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
20586     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
20587     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
20588     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
20589     \prg_break_point:Nn \prop_map_break: { }
20590   }
20591 \cs_new:Npn \__prop_map_tokens:nw #1
20592   \__prop_pair:wn #2 \s__prop #3
20593   \__prop_pair:wn #4 \s__prop #5
20594   \__prop_pair:wn #6 \s__prop #7
20595   \__prop_pair:wn #8 \s__prop #9
20596   {
20597     \if_false: #2 \fi: \use:n {#1} {#2} {#3}
20598     \if_false: #4 \fi: \use:n {#1} {#4} {#5}
20599     \if_false: #6 \fi: \use:n {#1} {#6} {#7}
20600     \if_false: #8 \fi: \use:n {#1} {#8} {#9}
20601     \__prop_map_tokens:nw {#1}
20602   }
20603 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }

```

(End of definition for `\prop_map_tokens:Nn` and `__prop_map_tokens:nw`. This function is documented on page 220.)

`\prop_map_break:` The break statements are based on the general `\prg_map_break:Nn`.
`\prop_map_break:n`

```
20604 \cs_new:Npn \prop_map_break:
20605   { \prg_map_break:Nn \prop_map_break: { } }
20606 \cs_new:Npn \prop_map_break:n
20607   { \prg_map_break:Nn \prop_map_break: }
```

(End of definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 221.)

63.9 Uses of mapping over property lists

`\prop_count:N` Counting the key–value pairs in a property list is done using the same approach as for
`\prop_count:c` other count functions: turn each entry into a +1 then use integer evaluation to actually
`__prop_count:nn` do the mathematics.

```
20608 \cs_new:Npn \prop_count:N #1
20609   {
20610     \int_eval:n
20611     {
20612       0
20613       \prop_map_function:NN #1 \__prop_count:nn
20614     }
20615   }
20616 \cs_new:Npn \__prop_count:nn #1#2 { + 1 }
20617 \cs_generate_variant:Nn \prop_count:N { c }
```

(End of definition for `\prop_count:N` and `__prop_count:nn`. This function is documented on page 218.)

`\prop_to_keyval:N` Each property name and value pair will be returned in the form `_{}{<name>}=_{}{<value>}`.
`__prop_to_keyval_exp_after:wN` As one of the main use cases for this macro is to pass the `<property list>` on to a
`__prop_to_keyval:nn` key–value parser, we have to make sure that the behaviour is as good as possible. Using
`__prop_to_keyval:nnw` a space before the opening brace we get the correct brace stripping behaviour for most of
the key–value parsers available in L^AT_EX. Iterate over the `<property list>` and remove
the leading comma afterwards. Only the value has to be protected in `__kernel_`
`exp_not:w` as the property name is always a string. After the loop the leading comma
is removed by `\use_none:n` and afterwards `__kernel_exp_not:w` eventually finds the
opening brace of its argument.

```
20618 \cs_new:Npn \prop_to_keyval:N #1
20619   {
20620     \__kernel_exp_not:w
20621     \prop_if_empty:NTF #1
20622     { {} }
20623     {
20624       \exp_after:wN \exp_after:wN \exp_after:wN
20625       {
20626         \tex_expanded:D
20627         {
20628           \exp_not:N \use_none:n
20629           \prop_map_function:NN #1 \__prop_to_keyval:nn
20630         }
20631       }
20632     }
20633   }
```

```

20634 \cs_new:Npn \__prop_to_keyval:nn #1#2
20635   { , ~ {#1} =~ { \__kernel_exp_not:w {#2} } }

```

(End of definition for `\prop_to_keyval:N` and others. This function is documented on page 218.)

63.10 Viewing property lists

`\prop_show:N` Experience shows one source of problems is very hard to debug: when a data structure such as a `seq` or `prop` gets corrupted. In the past, `\prop_show:N` would in some cases happily show items of such a `prop`, even though other more demanding `!3prop` functions would choke. It is thus best to make `\prop_show:N` check very thoroughly the structure and flag issues, even though that is very painful for linked props. Throughout the code below, we strive to remain as safe as possible, but in the explanations we only state what the arguments are when the prop is correctly formed, rather than saying at every step that various arguments can be arbitrary junk, made safe by using `\tl_to_str:n` generously.

The general `__kernel_chk_tl_type:NnnT` checks that its first argument is a token list, and if it is, then it e-expands its second argument and compares with the contents of its first argument. Thus, within this e-expansion it is safe to use `__prop_if_flat:NTF` to check if the prop is flat or linked. In the flat case we simply reconstruct the expected structure using `__prop_show_flat:w`, which loops through the prop and correctly turns all keys to strings for instance. In the linked case, we use `__prop_show_linked:w`, which ensures the form `__prop_flatten:w __prop <prefix> \s__prop {<prefix>} <rest>`, where `<prefix>` is made into a string and `<rest>` cannot be a brace group or multiple tokens since `__prop_show_linked:w` would in such cases give a different result from the original token list.

```

20636 \cs_new_protected:Npn \prop_show:N { \__prop_show:NN \msg_show:nneeee }
20637 \cs_generate_variant:Nn \prop_show:N { c }
20638 \cs_new_protected:Npn \prop_log:N { \__prop_show:NN \msg_log:nneeee }
20639 \cs_generate_variant:Nn \prop_log:N { c }
20640 \cs_new_protected:Npn \__prop_show:NN #1#2
20641   {
20642     \__kernel_chk_tl_type:NnnT #2 { prop }
20643     {
20644       \__prop_if_flat:NTF #2
20645       {
20646         \s__prop \__prop_chk:w
20647         \exp_after:wN \__prop_show_flat:w #2
20648         \s__prop { }
20649         \__prop_pair:wn \q__prop_recursion_tail \s__prop { }
20650         \q__prop_recursion_stop
20651       }
20652       { \exp_after:wN \__prop_show_linked:w #2 \s__prop ! ? \s__prop_stop }
20653     }
20654   {
20655     \__prop_if_flat:NTF #2
20656     { \__prop_show_finally:NNn #1 #2 { flat } }
20657     {
20658       \tl_set:Nn \l__prop_internal_tl { #1 #2 }
20659       \exp_after:wN \__prop_show_prepare:w #2 #2
20660     }
20661   }

```

```

20662 }
20663 \cs_new:Npn \__prop_show_flat:w #1 \__prop_pair:wn #2 \s__prop #3
20664 {
20665   \__prop_if_recursion_tail_stop:n {#2}
20666   \exp_not:N \__prop_pair:wn \tl_to_str:n {#2} \s__prop \exp_not:n { {#3} }
20667   \__prop_show_flat:w
20668 }
20669 \cs_new:Npn \__prop_show_linked:w #1 \s__prop #2#3#4 \s__prop_stop
20670 {
20671   \exp_not:N \__prop_flatten:w
20672   \exp_not:c { __prop ~ \tl_to_str:n {#2} }
20673   \s__prop { \tl_to_str:n {#2} }
20674   \exp_not:n {#3}
20675 }

```

For flat props we are done by using `\msg_show:nneeee` or `\msg_log:nneeee`. The auxiliary `__prop_show_finally:NNn` is eventually also used in the linked case after some more tests. To avoid having to bring along the message function and the property list, we store them into `\l__prop_internal_tl`.

```

20676 \cs_new_protected:Npn \__prop_show_finally:NNn #1#2#3
20677 {
20678   #1 { prop } { show }
20679   { \token_to_str:N #2 }
20680   { \prop_map_function:NN #2 \msg_show_item:nn }
20681   {#3} { }
20682 }

```

For linked props, we now know they have a reasonable form so that we are calling `__prop_show_prepare:w __prop_flatten:w __prop <prefix> \s__prop {<prefix>} <token> <property list>`, and the task is to loop through the linked list and check integrity. We first set things up: the auxiliary `__prop_tmp:w` will be in charge of checking that various tokens start with `__prop <prefix>` (in the sense of string representations), and calling one of `__prop_show_loop_key:wNNN`, `__prop_show_end:NNN`, `__prop_show_bad_name:NNN`.

```

20683 \cs_new_protected:Npn \__prop_show_prepare:w
20684   \__prop_flatten:w #1 \s__prop #2#3#4
20685 {
20686   \use:e
20687   {
20688     \cs_set_nopar:Npn \exp_not:N \__prop_tmp:w
20689       ##1 \token_to_str:N #1 ##2 \s__prop_mark ##3 \s__prop_stop
20690     {
20691       \exp_not:N \tl_if_empty:nTF {##1}
20692       {
20693         \exp_not:N \tl_if_head_is_space:nTF {##2}
20694         { \exp_not:N \exp_args:Nf \__prop_show_loop_key:wNNN }
20695         { \exp_not:N \tl_if_empty:nTF }
20696         {##2}
20697       }
20698       { \exp_not:N \use_ii:nn }
20699       \__prop_show_end:NNN
20700       \__prop_show_bad_name:NNN
20701     }
20702   }

```

```

20703     \exp_last_unbraced:NNNo \_prop_show_loop:NNw #1 #4 #4
20704 }

```

The loop will consist of calls to `_prop_show_loop:NNw _prop <prefix> <token> <expansion>`, where `<token>` is one of the items in the list, specifically the key container for `<keyi-1>` (starting at $i = 1$ with the property list variable itself), and `<expansion>` stands for the expansion of that token, which has already been checked, and takes the form `<junk> \s_prop {<value>} _prop <prefix> <keyi>`. Thus, the loop auxiliary receives the prefix command as #1, and the $(i - 1)$ -th and i -th key containers as #2 and #5. Then `_prop_tmp:w` checks that the name of the i -th key container is valid.

```

20705 \cs_new_protected:Npn \_prop_show_loop:NNw #1#2 #3 \s\_prop #4#5
20706 {
20707     \exp_last_two_unbraced:Noo \_prop_tmp:w
20708     { \token_to_str:N #5 \s\_prop_mark }
20709     { \token_to_str:N #1 \s\_prop_mark \s\_prop_stop }
20710     #1 #2 #5
20711 }

```

If the i -th key container has the wrong name we get `_prop_show_bad_name:NNN _prop <prefix> <previous container> <current container with bad name>`.

```

20712 \cs_new_protected:Npn \_prop_show_bad_name:NNN #1#2#3
20713 {
20714     \msg_error:nneeee { prop } { bad-link }
20715     { \tl_tail:N \l\_prop_internal_tl }
20716     { \token_to_str:N #2 }
20717     { \token_to_str:N #3 }
20718     { \token_to_str:N #1 }
20719 }

```

If the i -th key container has the name `_prop <prefix>` (without space), it is the trailing one. We check that it is the right kind of macro to be a token list, and that it has the right contents `\use_none:n <previous container>`. If so, we are done checking everything, and we display the property list using the message function and property list name stored in `\l_prop_internal_tl`. Note that we also use this `\l_prop_internal_tl` in the type argument of `_kernel_chk_tl_type:NnnT`, to build up the name “`<property list> prop entry`” used in error messages.

```

20720 \cs_new_protected:Npn \_prop_show_end:NNN #1#2#3
20721 {
20722     \_kernel_chk_tl_type:NnnT #3
20723     { \tl_tail:N \l\_prop_internal_tl prop~entry }
20724     { \exp_not:n { \use_none:n #2 } }
20725     {
20726         \exp_after:wN \_prop_show_finally:NNn
20727         \l\_prop_internal_tl { linked }
20728     }
20729 }

```

If the i -th container has a name `_prop <prefix> <key>` (with a space before the key), then we have a call to `_prop_show_loop_key:wNNN {<key>} <junk1> <junk2> _prop <prefix> <previous container> <current container>`. (with an `f`-expansion to eliminate the space). The first argument is the `<key>` without a leading space, thanks to a judicious `f`-expansion earlier on. We check that the `<current container>` is a token list with the expected structure `\use_none:n <previous container> _prop_pair:wn <string> \s_prop {<anything>} <single token>`. The auxiliary `_prop_show_flat:w`

is reused to produce the `__prop_pair:wn` part, and the last token is produced by `\tl_item:Nn` (we don't waste a specialized auxiliary to speed that up). If the check succeed, move on to the next item.

```

20730 \cs_new_protected:Npn \__prop_show_loop_key:wNnN #1#2#3#4#5#6
20731 {
20732   \__kernel_chk_tl_type:NnnT #6
20733   { \tl_tail:N \l__prop_internal_tl prop-entry }
20734   {
20735     \exp_not:n { \use_none:n #5 }
20736     \exp_after:wN \__prop_show_flat:w #6 \s__prop { }
20737     \__prop_pair:wn \q__prop_recursion_tail \s__prop { }
20738     \q__prop_recursion_stop
20739     \tl_item:Nn #6 { -1 }
20740   }
20741   { \exp_last_unbraced:NNNo \__prop_show_loop:NNw #4 #6 #6 }
20742 }

```

(End of definition for `\prop_show:N` and others. These functions are documented on page 221.)

```

20743 \endpackage

```

Chapter 64

l3skip implementation

```
20744 ⟨*package⟩
20745 ⟨@@=dim⟩
```

64.1 Length primitives renamed

```
⟨if_dim:w Primitives renamed.
⟨_dim_eval:w 20746 ⟨cs_new_eq:NN ⟨if_dim:w ⟨tex_ifdim:D
⟨_dim_eval_end: 20747 ⟨cs_new_eq:NN ⟨_dim_eval:w ⟨tex_dimexpr:D
20748 ⟨cs_new_eq:NN ⟨_dim_eval_end: ⟨tex_relax:D
```

(End of definition for ⟨if_dim:w, ⟨_dim_eval:w, and ⟨_dim_eval_end:. This function is documented on page 238.)

64.2 Internal auxiliaries

```
⟨s_dim_mark Internal scan marks.
⟨s_dim_stop 20749 ⟨scan_new:N ⟨s_dim_mark
20750 ⟨scan_new:N ⟨s_dim_stop
```

(End of definition for ⟨s_dim_mark and ⟨s_dim_stop.)

```
⟨_dim_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.
20751 ⟨cs_new:Npn ⟨_dim_use_none_delimit_by_s_stop:w #1 ⟨s_dim_stop { }
```

(End of definition for ⟨_dim_use_none_delimit_by_s_stop:w.)

64.3 Creating and initialising dim variables

```
⟨dim_new:N Allocating ⟨dim⟩ registers ...
⟨dim_new:c 20752 ⟨cs_new_protected:Npn ⟨dim_new:N #1
20753 {
20754 ⟨_kernel_chk_if_free_cs:N #1
20755 ⟨cs:w newdimen ⟨cs_end: #1
20756 }
20757 ⟨cs_generate_variant:Nn ⟨dim_new:N { c }
```

(End of definition for `\dim_new:N`. This function is documented on page 223.)

`\dim_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants. We cannot use `\dim_gset:Nn` because debugging code would complain that the constant is not a global variable. Since `\dim_const:Nn` does not need to be fast, use `\dim_eval:n` to avoid needing a debugging patch that wraps the expression in checking code.

```
20758 \cs_new_protected:Npn \dim_const:Nn #1#2
20759   {
20760     \dim_new:N #1
20761     \tex_global:D #1 = \dim_eval:n {#2} \scan_stop:
20762   }
20763 \cs_generate_variant:Nn \dim_const:Nn { c }
```

(End of definition for `\dim_const:Nn`. This function is documented on page 223.)

`\dim_zero:N` Reset the register to zero. Using `\c_zero_skip` deals with the case where the variable passed is incorrectly a skip (for example a L^AT_EX 2_ε length). Besides, these functions are then simply copied for `\skip_zero:N` and related functions.

```
\dim_zero:c
\dim_gzero:N
\dim_gzero:c
20764 \cs_new_protected:Npn \dim_zero:N #1 { #1 = \c_zero_skip }
20765 \cs_new_protected:Npn \dim_gzero:N #1
20766   { \tex_global:D #1 = \c_zero_skip }
20767 \cs_generate_variant:Nn \dim_zero:N { c }
20768 \cs_generate_variant:Nn \dim_gzero:N { c }
```

(End of definition for `\dim_zero:N` and `\dim_gzero:N`. These functions are documented on page 223.)

`\dim_zero_new:N` Create a register if needed, otherwise clear it.

```
\dim_zero_new:c
\dim_gzero_new:N
\dim_gzero_new:c
20769 \cs_new_protected:Npn \dim_zero_new:N #1
20770   { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
20771 \cs_new_protected:Npn \dim_gzero_new:N #1
20772   { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
20773 \cs_generate_variant:Nn \dim_zero_new:N { c }
20774 \cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(End of definition for `\dim_zero_new:N` and `\dim_gzero_new:N`. These functions are documented on page 223.)

`\dim_if_exist_p:N` Copies of the cs functions defined in l3basics.

```
\dim_if_exist_p:c
\dim_if_exist:NTF
\dim_if_exist:cTF
20775 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
20776   { TF , T , F , p }
20777 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
20778   { TF , T , F , p }
```

(End of definition for `\dim_if_exist:NTF`. This function is documented on page 224.)

64.4 Setting dim variables

`\dim_set:Nn` Setting dimensions is easy enough but when debugging we want both to check that the variable is correctly local/global and to wrap the expression in some code. The `\scan_stop:` deals with the case where the variable passed is a skip (for example a L^AT_EX 2_ε length).

```
\dim_set:c
\dim_gset:Nn
\dim_gset:c
20779 \cs_new_protected:Npn \dim_set:Nn #1#2
20780   { #1 = \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
```



```

20781 \cs_new_protected:Npn \dim_gset:Nn #1#2
20782 { \tex_global:D #1 = \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
20783 \cs_generate_variant:Nn \dim_set:Nn { c }
20784 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End of definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 224.)

`\dim_set_eq:NN` All straightforward, with a `\scan_stop:` to deal with the case where #1 is (incorrectly) a skip.

```

\dim_set_eq:cN
\dim_set_eq:Nc
\dim_set_eq:cc
\dim_gset_eq:NN
\dim_gset_eq:cN
\dim_gset_eq:Nc
\dim_gset_eq:cc
20785 \cs_new_protected:Npn \dim_set_eq:NN #1#2
20786 { #1 = #2 \scan_stop: }
20787 \cs_generate_variant:Nn \dim_set_eq:NN { c , Nc , cc }
20788 \cs_new_protected:Npn \dim_gset_eq:NN #1#2
20789 { \tex_global:D #1 = #2 \scan_stop: }
20790 \cs_generate_variant:Nn \dim_gset_eq:NN { c , Nc , cc }

```

(End of definition for `\dim_set_eq:NN` and `\dim_gset_eq:NN`. These functions are documented on page 224.)

`\dim_add:Nn` Using by here would slow things down just to detect nonsensical cases such as passing `\dimen 123` as the first argument. Using `\scan_stop:` deals with skip variables. Since debugging checks that the variable is correctly local/global, the global versions cannot be defined as `\tex_global:D` followed by the local versions.

```

\dim_add:cN
\dim_gadd:Nn
\dim_gadd:cN
\dim_sub:Nn
\dim_sub:cN
\dim_gsub:Nn
\dim_gsub:cN
20791 \cs_new_protected:Npn \dim_add:Nn #1#2
20792 { \tex_advance:D #1 \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
20793 \cs_new_protected:Npn \dim_gadd:Nn #1#2
20794 {
20795   \tex_global:D \tex_advance:D #1
20796   \__dim_eval:w #2 \__dim_eval_end: \scan_stop:
20797 }
20798 \cs_generate_variant:Nn \dim_add:Nn { c }
20799 \cs_generate_variant:Nn \dim_gadd:Nn { c }
20800 \cs_new_protected:Npn \dim_sub:Nn #1#2
20801 { \tex_advance:D #1 - \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
20802 \cs_new_protected:Npn \dim_gsub:Nn #1#2
20803 {
20804   \tex_global:D \tex_advance:D #1
20805   -\__dim_eval:w #2 \__dim_eval_end: \scan_stop:
20806 }
20807 \cs_generate_variant:Nn \dim_sub:Nn { c }
20808 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End of definition for `\dim_add:Nn` and others. These functions are documented on page 224.)

64.5 Utilities for dimension calculations

`\dim_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading - if present.

```

\__dim_abs:N
\dim_max:nn
\dim_min:nn
\__dim_maxmin:wwN
20809 \cs_new:Npn \dim_abs:n #1
20810 {
20811   \exp_after:wN \__dim_abs:N
20812   \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
20813 }

```

```

20814 \cs_new:Npn \__dim_abs:N #1
20815   { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
20816 \cs_new:Npn \dim_max:nn #1#2
20817   {
20818     \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
20819     \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
20820     \dim_use:N \__dim_eval:w #2 ;
20821     >
20822     \__dim_eval_end:
20823   }
20824 \cs_new:Npn \dim_min:nn #1#2
20825   {
20826     \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
20827     \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
20828     \dim_use:N \__dim_eval:w #2 ;
20829     <
20830     \__dim_eval_end:
20831   }
20832 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
20833   {
20834     \if_dim:w #1 #3 #2 ~
20835     #1
20836     \else:
20837     #2
20838     \fi:
20839   }

```

(End of definition for `\dim_abs:n` and others. These functions are documented on page 224.)

`\dim_ratio:nn` `__dim_ratio:n` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` does not work. Instead, the ratio part needs to be converted to an integer expression. Using `\int_value:w` forces everything into `sp`, avoiding any decimal parts.

```

20840 \cs_new:Npn \dim_ratio:nn #1#2
20841   { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
20842 \cs_new:Npn \__dim_ratio:n #1
20843   { \int_value:w \__dim_eval:w (#1) \__dim_eval_end: }

```

(End of definition for `\dim_ratio:nn` and `__dim_ratio:n`. This function is documented on page 225.)

64.6 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

```

\dim_compare:nNnTF
20844 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
20845   {
20846     \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
20847     \prg_return_true: \else: \prg_return_false: \fi:
20848   }

```

(End of definition for `\dim_compare:nNnTF`. This function is documented on page 225.)

`\dim_compare_p:n` `\dim_compare:nTF` `__dim_compare:w` `__dim_compare:wNN` `__dim_compare:=w` `__dim_compare!=w` `__dim_compare<:w` `__dim_compare>:w` `__dim_compare_error:` This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `__dim_compare_error:`. Just like for integers, the looping auxiliary `__dim_compare:wNN` closes a primitive conditional and opens a new one. It is actually easier to

grab a dimension operand than an integer one, because once evaluated, dimensions all end with `pt` (with category other). Thus we do not need specific auxiliaries for the three “simple” relations `<`, `=`, and `>`.

```

20849 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
20850 {
20851   \exp_after:wN \__dim_compare:w
20852   \dim_use:N \__dim_eval:w #1 \__dim_compare_error:
20853 }
20854 \cs_new:Npn \__dim_compare:w #1 \__dim_compare_error:
20855 {
20856   \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
20857   \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \s__dim_stop
20858 }
20859 \exp_args:Nno \use:nn
20860 { \cs_new:Npn \__dim_compare:wNN #1 } { \tl_to_str:n {pt} #2#3 }
20861 {
20862   \if_meaning:w = #3
20863   \use:c { __dim_compare_#2:w }
20864   \fi:
20865   #1 pt \exp_stop_f:
20866   \prg_return_false:
20867   \exp_after:wN \__dim_use_none_delimit_by_s_stop:w
20868   \fi:
20869   \reverse_if:N \if_dim:w #1 pt #2
20870   \exp_after:wN \__dim_compare:wNN
20871   \dim_use:N \__dim_eval:w #3
20872 }
20873 \cs_new:cpn { __dim_compare_ ! :w }
20874 #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
20875 \cs_new:cpn { __dim_compare_ = :w }
20876 #1 \__dim_eval:w = { #1 \__dim_eval:w }
20877 \cs_new:cpn { __dim_compare_ < :w }
20878 #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
20879 \cs_new:cpn { __dim_compare_ > :w }
20880 #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
20881 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \s__dim_stop
20882 { #1 \prg_return_false: \else: \prg_return_true: \fi: }
20883 \cs_new_protected:Npn \__dim_compare_error:
20884 {
20885   \if_int_compare:w \c_zero_int \c_zero_int \fi:
20886   =
20887   \__dim_compare_error:
20888 }

```

(End of definition for `\dim_compare:nTF` and others. This function is documented on page 226.)

`\dim_case:nn` For dimension cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\str_case:nnTF` as described in l3basics.

```

\dim_case:nnTF
\dim_case:nnTF
\__dim_case:nnTF
\__dim_case:nw
\__dim_case_end:nw
20889 \cs_new:Npn \dim_case:nnTF #1
20890 {
20891   \exp:w
20892   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
20893 }
20894 \cs_new:Npn \dim_case:nnT #1#2#3

```

```

20895 {
20896   \exp:w
20897   \exp_args:Nf \_dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
20898 }
20899 \cs_new:Npn \dim_case:nnF #1#2
20900 {
20901   \exp:w
20902   \exp_args:Nf \_dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
20903 }
20904 \cs_new:Npn \dim_case:nn #1#2
20905 {
20906   \exp:w
20907   \exp_args:Nf \_dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
20908 }
20909 \cs_new:Npn \_dim_case:nnTF #1#2#3#4
20910 { \_dim_case:nw {#1} #2 {#1} { } \s_dim_mark {#3} \s_dim_mark {#4} \s_dim_stop }
20911 \cs_new:Npn \_dim_case:nw #1#2#3
20912 {
20913   \dim_compare:nNnTF {#1} = {#2}
20914   { \_dim_case_end:nw {#3} }
20915   { \_dim_case:nw {#1} }
20916 }
20917 \cs_new:Npn \_dim_case_end:nw #1#2#3 \s_dim_mark #4#5 \s_dim_stop
20918 { \exp_end: #1 #4 }

```

(End of definition for `\dim_case:nnTF` and others. This function is documented on page 227.)

64.7 Dimension expression loops

`\dim_while_do:nn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_while_do:nn 20919 \cs_new:Npn \dim_while_do:nn #1#2
\dim_until_do:nn 20920 {
\dim_do_while:nn 20921   \dim_compare:nT {#1}
20922   {
\dim_do_until:nn 20923     #2
20924     \dim_while_do:nn {#1} {#2}
20925   }
20926 }
20927 \cs_new:Npn \dim_until_do:nn #1#2
20928 {
20929   \dim_compare:nF {#1}
20930   {
20931     #2
20932     \dim_until_do:nn {#1} {#2}
20933   }
20934 }
20935 \cs_new:Npn \dim_do_while:nn #1#2
20936 {
20937   #2
20938   \dim_compare:nT {#1}
20939   { \dim_do_while:nn {#1} {#2} }
20940 }

```

```

20941 \cs_new:Npn \dim_do_until:nn #1#2
20942 {
20943   #2
20944   \dim_compare:nF {#1}
20945   { \dim_do_until:nn {#1} {#2} }
20946 }

```

(End of definition for `\dim_while_do:nn` and others. These functions are documented on page 228.)

`\dim_while_do:nNnn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_until_do:nNnn
\dim_do_while:nNnn
\dim_do_until:nNnn
20947 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
20948 {
20949   \dim_compare:nNnT {#1} #2 {#3}
20950   {
20951     #4
20952     \dim_while_do:nNnn {#1} #2 {#3} {#4}
20953   }
20954 }
20955 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4
20956 {
20957   \dim_compare:nNnF {#1} #2 {#3}
20958   {
20959     #4
20960     \dim_until_do:nNnn {#1} #2 {#3} {#4}
20961   }
20962 }
20963 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
20964 {
20965   #4
20966   \dim_compare:nNnT {#1} #2 {#3}
20967   { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
20968 }
20969 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4
20970 {
20971   #4
20972   \dim_compare:nNnF {#1} #2 {#3}
20973   { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
20974 }

```

(End of definition for `\dim_while_do:nNnn` and others. These functions are documented on page 228.)

64.8 Dimension step functions

`\dim_step_function:nnnN` Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

20975 \cs_new:Npn \dim_step_function:nnnN #1#2#3
20976 {
20977   \exp_after:wN \_dim_step:wwwN
20978   \tex_the:D \_dim_eval:w #1 \exp_after:wN ;

```

```

20979     \tex_the:D \__dim_eval:w #2 \exp_after:wN ;
20980     \tex_the:D \__dim_eval:w #3 ;
20981   }
20982 \cs_new:Npn \__dim_step:wwwN #1; #2; #3; #4
20983   {
20984     \dim_compare:nNnTF {#2} > \c_zero_dim
20985     { \__dim_step:NnnnN > }
20986     {
20987       \dim_compare:nNnTF {#2} = \c_zero_dim
20988       {
20989         \msg_expandable_error:nnn { kernel } { zero-step } {#4}
20990         \use_none:nnnn
20991       }
20992       { \__dim_step:NnnnN < }
20993     }
20994     {#1} {#2} {#3} #4
20995   }
20996 \cs_new:Npn \__dim_step:NnnnN #1#2#3#4#5
20997   {
20998     \dim_compare:nNnF {#2} #1 {#4}
20999     {
21000       #5 {#2}
21001       \exp_args:Nnf \__dim_step:NnnnN
21002       #1 { \dim_eval:n { #2 + #3 } } {#3} {#4} #5
21003     }
21004   }

```

(End of definition for `\dim_step_function:nnnN`, `__dim_step:wwwN`, and `__dim_step:NnnnN`. This function is documented on page 228.)

`\dim_step_inline:nnnn`
`\dim_step_variable:nnnNn`
`__dim_step:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\dim_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

21005 \cs_new_protected:Npn \dim_step_inline:nnnn
21006   {
21007     \int_gincr:N \g__kernel_prg_map_int
21008     \exp_args:NNc \__dim_step:NNnnnn
21009     \cs_gset_protected:Npn
21010     { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
21011   }
21012 \cs_new_protected:Npn \dim_step_variable:nnnNn #1#2#3#4#5
21013   {
21014     \int_gincr:N \g__kernel_prg_map_int
21015     \exp_args:NNc \__dim_step:NNnnnn
21016     \cs_gset_protected:Npe
21017     { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
21018     {#1}{#2}{#3}
21019     {
21020       \tl_set:Nn \exp_not:N #4 {##1}
21021       \exp_not:n {#5}
21022     }

```

```

21023 }
21024 \cs_new_protected:Npn \__dim_step:NNnnnn #1#2#3#4#5#6
21025 {
21026   #1 #2 ##1 {#6}
21027   \dim_step_function:nnnN {#3} {#4} {#5} #2
21028   \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
21029 }

```

(End of definition for `\dim_step_inline:nnnn`, `\dim_step_variable:nnnN`, and `__dim_step:NNnnnn`. These functions are documented on page 228.)

64.9 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

21030 \cs_new:Npn \dim_eval:n #1
21031 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End of definition for `\dim_eval:n`. This function is documented on page 229.)

`\dim_sign:n` See `\dim_abs:n`. Contrarily to `\int_sign:n` the case of a zero dimension cannot be distinguished from a positive dimension by looking only at the first character, since `0.2pt` and `0pt` start the same way. We need explicit comparisons. We start by distinguishing the most common case of a positive dimension.

`__dim_sign:Nw`

```

21032 \cs_new:Npn \dim_sign:n #1
21033 {
21034   \int_value:w \exp_after:wN \__dim_sign:Nw
21035   \dim_use:N \__dim_eval:w #1 \__dim_eval_end: ;
21036   \exp_stop_f:
21037 }
21038 \cs_new:Npn \__dim_sign:Nw #1#2 ;
21039 {
21040   \if_dim:w #1#2 > \c_zero_dim
21041     1
21042   \else:
21043     \if_meaning:w - #1
21044     -1
21045   \else:
21046     0
21047   \fi:
21048   \fi:
21049 }

```

(End of definition for `\dim_sign:n` and `__dim_sign:Nw`. This function is documented on page 229.)

`\dim_use:N` Accessing a $\langle dim \rangle$. We hand-code the `c` variant for some speed gain.

```

21050 \cs_new_eq:NN \dim_use:N \tex_the:D
21051 \cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End of definition for `\dim_use:N`. This function is documented on page 229.)

`\dim_to_decimal:n` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

`__dim_to_decimal:w`

```

21052 \cs_new:Npn \dim_to_decimal:n #1
21053 {
21054   \exp_after:wN
21055   \__dim_to_decimal:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
21056 }
21057 \use:e
21058 {
21059   \cs_new:Npn \exp_not:N \__dim_to_decimal:w
21060   #1 . #2 \tl_to_str:n { pt }
21061 }
21062 {
21063   \int_compare:nNnTF {#2} > \c_zero_int
21064   { #1 . #2 }
21065   { #1 }
21066 }

```

(End of definition for `\dim_to_decimal:n` and `__dim_to_decimal:w`. This function is documented on page 229.)

`\dim_to_fp:n` Defined in `l3fp-convert`, documented here.

(End of definition for `\dim_to_fp:n`. This function is documented on page 231.)

64.10 Conversion of `dim` to other units

The conversion from `pt` or `sp` to other units is complicated by the fact that `TeX`'s conversion to `sp` involves rounding and hard-coded ratios. In order to give re-entrant outcomes, we therefore need to do quite a bit of work: see <https://github.com/latex3/latex3/issues/954> for detailed discussion. After dealing with the trivial case, we therefore have some work to do. The code to do this is contributed by Ruixi Zhang.

`\dim_to_decimal_in_sp:n` The one easy case: the only requirement here is that we avoid an overflow.

```

21067 \cs_new:Npn \dim_to_decimal_in_sp:n #1
21068 { \int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End of definition for `\dim_to_decimal_in_sp:n`. This function is documented on page 231.)

`\dim_to_decimal_in_bp:n` We first set up a helper macro `__dim_tmp:w` which takes two arguments. The first
`\dim_to_decimal_in_cc:n` argument is one of the following engine-defined units: `in`, `pc`, `cm`, `mm`, `bp`, `dd`, `cc`, `nd`,
`\dim_to_decimal_in_cm:n` and `nc`. The second argument is $\frac{1}{2}\delta^{-1}$ in reduced fraction, where $\delta > 1$ is the engine-
`\dim_to_decimal_in_dd:n` defined conversion factor for each unit. Note that δ must be strictly larger than 1 for the
`\dim_to_decimal_in_in:n` following algorithm to work.

`\dim_to_decimal_in_mm:n` Here is how the algorithm works: Suppose that a user inputs a non-negative di-
`\dim_to_decimal_in_pc:n` mension in a unit that has conversion factor $\delta > 1$. Then this dimension is internally
`__dim_to_decimal_aux:w` represented as X `sp`, where $X = \lfloor N\delta \rfloor$ for some integer $N \geq 0$. We then seek a formula
to express this N using X . The `\dim_to_decimal_in_<unit>:n` functions shall return
the number $N/2^{16}$ in decimal. This way, we guarantee the returned decimal followed by
the original unit will parse to exactly X `sp`.

So how do we get N from X ? Well, since $X = \lfloor N\delta \rfloor$, we have $X \leq N\delta < X + 1$ and $X\delta^{-1} \leq N < (X + 1)\delta^{-1}$. Let's focus on the midpoint of this bounding interval for N . The midpoint is $(X + \frac{1}{2})\delta^{-1}$. The fact $\delta > 1$ implies that the bounding interval is shorter than 1 in length. Thus, (1) midpoint + $\frac{1}{2} > N$ and (2) midpoint + $\frac{1}{2} < N + 1$. In other words, $N = \lfloor \text{midpoint} + \frac{1}{2} \rfloor$. As long as we can rewrite the midpoint as the result of

a “scaling operation” of ε -TeX, the $[\dots + \frac{1}{2}]$ part will follow naturally. Indeed we can: $\text{midpoint} = (2X + 1) \times (\frac{1}{2}\delta^{-1})$.

Addendum: If $\delta \geq 2$, then the bounding interval for N is at most $\frac{1}{2}$ wide in length. In this case, the leftpoint $X\delta^{-1}$ suffices as $N = \lfloor X\delta^{-1} + \frac{1}{2} \rfloor$. Six out of the nine units listed above can be handled in this way, which is much simpler than using midpoint. But three remaining units have $1 < \delta < 2$; they are **bp** ($\delta = 7227/7200$), **nd** ($\delta = 685/642$), and **dd** ($\delta = 1238/1157$), and these three must be handled using midpoint. For consistency, we shall use the midpoint approach for all nine units.

```

21069 \group_begin:
21070   \cs_set_protected:Npn \__dim_tmp:w #1#2
21071     {
21072       \cs_new:cpn { dim_to_decimal_in_ #1 :n } ##1
21073         {
21074           \exp_after:wN \__dim_to_decimal_aux:w
21075             \int_value:w \__dim_eval:w ##1 \__dim_eval_end: ; #2 ;
21076         }
21077     }

```

Conversions to other units are now coded. Consult the pdfTeX source for each conversion factor δ . Each factor $\frac{1}{2}\delta^{-1}$ is hand-coded for accuracy (and speed). As the units **nc** and **nd** are not supported by XeTeX or (u)pTeX, they are not included here.

```

21078   \__dim_tmp:w { in } { 50 / 7227 } % delta = 7227/100
21079   \__dim_tmp:w { pc } { 1 / 24 } % delta = 12/1
21080   \__dim_tmp:w { cm } { 127 / 7227 } % delta = 7227/254
21081   \__dim_tmp:w { mm } { 1270 / 7227 } % delta = 7227/2540
21082   \__dim_tmp:w { bp } { 400 / 803 } % delta = 7227/7200
21083   \__dim_tmp:w { dd } { 1157 / 2476 } % delta = 1238/1157
21084   \__dim_tmp:w { cc } { 1157 / 29712 } % delta = 14856/1157
21085 \group_end:

```

The tokens after `__dim_to_decimal_aux:w` shall have the following form: `<number>;<half of delta i` where `<number>` represents the input dimension in **sp** unit. If `<number>` is positive, then `#1` is its leading digit and `#2` (possibly empty) is all the remaining digits; If `<number>` is zero, then `#1` is `012` and `#2` is empty; If `<number>` is negative, then `#1` is its sign `-12` and `#2` is all its digits. In all three cases, `#1#2` is the original `<number>`. We can use `#1` to decide whether to use the `-1` formula or the `+1` formula.

```

21086 \cs_new:Npn \__dim_to_decimal_aux:w #1#2 ; #3 ;
21087   {
21088     \dim_to_decimal:n
21089     {

```

We need different formulae depending on whether the user input dimension is negative or not. For negative dimension (internally represented as X sp), the formula is $(2X - 1) \times (\frac{1}{2}\delta^{-1})$. For non-negative dimension, the formula is $(2X + 1) \times (\frac{1}{2}\delta^{-1})$. The intermediate step doubles the dimension X . To avoid overflow, we must invoke `\int_eval:n`.

```

21090       \int_eval:n
21091       { ( 2 * #1#2 \if:w #1 - - \else: + \fi: 1 ) * #3 }

```

Now we append **sp** to finish the dimension specification.

```

21092         sp
21093     }
21094 }

```

(End of definition for `\dim_to_decimal_in_bp:n` and others. These functions are documented on page 230.)

`\dim_to_decimal_in_unit:nn`

```
21095 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
21096 {
21097   \exp_after:wN \__dim_chk_unit:w
21098   \int_value:w \__dim_eval:w #2 \__dim_eval_end: ; {#1}
21099 }
```

(End of definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 231.)

`__dim_chk_unit:w` The tokens after `__dim_chk_unit:w` shall have the following form: `<number2>;{<dimexpr1>}`, where `<number2>` represents `<dimexpr2>` in `sp` unit. If `#1` is `012`, the “unit” `<dimexpr2>` must also be zero. So we throw out a “division by zero” error message at this point. Otherwise, if `#1` is `-12`, we shall negate both `<dimexpr1>` and `<dimexpr2>` for later procedures.

```
21100 \cs_new:Npn \__dim_chk_unit:w #1#2;#3
21101 {
21102   \token_if_eq_charcode:NNTF #1 0
21103   { \msg_expandable_error:nn { dim } { zero-unit } }
21104   {
21105     \exp_after:wN \__dim_branch_unit:w
21106     \int_value:w \if:w #1 - - \fi: \__dim_eval:w #3 \exp_after:wN ;
21107     \int_value:w \if:w #1 - - \fi: #1#2 ;
21108   }
21109 }
```

(End of definition for `__dim_chk_unit:w`.)

`__dim_branch_unit:w` The tokens after `__dim_branch_unit:w` shall have the following form: `<number1>;<number2>;`, where `<number1>` represents `<dimexpr1>` in `sp` unit (whose sign is taken care of) and `<number2>` represents the absolute value of `<dimexpr2>` in `sp` unit (which is strictly positive).

As explained, the formulae $(2X \pm 1) \times (\frac{1}{2} \delta^{-1})$ work if and only if $\delta = \langle \text{number2} \rangle / 65536 > 1$. This corresponds to `<dimexpr2>` strictly larger than 1 pt in absolute value. In this case, we simply call `__dim_to_decimal_aux:w` and supply $\frac{1}{2} \delta^{-1} = 32768 / \langle \text{number2} \rangle$ as `<half of delta inverse>`.

Otherwise if `<number2> = 65536`, then `<dimexpr2>` is 1 pt in absolute value and we call `\dim_to_decimal:n` directly.

Otherwise $0 < \langle \text{number2} \rangle < 65536$ and we shall proceed differently.

For unit less than 1 pt, write $n = \langle \text{number2} \rangle$, then $\delta = n / 65536 < 1$. The midpoint formulae are not optimal. Let’s go back to the inequalities $X \delta^{-1} \leq N < (X + 1) \delta^{-1}$. Since now $\delta < 1$, the bounding interval is wider than 1 in length. Consider the ceiling integer $M = \lceil X \delta^{-1} \rceil$, then $X \delta^{-1} \leq M < (X + 1) \delta^{-1}$, or equivalently $X \leq M \delta < X + 1$, and thus $\lfloor M \delta \rfloor = X$. The key point here is that we *don’t* need to solve for N ; in fact, any integer that can reproduce X (such as M) is good enough. So the algorithm goes like this: (1) Compute rounding of $X \delta^{-1}$, i.e., $M' = \lfloor X \delta^{-1} + \frac{1}{2} \rfloor$; this M' could be either M or $M - 1$. (2) Check if $\lfloor M' \delta \rfloor = X$, i.e., whether our candidate M' can reproduce X . If so, then this M' is good enough; if not, then we add one to M' .

But when $0 < n < 65536$, we cannot delay the problem of overflow any more. For $X \delta^{-1} = X \times 65536 / n$, where X can go up to $2^{30} - 1$ and n can be as small as 1, the result is well over $2^{31} - 1$ (largest integer allowed within `\numexpr`). For example, `\dim_to_decimal_in_unit:nn { \maxdimen } { 1sp }`. Here, all inputs are legal, so we should be able to output 1073741823 *without* causing arithmetic overflow.

As a workaround, let's write $X = qn + r$ with some $q \geq 0$ and $0 \leq r < n$. Then $X\delta^{-1} = 65536q + 65536r/n$, and so $M' = 65536q + \lfloor 65536r/n + \frac{1}{2} \rfloor = 65536q + R'$. Computing R' will never overflow. If this R' can reproduce r , then it is good enough; otherwise we add one to R' . In the end, we shall output $q + R'/65536$ in decimal.

Note: $q = \lfloor X/n \rfloor = \lfloor \frac{2X-n}{2n} + \frac{1}{2} \rfloor$ represents the “integer” part, while $0 \leq R' \leq 65536$ represents the “fractional” part. (Can $R' = 65536$ really happen? Didn't investigate.)

```

21110 \cs_new:Npn \__dim_branch_unit:w #1;#2;
21111 {
21112   \int_compare:nNnTF {#2} > { 65536 }
21113     { \__dim_to_decimal_aux:w #1 ; 32768 / #2 ; }
21114     {
21115       \int_compare:nNnTF {#2} = { 65536 }
21116         { \dim_to_decimal:n { #1sp } }
21117         { \__dim_get_quotient:w #1 ; #2 ; }
21118     }
21119 }

```

(End of definition for `__dim_branch_unit:w`.)

`__dim_get_quotient:w` We wish to get the quotient q via rounding of $\frac{2X-n}{2n}$. When $0 \leq X < n/2$, we have $\frac{2X-n}{2n} < 0$. So, strictly speaking, `\numexpr` performs its rounding as $\lceil \frac{2X-n}{2n} - \frac{1}{2} \rceil$, not exactly what we want. However, lucky for us, only $X = 0$ makes $\lceil \frac{2X-n}{2n} - \frac{1}{2} \rceil = -1 \neq 0$ (we want 0); all other $0 < X < n/2$ make $\lceil \frac{2X-n}{2n} - \frac{1}{2} \rceil = 0 = q$. Thus, let's filter out $X = 0$ early. If $X \neq 0$, we extract its sign and leave the sign to the back. The sign does not participate in any calculations (also the code works with positive integers only). The sign is used at the last stages when we parse the decimal output.

After `__dim_get_quotient:w` has done its job, either we have the decimal 0, or we have `__dim_get_remainder:w` followed by $q;|X|;n;<\text{sign of } X>;$.

```

21120 \cs_new:Npn \__dim_get_quotient:w #1#2;#3;
21121 {
21122   \token_if_eq_charcode:NNTF #1 0
21123     { 0 }
21124     {
21125       \token_if_eq_charcode:NNTF #1 -
21126         {
21127           \exp_after:wN \exp_after:wN \exp_after:wN \__dim_get_remainder:w
21128             \int_eval:n { ( 2 * #2 - #3 ) / ( 2 * #3 ) } ;
21129             #2 ; #3 ; - ;
21130         }
21131         {
21132           \exp_after:wN \exp_after:wN \exp_after:wN \__dim_get_remainder:w
21133             \int_eval:n { ( 2 * #1#2 - #3 ) / ( 2 * #3 ) } ;
21134             #1#2 ; #3 ; ;
21135         }
21136     }
21137 }

```

(End of definition for `__dim_get_quotient:w`.)

`__dim_get_remainder:w` `__dim_get_remainder:w` does not need to read the sign. After finding the remainder r , the number $|X|$ is no longer needed. We should then have `__dim_convert_remainder:w` followed by $r;n;q;<\text{sign of } X>;$.

```

21138 \cs_new:Npn \__dim_get_remainder:w #1;#2;#3;
21139 {
21140   \exp_after:wN \exp_after:wN \exp_after:wN \__dim_convert_remainder:w
21141   \int_eval:n { #2 - #1 * #3 } ;
21142   #3 ; #1 ;
21143 }

```

(End of definition for `__dim_get_remainder:w`.)

`__dim_convert_remainder:w` This is trivial. We compute $R' = \lfloor 65536r/n + \frac{1}{2} \rfloor$, then leave `__dim_test_candidate:w` followed by $R';r;n;q;<\text{sign of } X>$;

```

21144 \cs_new:Npn \__dim_convert_remainder:w #1;#2;
21145 {
21146   \exp_after:wN \exp_after:wN \exp_after:wN \__dim_test_candidate:w
21147   \int_eval:n { #1 * 65536 / #2 } ;
21148   #1 ; #2 ;
21149 }

```

(End of definition for `__dim_convert_remainder:w`.)

`__dim_test_candidate:w` Now the fun part: We take R' , r and n to test whether $r = \lfloor R'\delta \rfloor$. This is done as a dimension comparison. The left-hand side, r , is simply `r sp`. The right-hand side, $\lfloor R'\delta \rfloor$, is exactly `<R' as decimal><dimen = n sp>`. If the result is true, then we've found R' ; otherwise we add one to R' . After this step, r and n are no longer needed. We should then have `__dim_parse_decimal:w` followed by $R';q;<\text{sign of } X>$;

```

21150 \cs_new:Npn \__dim_test_candidate:w #1;#2;#3;
21151 {
21152   \dim_compare:nNnTF { #2sp } =
21153   { \dim_to_decimal:n { #1sp } \__dim_eval:w #3sp \__dim_eval_end: }
21154   { \__dim_parse_decimal:w #1 ; }
21155   {
21156     \__dim_parse_decimal:w \int_eval:n { #1 + 1 } ;
21157   }
21158 }

```

(End of definition for `__dim_test_candidate:w`.)

`__dim_parse_decimal:w` The Grand Finale: We sum q and $R'/65536$ together, and negate the result if necessary.
`__dim_parse_decimal_aux:w` These are all done expandably. If $0 < R'/65536 < 1$, the integer summation is naturally terminated at the decimal point. If $R'/65536 = 0$ (or 1?), the summation is terminated at the semicolon. The auxiliary function `__dim_parse_decimal_aux:w` takes care of both cases.

```

21159 \cs_new:Npn \__dim_parse_decimal:w #1;#2;#3;
21160 {
21161   \exp_after:wN \__dim_parse_decimal_aux:w
21162   \int_value:w #3 \int_eval:w #2 + \dim_to_decimal:n { #1sp } ;
21163 }
21164 \cs_new:Npn \__dim_parse_decimal_aux:w #1 ; {#1}

```

(End of definition for `__dim_parse_decimal:w` and `__dim_parse_decimal_aux:w`.)

64.11 Viewing dim variables

`\dim_show:N` Diagnostics.

```
\dim_show:c 21165 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N  
21166 \cs_generate_variant:Nn \dim_show:N { c }
```

(End of definition for \dim_show:N. This function is documented on page 231.)

`\dim_show:n` Diagnostics. We don't use the T_EX primitive `\showthe` to show dimension expressions: this gives a more unified output.

```
21167 \cs_new_protected:Npn \dim_show:n  
21168 { \__kernel_msg_show_eval:Nn \dim_eval:n }
```

(End of definition for \dim_show:n. This function is documented on page 231.)

`\dim_log:N` Diagnostics. Redirect output of `\dim_show:n` to the log.

```
\dim_log:c 21169 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N  
\dim_log:n 21170 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c  
21171 \cs_new_protected:Npn \dim_log:n  
21172 { \__kernel_msg_log_eval:Nn \dim_eval:n }
```

(End of definition for \dim_log:N and \dim_log:n. These functions are documented on page 231.)

64.12 Constant dimensions

`\c_zero_dim` Constant dimensions.

```
\c_max_dim 21173 \dim_const:Nn \c_zero_dim { 0 pt }  
21174 \dim_const:Nn \c_max_dim { 16383.99999 pt }
```

(End of definition for \c_zero_dim and \c_max_dim. These variables are documented on page 232.)

64.13 Scratch dimensions

`\l_tmpa_dim` We provide two local and two global scratch registers, maybe we need more or less.

```
\l_tmpb_dim 21175 \dim_new:N \l_tmpa_dim  
\g_tmpa_dim 21176 \dim_new:N \l_tmpb_dim  
\g_tmpb_dim 21177 \dim_new:N \g_tmpa_dim  
21178 \dim_new:N \g_tmpb_dim
```

(End of definition for \l_tmpa_dim and others. These variables are documented on page 232.)

64.14 Creating and initialising skip variables

```
21179 <@@=skip>
```

`\s__skip_stop` Internal scan marks.

```
21180 \scan_new:N \s__skip_stop
```

(End of definition for \s__skip_stop.)

`\skip_new:N` Allocation of a new internal registers.
`\skip_new:c`

```

21181 \cs_new_protected:Npn \skip_new:N #1
21182 {
21183   \__kernel_chk_if_free_cs:N #1
21184   \cs:w newskip \cs_end: #1
21185 }
21186 \cs_generate_variant:Nn \skip_new:N { c }

```

(End of definition for \skip_new:N. This function is documented on page 232.)

`\skip_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants. See
`\skip_const:cn` `\dim_const:Nn` for why we cannot use `\skip_gset:Nn`.

```

21187 \cs_new_protected:Npn \skip_const:Nn #1#2
21188 {
21189   \skip_new:N #1
21190   \tex_global:D #1 = \skip_eval:n {#2} \scan_stop:
21191 }
21192 \cs_generate_variant:Nn \skip_const:Nn { c }

```

(End of definition for \skip_const:Nn. This function is documented on page 232.)

`\skip_zero:N` Reset the register to zero.

```

\skip_zero:c 21193 \cs_new_eq:NN \skip_zero:N \dim_zero:N
\skip_gzero:N 21194 \cs_new_eq:NN \skip_gzero:N \dim_gzero:N
\skip_gzero:c 21195 \cs_generate_variant:Nn \skip_zero:N { c }
21196 \cs_generate_variant:Nn \skip_gzero:N { c }

```

(End of definition for \skip_zero:N and \skip_gzero:N. These functions are documented on page 232.)

`\skip_zero_new:N` Create a register if needed, otherwise clear it.

```

\skip_zero_new:c 21197 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N 21198 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c 21199 \cs_new_protected:Npn \skip_gzero_new:N #1
21200 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
21201 \cs_generate_variant:Nn \skip_zero_new:N { c }
21202 \cs_generate_variant:Nn \skip_gzero_new:N { c }

```

(End of definition for \skip_zero_new:N and \skip_gzero_new:N. These functions are documented on page 233.)

`\skip_if_exist_p:N` Copies of the cs functions defined in l3basics.

```

\skip_if_exist_p:c 21203 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
\skip_if_exist:NTF 21204 { TF , T , F , p }
\skip_if_exist:cTF 21205 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
21206 { TF , T , F , p }

```

(End of definition for \skip_if_exist:NTF. This function is documented on page 233.)

64.15 Setting skip variables

`\skip_set:Nn` Much the same as for dimensions.
`\skip_set:cn` 21207 `\cs_new_protected:Npn \skip_set:Nn #1#2`
`\skip_gset:Nn` 21208 `{ #1 = \tex_glueexpr:D #2 \scan_stop: }`
`\skip_gset:cn` 21209 `\cs_new_protected:Npn \skip_gset:Nn #1#2`
 21210 `{ \tex_global:D #1 = \tex_glueexpr:D #2 \scan_stop: }`
 21211 `\cs_generate_variant:Nn \skip_set:Nn { c }`
 21212 `\cs_generate_variant:Nn \skip_gset:Nn { c }`

(End of definition for `\skip_set:Nn` and `\skip_gset:Nn`. These functions are documented on page 233.)

`\skip_set_eq:NN` All straightforward.
`\skip_set_eq:cN` 21213 `\cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }`
`\skip_set_eq:Nc` 21214 `\cs_generate_variant:Nn \skip_set_eq:NN { c , Nc , cc }`
`\skip_set_eq:cc` 21215 `\cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }`
`\skip_gset_eq:NN` 21216 `\cs_generate_variant:Nn \skip_gset_eq:NN { c , Nc , cc }`

`\skip_gset_eq:cN`
`\skip_gset_eq:Nc`
`\skip_gset_eq:cc`
(End of definition for `\skip_set_eq:NN` and `\skip_gset_eq:NN`. These functions are documented on page 233.)

`\skip_add:Nn` Using by here deals with the (incorrect) case `\skip123`.
`\skip_add:cn` 21217 `\cs_new_protected:Npn \skip_add:Nn #1#2`
`\skip_gadd:Nn` 21218 `{ \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }`
`\skip_gadd:cn` 21219 `\cs_new_protected:Npn \skip_gadd:Nn #1#2`
`\skip_sub:Nn` 21220 `{ \tex_global:D \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }`
`\skip_sub:cn` 21221 `\cs_generate_variant:Nn \skip_add:Nn { c }`
`\skip_gsub:Nn` 21222 `\cs_generate_variant:Nn \skip_gadd:Nn { c }`
`\skip_gsub:cn` 21223 `\cs_new_protected:Npn \skip_sub:Nn #1#2`
 21224 `{ \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }`
 21225 `\cs_new_protected:Npn \skip_gsub:Nn #1#2`
 21226 `{ \tex_global:D \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }`
 21227 `\cs_generate_variant:Nn \skip_sub:Nn { c }`
 21228 `\cs_generate_variant:Nn \skip_gsub:Nn { c }`

(End of definition for `\skip_add:Nn` and others. These functions are documented on page 233.)

64.16 Skip expression conditionals

`\skip_if_eq_p:nn` Comparing skips means doing two expansions to make strings, and then testing them.
`\skip_if_eq_nnTF` As a result, only equality is tested.

21229 `\prg_new_conditional:Npnm \skip_if_eq:nn #1#2 { p , T , F , TF }`
 21230 `{`
 21231 `\str_if_eq:eeTF { \skip_eval:n {#1} } { \skip_eval:n {#2} }`
 21232 `{ \prg_return_true: }`
 21233 `{ \prg_return_false: }`
 21234 `}`

(End of definition for `\skip_if_eq:nnTF`. This function is documented on page 234.)

`\skip_if_finite_p:n` With ε -TeX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

```

21235 \cs_set_protected:Npn \__skip_tmp:w #1
21236   {
21237     \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
21238     {
21239       \exp_after:wN \__skip_if_finite:wwNw
21240       \skip_use:N \tex_glueexpr:D ##1 ; \prg_return_false:
21241       #1 ; \prg_return_true: \s__skip_stop
21242     }
21243     \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \s__skip_stop {##3}
21244   }
21245 \exp_args:No \__skip_tmp:w { \tl_to_str:n { fil } }

```

(End of definition for `\skip_if_finite:nTF` and `__skip_if_finite:wwNw`. This function is documented on page 234.)

64.17 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

21246 \cs_new:Npn \skip_eval:n #1
21247   { \skip_use:N \tex_glueexpr:D #1 \scan_stop: }

```

(End of definition for `\skip_eval:n`. This function is documented on page 234.)

`\skip_use:N` Accessing a $\langle skip \rangle$.

```

\skip_use:c 21248 \cs_new_eq:NN \skip_use:N \dim_use:N
21249 \cs_new_eq:NN \skip_use:c \dim_use:c

```

(End of definition for `\skip_use:N`. This function is documented on page 234.)

64.18 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.

```

\skip_horizontal:c 21250 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\skip_horizontal:n 21251 \cs_new:Npn \skip_horizontal:n #1
\skip_vertical:N 21252   { \skip_horizontal:N \tex_glueexpr:D #1 \scan_stop: }
\skip_vertical:c 21253 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
\skip_vertical:n 21254 \cs_new:Npn \skip_vertical:n #1
21255   { \skip_vertical:N \tex_glueexpr:D #1 \scan_stop: }
21256 \cs_generate_variant:Nn \skip_horizontal:N { c }
21257 \cs_generate_variant:Nn \skip_vertical:N { c }

```

(End of definition for `\skip_horizontal:N` and others. These functions are documented on page 235.)

64.19 Viewing skip variables

`\skip_show:N` Diagnostics.

```
\skip_show:c 21258 \cs_new_eq:NN \skip_show:N \__kernel_register_show:N
21259 \cs_generate_variant:Nn \skip_show:N { c }
```

(End of definition for \skip_show:N. This function is documented on page 234.)

`\skip_show:n` Diagnostics. We don't use the T_EX primitive `\showthe` to show skip expressions: this gives a more unified output.

```
21260 \cs_new_protected:Npn \skip_show:n
21261 { \__kernel_msg_show_eval:Nn \skip_eval:n }
```

(End of definition for \skip_show:n. This function is documented on page 234.)

`\skip_log:N` Diagnostics. Redirect output of `\skip_show:n` to the log.

```
\skip_log:c 21262 \cs_new_eq:NN \skip_log:N \__kernel_register_log:N
\skip_log:n 21263 \cs_new_eq:NN \skip_log:c \__kernel_register_log:c
21264 \cs_new_protected:Npn \skip_log:n
21265 { \__kernel_msg_log_eval:Nn \skip_eval:n }
```

(End of definition for \skip_log:N and \skip_log:n. These functions are documented on page 235.)

64.20 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.

```
\c_max_skip 21266 \skip_const:Nn \c_zero_skip { \c_zero_dim }
21267 \skip_const:Nn \c_max_skip { \c_max_dim }
```

(End of definition for \c_zero_skip and \c_max_skip. These functions are documented on page 235.)

64.21 Scratch skips

`\l_tmpa_skip` We provide two local and two global scratch registers, maybe we need more or less.

```
\l_tmpb_skip 21268 \skip_new:N \l_tmpa_skip
\g_tmpa_skip 21269 \skip_new:N \l_tmpb_skip
\g_tmpb_skip 21270 \skip_new:N \g_tmpa_skip
21271 \skip_new:N \g_tmpb_skip
```

(End of definition for \l_tmpa_skip and others. These variables are documented on page 235.)

64.22 Creating and initialising muskip variables

`\muskip_new:N` And then we add muskips.

```
\muskip_new:c 21272 \cs_new_protected:Npn \muskip_new:N #1
21273 {
21274 \__kernel_chk_if_free_cs:N #1
21275 \cs:w newmuskip \cs_end: #1
21276 }
21277 \cs_generate_variant:Nn \muskip_new:N { c }
```

(End of definition for \muskip_new:N. This function is documented on page 236.)

`\muskip_const:Nn` See `\skip_const:Nn`.
`\muskip_const:cn`

```

21278 \cs_new_protected:Npn \muskip_const:Nn #1#2
21279 {
21280   \muskip_new:N #1
21281   \tex_global:D #1 = \muskip_eval:n {#2} \scan_stop:
21282 }
21283 \cs_generate_variant:Nn \muskip_const:Nn { c }

```

(End of definition for `\muskip_const:Nn`. This function is documented on page 236.)

`\muskip_zero:N` Reset the register to zero.
`\muskip_zero:c`
`\muskip_gzero:N`
`\muskip_gzero:c`

```

21284 \cs_new_protected:Npn \muskip_zero:N #1
21285 { #1 = \c_zero_muskip }
21286 \cs_new_protected:Npn \muskip_gzero:N #1
21287 { \tex_global:D #1 = \c_zero_muskip }
21288 \cs_generate_variant:Nn \muskip_zero:N { c }
21289 \cs_generate_variant:Nn \muskip_gzero:N { c }

```

(End of definition for `\muskip_zero:N` and `\muskip_gzero:N`. These functions are documented on page 236.)

`\muskip_zero_new:N` Create a register if needed, otherwise clear it.
`\muskip_zero_new:c`
`\muskip_gzero_new:N`
`\muskip_gzero_new:c`

```

21290 \cs_new_protected:Npn \muskip_zero_new:N #1
21291 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
21292 \cs_new_protected:Npn \muskip_gzero_new:N #1
21293 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
21294 \cs_generate_variant:Nn \muskip_zero_new:N { c }
21295 \cs_generate_variant:Nn \muskip_gzero_new:N { c }

```

(End of definition for `\muskip_zero_new:N` and `\muskip_gzero_new:N`. These functions are documented on page 236.)

`\muskip_if_exist_p:N` Copies of the cs functions defined in l3basics.
`\muskip_if_exist_p:c`
`\muskip_if_exist:NTF`
`\muskip_if_exist:cTF`

```

21296 \prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
21297 { TF , T , F , p }
21298 \prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
21299 { TF , T , F , p }

```

(End of definition for `\muskip_if_exist:NTF`. This function is documented on page 236.)

64.23 Setting muskip variables

`\muskip_set:Nn` This should be pretty familiar.
`\muskip_set:cn`
`\muskip_gset:Nn`
`\muskip_gset:cn`

```

21300 \cs_new_protected:Npn \muskip_set:Nn #1#2
21301 { #1 = \tex_muexpr:D #2 \scan_stop: }
21302 \cs_new_protected:Npn \muskip_gset:Nn #1#2
21303 { \tex_global:D #1 = \tex_muexpr:D #2 \scan_stop: }
21304 \cs_generate_variant:Nn \muskip_set:Nn { c }
21305 \cs_generate_variant:Nn \muskip_gset:Nn { c }

```

(End of definition for `\muskip_set:Nn` and `\muskip_gset:Nn`. These functions are documented on page 237.)

`\muskip_set_eq:NN` All straightforward.

```

\muskip_set_eq:cN 21306 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\muskip_set_eq:Nc 21307 \cs_generate_variant:Nn \muskip_set_eq:NN { c , Nc , cc }
\muskip_set_eq:cc 21308 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\muskip_gset_eq:NN 21309 \cs_generate_variant:Nn \muskip_gset_eq:NN { c , Nc , cc }
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc

```

(End of definition for `\muskip_set_eq:NN` and `\muskip_gset_eq:NN`. These functions are documented on page 237.)

`\muskip_add:Nn` Using by here deals with the (incorrect) case `\muskip123`.

```

\muskip_add:cN 21310 \cs_new_protected:Npn \muskip_add:Nn #1#2
\muskip_gadd:Nn 21311 { \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
\muskip_gadd:cN 21312 \cs_new_protected:Npn \muskip_gadd:Nn #1#2
\muskip_sub:Nn 21313 { \tex_global:D \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
\muskip_sub:cN 21314 \cs_generate_variant:Nn \muskip_add:Nn { c }
\muskip_gsub:Nn 21315 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
\muskip_gsub:cN 21316 \cs_new_protected:Npn \muskip_sub:Nn #1#2
21317 { \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
21318 \cs_new_protected:Npn \muskip_gsub:Nn #1#2
21319 { \tex_global:D \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
21320 \cs_generate_variant:Nn \muskip_sub:Nn { c }
21321 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

```

(End of definition for `\muskip_add:Nn` and others. These functions are documented on page 236.)

64.24 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```

21322 \cs_new:Npn \muskip_eval:n #1
21323 { \muskip_use:N \tex_muexpr:D #1 \scan_stop: }

```

(End of definition for `\muskip_eval:n`. This function is documented on page 237.)

`\muskip_use:N` Accessing a $\langle muskip \rangle$.

```

\muskip_use:c 21324 \cs_new_eq:NN \muskip_use:N \dim_use:N
21325 \cs_new_eq:NN \muskip_use:c \dim_use:c

```

(End of definition for `\muskip_use:N`. This function is documented on page 237.)

64.25 Viewing muskip variables

`\muskip_show:N` Diagnostics.

```

\muskip_show:c 21326 \cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
21327 \cs_generate_variant:Nn \muskip_show:N { c }

```

(End of definition for `\muskip_show:N`. This function is documented on page 237.)

`\muskip_show:n` Diagnostics. We don't use the \TeX primitive `\showthe` to show muskip expressions: this gives a more unified output.

```

21328 \cs_new_protected:Npn \muskip_show:n
21329 { \__kernel_msg_show_eval:Nn \muskip_eval:n }

```

(End of definition for `\muskip_show:n`. This function is documented on page 238.)

`\muskip_log:N` Diagnostics. Redirect output of `\muskip_show:n` to the log.
`\muskip_log:c` 21330 `\cs_new_eq:NN \muskip_log:N __kernel_register_log:N`
`\muskip_log:n` 21331 `\cs_new_eq:NN \muskip_log:c __kernel_register_log:c`
21332 `\cs_new_protected:Npn \muskip_log:n`
21333 `{ __kernel_msg_log_eval:Nn \muskip_eval:n }`

(End of definition for `\muskip_log:N` and `\muskip_log:n`. These functions are documented on page 238.)

64.26 Constant muskips

`\c_zero_muskip` Constant muskips given by their value.
`\c_max_muskip` 21334 `\muskip_const:Nn \c_zero_muskip { 0 mu }`
21335 `\muskip_const:Nn \c_max_muskip { 16383.99999 mu }`

(End of definition for `\c_zero_muskip` and `\c_max_muskip`. These functions are documented on page 238.)

64.27 Scratch muskips

`\l_tmpa_muskip` We provide two local and two global scratch registers, maybe we need more or less.
`\l_tmpb_muskip` 21336 `\muskip_new:N \l_tmpa_muskip`
`\g_tmpa_muskip` 21337 `\muskip_new:N \l_tmpb_muskip`
`\g_tmpb_muskip` 21338 `\muskip_new:N \g_tmpa_muskip`
21339 `\muskip_new:N \g_tmpb_muskip`

(End of definition for `\l_tmpa_muskip` and others. These variables are documented on page 238.)

21340 `\</package>`

Chapter 65

l3keys implementation

21341 `<*package>`

65.1 Low-level interface

The low-level key parser's implementation is based heavily on `expkv`. Compared to `keyval` it adds a number of additional "safety" requirements and allows to process the parsed list of key-value pairs in a variety of ways. The net result is that this code needs around one and a half the amount of time as `keyval` to parse the same list of keys. To optimise speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level `expl3` interfaces.

21342 `<@@=keyval>`

```
\s__keyval_nil
\s__keyval_mark 21343 \scan_new:N \s__keyval_nil
\s__keyval_stop 21344 \scan_new:N \s__keyval_mark
\s__keyval_tail 21345 \scan_new:N \s__keyval_stop
                21346 \scan_new:N \s__keyval_tail
```

(End of definition for `\s__keyval_nil` and others.)

`\l__kernel_keyval_allow_blank_keys_bool`

The general behavior of the `l3keys` module is to throw an error on blank key names. However to support the usage of `\keyval_parse:nnn` in the `l3prop` module we allow this error to be switched off temporarily and just ignore blank names.

21347 `\bool_new:N \l__kernel_keyval_allow_blank_keys_bool`

(End of definition for `\l__kernel_keyval_allow_blank_keys_bool`.)

This temporary macro will be used since some of the definitions will need an active comma or equals sign. Inside of this macro `#1` will be the active comma and `#2` will be the active equals sign.

```
21348 \group_begin:
21349   \cs_set_protected:Npn \__keyval_tmp:w #1#2
21350   {
```

```
\keyval_parse:nnn
\keyval_parse:nnV
\keyval_parse:nnv
\keyval_parse:NNn
\keyval_parse:NNV
\keyval_parse:NNv
```

The main function starts the first of two loops. The outer loop splits the key-value list at active commas, the inner loop will do so at other commas. The use of `\s__keyval_mark` here prevents loss of braces from the key argument.

```

21351 \cs_new:Npn \keyval_parse:nnn ##1 ##2 ##3
21352 {
21353   \__kernel_exp_not:w \tex_expanded:D
21354   {
21355     {
21356       \__keyval_loop_active:nnw {##1} {##2}
21357       \s__keyval_mark ##3 #1 \s__keyval_tail #1
21358     }
21359   }
21360 }
21361 \cs_new_eq:NN \keyval_parse:NNn \keyval_parse:nnn

```

(End of definition for `\keyval_parse:nnn` and `\keyval_parse:NNn`. These functions are documented on page 253.)

`__keyval_loop_active:nnw` First a fast test for the end of the loop is done, it'll gobble everything up to a `\s__keyval_tail`. The loop ending macro will gobble everything to the last comma in this definition. If the end isn't reached yet, start the second loop splitting at other commas, the next iteration of this first loop will be inserted by the end of `__keyval_loop_other:nnw`.

```

21362 \cs_new:Npn \__keyval_loop_active:nnw ##1 ##2 ##3 #1
21363 {
21364   \__keyval_if_recursion_tail:w ##3
21365   \__keyval_end_loop_active:w \s__keyval_tail
21366   \__keyval_loop_other:nnw {##1} {##2} ##3 , \s__keyval_tail ,
21367 }

```

(End of definition for `__keyval_loop_active:nnw`.)

`__keyval_split_other:w` These two macros allow to split at the first equals sign of category 12 or 13. At the same time they also execute branching by inserting the first token following `\s__keyval_mark` that followed the equals sign. Hence they also test for the presence of such an equals sign simultaneously.

```

21368 \cs_new:Npn \__keyval_split_other:w ##1 = ##2 \s__keyval_mark ##3
21369 { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }
21370 \cs_new:Npn \__keyval_split_active:w ##1 #2 ##2 \s__keyval_mark ##3
21371 { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }

```

(End of definition for `__keyval_split_other:w` and `__keyval_split_active:w`.)

`__keyval_loop_other:nnw` The second loop uses the same test for its end as the first loop, next it splits at the first active equals sign using `__keyval_split_active:w`. The `\s__keyval_nil` prevents accidental brace stripping and acts as a delimiter in the next steps. First testing for an active equals sign will reduce the number of necessary expansion steps for the expected average use case of other equals signs and hence perform better on average.

```

21372 \cs_new:Npn \__keyval_loop_other:nnw ##1 ##2 ##3 ,
21373 {
21374   \__keyval_if_recursion_tail:w ##3
21375   \__keyval_end_loop_other:w \s__keyval_tail
21376   \__keyval_split_active:w ##3 \s__keyval_nil
21377   \s__keyval_mark \__keyval_split_active_auxi:w
21378   #2 \s__keyval_mark \__keyval_clean_up_active:w
21379   {##1} {##2}
21380   \s__keyval_mark
21381 }

```

(End of definition for `__keyval_loop_other:nw`.)

`__keyval_split_active_auxi:w`
`__keyval_split_active_auxii:w`
`__keyval_split_active_auxiii:w`
`__keyval_split_active_auxiv:w`
`__keyval_split_active_auxv:w`

After `__keyval_split_active:w` the following will only be called if there was at least one active equals sign in the current key–value pair. Therefore this is the execution branch for a key–value pair with an active equals sign. `##1` will be everything up to the first active equals sign. First it tests for other equals signs in the key name, which will eventually throw an error via `__keyval_misplaced_equal_after_active_error:w`. If none was found we forward the key to `__keyval_split_active_auxii:w`.

```
21382     \cs_new:Npn \__keyval_split_active_auxi:w ##1 \s__keyval_stop
21383     {
21384         \__keyval_split_other:w ##1 \s__keyval_nil
21385         \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
21386         = \s__keyval_mark \__keyval_split_active_auxii:w
21387     }
```

`__keyval_split_active_auxii:w` gets the correct key name with a leading `\s__keyval_mark` as `##1`. It has to sanitise the remainder of the previous test and trims the key name which will be forwarded to `__keyval_split_active_auxiii:w`.

```
21388     \cs_new:Npn \__keyval_split_active_auxii:w
21389     ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
21390     \s__keyval_stop \s__keyval_mark
21391     ##2 \s__keyval_nil #2 \s__keyval_mark \__keyval_clean_up_active:w
21392     { \__keyval_trim:nN {##1} \__keyval_split_active_auxiii:w ##2 \s__keyval_nil }
```

Next we test for a misplaced active equals sign in the value, if none is found `__keyval_split_active_auxiv:w` will be called.

```
21393     \cs_new:Npn \__keyval_split_active_auxiii:w ##1 ##2 \s__keyval_nil
21394     {
21395         \__keyval_split_active:w ##2 \s__keyval_nil
21396         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
21397         #2 \s__keyval_mark \__keyval_split_active_auxiv:w
21398         {##1}
21399     }
```

This runs the last test after sanitising the remainder of the previous one. This time test for a misplaced equals sign of category 12 in the value. Finally the last auxiliary macro will be called.

```
21400     \cs_new:Npn \__keyval_split_active_auxiv:w
21401     ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
21402     \s__keyval_stop \s__keyval_mark
21403     {
21404         \__keyval_split_other:w ##1 \s__keyval_nil
21405         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
21406         = \s__keyval_mark \__keyval_split_active_auxv:w
21407     }
```

This last macro in this execution branch sanitises the last test, trims the value and passes it to `__keyval_pair:nwnn`.

```
21408     \cs_new:Npn \__keyval_split_active_auxv:w
21409     ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
21410     \s__keyval_stop \s__keyval_mark
21411     { \__keyval_trim:nN { ##1 } \__keyval_pair:nwnn }
```

(End of definition for `__keyval_split_active_auxi:w` and others.)

`__keyval_clean_up_active:w` The following is the branch taken if the key–value pair doesn’t contain an active equals sign. The remainder of that test will be cleaned up by `__keyval_clean_up_active:w` which will then split at an equals sign of category other.

```

21412     \cs_new:Npn \__keyval_clean_up_active:w
21413         ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_active_auxi:w \s__keyval_stop
21414     {
21415         \__keyval_split_other:w ##1 \s__keyval_nil
21416         \s__keyval_mark \__keyval_split_other_auxi:w
21417         = \s__keyval_mark \__keyval_clean_up_other:w
21418     }

```

(End of definition for `__keyval_clean_up_active:w`.)

`__keyval_split_other_auxi:w` This is executed if the key–value pair doesn’t contain an active equals sign but at least one other. `##1` of `__keyval_split_other_auxi:w` will contain the complete key name, which is trimmed and forwarded to the next auxiliary macro.

```

21419     \cs_new:Npn \__keyval_split_other_auxi:w ##1 \s__keyval_stop
21420     { \__keyval_trim:nN { ##1 } \__keyval_split_other_auxii:w }

```

We know that the value doesn’t contain misplaced active equals signs but we have to test for others. Also we need to sanitise the previous test, which is done here and not earlier to avoid superfluous argument grabbing.

```

21421     \cs_new:Npn \__keyval_split_other_auxii:w
21422         ##1 ##2 \s__keyval_nil = \s__keyval_mark \__keyval_clean_up_other:w
21423     {
21424         \__keyval_split_other:w ##2 \s__keyval_nil
21425         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
21426         = \s__keyval_mark \__keyval_split_other_auxiii:w
21427         { ##1 }
21428     }

```

`__keyval_split_other_auxiii:w` sanitises the test for other equals signs, trims the value and forwards it to `__keyval_pair:n`.

```

21429     \cs_new:Npn \__keyval_split_other_auxiii:w
21430         ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
21431         \s__keyval_stop \s__keyval_mark
21432     { \__keyval_trim:nN { ##1 } \__keyval_pair:n }

```

(End of definition for `__keyval_split_other_auxi:w`, `__keyval_split_other_auxii:w`, and `__keyval_split_other_auxiii:w`.)

`__keyval_clean_up_other:w` `__keyval_clean_up_other:w` is the last branch that might exist. It is called if no equals sign was found, hence the only possibilities left are a blank list element, which is to be skipped, or a lonely key. If it’s no empty list element this will trim the key name and forward it to `__keyval_key:n`.

```

21433     \cs_new:Npn \__keyval_clean_up_other:w
21434         ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_other_auxi:w \s__keyval_stop \
21435     {
21436         \__keyval_if_blank:w ##1 \s__keyval_nil \s__keyval_stop \__keyval_blank_true:w
21437         \s__keyval_mark \s__keyval_stop
21438         \__keyval_trim:nN { ##1 } \__keyval_key:n
21439     }

```

(End of definition for `__keyval_clean_up_other:w`.)

keyval_misplaced_equal_after_active_error:w All these two macros do is gobble the remainder of the current other loop execution and
 _keyval_misplaced_equal_in_split_error:w throw an error. Afterwards they have to insert the next loop iteration.

```

21440 \cs_new:Npn \_keyval_misplaced_equal_after_active_error:w
21441 \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
21442 = \s__keyval_mark \_keyval_split_active_auxii:w
21443 \s__keyval_mark ##3 \s__keyval_nil
21444 #2 \s__keyval_mark \_keyval_clean_up_active:w
21445 {
21446 \msg_expandable_error:nn
21447 { keyval } { misplaced-equals-sign }
21448 \_keyval_loop_other:nnw
21449 }
21450 \cs_new:Npn \_keyval_misplaced_equal_in_split_error:w
21451 \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
21452 ##3 \s__keyval_mark ##4 ##5
21453 {
21454 \msg_expandable_error:nn
21455 { keyval } { misplaced-equals-sign }
21456 \_keyval_loop_other:nnw
21457 }

```

(End of definition for _keyval_misplaced_equal_after_active_error:w and _keyval_misplaced_equal_in_split_error:w.)

_keyval_end_loop_other:w All that's left for the parsing loops are the macros which end the recursion. Both just
 _keyval_end_loop_active:w gobble the remaining tokens of the respective loop including the next recursion call.
 _keyval_end_loop_other:w also has to insert the next iteration of the active loop.

```

21458 \cs_new:Npn \_keyval_end_loop_other:w
21459 \s__keyval_tail
21460 \_keyval_split_active:w
21461 \s__keyval_mark \s__keyval_tail
21462 \s__keyval_nil \s__keyval_mark
21463 \_keyval_split_active_auxi:w
21464 #2 \s__keyval_mark \_keyval_clean_up_active:w
21465 { \_keyval_loop_active:nnw }
21466 \cs_new:Npn \_keyval_end_loop_active:w
21467 \s__keyval_tail
21468 \_keyval_loop_other:nnw ##1 \s__keyval_mark \s__keyval_tail , \s__keyval_tail ,
21469 { }

```

(End of definition for _keyval_end_loop_other:w and _keyval_end_loop_active:w.)

The parsing loops are done, so here ends the definition of _keyval_tmp:w, which will finally set up the macros.

```

21470 }
21471 \char_set_catcode_active:n { '\ , }
21472 \char_set_catcode_active:n { '= }
21473 \_keyval_tmp:w , =
21474 \group_end:
21475 \cs_generate_variant:Nn \keyval_parse:NNn { NNv , NNv }
21476 \cs_generate_variant:Nn \keyval_parse:nnn { nnv , nnv }

```

_keyval_pair:nnnn These macros will be called on the parsed keys and values of the key–value list. All
 _keyval_key:nn arguments are completely trimmed. They test for blank key names and call the func-

tions passed to `\keyval_parse:nnn` inside of `\exp_not:n` with the correct arguments. Afterwards they insert the next iteration of the other loop.

```

21477 \group_begin:
21478   \cs_set_protected:Npn \__keyval_tmp:w #1#2
21479   {
21480     \cs_new:Npn \__keyval_pair:nmmm ##1 ##2 ##3 ##4
21481     {
21482       \__keyval_if_blank:w \s__keyval_mark ##2 \s__keyval_nil \s__keyval_stop \__keyval
21483       \s__keyval_mark \s__keyval_stop
21484       #1
21485       \exp_not:n { ##4 {##2} {##1} }
21486       #2
21487       \__keyval_loop_other:nnw {##3} {##4}
21488     }
21489     \cs_new:Npn \__keyval_key:nn ##1 ##2
21490     {
21491       \__keyval_if_blank:w \s__keyval_mark ##1 \s__keyval_nil \s__keyval_stop \__keyval
21492       \s__keyval_mark \s__keyval_stop
21493       #1
21494       \exp_not:n { ##2 {##1} }
21495       #2
21496       \__keyval_loop_other:nnw {##2}
21497     }
21498   }
21499   \__keyval_tmp:w { } { }
21500 \group_end:

```

(End of definition for `__keyval_pair:nmmm` and `__keyval_key:nn`.)

`__keyval_if_empty:w` `__keyval_if_blank:w` `__keyval_if_recursion_tail:w` All these tests work by gobbling tokens until a certain combination is met, which makes them pretty fast. The test for a blank argument should be called with an arbitrary token following the argument. Each of these utilize the fact that the argument will contain a leading `\s__keyval_mark`.

```

21501 \cs_new:Npn \__keyval_if_empty:w #1 \s__keyval_mark \s__keyval_stop { }
21502 \cs_new:Npn \__keyval_if_blank:w \s__keyval_mark #1 { \__keyval_if_empty:w \s__keyval_mark
21503 \cs_new:Npn \__keyval_if_recursion_tail:w \s__keyval_mark #1 \s__keyval_tail { }

```

(End of definition for `__keyval_if_empty:w`, `__keyval_if_blank:w`, and `__keyval_if_recursion_tail:w`.)

`__keyval_blank_true:w` `__keyval_blank_key_error:w` These macros will be called if the tests above didn't gobble them, they execute the branching.

```

21504 \cs_new:Npn \__keyval_blank_true:w \s__keyval_mark \s__keyval_stop \__keyval_trim:nN #1 \__
21505 { \__keyval_loop_other:nnw }
21506 \cs_new:Npn \__keyval_blank_key_error:w \s__keyval_mark \s__keyval_stop #1 \__keyval_loop_o
21507 {
21508   \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
21509   { #1 }
21510   { \msg_expandable_error:nn { keyval } { blank-key-name } }
21511   \__keyval_loop_other:nnw
21512 }

```

(End of definition for `__keyval_blank_true:w` and `__keyval_blank_key_error:w`.)

Two messages for the low level parsing system.

```

21513 \msg_new:nnn { keyval } { misplaced-equals-sign }
21514 { Misplaced-''-in-key-value-input-\msg_line_context: }
21515 \msg_new:nnn { keyval } { blank-key-name }
21516 { Blank-key-name-in-key-value-input-\msg_line_context: }
21517 \prop_gput:Nnn \g_msg_module_name_prop { keyval } { LaTeX }
21518 \prop_gput:Nnn \g_msg_module_type_prop { keyval } { }

```

`__keyval_trim:nN` And an adapted version of `__tl_trim_spaces:nn` which is a bit faster for our use case, as it can strip the braces at the end. This is pretty much the same concept, so I won't comment on it here. The speed gain by using this instead of `\tl_trim_spaces_apply:nN` is about 10% of the total time for `\keyval_parse:NNn` with one key and one key–value pair, so I think it's worth it.

```

21519 \group_begin:
21520   \cs_set_protected:Npn \__keyval_tmp:w #1
21521     {
21522       \cs_new:Npn \__keyval_trim:nN ##1
21523         {
21524           \__keyval_trim_auxi:w
21525             ##1
21526           \s_keyval_nil
21527           \s_keyval_mark #1 { }
21528           \s_keyval_mark \__keyval_trim_auxii:w
21529           \__keyval_trim_auxiii:w
21530           #1 \s_keyval_nil
21531           \__keyval_trim_auxiv:w
21532         }
21533       \cs_new:Npn \__keyval_trim_auxi:w ##1 \s_keyval_mark #1 ##2 \s_keyval_mark ##3
21534         {
21535           ##3
21536           \__keyval_trim_auxi:w
21537           \s_keyval_mark
21538           ##2
21539           \s_keyval_mark #1 {##1}
21540         }
21541       \cs_new:Npn \__keyval_trim_auxii:w \__keyval_trim_auxi:w \s_keyval_mark \s_keyval_m
21542         {
21543           \__keyval_trim_auxiii:w
21544           ##1
21545         }
21546       \cs_new:Npn \__keyval_trim_auxiii:w ##1 #1 \s_keyval_nil ##2
21547         {
21548           ##2
21549           ##1 \s_keyval_nil
21550           \__keyval_trim_auxiii:w
21551         }

```

This is the one macro which differs from the original definition.

```

21552       \cs_new:Npn \__keyval_trim_auxiv:w
21553         \s_keyval_mark ##1 \s_keyval_nil
21554         \__keyval_trim_auxiii:w \s_keyval_nil \__keyval_trim_auxiii:w
21555         ##2
21556         { ##2 { ##1 } }
21557     }
21558   \__keyval_tmp:w { ~ }

```

21559 `\group_end:`

(End of definition for `__keyval_trim:nN` and others.)

65.2 Constants and variables

21560 `<@@=keys>`

`\c__keys_code_root_str` Various storage areas for the different data which make up keys.
`\c__keys_check_root_str` 21561 `\str_const:Nn \c__keys_code_root_str { key~code~>~ }`
`\c__keys_default_root_str` 21562 `\str_const:Nn \c__keys_check_root_str { key~check~>~ }`
`\c__keys_groups_root_str` 21563 `\str_const:Nn \c__keys_default_root_str { key~default~>~ }`
`\c__keys_inherit_root_str` 21564 `\str_const:Nn \c__keys_groups_root_str { key~groups~>~ }`
`\c__keys_type_root_str` 21565 `\str_const:Nn \c__keys_inherit_root_str { key~inherit~>~ }`
21566 `\str_const:Nn \c__keys_type_root_str { key~type~>~ }`

(End of definition for `\c__keys_code_root_str` and others.)

`\c__keys_props_root_str` The prefix for storing properties.

21567 `\str_const:Nn \c__keys_props_root_str { key~prop~>~ }`

(End of definition for `\c__keys_props_root_str`.)

`\l_keys_choice_int` Publicly accessible data on which choice is being used when several are generated as a set.
`\l_keys_choice_tl`

21568 `\int_new:N \l_keys_choice_int`

21569 `\tl_new:N \l_keys_choice_tl`

(End of definition for `\l_keys_choice_int` and `\l_keys_choice_tl`. These variables are documented on page 246.)

`\l__keys_groups_clist` Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

21570 `\clist_new:N \l__keys_groups_clist`

(End of definition for `\l__keys_groups_clist`.)

`\l__keys_inherit_clist` For normalisation.

21571 `\clist_new:N \l__keys_inherit_clist`

(End of definition for `\l__keys_inherit_clist`.)

`\l_keys_key_str` The name of a key itself: needed when setting keys.

21572 `\str_new:N \l_keys_key_str`

(End of definition for `\l_keys_key_str`. This variable is documented on page 249.)

`\l_keys_key_tl` The `tl` version is deprecated but has to be handled manually.

21573 `\tl_new:N \l_keys_key_tl`

(End of definition for `\l_keys_key_tl`.)

`\l__keys_module_str` The module for an entire set of keys.

21574 `\str_new:N \l__keys_module_str`

(End of definition for `\l__keys_module_str`.)

`\l__keys_no_value_bool` A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.
21575 `\bool_new:N \l__keys_no_value_bool`
(End of definition for \l__keys_no_value_bool.)

`\l__keys_only_known_bool` Used to track if only “known” keys are being set.
21576 `\bool_new:N \l__keys_only_known_bool`
(End of definition for \l__keys_only_known_bool.)

`\l_keys_path_str` The “path” of the current key is stored here: this is available to the programmer and so is public.
21577 `\str_new:N \l_keys_path_str`
(End of definition for \l_keys_path_str. This variable is documented on page 249.)

`\l_keys_path_tl` The older version is deprecated but has to be handled manually.
21578 `\tl_new:N \l_keys_path_tl`
(End of definition for \l_keys_path_tl.)

`\l__keys_inherit_str`
21579 `\str_new:N \l__keys_inherit_str`
(End of definition for \l__keys_inherit_str.)

`\l__keys_relative_tl` The relative path for passing keys back to the user. As this can be explicitly no-value, it must be a token list.
21580 `\tl_new:N \l__keys_relative_tl`
21581 `\tl_set:Nn \l__keys_relative_tl { \q__keys_no_value }`
(End of definition for \l__keys_relative_tl.)

`\l__keys_property_str` The “property” begin set for a key at definition time is stored here.
21582 `\str_new:N \l__keys_property_str`
(End of definition for \l__keys_property_str.)

`\l__keys_selective_bool` Two booleans for using key groups: one to indicate that “selective” setting is active, a
`\l__keys_exclude_bool` second to specify which type (“opt-in” or “opt-out”).
21583 `\bool_new:N \l__keys_selective_bool`
21584 `\bool_new:N \l__keys_exclude_bool`
(End of definition for \l__keys_selective_bool and \l__keys_exclude_bool.)

`\l__keys_selective_clist` The list of key groups being filtered in or out during selective setting.
21585 `\seq_new:N \l__keys_selective_clist`
(End of definition for \l__keys_selective_clist.)

`\l__keys_tmp_clist` Scratch space used as a data dump.
21586 `\seq_new:N \l__keys_tmp_clist`
(End of definition for \l__keys_tmp_clist.)

`\l__keys_unused_clist` Used when setting only some keys to store those left over.
 21587 `\clist_new:N \l__keys_unused_clist`
 (End of definition for `\l__keys_unused_clist`.)

`\l_keys_value_tl` The value given for a key: may be empty if no value was given.
 21588 `\tl_new:N \l_keys_value_tl`
 (End of definition for `\l_keys_value_tl`. This variable is documented on page 249.)

`\l__keys_tmp_bool` Scratch space.
`\l__keys_tmpa_tl` 21589 `\bool_new:N \l__keys_tmp_bool`
`\l__keys_tmpb_tl` 21590 `\tl_new:N \l__keys_tmpa_tl`
 21591 `\tl_new:N \l__keys_tmpb_tl`
 (End of definition for `\l__keys_tmp_bool`, `\l__keys_tmpa_tl`, and `\l__keys_tmpb_tl`.)

`\l__keys_precompile_bool` For digesting keys.
`\l__keys_precompile_tl` 21592 `\bool_new:N \l__keys_precompile_bool`
 21593 `\tl_new:N \l__keys_precompile_tl`
 (End of definition for `\l__keys_precompile_bool` and `\l__keys_precompile_tl`.)

`\l_keys_usage_load_prop` Global data for document-level information.
`\l_keys_usage_preamble_prop` 21594 `\prop_new:N \l_keys_usage_load_prop`
 21595 `\prop_new:N \l_keys_usage_preamble_prop`
 (End of definition for `\l_keys_usage_load_prop` and `\l_keys_usage_preamble_prop`. These variables are documented on page 248.)

65.2.1 Internal auxiliaries

`\s__keys_nil` Internal scan marks.
`\s__keys_mark` 21596 `\scan_new:N \s__keys_nil`
`\s__keys_stop` 21597 `\scan_new:N \s__keys_mark`
 21598 `\scan_new:N \s__keys_stop`
 (End of definition for `\s__keys_nil`, `\s__keys_mark`, and `\s__keys_stop`.)

`\q__keys_no_value` Internal quarks.
 21599 `\quark_new:N \q__keys_no_value`
 (End of definition for `\q__keys_no_value`.)

`__keys_quark_if_no_value_p:N` Branching quark conditional.
`__keys_quark_if_no_value:NTF` 21600 `__kernel_quark_new_conditional:Nn __keys_quark_if_no_value:N { TF }`
 (End of definition for `__keys_quark_if_no_value:NTF`.)

`__keys_precompile:n` An auxiliary to allow cleaner showing of code.
 21601 `\cs_new_protected:Npn __keys_precompile:n #1`
 21602 `{`
 21603 `\bool_if:NTF \l__keys_precompile_bool`
 21604 `{ \tl_put_right:Nn \l__keys_precompile_tl }`
 21605 `{ \use:n }`
 21606 `{#1}`
 21607 `}`

(End of definition for `__keys_precompile:n`.)

`__keys_cs_undefine:c` Local version of `\cs_undefine:c` to avoid sprinkling `\tex_undefined:D` everywhere.

```
21608 \cs_new_protected:Npn \__keys_cs_undefine:c #1
21609   {
21610     \if_cs_exist:w #1 \cs_end:
21611     \else:
21612       \use_i:nnnn
21613     \fi:
21614     \cs_set_eq:cN {#1} \tex_undefined:D
21615   }
```

(End of definition for `__keys_cs_undefine:c`.)

65.3 The key defining mechanism

`\keys_define:nm` The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

`\keys_define:ne`

`\keys_define:nx`

`__keys_define:nnn`

`__keys_define:onn`

```
21616 \cs_new_protected:Npn \keys_define:nm
21617   { \__keys_define:onn \l__keys_module_str }
21618 \cs_generate_variant:Nn \keys_define:nm { ne , nx }
21619 \cs_new_protected:Npn \__keys_define:nnn #1#2#3
21620   {
21621     \str_set:Ne \l__keys_module_str { \__keys_trim_spaces:n {#2} }
21622     \keyval_parse:NNn \__keys_define:n \__keys_define:nm {#3}
21623     \str_set:Nn \l__keys_module_str {#1}
21624   }
21625 \cs_generate_variant:Nn \__keys_define:nnn { o }
```

(End of definition for `\keys_define:nm` and `__keys_define:nnn`. This function is documented on page 240.)

`__keys_define:n`

`__keys_define:nn`

`__keys_define_aux:nn`

The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```
21626 \cs_new_protected:Npn \__keys_define:n #1
21627   {
21628     \bool_set_true:N \l__keys_no_value_bool
21629     \__keys_define_aux:nn {#1} { }
21630   }
21631 \cs_new_protected:Npn \__keys_define:nn #1#2
21632   {
21633     \bool_set_false:N \l__keys_no_value_bool
21634     \__keys_define_aux:nn {#1} {#2}
21635   }
21636 \cs_new_protected:Npn \__keys_define_aux:nn #1#2
21637   {
21638     \__keys_property_find:n {#1}
21639     \cs_if_exist:cTF { \c__keys_props_root_str \l__keys_property_str }
21640     { \__keys_define_code:n {#2} }
21641     {
21642       \str_if_empty:NF \l__keys_property_str
```

```

21643     {
21644         \msg_error:nnee { keys } { property-unknown }
21645         \l__keys_property_str \l_keys_path_str
21646     }
21647 }
21648 }

```

(End of definition for `__keys_define:n`, `__keys_define:nm`, and `__keys_define_aux:nn`.)

`__keys_property_find:n` Searching for a property means finding the last `.` in the input, and storing the text before and after it. Everything is first turned into strings, so there is no problem using `\cs_set_nopar:Npe` instead of `\str_set:Ne` to set `\l_keys_path_str`. To gain further speed, brace tricks are used and `__keys_property_find_auxiv:w` is defined as expandable. Since spaces will already be trimmed from the module we can omit it from the argument to `__keys_trim_spaces:n`.

```

21649 \cs_new_protected:Npn \__keys_property_find:n #1
21650 {
21651     \exp_after:wN \__keys_property_find_auxi:w \tl_to_str:n {#1}
21652     \s__keys_nil \__keys_property_find_auxii:w
21653     . \s__keys_nil \__keys_property_find_err:w
21654 }
21655 \cs_new:Npn \__keys_property_find_auxi:w #1 . #2 \s__keys_nil #3
21656 {
21657     #3 #1 \s__keys_mark #2 \s__keys_nil #3
21658 }
21659 \cs_new_protected:Npn \__keys_property_find_auxii:w
21660 #1 \s__keys_mark #2 \s__keys_nil \__keys_property_find_auxii:w . \s__keys_nil
21661 \__keys_property_find_err:w
21662 {
21663     \cs_set_nopar:Npe \l_keys_path_str
21664     {
21665         \str_if_empty:NF \l__keys_module_str { \l__keys_module_str / }
21666         \exp_after:wN \__keys_trim_spaces:n \tex_expanded:D {
21667             #1
21668             \if_false: }}}} \fi:
21669     \__keys_property_find_auxi:w #2 \s__keys_nil \__keys_property_find_auxiii:w
21670     . \s__keys_nil \__keys_property_find_auxiv:w
21671 }
21672 \cs_new:Npn \__keys_property_find_auxiii:w #1 \s__keys_mark #2 . #3 \s__keys_nil #4
21673 {
21674     . #1 #4 #2 \s__keys_mark #3 \s__keys_nil #4
21675 }
21676 \cs_new:Npn \__keys_property_find_auxiv:w
21677 #1 \s__keys_nil \__keys_property_find_auxiii:w
21678 \s__keys_mark \s__keys_nil \__keys_property_find_auxiv:w
21679 {
21680     \if_false: {{{ \fi: }}}
21681     \cs_set_nopar:Npe \l__keys_property_str { . #1 }
21682     \tl_set_eq:NN \l_keys_path_tl \l_keys_path_str
21683 }
21684 \cs_new_protected:Npn \__keys_property_find_err:w
21685 #1 \s__keys_nil #2 \__keys_property_find_err:w
21686 {
21687     \str_clear:N \l__keys_property_str

```



```

21688     \msg_error:nnn { keys } { no-property } {#1}
21689   }

```

(End of definition for `__keys_property_find:n` and others.)

`__keys_define_code:n` Two possible cases. If there is a value for the key, then just use the function. If not, then
`__keys_define_code:w` a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a `:` as if it was missing the earlier tests would have failed.

```

21690 \cs_new_protected:Npn \__keys_define_code:n #1
21691   {
21692     \bool_if:NTF \l__keys_no_value_bool
21693     {
21694       \exp_after:wN \__keys_define_code:w
21695       \l__keys_property_str \s__keys_stop
21696       { \use:c { \c__keys_props_root_str \l__keys_property_str } }
21697     {
21698       \msg_error:nnee { keys } { property-requires-value }
21699       \l__keys_property_str \l_keys_path_str
21700     }
21701   }
21702   { \use:c { \c__keys_props_root_str \l__keys_property_str } {#1} }
21703 }
21704 \exp_last_unbraced:NNNNo
21705 \cs_new:Npn \__keys_define_code:w #1 \c_colon_str #2 \s__keys_stop
21706   { \tl_if_empty:nTF {#2} }

```

(End of definition for `__keys_define_code:n` and `__keys_define_code:w`.)

65.4 Turning properties into actions

`__keys_bool_set:Nn` Boolean keys are really just choices, but all done by hand. The second argument here is
`__keys_bool_set:cn` the scope: either empty or `g` for global.

```

\__keys_bool_set:Nn 21707 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
\__keys_bool_set:cn 21708   { \__keys_bool_set:Nnnn #1 {#2} { true } { false } }
\__keys_bool_set:Nn 21709 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }
\__keys_bool_set:cn 21710 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
\__keys_bool_set:Nnnn 21711   { \__keys_bool_set:Nnnn #1 {#2} { false } { true } }
21712 \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }
21713 \cs_new_protected:Npn \__keys_bool_set:Nnnn #1#2#3#4
21714   {
21715     \bool_if_exist:NF #1 { \bool_new:N #1 }
21716     \__keys_choice_make:
21717     \__keys_cmd_set:ne { \l_keys_path_str / true }
21718     { \exp_not:c { bool_ #2 set_ #3 :N } \exp_not:N #1 }
21719     \__keys_cmd_set:ne { \l_keys_path_str / false }
21720     { \exp_not:c { bool_ #2 set_ #4 :N } \exp_not:N #1 }
21721     \__keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
21722     {
21723       \msg_error:nne { keys } { boolean-values-only }
21724       \l_keys_path_str
21725     }
21726     \__keys_default_set:n { true }

```

```

21727 }
21728 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

```

(End of definition for __keys_bool_set:Nn, __keys_bool_set_inverse:Nn, and __keys_bool_set:Nnn.)

__keys_choice_make: To make a choice from a key, two steps: set the code, and set the unknown key. As multichoices and choices are essentially the same bar one function, the code is given together.

```

\__keys_multichoices_make:
  \__keys_choice_make:N
\__keys_choice_make_aux:N
21729 \cs_new_protected:Npn \__keys_choice_make:
21730 { \__keys_choice_make:N \__keys_choice_find:n }
21731 \cs_new_protected:Npn \__keys_multichoices_make:
21732 { \__keys_choice_make:N \__keys_multichoices_find:n }
21733 \cs_new_protected:Npn \__keys_choice_make:N #1
21734 {
21735   \cs_if_exist:cTF
21736     { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }
21737     {
21738       \str_if_eq:vnTF
21739         { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }
21740         { choice }
21741         {
21742           \msg_error:nnee { keys } { nested-choice-key }
21743           \l_keys_path_tl { \__keys_parent:o \l_keys_path_str }
21744         }
21745         { \__keys_choice_make_aux:N #1 }
21746       }
21747     { \__keys_choice_make_aux:N #1 }
21748   }
21749 \cs_new_protected:Npn \__keys_choice_make_aux:N #1
21750 {
21751   \cs_set_nopar:cpn { \c__keys_type_root_str \l_keys_path_str }
21752   { choice }
21753   \__keys_cmd_set_direct:nn \l_keys_path_str { #1 {##1} }
21754   \__keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
21755   {
21756     \msg_error:nnee { keys } { choice-unknown }
21757     \l_keys_path_str {##1}
21758   }
21759 }

```

(End of definition for __keys_choice_make: and others.)

__keys_choices_make:nn Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```

\__keys_multichoices_make:nn
  \__keys_choices_make:Nnn
21760 \cs_new_protected:Npn \__keys_choices_make:nn
21761 { \__keys_choices_make:Nnn \__keys_choice_make: }
21762 \cs_new_protected:Npn \__keys_multichoices_make:nn
21763 { \__keys_choices_make:Nnn \__keys_multichoices_make: }
21764 \cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
21765 {
21766   #1
21767   \int_zero:N \l_keys_choice_int
21768   \clist_map_inline:nn {#2}

```

```

21769     {
21770     \int_incr:N \l_keys_choice_int
21771     \__keys_cmd_set:ne
21772     { \l_keys_path_str / \__keys_trim_spaces:n {##1} }
21773     {
21774     \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
21775     \int_set:Nn \exp_not:N \l_keys_choice_int
21776     { \int_use:N \l_keys_choice_int }
21777     \exp_not:n {#3}
21778     }
21779     }
21780 }

```

(End of definition for `__keys_choices_make:nn`, `__keys_multichoice_make:nn`, and `__keys_choices_make:Nnn`.)

```

\__keys_cmd_set:nn  Setting the code for a key first logs if appropriate that we are defining a new key, then
\__keys_cmd_set:Vn  saves the code.
\__keys_cmd_set:ne  21781 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
\__keys_cmd_set:Vo  21782 { \__keys_cmd_set_direct:nn {#1} { \__keys_precompile:n {#2} } }
\__keys_cmd_set_direct:nn 21783 \cs_generate_variant:Nn \__keys_cmd_set:nn { ne , Vn , Vo }
21784 \cs_new_protected:Npn \__keys_cmd_set_direct:nn #1#2
21785 { \cs_set_protected:cpn { \c__keys_code_root_str #1 } ##1 {#2} }

```

(End of definition for `__keys_cmd_set:nn` and `__keys_cmd_set_direct:nn`.)

```

\__keys_cs_set:NNpn  Creating control sequences is a bit more tricky than other cases as we need to pick up
\__keys_cs_set:Ncpn  the p argument. To make the internals look clearer, the trailing n argument here is just
                    for appearance.

```

```

21786 \cs_new_protected:Npn \__keys_cs_set:NNpn #1#2#3#
21787 {
21788   \cs_set_protected:cpe { \c__keys_code_root_str \l_keys_path_str } ##1
21789   {
21790     \__keys_precompile:n
21791     { #1 \exp_not:N #2 \exp_not:n {#3} {##1} }
21792   }
21793   \use_none:n
21794 }
21795 \cs_generate_variant:Nn \__keys_cs_set:NNpn { Nc }

```

(End of definition for `__keys_cs_set:NNpn`.)

```

\__keys_default_set:n  Setting a default value is easy. These are stored using \cs_set_nopar:cpe as this avoids
                    any worries about whether a token list exists.

```

```

21796 \cs_new_protected:Npn \__keys_default_set:n #1
21797 {
21798   \tl_if_empty:nTF {#1}
21799   {
21800     \__keys_cs_undefine:c
21801     { \c__keys_default_root_str \l_keys_path_str }
21802   }
21803   {
21804     \cs_set_nopar:cpe
21805     { \c__keys_default_root_str \l_keys_path_str }

```

```

21806         { \exp_not:n {#1} }
21807         \__keys_value_requirement:nn { required } { false }
21808     }
21809 }

```

(End of definition for __keys_default_set:n.)

`__keys_groups_set:n` Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the check-declarations code.

```

21810 \cs_new_protected:Npn \__keys_groups_set:n #1
21811 {
21812     \clist_set:Ne \l__keys_groups_clist { \tl_to_str:n {#1} }
21813     \clist_if_empty:NTF \l__keys_groups_clist
21814     {
21815         \__keys_cs_undefine:c
21816         { \c__keys_groups_root_str \l_keys_path_str }
21817     }
21818     {
21819         \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
21820         \l__keys_groups_clist
21821     }
21822 }

```

(End of definition for __keys_groups_set:n.)

`__keys_inherit:n` Inheritance means ignoring anything already said about the key: zap the lot and set up.

```

21823 \cs_new_protected:Npn \__keys_inherit:n #1
21824 {
21825     \__keys_undefine:
21826     \clist_set:Nn \l__keys_inherit_clist {#1}
21827     \cs_set_eq:cN { \c__keys_inherit_root_str \l_keys_path_str }
21828     \l__keys_inherit_clist
21829 }

```

(End of definition for __keys_inherit:n.)

`__keys_initialise:n` A set up for initialisation: just run the code if it exists. We need to set the key string here, using the deprecated `tl` as a piece of scratch space.

```

21830 \cs_new_protected:Npn \__keys_initialise:n #1
21831 {
21832     \cs_if_exist:cTF
21833     { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21834     { \__keys_execute_inherit: }
21835     {
21836         \str_clear:N \l__keys_inherit_str
21837         \cs_if_exist:cF { \c__keys_code_root_str \l_keys_path_str }
21838         {
21839             \exp_after:wN \__keys_find_key_module:wNN
21840             \l_keys_path_str \s__keys_stop
21841             \l_keys_key_tl \l_keys_key_str
21842             \tl_set_eq:NN \l_keys_key_tl \l_keys_key_str
21843             \tl_set:Nn \l_keys_value_tl {#1}

```

```

21844         \_keys_execute:no \l_keys_path_str \l_keys_value_tl
21845     }
21846 }
21847 }

```

(End of definition for _keys_initialise:n.)

_keys_legacy_if_set:nn Much the same as expl3 booleans, except we assume that the switch exists.

```

\_keys_legacy_if_inverse:nn 21848 \cs_new_protected:Npn \_keys_legacy_if_set:nn #1#2
\_keys_legacy_if_inverse:nnn 21849 { \_keys_legacy_if_set:nnnn {#1} {#2} { true } { false } }
21850 \cs_new_protected:Npn \_keys_legacy_if_set_inverse:nn #1#2
21851 { \_keys_legacy_if_set:nnnn {#1} {#2} { false } { true } }
21852 \cs_new_protected:Npn \_keys_legacy_if_set:nnnn #1#2#3#4
21853 {
21854     \_keys_choice_make:
21855     \_keys_cmd_set:ne { \l_keys_path_str / true }
21856     { \exp_not:c { legacy_if_#2 set_ #3 :n } { \exp_not:n {#1} } }
21857     \_keys_cmd_set:ne { \l_keys_path_str / false }
21858     { \exp_not:c { legacy_if_#2 set_ #4 :n } { \exp_not:n {#1} } }
21859     \_keys_cmd_set:nn { \l_keys_path_str / unknown }
21860     {
21861         \msg_error:nne { keys } { boolean-values-only }
21862         \l_keys_path_str
21863     }
21864     \_keys_default_set:n { true }
21865     \cs_if_exist:cF { if#1 }
21866     {
21867         \cs:w newif \exp_after:wN \cs_end:
21868         \cs:w if#1 \cs_end:
21869     }
21870 }

```

(End of definition for _keys_legacy_if_set:nn, _keys_legacy_if_inverse:nn, and _keys_legacy_if_inverse:nnnn.)

_keys_meta_make:n To create a meta-key, simply set up to pass data through. The internal function is used here as a meta key should respect the prevailing filtering, etc.

```

21871 \cs_new_protected:Npn \_keys_meta_make:n #1
21872 {
21873     \exp_args:NVo \_keys_cmd_set_direct:nn \l_keys_path_str
21874     {
21875         \exp_after:wN \_keys_set:nn \exp_after:wN
21876         { \l_keys_module_str } {#1}
21877     }
21878 }
21879 \cs_new_protected:Npn \_keys_meta_make:nn #1#2
21880 {
21881     \exp_args:NV \_keys_cmd_set_direct:nn
21882     \l_keys_path_str { \_keys_set:nn {#1} {#2} }
21883 }

```

(End of definition for _keys_meta_make:n and _keys_meta_make:nn.)

`__keys_prop_put:Nn` Much the same as other variables, but needs a dedicated auxiliary.

```

\__keys_prop_put:cn
21884 \cs_new_protected:Npn \__keys_prop_put:Nn #1#2
21885 {
21886   \prop_if_exist:NF #1 { \prop_new:N #1 }
21887   \exp_after:wN \__keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
21888   \l__keys_tmpa_tl \l__keys_tmpb_tl
21889   \__keys_cmd_set:ne \l_keys_path_str
21890   {
21891     \exp_not:c { prop_ #2 put:Nnn }
21892     \exp_not:N #1
21893     { \l__keys_tmpb_tl }
21894     \exp_not:n { {##1} }
21895   }
21896 }
21897 \cs_generate_variant:Nn \__keys_prop_put:Nn { c }

```

(End of definition for __keys_prop_put:Nn.)

`__keys_undefine:` Undefined a key has to be done without `\cs_undefine:c` as that function acts globally.

```

21898 \cs_new_protected:Npn \__keys_undefine:
21899 {
21900   \clist_map_inline:nn
21901   { code , default , groups , inherit , type , check }
21902   {
21903     \__keys_cs_undefine:c
21904     { \tl_use:c { c__keys_ ##1 _root_str } \l_keys_path_str }
21905   }
21906 }

```

(End of definition for __keys_undefine:.)

`__keys_value_requirement:nn` Validating key input is done using a second function which runs before the main key code. Setting that up means setting it equal to a generic stub which does the check. This approach makes the lookup very fast at the cost of one additional csname per key that needs it. The cleanup here has to know the structure of the following code.

`__keys_check_forbidden:`
`__keys_check_required:`

```

21907 \cs_new_protected:Npn \__keys_value_requirement:nn #1#2
21908 {
21909   \str_case:nnF {#2}
21910   {
21911     { true }
21912     {
21913       \cs_set_eq:cc
21914       { \c__keys_check_root_str \l_keys_path_str }
21915       { __keys_check_ #1 : }
21916     }
21917     { false }
21918     {
21919       \cs_if_eq:ccT
21920       { \c__keys_check_root_str \l_keys_path_str }
21921       { __keys_check_ #1 : }
21922       {
21923         \__keys_cs_undefine:c
21924         { \c__keys_check_root_str \l_keys_path_str }
21925       }
21926     }
21927   }
21928 }

```

```

21926     }
21927   }
21928   {
21929     \msg_error:nne { keys }
21930     { boolean-values-only }
21931     { .value_ #1 :n }
21932   }
21933 }
21934 \cs_new_protected:Npn \__keys_check_forbidden:
21935 {
21936   \bool_if:NF \l__keys_no_value_bool
21937   {
21938     \msg_error:nnee { keys } { value-forbidden }
21939     \l_keys_path_str \l_keys_value_tl
21940     \use_none:nnn
21941   }
21942 }
21943 \cs_new_protected:Npn \__keys_check_required:
21944 {
21945   \bool_if:NT \l__keys_no_value_bool
21946   {
21947     \msg_error:nne { keys } { value-required }
21948     \l_keys_path_str
21949     \use_none:nnn
21950   }
21951 }

```

(End of definition for `__keys_value_requirement:nn`, `__keys_check_forbidden:`, and `__keys-check_required:`.)

```

\__keys_usage:n Save the relevant data.
\__keys_usage:NN
\__keys_usage:w
21952 \cs_new_protected:Npn \__keys_usage:n #1
21953 {
21954   \str_case:nnF {#1}
21955   {
21956     { general }
21957     {
21958       \__keys_usage:NN \l_keys_usage_load_prop
21959       \c_false_bool
21960       \__keys_usage:NN \l_keys_usage_preamble_prop
21961       \c_false_bool
21962     }
21963     { load }
21964     {
21965       \__keys_usage:NN \l_keys_usage_load_prop
21966       \c_true_bool
21967       \__keys_usage:NN \l_keys_usage_preamble_prop
21968       \c_false_bool
21969     }
21970     { preamble }
21971     {
21972       \__keys_usage:NN \l_keys_usage_load_prop
21973       \c_false_bool
21974       \__keys_usage:NN \l_keys_usage_preamble_prop

```

```

21975         \c_true_bool
21976     }
21977 }
21978 {
21979     \msg_error:nnnn { keys }
21980     { choice-unknown }
21981     { .usage:n }
21982     {#1}
21983 }
21984 }
21985 \cs_new_protected:Npn \__keys_usage:NN #1#2
21986 {
21987     \prop_get:NVNF #1 \l__keys_module_str \l__keys_tmpa_tl
21988     { \tl_clear:N \l__keys_tmpa_tl }
21989     \tl_set:Ne \l__keys_tmpb_tl
21990     { \exp_after:wN \__keys_usage:w \l__keys_path_str \s__keys_stop }
21991     \bool_if:NTF #2
21992     { \clist_put_right:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
21993     { \clist_remove_all:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
21994     \prop_put:NVV #1 \l__keys_module_str
21995     \l__keys_tmpa_tl
21996 }
21997 \cs_new:Npn \__keys_usage:w #1 / #2 \s__keys_stop {#2}

```

(End of definition for `__keys_usage:n`, `__keys_usage:NN`, and `__keys_usage:w`.)

`__keys_variable_set:NnnN` Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

`__keys_variable_set:cnnN`
`__keys_variable_set_required:NnnN`
`__keys_variable_set_required:cnnN`

```

21998 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
21999 {
22000     \use:c { #2_if_exist:NF } #1 { \use:c { #2_new:N } #1 }
22001     \__keys_cmd_set:ne \l__keys_path_str
22002     {
22003         \exp_not:c { #2 _ #3 set:N #4 }
22004         \exp_not:N #1
22005         \exp_not:n { {##1} }
22006     }
22007 }
22008 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
22009 \cs_new_protected:Npn \__keys_variable_set_required:NnnN #1#2#3#4
22010 {
22011     \__keys_variable_set:NnnN #1 {#2} {#3} #4
22012     \__keys_value_requirement:nn { required } { true }
22013 }
22014 \cs_generate_variant:Nn \__keys_variable_set_required:NnnN { c }

```

(End of definition for `__keys_variable_set:NnnN` and `__keys_variable_set_required:NnnN`.)

65.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is

handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

```
.bool_set:N One function for this.
.bool_set:c 22015 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set:N } #1
.bool_gset:N 22016 { \__keys_bool_set:Nn #1 { } }
.bool_gset:c 22017 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set:c } #1
22018 { \__keys_bool_set:cn {#1} { } }
22019 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:N } #1
22020 { \__keys_bool_set:Nn #1 { g } }
22021 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:c } #1
22022 { \__keys_bool_set:cn {#1} { g } }
```

(End of definition for .bool_set:N and .bool_gset:N. These functions are documented on page 241.)

```
.bool_set_inverse:N One function for this.
.bool_set_inverse:c 22023 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:N } #1
.bool_gset_inverse:N 22024 { \__keys_bool_set_inverse:Nn #1 { } }
.bool_gset_inverse:c 22025 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:c } #1
22026 { \__keys_bool_set_inverse:cn {#1} { } }
22027 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:N } #1
22028 { \__keys_bool_set_inverse:Nn #1 { g } }
22029 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:c } #1
22030 { \__keys_bool_set_inverse:cn {#1} { g } }
```

(End of definition for .bool_set_inverse:N and .bool_gset_inverse:N. These functions are documented on page 241.)

```
.choice: Making a choice is handled internally, as it is also needed by .generate_choices:n.
22031 \cs_new_protected:cpn { \c__keys_props_root_str .choice: }
22032 { \__keys_choice_make: }
```

(End of definition for .choice:. This function is documented on page 241.)

```
.choices:mn For auto-generation of a series of mutually-exclusive choices. Here, #1 consists of two
.choices:Vn separate arguments, hence the slightly odd-looking implementation.
.choices:en 22033 \cs_new_protected:cpn { \c__keys_props_root_str .choices:mn } #1
.choices:on 22034 { \__keys_choices_make:mn #1 }
.choices:xn 22035 \cs_new_protected:cpn { \c__keys_props_root_str .choices:Vn } #1
22036 { \exp_args:NV \__keys_choices_make:mn #1 }
22037 \cs_new_protected:cpn { \c__keys_props_root_str .choices:en } #1
22038 { \exp_args:Ne \__keys_choices_make:mn #1 }
22039 \cs_new_protected:cpn { \c__keys_props_root_str .choices:on } #1
22040 { \exp_args:No \__keys_choices_make:mn #1 }
22041 \cs_new_protected:cpn { \c__keys_props_root_str .choices:xn } #1
22042 { \exp_args:Nx \__keys_choices_make:mn #1 }
```

(End of definition for .choices:mn. This function is documented on page 241.)

```
.code:n Creating code is simply a case of passing through to the underlying set function.
22043 \cs_new_protected:cpn { \c__keys_props_root_str .code:n } #1
22044 { \__keys_cmd_set:nn \l_keys_path_str {#1} }
```

(End of definition for .code:n. This function is documented on page 242.)

```

.clist_set:N
.clist_set:c 22045 \cs_new_protected:cpn { \c__keys_props_root_str .clist_set:N } #1
.clist_gset:N 22046 { \__keys_variable_set:NnnN #1 { clist } { } n }
.clist_gset:c 22047 \cs_new_protected:cpn { \c__keys_props_root_str .clist_set:c } #1
22048 { \__keys_variable_set:cnnN {#1} { clist } { } n }
22049 \cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:N } #1
22050 { \__keys_variable_set:NnnN #1 { clist } { g } n }
22051 \cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:c } #1
22052 { \__keys_variable_set:cnnN {#1} { clist } { g } n }

```

(End of definition for .clist_set:N and .clist_gset:N. These functions are documented on page 241.)

```

.cs_set:Np
.cs_set:cp 22053 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set:Np } #1
.cs_set_protected:Np 22054 { \__keys_cs_set:NNpn \cs_set:Npn #1 { } }
.cs_set_protected:cp 22055 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set:cp } #1
.cs_gset:Np 22056 { \__keys_cs_set:Ncpn \cs_set:Npn #1 { } }
.cs_gset:cp 22057 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:Np } #1
.cs_gset_protected:Np 22058 { \__keys_cs_set:NNpn \cs_set_protected:Npn #1 { } }
.cs_gset_protected:cp 22059 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:cp } #1
22060 { \__keys_cs_set:Ncpn \cs_set_protected:Npn #1 { } }
22061 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:Np } #1
22062 { \__keys_cs_set:NNpn \cs_gset:Npn #1 { } }
22063 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:cp } #1
22064 { \__keys_cs_set:Ncpn \cs_gset:Npn #1 { } }
22065 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:Np } #1
22066 { \__keys_cs_set:NNpn \cs_gset_protected:Npn #1 { } }
22067 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:cp } #1
22068 { \__keys_cs_set:Ncpn \cs_gset_protected:Npn #1 { } }

```

(End of definition for .cs_set:Np and others. These functions are documented on page 242.)

```

.default:n Expansion is left to the internal functions.
.default:V 22069 \cs_new_protected:cpn { \c__keys_props_root_str .default:n } #1
.default:e 22070 { \__keys_default_set:n {#1} }
.default:o 22071 \cs_new_protected:cpn { \c__keys_props_root_str .default:V } #1
.default:x 22072 { \exp_args:NV \__keys_default_set:n #1 }
22073 \cs_new_protected:cpn { \c__keys_props_root_str .default:e } #1
22074 { \exp_args:Ne \__keys_default_set:n {#1} }
22075 \cs_new_protected:cpn { \c__keys_props_root_str .default:o } #1
22076 { \exp_args:No \__keys_default_set:n {#1} }
22077 \cs_new_protected:cpn { \c__keys_props_root_str .default:x } #1
22078 { \exp_args:Nx \__keys_default_set:n {#1} }

```

(End of definition for .default:n. This function is documented on page 242.)

```

.dim_set:N Setting a variable is very easy: just pass the data along.
.dim_set:c 22079 \cs_new_protected:cpn { \c__keys_props_root_str .dim_set:N } #1
.dim_gset:N 22080 { \__keys_variable_set_required:NnnN #1 { dim } { } n }
.dim_gset:c 22081 \cs_new_protected:cpn { \c__keys_props_root_str .dim_set:c } #1
22082 { \__keys_variable_set_required:cnnN {#1} { dim } { } n }
22083 \cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:N } #1
22084 { \__keys_variable_set_required:NnnN #1 { dim } { g } n }
22085 \cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:c } #1
22086 { \__keys_variable_set_required:cnnN {#1} { dim } { g } n }

```

(End of definition for `.dim_set:N` and `.dim_gset:N`. These functions are documented on page 242.)

`.fp_set:N` Setting a variable is very easy: just pass the data along.
`.fp_set:c` 22087 `\cs_new_protected:cpn { \c__keys_props_root_str .fp_set:N } #1`
`.fp_gset:N` 22088 `{ __keys_variable_set_required:NnnN #1 { fp } { } n }`
`.fp_gset:c` 22089 `\cs_new_protected:cpn { \c__keys_props_root_str .fp_set:c } #1`
22090 `{ __keys_variable_set_required:cnnN {#1} { fp } { } n }`
22091 `\cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:N } #1`
22092 `{ __keys_variable_set_required:NnnN #1 { fp } { g } n }`
22093 `\cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:c } #1`
22094 `{ __keys_variable_set_required:cnnN {#1} { fp } { g } n }`

(End of definition for `.fp_set:N` and `.fp_gset:N`. These functions are documented on page 242.)

`.groups:n` A single property to create groups of keys.
22095 `\cs_new_protected:cpn { \c__keys_props_root_str .groups:n } #1`
22096 `{ __keys_groups_set:n {#1} }`

(End of definition for `.groups:n`. This function is documented on page 243.)

`.inherit:n` Nothing complex: only one variant at the moment!
22097 `\cs_new_protected:cpn { \c__keys_props_root_str .inherit:n } #1`
22098 `{ __keys_inherit:n {#1} }`

(End of definition for `.inherit:n`. This function is documented on page 243.)

`.initial:n` The standard hand-off approach.
`.initial:V` 22099 `\cs_new_protected:cpn { \c__keys_props_root_str .initial:n } #1`
`.initial:e` 22100 `{ __keys_initialise:n {#1} }`
`.initial:o` 22101 `\cs_new_protected:cpn { \c__keys_props_root_str .initial:V } #1`
`.initial:x` 22102 `{ \exp_args:NV __keys_initialise:n #1 }`
22103 `\cs_new_protected:cpn { \c__keys_props_root_str .initial:e } #1`
22104 `{ \exp_args:Ne __keys_initialise:n {#1} }`
22105 `\cs_new_protected:cpn { \c__keys_props_root_str .initial:o } #1`
22106 `{ \exp_args:No __keys_initialise:n {#1} }`
22107 `\cs_new_protected:cpn { \c__keys_props_root_str .initial:x } #1`
22108 `{ \exp_args:Nx __keys_initialise:n {#1} }`

(End of definition for `.initial:n`. This function is documented on page 243.)

`.int_set:N` Setting a variable is very easy: just pass the data along.
`.int_set:c` 22109 `\cs_new_protected:cpn { \c__keys_props_root_str .int_set:N } #1`
`.int_gset:N` 22110 `{ __keys_variable_set_required:NnnN #1 { int } { } n }`
`.int_gset:c` 22111 `\cs_new_protected:cpn { \c__keys_props_root_str .int_set:c } #1`
22112 `{ __keys_variable_set_required:cnnN {#1} { int } { } n }`
22113 `\cs_new_protected:cpn { \c__keys_props_root_str .int_gset:N } #1`
22114 `{ __keys_variable_set_required:NnnN #1 { int } { g } n }`
22115 `\cs_new_protected:cpn { \c__keys_props_root_str .int_gset:c } #1`
22116 `{ __keys_variable_set_required:cnnN {#1} { int } { g } n }`

(End of definition for `.int_set:N` and `.int_gset:N`. These functions are documented on page 243.)

```

.legacy_if_set:n
.legacy_if_gset:n
.legacy_if_set_inverse:n
.legacy_if_gset_inverse:n
22117 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_set:n } #1
22118 { \__keys_legacy_if_set:nn {#1} { } }
22119 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_gset:n } #1
22120 { \__keys_legacy_if_set:nn {#1} { g } }
22121 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_set_inverse:n } #1
22122 { \__keys_legacy_if_set_inverse:nn {#1} { } }
22123 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_gset_inverse:n } #1
22124 { \__keys_legacy_if_set_inverse:nn {#1} { g } }

```

(End of definition for `.legacy_if_set:n` and others. These functions are documented on page 243.)

```

.meta:n Making a meta is handled internally.
22125 \cs_new_protected:cpn { \c__keys_props_root_str .meta:n } #1
22126 { \__keys_meta_make:n {#1} }

```

(End of definition for `.meta:n`. This function is documented on page 243.)

```

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.
22127 \cs_new_protected:cpn { \c__keys_props_root_str .meta:nn } #1
22128 { \__keys_meta_make:nn #1 }

```

(End of definition for `.meta:nn`. This function is documented on page 244.)

```

.multichoice: The same idea as .choice: and .choices:nn, but where more than one choice is allowed.
.multichoices:nn
.multichoices:Vn
.multichoices:en
.multichoices:on
.multichoices:xn
22129 \cs_new_protected:cpn { \c__keys_props_root_str .multichoice: }
22130 { \__keys_multichoice_make: }
22131 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:nn } #1
22132 { \__keys_multichoices_make:nn #1 }
22133 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:Vn } #1
22134 { \exp_args:NV \__keys_multichoices_make:nn #1 }
22135 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:en } #1
22136 { \exp_args:Ne \__keys_multichoices_make:nn #1 }
22137 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:on } #1
22138 { \exp_args:No \__keys_multichoices_make:nn #1 }
22139 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:xn } #1
22140 { \exp_args:Nx \__keys_multichoices_make:nn #1 }

```

(End of definition for `.multichoice:` and `.multichoices:nn`. These functions are documented on page 244.)

```

.muskip_set:N Setting a variable is very easy: just pass the data along.
.muskip_set:c
.muskip_gset:N
.muskip_gset:c
22141 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:N } #1
22142 { \__keys_variable_set_required:NnnN #1 { muskip } { } n }
22143 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:c } #1
22144 { \__keys_variable_set_required:cnnN {#1} { muskip } { } n }
22145 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:N } #1
22146 { \__keys_variable_set_required:NnnN #1 { muskip } { g } n }
22147 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:c } #1
22148 { \__keys_variable_set_required:cnnN {#1} { muskip } { g } n }

```

(End of definition for `.muskip_set:N` and `.muskip_gset:N`. These functions are documented on page 244.)

```

.prop_put:N Setting a variable is very easy: just pass the data along.
.prop_put:c 22149 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:N } #1
.prop_gput:N 22150 { \__keys_prop_put:Nn #1 { } }
.prop_gput:c 22151 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:c } #1
22152 { \__keys_prop_put:cn {#1} { } }
22153 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:N } #1
22154 { \__keys_prop_put:Nn #1 { g } }
22155 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:c } #1
22156 { \__keys_prop_put:cn {#1} { g } }

```

(End of definition for .prop_put:N and .prop_gput:N. These functions are documented on page 244.)

```

.skip_set:N Setting a variable is very easy: just pass the data along.
.skip_set:c 22157 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:N } #1
.skip_gset:N 22158 { \__keys_variable_set_required:NnnN #1 { skip } { } n }
.skip_gset:c 22159 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:c } #1
22160 { \__keys_variable_set_required:cnnN {#1} { skip } { } n }
22161 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:N } #1
22162 { \__keys_variable_set_required:NnnN #1 { skip } { g } n }
22163 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:c } #1
22164 { \__keys_variable_set_required:cnnN {#1} { skip } { g } n }

```

(End of definition for .skip_set:N and .skip_gset:N. These functions are documented on page 244.)

```

.str_set:N Setting a variable is very easy: just pass the data along.
.str_set:c 22165 \cs_new_protected:cpn { \c__keys_props_root_str .str_set:N } #1
.str_gset:N 22166 { \__keys_variable_set:NnnN #1 { str } { } n }
.str_gset:c 22167 \cs_new_protected:cpn { \c__keys_props_root_str .str_set:c } #1
.str_set_e:N 22168 { \__keys_variable_set:cnnN {#1} { str } { } n }
.str_set_e:c 22169 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_e:N } #1
.str_gset_e:N 22170 { \__keys_variable_set:NnnN #1 { str } { } e }
.str_gset_e:c 22171 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_e:c } #1
22172 { \__keys_variable_set:cnnN {#1} { str } { } e }
22173 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset:N } #1
22174 { \__keys_variable_set:NnnN #1 { str } { g } n }
22175 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset:c } #1
22176 { \__keys_variable_set:cnnN {#1} { str } { g } n }
22177 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_e:N } #1
22178 { \__keys_variable_set:NnnN #1 { str } { g } e }
22179 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_e:c } #1
22180 { \__keys_variable_set:cnnN {#1} { str } { g } e }

```

(End of definition for .str_set:N and others. These functions are documented on page 244.)

```

.tl_set:N Setting a variable is very easy: just pass the data along.
.tl_set:c 22181 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:N } #1
.tl_gset:N 22182 { \__keys_variable_set:NnnN #1 { tl } { } n }
.tl_gset:c 22183 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:c } #1
.tl_set_e:N 22184 { \__keys_variable_set:cnnN {#1} { tl } { } n }
.tl_set_e:c 22185 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_e:N } #1
.tl_gset_e:N 22186 { \__keys_variable_set:NnnN #1 { tl } { } e }
.tl_gset_e:c 22187 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_e:c } #1
22188 { \__keys_variable_set:cnnN {#1} { tl } { } e }
22189 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:N } #1

```

```

22190 { \__keys_variable_set:NnnN #1 { t1 } { g } n }
22191 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:c } #1
22192 { \__keys_variable_set:cnnN {#1} { t1 } { g } n }
22193 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_e:N } #1
22194 { \__keys_variable_set:NnnN #1 { t1 } { g } e }
22195 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_e:c } #1
22196 { \__keys_variable_set:cnnN {#1} { t1 } { g } e }

```

(End of definition for .tl_set:N and others. These functions are documented on page 245.)

.undefine: Another simple wrapper.

```

22197 \cs_new_protected:cpn { \c__keys_props_root_str .undefine: }
22198 { \__keys_undefine: }

```

(End of definition for .undefine:. This function is documented on page 245.)

.usage:n

```

22199 \cs_new_protected:cpn { \c__keys_props_root_str .usage:n } #1
22200 { \__keys_usage:n {#1} }

```

(End of definition for .usage:n. This function is documented on page 248.)

.value_forbidden:n These are very similar, so both call the same function.

```

22201 \cs_new_protected:cpn { \c__keys_props_root_str .value_forbidden:n } #1
22202 { \__keys_value_requirement:nn { forbidden } {#1} }
22203 \cs_new_protected:cpn { \c__keys_props_root_str .value_required:n } #1
22204 { \__keys_value_requirement:nn { required } {#1} }

```

(End of definition for .value_forbidden:n and .value_required:n. These functions are documented on page 245.)

65.6 Setting keys

```

\__keys_set:nnnnNn
\__keys_set:nnnnnnNn
\__keys_reset_bool:N
\__keys_reset_var:N
  \__keys_set:nn
  \__keys_set:nnn

```

The aim here is to allow nesting of key setting without needing lots of tracking. That is done by expanding the appropriate tokens “around” the core keyval parsing. As there are several different sub-paths, this needs a few steps and some generic auxiliaries. The arguments here are

1. The root for keys
2. The key groups
3. The keys themselves
4. The relative root for return of unset keys
5. The `clist` var for returning unset keys
6. The code to set up the correct selection approach

```

22205 \cs_new_protected:Npn \__keys_set:nnnnNn
22206 {
22207   \exp_args:Nooo \__keys_set:nnnnnnNn
22208     \l__keys_unused_clist
22209     \l__keys_selective_clist
22210     \l__keys_relative_tl
22211 }
22212 \cs_new_protected:Npn \__keys_set:nnnnnnNn #1#2#3#4#5#6#7#8#9
22213 {
22214   \clist_clear:N \l__keys_unused_clist
22215   \clist_set:Ne \l__keys_selective_clist { \tl_to_str:n {#5} }
22216   \tl_set:Nn \l__keys_relative_tl {#7}
22217   \use:e
22218     {
22219       \exp_not:n
22220         {
22221           #9
22222           \__keys_set:nn {#4} {#6}
22223         }
22224       \__keys_reset_bool:N \l__keys_only_known_bool
22225       \__keys_reset_bool:N \l__keys_exclude_bool
22226       \__keys_reset_bool:N \l__keys_selective_bool
22227     }
22228   \clist_set_eq:NN #8 \l__keys_unused_clist
22229   \__kernel_tl_set:Nx \l__keys_unused_clist { \exp_not:n {#1} }
22230   \__kernel_tl_set:Nx \l__keys_selective_clist {#2}
22231   \__kernel_tl_set:Nx \l__keys_relative_tl { \exp_not:n {#3} }
22232 }
22233 \cs_new:Npn \__keys_reset_bool:N #1
22234 {
22235   \exp_not:c
22236     { bool_set_ \bool_if:NTF #1 { true } { false } :N }
22237   \exp_not:N #1
22238 }
22239 \cs_new_protected:Npn \__keys_set:nn #1#2
22240 { \exp_args:No \__keys_set:nnn \l__keys_module_str {#1} {#2} }
22241 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
22242 {
22243   \str_set:Ne \l__keys_module_str { \__keys_trim_spaces:n {#2} }
22244   \keyval_parse:NNn \__keys_set_keyval:n \__keys_set_keyval:nn {#3}
22245   \str_set:Nn \l__keys_module_str {#1}
22246 }

```

(End of definition for __keys_set:nnnnNn and others.)

```

\keys_set:nn A simple wrapper allowing for nesting.
\keys_set:nV 22247 \cs_new_protected:Npn \keys_set:nn #1#2
\keys_set:nv 22248 {
\keys_set:ne 22249   \__keys_set:nnnnNn
\keys_set:no 22250   {#1} { } {#2} { \q__keys_no_value } \l__keys_tmp_clist
\keys_set:nx 22251   {
22252     \bool_set_false:N \l__keys_only_known_bool
22253     \bool_set_false:N \l__keys_exclude_bool
22254     \bool_set_false:N \l__keys_selective_bool

```

```

22255     }
22256   }
22257 \cs_generate_variant:Nn \keys_set:nn { nV , nv , ne , no , nx }

```

(End of definition for `\keys_set:nn`. This function is documented on page 248.)

`\keys_set_known:nnnN`

Simply set the right variables.

```

\keys_set_known:nVnN 22258 \cs_new_protected:Npn \keys_set_known:nnnN #1#2#3#4
\keys_set_known:nvN 22259 {
\keys_set_known:nenN 22260   \__keys_set:nnnnNn
\keys_set_known:nonN 22261   {#1} { } {#2} {#3} #4
\keys_set_known:nnN 22262   {
\keys_set_known:nVN 22263     \bool_set_true:N \l__keys_only_known_bool
\keys_set_known:nvN 22264     \bool_set_false:N \l__keys_exclude_bool
\keys_set_known:neN 22265     \bool_set_false:N \l__keys_selective_bool
\keys_set_known:noN 22266   }
\keys_set_known:nn 22267 }
\keys_set_known:nV 22268 \cs_generate_variant:Nn \keys_set_known:nnnN { nV , nv , ne , no }
\keys_set_known:nv 22269 \cs_new_protected:Npn \keys_set_known:nnN #1#2#3
\keys_set_known:ne 22270 { \keys_set_known:nnnN {#1} {#2} { \q_keys_no_value } #3 }
\keys_set_known:no 22271 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , ne , no }
22272 \cs_new_protected:Npn \keys_set_known:nn #1#2
22273 { \keys_set_known:nnnN {#1} {#2} { \q_keys_no_value } \l__keys_tmp_clist }
22274 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , ne , no }

```

(End of definition for `\keys_set_known:nnnN`, `\keys_set_known:nnN`, and `\keys_set_known:nn`. These functions are documented on page 249.)

`\keys_set_exclude_groups:nnnN`

The same for (exclusion) groups.

```

\keys_set_exclude_groups:nnVN 22275 \cs_new_protected:Npn \keys_set_exclude_groups:nnnnN #1#2#3#4#5
\keys_set_exclude_groups:nnvN 22276 {
\keys_set_exclude_groups:nnoN 22277   \__keys_set:nnnnNn
\keys_set_exclude_groups:nnVnN 22278   {#1} {#2} {#3} {#4} #5
\keys_set_exclude_groups:nnvN 22279   {
\keys_set_exclude_groups:nnonN 22280     \bool_set_false:N \l__keys_only_known_bool
\keys_set_exclude_groups:nnonN 22281     \bool_set_true:N \l__keys_exclude_bool
\keys_set_exclude_groups:nnonN 22282     \bool_set_true:N \l__keys_selective_bool
\keys_set_exclude_groups:nnV 22283   }
\keys_set_exclude_groups:nnv 22284 }
\keys_set_exclude_groups:nno 22285 \cs_generate_variant:Nn \keys_set_exclude_groups:nnnnN { nnV , nnv , nno }
\keys_set_exclude_groups:nnv 22286 \cs_new_protected:Npn \keys_set_exclude_groups:nnnN #1#2#3#4
\keys_set_exclude_groups:nnonN 22287 { \keys_set_exclude_groups:nnnnN {#1} {#2} {#3} { \q_keys_no_value } #4 }
\keys_set_groups:nnVN 22288 \cs_generate_variant:Nn \keys_set_exclude_groups:nnnN { nnV , nnv , nno }
\keys_set_groups:nnvN 22289 \cs_new_protected:Npn \keys_set_exclude_groups:nnn #1#2#3
\keys_set_groups:nnoN 22290 {
\keys_set_groups:nnnnN 22291   \keys_set_exclude_groups:nnnnN {#1} {#2} {#3}
\keys_set_groups:nnVnN 22292   { \q_keys_no_value } \l__keys_tmp_clist
\keys_set_groups:nnvN 22293 }
\keys_set_groups:nnonN 22294 \cs_generate_variant:Nn \keys_set_exclude_groups:nnn { nnV , nnv , nno }
\keys_set_groups:nnn 22295 \cs_new_protected:Npn \keys_set_groups:nnnnN #1#2#3#4#5
\keys_set_groups:nnV 22296 {
\keys_set_groups:nnv 22297   \__keys_set:nnnnNn
\keys_set_groups:nnv 22298   {#1} {#2} {#3} {#4} #5
\keys_set_groups:nno 22299   {
22300     \bool_set_false:N \l__keys_only_known_bool

```



```

22301         \bool_set_false:N \l__keys_exclude_bool
22302         \bool_set_true:N \l__keys_selective_bool
22303     }
22304 }
22305 \cs_generate_variant:Nn \keys_set_groups:nnnnN { nnV , nnv , nno }
22306 \cs_new_protected:Npn \keys_set_groups:nnnN #1#2#3#4
22307 { \keys_set_groups:nnnnN {#1} {#2} {#3} { \q__keys_no_value } #4 }
22308 \cs_generate_variant:Nn \keys_set_groups:nnnN { nnV , nnv , nno }
22309 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
22310 {
22311     \keys_set_groups:nnnnN {#1} {#2} {#3}
22312     { \q__keys_no_value } \l__keys_tmp_clist
22313 }
22314 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }

```

(End of definition for `\keys_set_exclude_groups:nnnN` and others. These functions are documented on page 250.)

`\keys_precompile:nnN` A simple wrapper.

```

22315 \cs_new_protected:Npn \keys_precompile:nnN #1#2#3
22316 {
22317     \bool_set_true:N \l__keys_precompile_bool
22318     \tl_clear:N \l__keys_precompile_tl
22319     \keys_set:nn {#1} {#2}
22320     \bool_set_false:N \l__keys_precompile_bool
22321     \tl_set_eq:NN #3 \l__keys_precompile_tl
22322 }

```

(End of definition for `\keys_precompile:nnN`. This function is documented on page 251.)

```

\__keys_set_keyval:n A shared system once again. First, set the current path and add a default if needed.
\__keys_set_keyval:nn There are then checks to see if a value is required or forbidden. If everything passes,
\__keys_set_keyval:nnn move on to execute the code.
\__keys_set_keyval:onn
\__keys_find_key_module:wNN
\__keys_find_key_module_auxi:Nw
\__keys_find_key_module_auxii:Nw
\__keys_find_key_module_auxiii:Nn
\__keys_find_key_module_auxiv:Nw
\__keys_set_selective:
22323 \cs_new_protected:Npn \__keys_set_keyval:n #1
22324 {
22325     \bool_set_true:N \l__keys_no_value_bool
22326     \__keys_set_keyval:onn \l__keys_module_str {#1} { }
22327 }
22328 \cs_new_protected:Npn \__keys_set_keyval:nn #1#2
22329 {
22330     \bool_set_false:N \l__keys_no_value_bool
22331     \__keys_set_keyval:onn \l__keys_module_str {#1} {#2}
22332 }

```

The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

```

22333 \cs_new_protected:Npn \__keys_set_keyval:nnn #1#2#3
22334 {
22335     \__kernel_tl_set:Nx \l__keys_path_str
22336     {
22337         \tl_if_blank:nF {#1}
22338         { #1 / }
22339         \__keys_trim_spaces:n {#2}

```

```

22340     }
22341     \str_clear:N \l__keys_module_str
22342     \str_clear:N \l__keys_inherit_str
22343     \exp_after:wN \__keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
22344     \l__keys_module_str \l_keys_key_str
22345     \tl_set_eq:NN \l_keys_key_tl \l_keys_key_str
22346     \__keys_value_or_default:n {#3}
22347     \bool_if:NTF \l__keys_selective_bool
22348     \__keys_set_selective:
22349     \__keys_execute:
22350     \str_set:Nn \l__keys_module_str {#1}
22351   }
22352 \cs_generate_variant:Nn \__keys_set_keyval:nnn { o }

```

This function uses `\cs_set_nopar:Npe` internally for performance reasons, the argument #1 is already a string in every usage, so turning it into a string again seems unnecessary.

```

22353 \cs_new_protected:Npn \__keys_find_key_module:wNN #1 \s__keys_stop #2 #3
22354 {
22355   \__keys_find_key_module_auxi:Nw #2 #1 \s__keys_nil \__keys_find_key_module_auxii:Nw
22356   / \s__keys_nil \__keys_find_key_module_auxiv:Nw #3
22357 }
22358 \cs_new_protected:Npn \__keys_find_key_module_auxi:Nw #1 #2 / #3 \s__keys_nil #4
22359 {
22360   #4 #1 #2 \s__keys_mark #3 \s__keys_nil #4
22361 }
22362 \cs_new_protected:Npn \__keys_find_key_module_auxii:Nw
22363 #1 #2 \s__keys_mark #3 \s__keys_nil \__keys_find_key_module_auxii:Nw
22364 {
22365   \cs_set_nopar:Npe #1 { \tl_if_empty:NF #1 { #1 / } #2 }
22366   \__keys_find_key_module_auxi:Nw #1 #3 \s__keys_nil \__keys_find_key_module_auxiii:Nw
22367 }
22368 \cs_new_protected:Npn \__keys_find_key_module_auxiii:Nw #1 #2 \s__keys_mark
22369 {
22370   \cs_set_nopar:Npe #1 { \tl_if_empty:NF #1 { #1 / } #2 }
22371   \__keys_find_key_module_auxi:Nw #1
22372 }
22373 \cs_new_protected:Npn \__keys_find_key_module_auxiv:Nw
22374 #1 #2 \s__keys_nil #3 \s__keys_mark
22375 \s__keys_nil \__keys_find_key_module_auxiv:Nw #4
22376 {
22377   \cs_set_nopar:Npn #4 { #2 }
22378 }

```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```

22379 \cs_new_protected:Npn \__keys_set_selective:
22380 {
22381   \cs_if_exist:cTF { \c__keys_groups_root_str \l_keys_path_str }
22382   {
22383     \clist_set_eq:Nc \l__keys_groups_clist
22384     { \c__keys_groups_root_str \l_keys_path_str }
22385     \__keys_check_groups:
22386   }
22387   {

```

```

22388         \bool_if:NTF \l__keys_exclude_bool
22389         \__keys_execute:
22390         \__keys_store_unused:
22391     }
22392 }

```

In the case where selective setting requires a comparison of the list of groups which apply to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set. It is safe to use `\clist_if_in:NnTF` because both `\l__keys_selective_clist` and `\l__keys_groups_clist` contain the groups as strings, without leading/trailing spaces in any item, since the `\clist` functions were applied to the result of applying `\tl_to_str:n`.

```

22393 \cs_new_protected:Npn \__keys_check_groups:
22394 {
22395     \bool_set_false:N \l__keys_tmp_bool
22396     \clist_map_inline:Nn \l__keys_selective_clist
22397     {
22398         \clist_if_in:NnT \l__keys_groups_clist {##1}
22399         {
22400             \bool_set_true:N \l__keys_tmp_bool
22401             \clist_map_break:
22402         }
22403     }
22404     \bool_if:NTF \l__keys_tmp_bool
22405     {
22406         \bool_if:NTF \l__keys_exclude_bool
22407         \__keys_store_unused:
22408         \__keys_execute:
22409     }
22410     {
22411         \bool_if:NTF \l__keys_exclude_bool
22412         \__keys_execute:
22413         \__keys_store_unused:
22414     }
22415 }

```

(End of definition for `__keys_set_keyval:n` and others.)

`__keys_value_or_default:n` If a value is given, return it as #1, otherwise send a default if available.

```

\__keys_default_inherit:
22416 \cs_new_protected:Npn \__keys_value_or_default:n #1
22417 {
22418     \bool_if:NTF \l__keys_no_value_bool
22419     {
22420         \cs_if_exist:cTF { \c__keys_default_root_str \l_keys_path_str }
22421         {
22422             \tl_set_eq:Nc
22423             \l_keys_value_tl
22424             { \c__keys_default_root_str \l_keys_path_str }
22425         }
22426         {
22427             \tl_clear:N \l_keys_value_tl
22428             \cs_if_exist:cT
22429             { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
22430             { \__keys_default_inherit: }

```

```

22431     }
22432   }
22433   { \tl_set:Nn \l_keys_value_tl {#1} }
22434 }
22435 \cs_new_protected:Npn \__keys_default_inherit:
22436 {
22437   \clist_map_inline:cn
22438   { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
22439   {
22440     \cs_if_exist:cT
22441     { \c__keys_default_root_str ##1 / \l_keys_key_str }
22442     {
22443       \tl_set_eq:Nc
22444       \l_keys_value_tl
22445       { \c__keys_default_root_str ##1 / \l_keys_key_str }
22446       \clist_map_break:
22447     }
22448   }
22449 }

```

(End of definition for __keys_value_or_default:n and __keys_default_inherit:.)

__keys_execute: Actually executing a key is done in two parts. First, look for the key itself, then look for the **unknown** key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```

\__keys_execute_inherit:
\__keys_execute_unknown:
  \__keys_execute:nn
  \__keys_execute:no
  \__keys_store_unused:
\__keys_store_unused_aux:
22450 \cs_new_protected:Npn \__keys_execute:
22451 {
22452   \cs_if_exist:cTF { \c__keys_code_root_str \l_keys_path_str }
22453   {
22454     \cs_if_exist_use:c { \c__keys_check_root_str \l_keys_path_str }
22455     \__keys_execute:no \l_keys_path_str \l_keys_value_tl
22456   }
22457   {
22458     \cs_if_exist:cTF
22459     { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
22460     { \__keys_execute_inherit: }
22461     { \__keys_execute_unknown: }
22462   }
22463 }

```

To deal with the case where there is no hit, we leave __keys_execute_unknown: in the input stream and clean it up using the break function: that avoids needing a boolean.

```

22464 \cs_new_protected:Npn \__keys_execute_inherit:
22465 {
22466   \clist_map_inline:cn
22467   { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
22468   {
22469     \cs_if_exist:cT
22470     { \c__keys_code_root_str ##1 / \l_keys_key_str }
22471     {
22472       \str_set:Nn \l__keys_inherit_str {##1}
22473       \cs_if_exist_use:c { \c__keys_check_root_str ##1 / \l_keys_key_str }
22474       \__keys_execute:no { ##1 / \l_keys_key_str } \l_keys_value_tl

```

```

22475         \clist_map_break:n \use_none:n
22476     }
22477 }
22478 \__keys_execute_unknown:
22479 }
22480 \cs_new_protected:Npn \__keys_execute_unknown:
22481 {
22482     \bool_if:NTF \l__keys_only_known_bool
22483     { \__keys_store_unused: }
22484     {
22485         \cs_if_exist:cTF
22486         { \c__keys_code_root_str \l__keys_module_str / unknown }
22487         {
22488             \bool_if:NT \l__keys_no_value_bool
22489             {
22490                 \cs_if_exist:cT
22491                 { \c__keys_default_root_str \l__keys_module_str / unknown }
22492                 {
22493                     \tl_set_eq:Nc
22494                     \l_keys_value_tl
22495                     { \c__keys_default_root_str \l__keys_module_str / unknown }
22496                 }
22497             }
22498             \__keys_execute:no { \l__keys_module_str / unknown } \l_keys_value_tl
22499         }
22500     {
22501         \msg_error:nnee { keys } { unknown }
22502         \l_keys_path_str \l__keys_module_str
22503     }
22504 }
22505 }

```

A key's code is in the control sequence with csname `\c__keys_code_root_str #1`. We expand it once to get the replacement text (with argument #2) and call `\use:n` with this replacement as its argument. This ensures that any undefined control sequence error in the key's code will lead to an error message of the form `<argument>...<control sequence>` in which one can read the (undefined) `<control sequence>` in full, rather than an error message that starts with the potentially very long key name, which would make the (undefined) `<control sequence>` be truncated or sometimes completely hidden. See <https://github.com/latex3/latex2e/issues/351>.

```

22506 \cs_new:Npn \__keys_execute:nn #1#2
22507 { \__keys_execute:no {#1} { \prg_do_nothing: #2 } }
22508 \cs_new:Npn \__keys_execute:no #1#2
22509 {
22510     \exp_args:NNo \exp_args:No \use:n
22511     {
22512         \cs:w \c__keys_code_root_str #1 \exp_after:wN \cs_end:
22513         \exp_after:wN {#2}
22514     }
22515 }

```

When there is no relative path, things here are easy: just save the key name and value. When we are working with a relative path, first we need to turn it into a string: that can't happen earlier as we need to store `\q__keys_no_value`. Then, use a standard delimited

approach to fish out the partial path.

```

22516 \cs_new_protected:Npn \__keys_store_unused:
22517 {
22518   \__keys_quark_if_no_value:NTF \l__keys_relative_tl
22519   {
22520     \clist_put_right:Ne \l__keys_unused_clist
22521     {
22522       \l_keys_key_str
22523       \bool_if:NF \l__keys_no_value_bool
22524       { = { \exp_not:o \l_keys_value_tl } }
22525     }
22526   }
22527   {
22528     \tl_if_empty:NTF \l__keys_relative_tl
22529     {
22530       \clist_put_right:Ne \l__keys_unused_clist
22531       {
22532         \l_keys_path_str
22533         \bool_if:NF \l__keys_no_value_bool
22534         { = { \exp_not:o \l_keys_value_tl } }
22535       }
22536     }
22537     { \__keys_store_unused_aux: }
22538   }
22539 }
22540 \cs_new_protected:Npn \__keys_store_unused_aux:
22541 {
22542   \__kernel_tl_set:Nx \l__keys_relative_tl
22543   { \exp_args:No \__keys_trim_spaces:n \l__keys_relative_tl }
22544   \use:e
22545   {
22546     \cs_set_protected:Npn \__keys_store_unused:w
22547     ##1 \l__keys_relative_tl /
22548     ##2 \l__keys_relative_tl /
22549     ##3 \s__keys_stop
22550   }
22551   {
22552     \tl_if_blank:nF {##1}
22553     {
22554       \msg_error:nnee { keys } { bad-relative-key-path }
22555       \l_keys_path_str
22556       \l__keys_relative_tl
22557     }
22558     \clist_put_right:Ne \l__keys_unused_clist
22559     {
22560       \exp_not:n {##2}
22561       \bool_if:NF \l__keys_no_value_bool
22562       { = { \exp_not:o \l_keys_value_tl } }
22563     }
22564   }
22565   \use:e
22566   {
22567     \__keys_store_unused:w \l_keys_path_str
22568     \l__keys_relative_tl / \l__keys_relative_tl /

```

```

22569         \s__keys_stop
22570     }
22571 }
22572 \cs_new_protected:Npn \__keys_store_unused:w { }

```

(End of definition for `__keys_execute:` and others.)

`__keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the
`__keys_choice_find:nn` unknown key. That always exists, as it is created when a choice is first made. So there
`__keys_multichoice_find:n` is no need for any escape code. For multiple choices, the same code ends up used in a
mapping.

```

22573 \cs_new:Npn \__keys_choice_find:n #1
22574 {
22575     \str_if_empty:NTF \l__keys_inherit_str
22576     { \__keys_choice_find:nn \l_keys_path_str {#1} }
22577     {
22578         \__keys_choice_find:nn
22579         { \l__keys_inherit_str / \l_keys_key_str } {#1}
22580     }
22581 }
22582 \cs_new:Npn \__keys_choice_find:nn #1#2
22583 {
22584     \cs_if_exist:cTF { \c__keys_code_root_str #1 / \__keys_trim_spaces:n {#2} }
22585     { \__keys_execute:nn { #1 / \__keys_trim_spaces:n {#2} } {#2} }
22586     { \__keys_execute:nn { #1 / unknown } {#2} }
22587 }
22588 \cs_new:Npn \__keys_multichoice_find:n #1
22589 { \clist_map_function:nN {#1} \__keys_choice_find:n }

```

(End of definition for `__keys_choice_find:n`, `__keys_choice_find:nn`, and `__keys_multichoice_find:n`.)

65.7 Utilities

`__keys_parent:o` Used to strip off the ending part of the key path after the last /.

```

\__keys_parent_auxi:w
\__keys_parent_auxii:w
\__keys_parent_auxiii:n
\__keys_parent_auxiv:w
22590 \cs_new:Npn \__keys_parent:o #1
22591 {
22592     \exp_after:wN \__keys_parent_auxi:w #1 \q_nil \__keys_parent_auxii:w
22593     / \q_nil \__keys_parent_auxiv:w
22594 }
22595 \cs_new:Npn \__keys_parent_auxi:w #1 / #2 \q_nil #3
22596 {
22597     #3 { #1 } #2 \q_nil #3
22598 }
22599 \cs_new:Npn \__keys_parent_auxii:w #1 #2 \q_nil \__keys_parent_auxii:w
22600 {
22601     #1 \__keys_parent_auxi:w #2 \q_nil \__keys_parent_auxiii:n
22602 }
22603 \cs_new:Npn \__keys_parent_auxiii:n #1
22604 {
22605     / #1 \__keys_parent_auxi:w
22606 }
22607 \cs_new:Npn \__keys_parent_auxiv:w #1 \q_nil \__keys_parent_auxiv:w

```

```

22608 {
22609 }

```

(End of definition for `__keys_parent:o` and others.)

`__keys_trim_spaces:n` Space stripping has to allow for the fact that the key here might have several parts, and spaces need to be stripped from each part. Since the key name is turned into a string groups can't be stripped accidentally and the precautions of `\tl_trim_spaces:n` aren't necessary, in this case it is much faster to just directly strip spaces around `/`.

```

22610 \group_begin:
22611   \cs_set:Npn \__keys_tmp:w #1
22612     {
22613       \cs_new:Npn \__keys_trim_spaces:n ##1
22614         {
22615           \exp_after:wN \__keys_trim_spaces_auxi:w \tl_to_str:n { / ##1 } /
22616           \s__keys_nil \__keys_trim_spaces_auxi:w
22617           \s__keys_mark \__keys_trim_spaces_auxii:w
22618           #1 / #1
22619           \s__keys_nil \__keys_trim_spaces_auxii:w
22620           \s__keys_mark \__keys_trim_spaces_auxiii:w
22621         }
22622       }
22623   \__keys_tmp:w { ~ }
22624 \group_end:
22625 \cs_new:Npn \__keys_trim_spaces_auxi:w #1 ~ / #2 \s__keys_nil #3
22626   {
22627     #3 #1 / #2 \s__keys_nil #3
22628   }
22629 \cs_new:Npn \__keys_trim_spaces_auxii:w #1 / ~ #2 \s__keys_mark #3
22630   {
22631     #3 #1 / #2 \s__keys_mark #3
22632   }
22633 \cs_new:Npn \__keys_trim_spaces_auxiii:w
22634   / #1 /
22635   \s__keys_nil \__keys_trim_spaces_auxi:w
22636   \s__keys_mark \__keys_trim_spaces_auxii:w
22637   /
22638   \s__keys_nil \__keys_trim_spaces_auxii:w
22639   \s__keys_mark \__keys_trim_spaces_auxiii:w
22640   {
22641     #1
22642   }

```

(End of definition for `__keys_trim_spaces:n` and others.)

`\keys_if_exist_p:nn` A utility for others to see if a key exists.

`\keys_if_exist:nnTF`

```

22643 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
22644   {
22645     \cs_if_exist:cTF
22646       { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 } }
22647       { \prg_return_true: }
22648       { \prg_return_false: }
22649   }
22650 \prg_generate_conditional_variant:Nnn \keys_if_exist:nn { ne } { p , T , F , TF }

```


(End of definition for `\keys_if_exist:nnTF`. This function is documented on page 251.)

`\keys_if_choice_exist_p:nnn` Just an alternative view on `\keys_if_exist:nnTF`.

```
\keys_if_choice_exist:nnnTF 22651 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
                               22652 { p , T , F , TF }
                               22653 {
                               22654   \cs_if_exist:cTF
                               22655     { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 / #3 } }
                               22656     { \prg_return_true: }
                               22657     { \prg_return_false: }
                               22658 }
```

(End of definition for `\keys_if_choice_exist:nnnTF`. This function is documented on page 251.)

`\keys_show:nn` To show a key, show its code using a message.

```
\keys_log:nn 22659 \cs_new_protected:Npn \keys_show:nn
\__keys_show:Nnn 22660 { \__keys_show:Nnn \msg_show:nneeee }
  \__keys_show:n 22661 \cs_new_protected:Npn \keys_log:nn
  \__keys_show:w 22662 { \__keys_show:Nnn \msg_log:nneeee }
\__keys_show:Nw 22663 \cs_new_protected:Npn \__keys_show:Nnn #1#2#3
                22664 {
                22665   #1 { keys } { show-key }
                22666   { \__keys_trim_spaces:n { #2 / #3 } }
                22667   {
                22668     \keys_if_exist:nnT {#2} {#3}
                22669     {
                22670       \exp_args:Nnf \msg_show_item_unbraced:nn { code }
                22671       {
                22672         \exp_args:Ne \__keys_show:n
                22673         {
                22674           \exp_args:Nc \cs_replacement_spec:N
                22675           {
                22676             \c__keys_code_root_str
                22677             \__keys_trim_spaces:n { #2 / #3 }
                22678           }
                22679         }
                22680       }
                22681     }
                22682   }
                22683 } { } { }
                22684 }
22685 \cs_new:Npe \__keys_show:n #1
22686 {
22687   \exp_not:N \__keys_show:w
22688   #1
22689   \tl_to_str:n { \__keys_precompile:n }
22690   #1
22691   \tl_to_str:n { \__keys_precompile:n }
22692   \exp_not:N \s__keys_stop
22693 }
22694 \use:e
22695 {
22696   \cs_new:Npn \exp_not:N \__keys_show:w
22697   #1 \tl_to_str:n { \__keys_precompile:n }
```

```

22698     #2 \tl_to_str:n { \__keys_precompile:n }
22699     #3 \exp_not:N \s__keys_stop
22700   }
22701   {
22702     \tl_if_blank:nTF {#2}
22703     {#1}
22704     { \__keys_show:Nw #2 \s__keys_stop }
22705   }
22706 \use:e
22707   {
22708     \cs_new:Npn \exp_not:N \__keys_show:Nw #1#2
22709     \c_right_brace_str \exp_not:N \s__keys_stop
22710   }
22711   {#2}

```

(End of definition for `\keys_show:nn` and others. These functions are documented on page 251.)

65.8 Messages

For when there is a need to complain.

```

22712 \msg_new:nnnn { keys } { bad-relative-key-path }
22713 { The-key~'#1'~is-not~inside~the~'#2'~path. }
22714 { The-key~'#1'~cannot-be-expressed-relative-to-path~'#2'. }
22715 \msg_new:nnnn { keys } { boolean-values-only }
22716 { Key~'#1'~accepts~boolean-values-only. }
22717 { The-key~'#1'~only-accepts-the-values~'true'~and~'false'. }
22718 \msg_new:nnnn { keys } { choice-unknown }
22719 { Key~'#1'~accepts-only-a-fixed-set-of-choices. }
22720 {
22721   The-key~'#1'~only-accepts-predefined-values,~
22722   and~'#2'~is-not-one-of-these.
22723 }
22724 \msg_new:nnnn { keys } { unknown }
22725 { The-key~'#1'~is-unknown-and-is-being-ignored. }
22726 {
22727   The-module~'#2'~does-not-have-a-key-called~'#1'.\\
22728   Check-that-you-have-spelled-the-key-name-correctly.
22729 }
22730 \msg_new:nnnn { keys } { nested-choice-key }
22731 { Attempt-to-define~'#1'~as-a-nested-choice-key. }
22732 {
22733   The-key~'#1'~cannot-be-defined-as-a-choice-as-the-parent-key~'#2'~is~
22734   itself-a-choice.
22735 }
22736 \msg_new:nnnn { keys } { value-forbidden }
22737 { The-key~'#1'~does-not-take-a-value. }
22738 {
22739   The-key~'#1'~should-be-given-without-a-value.\\
22740   The-value~'#2'~was-present:-the-key-will-be-ignored.
22741 }
22742 \msg_new:nnnn { keys } { value-required }
22743 { The-key~'#1'~requires-a-value. }
22744 {

```

```

22745     The-key-~'#1'~must-have-a-value.\\
22746     No-value-was-present:~the-key-will-be-ignored.
22747   }
22748   \msg_new:nnn { keys } { show-key }
22749   {
22750     The-key-#1~
22751     \tl_if_empty:nTF {#2}
22752       { is-undefined. }
22753       { has-the-properties: #2 . }
22754   }
22755   \prop_gput:Nnn \g_msg_module_name_prop { keys } { LaTeX }
22756   \prop_gput:Nnn \g_msg_module_type_prop { keys } { }
22757 </package>

```

Chapter 66

l3intarray implementation

```
22758 (*package)
```

```
22759 (@@=intarray)
```

There are two implementations for this module: One `\fontdimen` based one for more traditional \TeX engines and a Lua based one for engines with Lua support.

Both versions do not allow negative array sizes.

```
22760 (*tex)
```

```
22761 \msg_new:nnn { kernel } { negative-array-size }
```

```
22762 { Size-of-array-may-not-be-negative:~#1 }
```

`\l__intarray_loop_int` A loop index.

```
22763 \int_new:N \l__intarray_loop_int
```

(End of definition for `\l__intarray_loop_int`.)

66.1 Lua implementation

First, let's look at the Lua variant:

We select the Lua version if the Lua helpers were defined. This can be detected by the presence of `__intarray_gset_count:Nw`.

```
22764 \cs_if_exist:NTF \__intarray_gset_count:Nw
```

```
22765 {
```

66.1.1 Allocating arrays

`\g__intarray_table_int` Used to differentiate intarrays in Lua and to record an invalid index.

```
\l__intarray_bad_index_int 22766 \int_new:N \g__intarray_table_int
```

```
22767 \int_new:N \l__intarray_bad_index_int
```

```
22768 </tex>
```

(End of definition for `\g__intarray_table_int` and `\l__intarray_bad_index_int`.)

`__intarray:w` Used as marker for intarrays in Lua. Followed by an unbraced number identifying the array and a single space. This format is used to make it easy to scan from Lua.

```
22769 (*lua)
```

```
22770 \luacmd{'__intarray:w', function()
```

```
22771 scan_int()
```

```

22772 tex.error'LaTeX Error: Isolated intarray ignored'
22773 end, 'protected', 'global')
22774 </lua>

```

(End of definition for `_intarray:w`.)

`\intarray_new:Nn` Declare #1 as a tokenlist with the scanmark and a unique number. Pass the array's size to the Lua helper. Every `intarray` must be global; it's enough to run this check in `_intarray_new:N`

`\intarray_new:cn` to the Lua helper. Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

`_intarray_new:N` `\intarray_new:Nn`.

```

22775 (*tex)
22776   \cs_new_protected:Npn \_intarray_new:N #1
22777     {
22778       \_kernel_chk_if_free_cs:N #1
22779       \int_gincr:N \g\_intarray_table_int
22780       \cs_gset_nopar:Npe #1 { \_intarray:w \int_use:N \g\_intarray_table_int \c_space_tl
22781     }
22782   \cs_new_protected:Npn \intarray_new:Nn #1#2
22783     {
22784       \_intarray_new:N #1
22785       \_intarray_gset_count:Nw #1 \int_eval:n {#2} \scan_stop:
22786       \int_compare:nNnT { \intarray_count:N #1 } < 0
22787         {
22788           \msg_error:nne { kernel } { negative-array-size }
22789           { \intarray_count:N #1 }
22790         }
22791     }
22792   \cs_generate_variant:Nn \intarray_new:Nn { c }
22793 </tex>

```

(End of definition for `\intarray_new:Nn` and `_intarray_new:N`. This function is documented on page 255.)

Before we get to the first command implemented in Lua, we first need some definitions. Since `token.create` only works correctly if `TEX` has seen the tokens before, we first run a short `TEX` sequence to ensure that all relevant control sequences are known.

```

22794 (*lua)
22795
22796 local scan_token = token.scan_token
22797 local put_next = token.put_next
22798 local intarray_marker = token_create_safe'_'intarray:w'
22799 local use_none = token_create_safe'use_none:n'
22800 local use_i = token_create_safe'use:n'
22801 local expand_after_scan_stop = {token_create_safe'exp_after:wN',
22802                                token_create_safe'scan_stop:'}
22803 local comma = token_create(string.byte',')

```

`_intarray_table` Internal helper to scan an `intarray` token, extract the associated Lua table and return an error if the input is invalid.

```

22804 local \_intarray_table do
22805   local tables = get_lua_data and get_lua_data'_'intarray' or {[0] = {}}
22806   function \_intarray_table()
22807     local t = scan_token()
22808     if t ~= intarray_marker then
22809       put_next(t)
22810       tex.error'LaTeX Error: intarray expected'

```

```

22811     return tables[0]
22812 end
22813 local i = scan_int()
22814 local current_table = tables[i]
22815 if current_table then return current_table end
22816 current_table = {}
22817 tables[i] = current_table
22818 return current_table
22819 end

```

Since in L^AT_EX this is loaded in the format, we want to preserve any intarrays which are created while format building for the actual run.

To do this, we use the `register_luadata` mechanism from l3`luatex`: Directly before the format get dumped, the following function gets invoked and serializes all existing tables into a string. This string gets compiled and dumped into the format and is made available at the beginning of regular runs as `get_luadata'@@'`.

```

22820 if register_luadata then
22821   register_luadata('__intarray', function()
22822     local t = "{[0]={},"
22823     for i=1, #tables do
22824       t = string.format("%s{%s}," , t, table.concat(tables[i], ','))
22825     end
22826     return t .. "}"
22827   end)
22828 end
22829 end

```

(End of definition for `__intarray_table`.)

`\intarray_count:N` Set and get the size of an array. “Setting the size” means in this context that we add zeros until we reach the desired size.

`\intarray_count:c`

`__intarray_gset_count:Nw`

```

22830
22831 local sprint = tex.sprint
22832
22833 luacmd('__intarray_gset_count:Nw', function()
22834   local t = __intarray_table()
22835   local n = scan_int()
22836   for i=#t+1, n do t[i] = 0 end
22837 end, 'protected', 'global')
22838
22839 luacmd('intarray_count:N', function()
22840   sprint(-2, #__intarray_table())
22841 end, 'global')
22842 \lua
22843 \*tex
22844 \cs_generate_variant:Nn \intarray_count:N { c }
22845 \*tex

```

(End of definition for `\intarray_count:N` and `__intarray_gset_count:Nw`. This function is documented on page 256.)

66.1.2 Array items

`__intarray_gset:wF` The setter provided by Lua. The argument order somewhat emulates the `\fontdimen:`
`__intarray_gset:w` First the array index, followed by the intarray and then the new value. This has been chosen over a more conventional order to provide a delimiter for the numbers.

```

22846 (*lua)
22847 luacmd('__intarray_gset:wF', function()
22848   local i = scan_int()
22849   local t = __intarray_table()
22850   if t[i] then
22851     t[i] = scan_int()
22852     put_next(use_none)
22853   else
22854     tex.count.l__intarray_bad_index_int = i
22855     scan_int()
22856     put_next(use_i)
22857   end
22858 end, 'protected', 'global')
22859
22860 luacmd('__intarray_gset:w', function()
22861   local i = scan_int()
22862   local t = __intarray_table()
22863   t[i] = scan_int()
22864 end, 'protected', 'global')
22865 </lua>

```

(End of definition for `__intarray_gset:wF` and `__intarray_gset:w`.)

`\intarray_gset:Nnn` The `__kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its
`\intarray_gset:cnn` arguments must be suitable for `\int_value:w`. The user version checks the position and
`__kernel_intarray_gset:Nnn` value are within bounds.

```

22866 (*tex)
22867   \cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
22868     { \__intarray_gset:w #2 #1 #3 \scan_stop: }
22869   \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
22870     {
22871       \__intarray_gset:wF \int_eval:n {#2} #1 \int_eval:n{#3}
22872       {
22873         \msg_error:nneee { kernel } { out-of-bounds }
22874         { \token_to_str:N #1 } { \int_use:N \l__intarray_bad_index_int } { \intarray_
22875       }
22876     }
22877   \cs_generate_variant:Nn \intarray_gset:Nnn { c }
22878 </tex>

```

(End of definition for `\intarray_gset:Nnn` and `__kernel_intarray_gset:Nnn`. This function is documented on page 256.)

`\intarray_gzero:N` Set the appropriate array entry to zero. No bound checking needed.

```

\intarray_gzero:c
22879 (*lua)
22880 luacmd('intarray_gzero:N', function()
22881   local t = __intarray_table()
22882   for i=1, #t do
22883     t[i] = 0

```

```

22884   end
22885 end, 'global', 'protected')
22886 </lua>
22887 <*tex>
22888   \cs_generate_variant:Nn \intarray_gzero:N { c }
22889 </tex>

```

(End of definition for `\intarray_gzero:N`. This function is documented on page 255.)

`\intarray_item:Nn` Get the appropriate entry and perform bound checks. The `__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

```

\__kernel_intarray_item:Nn
\__intarray_item:wF
\__intarray_item:w
22890 <*lua>
22891 luacmd('\__intarray_item:wF', function()
22892   local i = scan_int()
22893   local t = __intarray_table()
22894   local item = t[i]
22895   if item then
22896     put_next(use_none)
22897   else
22898     tex.l__intarray_bad_index_int = i
22899     put_next(use_i)
22900   end
22901   put_next(expand_after_scan_stop)
22902   scan_token()
22903   if item then
22904     sprint(-2, item)
22905   end
22906 end, 'global')
22907
22908 luacmd('\__intarray_item:w', function()
22909   local i = scan_int()
22910   local t = __intarray_table()
22911   sprint(-2, t[i])
22912 end, 'global')
22913 </lua>
22914 <*tex>
22915   \cs_new:Npn \__kernel_intarray_item:Nn #1#2
22916     { \__intarray_item:w #2 #1 }
22917   \cs_new:Npn \intarray_item:Nn #1#2
22918     {
22919       \__intarray_item:wF \int_eval:n {#2} #1
22920       {
22921         \msg_expandable_error:nnfff { kernel } { out-of-bounds }
22922         { \token_to_str:N #1 } { \int_use:N \l__intarray_bad_index_int } { \intarray_
22923           0
22924         }
22925       }
22926   \cs_generate_variant:Nn \intarray_item:Nn { c }

```

(End of definition for `\intarray_item:Nn` and others. This function is documented on page 256.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.

```

\intarray_rand_item:c
22927   \cs_new:Npn \intarray_rand_item:N #1

```



```

22928     { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
22929     \cs_generate_variant:Nn \intarray_rand_item:N { c }

```

(End of definition for `\intarray_rand_item:N`. This function is documented on page 256.)

66.1.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn`
`\intarray_const_from_clist:cn` We use the `__kernel_intarray_gset:Nnn` which does not do bounds checking and instead automatically resizes the array. This is not implemented in Lua to ensure that the clist parsing is consistent with the clist module.

```

22930     \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
22931     {
22932         \__intarray_new:N #1
22933         \int_zero:N \l__intarray_loop_int
22934         \clist_map_inline:nn {#2}
22935         {
22936             \int_incr:N \l__intarray_loop_int
22937             \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int { \int_eval:n {##1} } }
22938     }
22939     \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }

```

(End of definition for `\intarray_const_from_clist:Nn`. This function is documented on page 255.)

`__intarray_to_clist:Nn`
`__intarray_to_clist:w` The `__intarray_to_clist:Nn` auxiliary allows to choose the delimiter and is also used in `\intarray_show:N`. Here we just pass the information to Lua and let `table.concat` do the actual work. We discard the category codes of the passed delimiter but this is not an issue since the delimiter is always just a comma or a comma and a space. In both cases `sprint(2, ...)` provides the right catcodes.

```

22940 \end{tex}
22941 \begin{lua}
22942 local concat = table.concat
22943 luacmd('\__intarray_to_clist:Nn', function()
22944     local t = __intarray_table()
22945     local sep = token.scan_string()
22946     sprint(-2, concat(t, sep))
22947 end, 'global')
22948 \end{lua}

```

(End of definition for `__intarray_to_clist:Nn` and `__intarray_to_clist:w`.)

`__kernel_intarray_range_to_clist:Nnn`
`__intarray_range_to_clist:w` Loop through part of the array.

```

22949 \begin{tex}
22950     \cs_new:Npn \__kernel_intarray_range_to_clist:Nnn #1#2#3
22951     {
22952         \__intarray_range_to_clist:w #1
22953         \int_eval:n {#2} ~ \int_eval:n {#3} ~
22954     }
22955 \end{tex}
22956 \begin{lua}
22957 luacmd('\__intarray_range_to_clist:w', function()
22958     local t = __intarray_table()
22959     local from = scan_int()
22960     local to = scan_int()
22961     sprint(-2, concat(t, ', ', from, to))

```

```

22962 end, 'global')
22963 </lua>

```

(End of definition for `_kernel_intarray_range_to_clist:Nnn` and `_intarray_range_to_clist:w`.)

```

\_kernel_intarray_gset_range_from_clist:Nnn
\_intarray_gset_range:nNw

```

Loop through part of the array. We allow additional commas at the end.

```

22964 (*tex)
22965   \cs_new_protected:Npn \_kernel_intarray_gset_range_from_clist:Nnn #1#2#3
22966     {
22967       \_intarray_gset_range:w \int_eval:w #2 #1 #3 , , \scan_stop:
22968     }
22969 </tex>
22970 (*lua)
22971 luacmd('\_intarray_gset_range:w', function()
22972   local from = scan_int()
22973   local t = \_intarray_table()
22974   while true do
22975     local tok = scan_token()
22976     if tok == comma then
22977       repeat
22978         tok = scan_token()
22979       until tok ~= comma
22980       break
22981     else
22982       put_next(tok)
22983     end
22984     t[from] = scan_int()
22985     scan_token()
22986     from = from + 1
22987   end
22988   end, 'global', 'protected')
22989 </lua>

```

(End of definition for `_kernel_intarray_gset_range_from_clist:Nnn` and `_intarray_gset_range:nNw`.)

```

\_intarray_gset_overflow_test:nw

```

In order to allow some code sharing later we provide the `_intarray_gset_overflow_test:nw` name here. It doesn't actually test anything since the Lua implementation accepts all integers which could be tested with `\tex_ifabsnum:D`.

```

22990 (*tex)
22991   \cs_new_protected:Npn \_intarray_gset_overflow_test:nw #1
22992     {
22993     }

```

(End of definition for `_intarray_gset_overflow_test:nw`.)

66.2 Font dimension based implementation

Go to the false branch of the conditional above.

```

22994   }
22995   {

```

66.2.1 Allocating arrays

`__intarray_entry:w` We use these primitives quite a lot in this module.
`__intarray_count:w` 22996 `\cs_new_eq:NN __intarray_entry:w \tex_fontdimen:D`
22997 `\cs_new_eq:NN __intarray_count:w \tex_hyphenchar:D`
(End of definition for __intarray_entry:w and __intarray_count:w.)

`\c__intarray_sp_dim` Used to convert integers to dimensions fast.
22998 `\dim_const:Nn \c__intarray_sp_dim { 1 sp }`
(End of definition for \c__intarray_sp_dim.)

`\g__intarray_font_int` Used to assign one font per array.
22999 `\int_new:N \g__intarray_font_int`
(End of definition for \g__intarray_font_int.)

`\intarray_new:Nn` Declare #1 to be a font (arbitrarily cmr10 at a never-used size). Store the array's size as the `\hyphenchar` of that font and make sure enough `\fontdimen` are allocated, by setting the last one. Then clear any `\fontdimen` that cmr10 starts with. It seems LuaTeX's cmr10 has an extra `\fontdimen` parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

```

23000 \cs_new_protected:Npn \__intarray_new:N #1
23001 {
23002   \__kernel_chk_if_free_cs:N #1
23003   \int_gincr:N \g__intarray_font_int
23004   \tex_global:D \tex_font:D #1
23005   = cmr10-at~ \g__intarray_font_int \c__intarray_sp_dim \scan_stop:
23006   \int_step_inline:nn { 8 }
23007   { \__kernel_intarray_gset:Nnn #1 {##1} \c_zero_int }
23008 }
23009 \cs_new_protected:Npn \intarray_new:Nn #1#2
23010 {
23011   \__intarray_new:N #1
23012   \__intarray_count:w #1 = \int_eval:n {#2} \scan_stop:
23013   \int_compare:nNnT { \intarray_count:N #1 } < 0
23014   {
23015     \msg_error:nne { kernel } { negative-array-size }
23016     { \intarray_count:N #1 }
23017   }
23018   \int_compare:nNnT { \intarray_count:N #1 } > 0
23019   { \__kernel_intarray_gset:Nnn #1 { \intarray_count:N #1 } { 0 } }
23020 }
23021 \cs_generate_variant:Nn \intarray_new:Nn { c }

```

(End of definition for \intarray_new:Nn and __intarray_new:N. This function is documented on page 255.)

`\intarray_count:N` Size of an array.
`\intarray_count:c` 23022 `\cs_new:Npn \intarray_count:N #1 { \int_value:w __intarray_count:w #1 }`
23023 `\cs_generate_variant:Nn \intarray_count:N { c }`

(End of definition for \intarray_count:N. This function is documented on page 256.)

66.2.2 Array items

`__intarray_signed_max_dim:n` Used when an item to be stored is larger than `\c_max_dim` in absolute value; it is replaced by $\pm\c_max_dim$.

```
23024 \cs_new:Npn \__intarray_signed_max_dim:n #1
23025 { \int_value:w \int_compare:nNnT {#1} < 0 { - } \c_max_dim }
```

(End of definition for `__intarray_signed_max_dim:n`.)

`__intarray_bounds:NNnTF` The functions `\intarray_gset:Nnn` and `\intarray_item:Nn` share bounds checking. `__intarray_bounds_error:NNnw` The T branch is used if #3 is within bounds of the array #2.

```
23026 \cs_new:Npn \__intarray_bounds:NNnTF #1#2#3
23027 {
23028   \if_int_compare:w 1 > #3 \exp_stop_f:
23029   \__intarray_bounds_error:NNnw #1 #2 {#3}
23030   \else:
23031   \if_int_compare:w #3 > \intarray_count:N #2 \exp_stop_f:
23032   \__intarray_bounds_error:NNnw #1 #2 {#3}
23033   \fi:
23034   \fi:
23035   \use_i:nn
23036 }
23037 \cs_new:Npn \__intarray_bounds_error:NNnw #1#2#3#4 \use_i:nn #5#6
23038 {
23039   #4
23040   #1 { kernel } { out-of-bounds }
23041   { \token_to_str:N #2 } {#3} { \intarray_count:N #2 }
23042   #6
23043 }
```

(End of definition for `__intarray_bounds:NNnTF` and `__intarray_bounds_error:NNnw`.)

`\intarray_gset:Nnn` Set the appropriate `\fontdimen`. The `__kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its arguments must be suitable for `\int_value:w`. The user version checks the position and value are within bounds.

`\intarray_gset:cnn`

`__kernel_intarray_gset:Nnn`

`__intarray_gset:Nnn`

`__intarray_gset_overflow:Nnn`

```
23044 \cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
23045 { \__intarray_entry:w #2 #1 #3 \c__intarray_sp_dim }
23046 \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
23047 {
23048   \exp_after:wN \__intarray_gset:Nww
23049   \exp_after:wN #1
23050   \int_value:w \int_eval:n {#2} \exp_after:wN ;
23051   \int_value:w \int_eval:n {#3} ;
23052 }
23053 \cs_generate_variant:Nn \intarray_gset:Nnn { c }
23054 \cs_new_protected:Npn \__intarray_gset:Nnw #1#2 ; #3 ;
23055 {
23056   \__intarray_bounds:NNnTF \msg_error:nnee #1 {#2}
23057   {
23058     \__intarray_gset_overflow_test:nw {#3}
23059     \__kernel_intarray_gset:Nnn #1 {#2} {#3}
23060   }
23061   { }
23062 }
```

```

23063 \cs_if_exist:NTF \tex_ifabsnum:D
23064 {
23065   \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
23066   {
23067     \tex_ifabsnum:D #1 > \c_max_dim
23068     \exp_after:wN \__intarray_gset_overflow:NNnn
23069     \fi:
23070   }
23071 }
23072 {
23073   \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
23074   {
23075     \if_int_compare:w \int_abs:n {#1} > \c_max_dim
23076     \exp_after:wN \__intarray_gset_overflow:NNnn
23077     \fi:
23078   }
23079 }
23080 \cs_new_protected:Npn \__intarray_gset_overflow:NNnn #1#2#3#4
23081 {
23082   \msg_error:nneeee { kernel } { overflow }
23083   { \token_to_str:N #2 } {#3} {#4} { \__intarray_signed_max_dim:n {#4} }
23084   #1 #2 {#3} { \__intarray_signed_max_dim:n {#4} }
23085 }

```

(End of definition for `\intarray_gset:Nnn` and others. This function is documented on page 256.)

`\intarray_gzero:N` Set the appropriate `\fontdimen` to zero. No bound checking needed. The `\prg_replicate:nn` possibly uses quite a lot of memory, but this is somewhat comparable to the size of the array, and it is much faster than an `\int_step_inline:nn` loop.

`\intarray_gzero:c`

```

23086 \cs_new_protected:Npn \intarray_gzero:N #1
23087 {
23088   \int_zero:N \l__intarray_loop_int
23089   \prg_replicate:nn { \intarray_count:N #1 }
23090   {
23091     \int_incr:N \l__intarray_loop_int
23092     \__intarray_entry:w \l__intarray_loop_int #1 \c_zero_dim
23093   }
23094 }
23095 \cs_generate_variant:Nn \intarray_gzero:N { c }

```

(End of definition for `\intarray_gzero:N`. This function is documented on page 255.)

`\intarray_item:Nn` Get the appropriate `\fontdimen` and perform bound checks. The `__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

`\intarray_item:cn`

`__kernel_intarray_item:Nn`

`__intarray_item:Nw`

```

23096 \cs_new:Npn \__kernel_intarray_item:Nn #1#2
23097 { \int_value:w \__intarray_entry:w #2 #1 }
23098 \cs_new:Npn \intarray_item:Nn #1#2
23099 {
23100   \exp_after:wN \__intarray_item:Nw
23101   \exp_after:wN #1
23102   \int_value:w \int_eval:n {#2} ;
23103 }
23104 \cs_generate_variant:Nn \intarray_item:Nn { c }

```

```

23105 \cs_new:Npn \__intarray_item:Nw #1#2 ;
23106 {
23107   \__intarray_bounds:NNnTF \msg_expandable_error:nnfff #1 {#2}
23108   { \__kernel_intarray_item:Nn #1 {#2} }
23109   { 0 }
23110 }

```

(End of definition for `\intarray_item:Nn`, `__kernel_intarray_item:Nn`, and `__intarray_item:Nw`. This function is documented on page 256.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.

```

\intarray_rand_item:c 23111 \cs_new:Npn \intarray_rand_item:N #1
23112 { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
23113 \cs_generate_variant:Nn \intarray_rand_item:N { c }

```

(End of definition for `\intarray_rand_item:N`. This function is documented on page 256.)

66.2.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn` Similar to `\intarray_new:Nn` (which we don't use because when debugging is enabled that function checks the variable name starts with `g_`). We make use of the fact that `TeX` allows allocation of successive `\fontdimen` as long as no other font has been declared: no need to count the comma list items first. We need the code in `\intarray_gset:Nnn` that checks the item value is not too big, namely `__intarray_gset_overflow_test:nw`, but not the code that checks bounds. At the end, set the size of the intarray.

```

\intarray_const_from_clist:cn 23114 \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
\__intarray_const_from_clist:nN 23115 {
23116   \__intarray_new:N #1
23117   \int_zero:N \l__intarray_loop_int
23118   \clist_map_inline:nn {#2}
23119   { \exp_args:Nf \__intarray_const_from_clist:nN { \int_eval:n {##1} } #1 }
23120   \__intarray_count:w #1 \l__intarray_loop_int
23121 }
23122 \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
23123 \cs_new_protected:Npn \__intarray_const_from_clist:nN #1#2
23124 {
23125   \int_incr:N \l__intarray_loop_int
23126   \__intarray_gset_overflow_test:nw {#1}
23127   \__kernel_intarray_gset:Nnn #2 \l__intarray_loop_int {#1}
23128 }

```

(End of definition for `\intarray_const_from_clist:Nn` and `__intarray_const_from_clist:nN`. This function is documented on page 255.)

`__intarray_to_clist:Nn` Loop through the array, putting a comma before each item. Remove the leading comma with `f`-expansion. We also use the auxiliary in `\intarray_show:N` with argument comma, space.

```

\__intarray_to_clist:w 23129 \cs_new:Npn \__intarray_to_clist:Nn #1#2
23130 {
23131   \int_compare:nNnF { \intarray_count:N #1 } = \c_zero_int
23132   {
23133     \exp_last_unbraced:Nf \use_none:n
23134     { \__intarray_to_clist:w 1 ; #1 {#2} \prg_break_point: }
23135   }

```

```

23136     }
23137 \cs_new:Npn \__intarray_to_clist:w #1 ; #2#3
23138     {
23139     \if_int_compare:w #1 > \__intarray_count:w #2
23140     \prg_break:n
23141     \fi:
23142     #3 \__kernel_intarray_item:Nn #2 {#1}
23143     \exp_after:wN \__intarray_to_clist:w
23144     \int_value:w \int_eval:w #1 + \c_one_int ; #2 {#3}
23145     }

```

(End of definition for __intarray_to_clist:Nn and __intarray_to_clist:w.)

__kernel_intarray_range_to_clist:Nnn Loop through part of the array.

```

\__intarray_range_to_clist:w 23146 \cs_new:Npn \__kernel_intarray_range_to_clist:Nnn #1#2#3
23147     {
23148     \exp_last_unbraced:Nf \use_none:n
23149     {
23150     \exp_after:wN \__intarray_range_to_clist:ww
23151     \int_value:w \int_eval:w #2 \exp_after:wN ;
23152     \int_value:w \int_eval:w #3 ;
23153     #1 \prg_break_point:
23154     }
23155     }
23156 \cs_new:Npn \__intarray_range_to_clist:ww #1 ; #2 ; #3
23157     {
23158     \if_int_compare:w #1 > #2 \exp_stop_f:
23159     \prg_break:n
23160     \fi:
23161     , \__kernel_intarray_item:Nn #3 {#1}
23162     \exp_after:wN \__intarray_range_to_clist:ww
23163     \int_value:w \int_eval:w #1 + \c_one_int ; #2 ; #3
23164     }

```

(End of definition for __kernel_intarray_range_to_clist:Nnn and __intarray_range_to_clist:ww.)

__kernel_intarray_gset_range_from_clist:Nnn Loop through part of the array.

```

\__intarray_gset_range:Nw 23165 \cs_new_protected:Npn \__kernel_intarray_gset_range_from_clist:Nnn #1#2#3
23166     {
23167     \int_set:Nn \l__intarray_loop_int {#2}
23168     \__intarray_gset_range:Nw #1 #3 , , \prg_break_point:
23169     }
23170 \cs_new_protected:Npn \__intarray_gset_range:Nw #1 #2 ,
23171     {
23172     \if_catcode:w \scan_stop: \tl_to_str:n {#2} \scan_stop:
23173     \prg_break:n
23174     \fi:
23175     \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
23176     \int_incr:N \l__intarray_loop_int
23177     \__intarray_gset_range:Nw #1
23178     }

```

(End of definition for __kernel_intarray_gset_range_from_clist:Nnn and __intarray_gset_range:Nw.)

```

23179     }

```

66.3 Common parts

`\intarray_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\intarray_if_exist_p:c 23180 \prg_new_eq_conditional:NNn \intarray_if_exist:N \cs_if_exist:N
\intarray_if_exist:NTF 23181 { TF , T , F , p }
\intarray_if_exist:cTF 23182 \prg_new_eq_conditional:NNn \intarray_if_exist:c \cs_if_exist:c
23183 { TF , T , F , p }

```

(End of definition for `\intarray_if_exist:NTF`. This function is documented on page 256.)

`\intarray_show:N` Convert the list to a comma list (with spaces after each comma)

```

\intarray_show:c 23184 \cs_new_protected:Npn \intarray_show:N { \__intarray_show:NN \msg_show:nneeee }
\intarray_log:N 23185 \cs_generate_variant:Nn \intarray_show:N { c }
\intarray_log:c 23186 \cs_new_protected:Npn \intarray_log:N { \__intarray_show:NN \msg_log:nneeee }
23187 \cs_generate_variant:Nn \intarray_log:N { c }
23188 \cs_new_protected:Npn \__intarray_show:NN #1#2
23189 {
23190   \__kernel_chk_defined:NT #2
23191   {
23192     #1 { intarray } { show }
23193     { \token_to_str:N #2 }
23194     { \intarray_count:N #2 }
23195     { >~ \__intarray_to_clist:Nn #2 { , ~ } }
23196     { }
23197   }
23198 }

```

(End of definition for `\intarray_show:N` and `\intarray_log:N`. These functions are documented on page 256.)

```

23199 </tex>
23200 </package>

```


Chapter 67

l3fp implementation

Nothing to see here: everything is in the subfiles!

Chapter 68

l3fp-aux implementation

```
23201 \*package)
```

```
23202 \@@=fp)
```

68.1 Access to primitives

```
\__fp_int_eval:w Largely for performance reasons, we need to directly access primitives rather than use  
\__fp_int_eval_end: \int_eval:n. This happens a lot, so we use private names. The same is true for  
\__fp_int_to_roman:w \romannumeral, although it is used much less widely.
```

```
23203 \cs_new_eq:NN \__fp_int_eval:w \tex_numexpr:D
```

```
23204 \cs_new_eq:NN \__fp_int_eval_end: \scan_stop:
```

```
23205 \cs_new_eq:NN \__fp_int_to_roman:w \tex_romannumeral:D
```

(End of definition for `__fp_int_eval:w`, `__fp_int_eval_end:`, and `__fp_int_to_roman:w`.)

68.2 Internal representation

Internally, a floating point number $\langle X \rangle$ is a token list containing

```
\s__fp \__fp_chk:w \langle case \rangle \langle sign \rangle \langle body \rangle ;
```

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to `f`-expansion. They must leave a recognizable mark after `f`-expansion, to prevent the floating point number from being re-parsed. Thus, `\s__fp` is simply another name for `\relax`.

When used directly without an accessor function, floating points should produce an error: this is the role of `__fp_chk:w`. We could make floating point variables be protected to prevent them from expanding under `e/x`-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their `\langle case \rangle`, which is a single digit:

0 zeros: `+0` and `-0`,

1 “normal” numbers (positive and negative),

Table 3: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s__fp... ;	Positive zero.
0 2 \s__fp... ;	Negative zero.
1 0 {<exponent>} {<X ₁ >} {<X ₂ >} {<X ₃ >} {<X ₄ >} ;	Positive floating point.
1 2 {<exponent>} {<X ₁ >} {<X ₂ >} {<X ₃ >} {<X ₄ >} ;	Negative floating point.
2 0 \s__fp... ;	Positive infinity.
2 2 \s__fp... ;	Negative infinity.
3 1 \s__fp... ;	Quiet nan.
3 1 \s__fp... ;	Signalling nan.

2 infinities: `+inf` and `-inf`,

3 quiet and signalling `nan`.

The `<sign>` is 0 (positive) or 2 (negative), except in the case of `nan`, which have `<sign> = 1`. This ensures that changing the `<sign>` digit to $2 - \langle sign \rangle$ is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

```
\s__fp \__fp_chk:w <case> <sign> \s__fp... ;
```

where `\s__fp...` is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers (`<case> = 1`) have the form

```
\s__fp \__fp_chk:w 1 <sign> {<exponent>} {<X1>} {<X2>} {<X3>} {<X4>} ;
```

Here, the `<exponent>` is an integer, between -10000 and 10000 . The body consists in four blocks of exactly 4 digits, $0000 \leq \langle X_i \rangle \leq 9999$, and the floating point is

$$(-1)^{\langle sign \rangle / 2} \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle \cdot 10^{\langle exponent \rangle - 16}$$

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the `<exponent>` is minimal, in other words, $1000 \leq \langle X_1 \rangle \leq 9999$.

Calculations are done in base 10000, *i.e.* one myriad.

68.3 Using arguments and semicolons

`__fp_use_none_stop_f:n` This function removes an argument (typically a digit) and replaces it by `\exp_stop_f:`, a marker which stops `f`-type expansion.

```
23206 \cs_new:Npn \__fp_use_none_stop_f:n #1 { \exp_stop_f: }
```

(End of definition for `__fp_use_none_stop_f:n`.)

`__fp_use_s:n` Those functions place a semicolon after one or two arguments (typically digits).
`__fp_use_s:nn`

```
23207 \cs_new:Npn \__fp_use_s:n #1 { #1; }
```

```
23208 \cs_new:Npn \__fp_use_s:nn #1#2 { #1#2; }
```

(End of definition for `__fp_use_s:n` and `__fp_use_s:nn`.)

`__fp_use_none_until_s:w` Those functions select specific arguments among a set of arguments delimited by a semicolon.

`__fp_use_i_until_s:nw`

`__fp_use_ii_until_s:nnw`

```

23209 \cs_new:Npn \__fp_use_none_until_s:w #1; { }
23210 \cs_new:Npn \__fp_use_i_until_s:nw #1#2; {#1}
23211 \cs_new:Npn \__fp_use_ii_until_s:nnw #1#2#3; {#2}

```

(End of definition for __fp_use_none_until_s:w, __fp_use_i_until_s:nw, and __fp_use_ii_until_s:nnw.)

`__fp_reverse_args:Nww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

```

23212 \cs_new:Npn \__fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }

```

(End of definition for __fp_reverse_args:Nww.)

`__fp_rrot:www` Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT, hence the name.

```

23213 \cs_new:Npn \__fp_rrot:www #1; #2; #3; { #2; #3; #1; }

```

(End of definition for __fp_rrot:www.)

`__fp_use_i:ww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.

`__fp_use_i:www`

```

23214 \cs_new:Npn \__fp_use_i:ww #1; #2; { #1; }
23215 \cs_new:Npn \__fp_use_i:www #1; #2; #3; { #1; }

```

(End of definition for __fp_use_i:ww and __fp_use_i:www.)

68.4 Constants, and structure of floating points

`__fp_misused:n` This receives a floating point object (floating point number or tuple) and generates an error stating that it was misused. This is called when for instance an fp variable is left in the input stream and its contents reach T_EX's stomach.

```

23216 \cs_new_protected:Npn \__fp_misused:n #1
23217   { \msg_error:nne { fp } { misused } { \fp_to_tl:n {#1} } }

```

(End of definition for __fp_misused:n.)

`\s__fp` Floating points numbers all start with `\s__fp` `__fp_chk:w`, where `\s__fp` is equal to the T_EX primitive `\relax`, and `__fp_chk:w` is protected. The rest of the floating point number is made of characters (or `\relax`). This ensures that nothing expands under f-expansion, nor under e/x-expansion. However, when typeset, `\s__fp` does nothing, and `__fp_chk:w` is expanded. We define `__fp_chk:w` to produce an error.

`__fp_chk:w`

```

23218 \scan_new:N \s__fp
23219 \cs_new_protected:Npn \__fp_chk:w #1 ;
23220   { \__fp_misused:n { \s__fp \__fp_chk:w #1 ; } }

```

(End of definition for \s__fp and __fp_chk:w.)

`\s__fp_expr_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

`\s__fp_expr_stop`

```

23221 \scan_new:N \s__fp_expr_mark
23222 \scan_new:N \s__fp_expr_stop

```

(End of definition for \s__fp_expr_mark and \s__fp_expr_stop.)

`\s__fp_mark` Generic scan marks used throughout the module.

`\s__fp_stop` 23223 `\scan_new:N \s__fp_mark`
23224 `\scan_new:N \s__fp_stop`

(End of definition for `\s__fp_mark` and `\s__fp_stop`.)

`_fp_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

23225 `\cs_new:Npn _fp_use_i_delimit_by_s_stop:nw #1 #2 \s__fp_stop {#1}`

(End of definition for `_fp_use_i_delimit_by_s_stop:nw`.)

`\s__fp_invalid` A couple of scan marks used to indicate where special floating point numbers come from.

`\s__fp_underflow` 23226 `\scan_new:N \s__fp_invalid`
`\s__fp_overflow` 23227 `\scan_new:N \s__fp_underflow`
`\s__fp_division` 23228 `\scan_new:N \s__fp_overflow`
`\s__fp_exact` 23229 `\scan_new:N \s__fp_division`
23230 `\scan_new:N \s__fp_exact`

(End of definition for `\s__fp_invalid` and others.)

`\c_zero_fp` The special floating points. We define the floating points here as “exact”.

`\c_minus_zero_fp` 23231 `\tl_const:Nn \c_zero_fp { \s__fp _fp_chk:w 0 0 \s__fp_exact ; }`
`\c_inf_fp` 23232 `\tl_const:Nn \c_minus_zero_fp { \s__fp _fp_chk:w 0 2 \s__fp_exact ; }`
`\c_minus_inf_fp` 23233 `\tl_const:Nn \c_inf_fp { \s__fp _fp_chk:w 2 0 \s__fp_exact ; }`
`\c_nan_fp` 23234 `\tl_const:Nn \c_minus_inf_fp { \s__fp _fp_chk:w 2 2 \s__fp_exact ; }`
23235 `\tl_const:Nn \c_nan_fp { \s__fp _fp_chk:w 3 1 \s__fp_exact ; }`

(End of definition for `\c_zero_fp` and others. These variables are documented on page 269.)

`\c__fp_prec_int` The number of digits of floating points.

`\c__fp_half_prec_int` 23236 `\int_const:Nn \c__fp_prec_int { 16 }`
`\c__fp_block_int` 23237 `\int_const:Nn \c__fp_half_prec_int { 8 }`
23238 `\int_const:Nn \c__fp_block_int { 4 }`

(End of definition for `\c__fp_prec_int`, `\c__fp_half_prec_int`, and `\c__fp_block_int`.)

`\c__fp_myriad_int` Blocks have 4 digits so this integer is useful.

23239 `\int_const:Nn \c__fp_myriad_int { 10000 }`

(End of definition for `\c__fp_myriad_int`.)

`\c__fp_minus_min_exponent_int` Normal floating point numbers have an exponent between `-minus_min_exponent` and
`\c__fp_max_exponent_int` `max_exponent` inclusive. Larger numbers are rounded to $\pm\infty$. Smaller numbers are rounded to ± 0 . It would be more natural to define a `min_exponent` with the opposite sign but that would waste one TeX count.

23240 `\int_const:Nn \c__fp_minus_min_exponent_int { 10000 }`
23241 `\int_const:Nn \c__fp_max_exponent_int { 10000 }`

(End of definition for `\c__fp_minus_min_exponent_int` and `\c__fp_max_exponent_int`.)

`\c__fp_max_exp_exponent_int` If a number’s exponent is larger than that, its exponential overflows/underflows.

23242 `\int_const:Nn \c__fp_max_exp_exponent_int { 5 }`

(End of definition for `\c__fp_max_exp_exponent_int`.)

`\c__fp_overflowing_fp` A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.

```

23243 \tl_const:Ne \c__fp_overflowing_fp
23244 {
23245   \s__fp \__fp_chk:w 1 0
23246   { \int_eval:n { \c__fp_max_exponent_int + 1 } }
23247   {1000} {0000} {0000} {0000} ;
23248 }

```

(End of definition for \c__fp_overflowing_fp.)

`__fp_zero_fp:N` In case of overflow or underflow, we have to output a zero or infinity with a given sign.

```

\__fp_inf_fp:N
23249 \cs_new:Npn \__fp_zero_fp:N #1
23250 { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
23251 \cs_new:Npn \__fp_inf_fp:N #1
23252 { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }

```

(End of definition for __fp_zero_fp:N and __fp_inf_fp:N.)

`__fp_exponent:w` For normal numbers, the function expands to the exponent, otherwise to 0. This is used in `l3str-format`.

```

23253 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
23254 {
23255   \if_meaning:w 1 #1
23256   \exp_after:wN \__fp_use_ii_until_s:nnw
23257   \else:
23258   \exp_after:wN \__fp_use_i_until_s:nw
23259   \exp_after:wN 0
23260   \fi:
23261 }

```

(End of definition for __fp_exponent:w.)

`__fp_neg_sign:N` When appearing in an integer expression or after `\int_value:w`, this expands to the sign opposite to #1, namely 0 (positive) is turned to 2 (negative), 1 (`nan`) to 1, and 2 to 0.

```

23262 \cs_new:Npn \__fp_neg_sign:N #1
23263 { \__fp_int_eval:w 2 - #1 \__fp_int_eval_end: }

```

(End of definition for __fp_neg_sign:N.)

`__fp_kind:w` Expands to 0 for zeros, 1 for normal floating point numbers, 2 for infinities, 3 for `nan`, 4 for tuples.

```

23264 \cs_new:Npn \__fp_kind:w #1
23265 {
23266   \__fp_if_type_fp:NTwFw
23267   #1 \__fp_use_ii_until_s:nnw
23268   \s__fp { \__fp_use_i_until_s:nw 4 }
23269   \s__fp_stop
23270 }

```

(End of definition for __fp_kind:w.)

68.5 Overflow, underflow, and exact zero

`__fp_sanitize:Nw` Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to ± 0 or overflowed to $\pm\infty$. The functions `__fp_underflow:w` and `__fp_overflow:w` are defined in `l3fp-traps`.

```

23271 \cs_new:Npn \__fp_sanitize:Nw #1 #2;
23272   {
23273     \if_case:w
23274       \if_int_compare:w #2 > \c__fp_max_exponent_int 1 ~ \else:
23275       \if_int_compare:w #2 < - \c__fp_minus_min_exponent_int 2 ~ \else:
23276       \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
23277       \or: \exp_after:wN \__fp_overflow:w
23278       \or: \exp_after:wN \__fp_underflow:w
23279       \or: \exp_after:wN \__fp_sanitize_zero:w
23280       \fi:
23281       \s__fp \__fp_chk:w 1 #1 {#2}
23282   }
23283 \cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
23284 \cs_new:Npn \__fp_sanitize_zero:w \s__fp \__fp_chk:w #1 #2 #3;
23285   { \c_zero_fp }

```

(End of definition for `__fp_sanitize:Nw`, `__fp_sanitize:wN`, and `__fp_sanitize_zero:w`.)

68.6 Expanding after a floating point number

`__fp_exp_after_o:w`
`__fp_exp_after_f:nw`

`__fp_exp_after_o:w` *<floating point>*
`__fp_exp_after_f:nw` *{<tokens>}* *<floating point>*

Places *<tokens>* (empty in the case of `__fp_exp_after_o:w`) between the *<floating point>* and the following tokens, then hits those tokens with `o` or `f`-expansion, and leaves the floating point number unchanged.

We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

```

23286 \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
23287   {
23288     \if_meaning:w 1 #1
23289     \exp_after:wN \__fp_exp_after_normal:nNNw
23290   \else:
23291     \exp_after:wN \__fp_exp_after_special:nNNw
23292   \fi:
23293   { }
23294   #1
23295   }
23296 \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
23297   {
23298     \if_meaning:w 1 #2
23299     \exp_after:wN \__fp_exp_after_normal:nNNw
23300   \else:
23301     \exp_after:wN \__fp_exp_after_special:nNNw
23302   \fi:
23303   { \exp:w \exp_end_continue_f:w #1 }
23304   #2

```

```
23305 }
```

(End of definition for `_fp_exp_after_o:w` and `_fp_exp_after_f:nw`.)

```
\_fp_exp_after_special:nNNw \_fp_exp_after_special:nNNw {<after>} <case> <sign> <scan mark> ;  
Special floating point numbers are easy to jump over since they contain few tokens.
```

```
23306 \cs_new:Npn \_fp_exp_after_special:nNNw #1#2#3#4;  
23307 {  
23308   \exp_after:wN \s__fp  
23309   \exp_after:wN \_fp_chk:w  
23310   \exp_after:wN #2  
23311   \exp_after:wN #3  
23312   \exp_after:wN #4  
23313   \exp_after:wN ;  
23314   #1  
23315 }
```

(End of definition for `_fp_exp_after_special:nNNw`.)

```
\_fp_exp_after_normal:nNNw For normal floating point numbers, life is slightly harder, since we have many tokens to  
jump over. Here it would be slightly better if the digits were not braced but instead were  
delimited arguments (for instance delimited by ,). That may be changed some day.
```

```
23316 \cs_new:Npn \_fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;  
23317 {  
23318   \exp_after:wN \_fp_exp_after_normal:Nwwwww  
23319   \exp_after:wN #2  
23320   \int_value:w #3 \exp_after:wN ;  
23321   \int_value:w 1 #4 \exp_after:wN ;  
23322   \int_value:w 1 #5 \exp_after:wN ;  
23323   \int_value:w 1 #6 \exp_after:wN ;  
23324   \int_value:w 1 #7 \exp_after:wN ; #1  
23325 }  
23326 \cs_new:Npn \_fp_exp_after_normal:Nwwwww  
23327   #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;  
23328   { \s__fp \_fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }
```

(End of definition for `_fp_exp_after_normal:nNNw`.)

68.7 Other floating point types

```
\s__fp_tuple Floating point tuples take the form \s__fp_tuple \_fp_tuple_chk:w { <fp 1> <fp 2>  
\_fp_tuple_chk:w ... } ; where each <fp> is a floating point number or tuple, hence ends with ; itself.  
\c__fp_empty_tuple_fp When a tuple is typeset, \_fp_tuple_chk:w produces an error, just like usual floating  
point numbers. Tuples may have zero or one element.
```

```
23329 \scan_new:N \s__fp_tuple  
23330 \cs_new_protected:Npn \_fp_tuple_chk:w #1 ;  
23331 { \_fp_misused:n { \s__fp_tuple \_fp_tuple_chk:w #1 ; } }  
23332 \tl_const:Nn \c__fp_empty_tuple_fp  
23333 { \s__fp_tuple \_fp_tuple_chk:w { } ; }
```

(End of definition for `\s__fp_tuple`, `_fp_tuple_chk:w`, and `\c__fp_empty_tuple_fp`.)

`__fp_tuple_count:w` Count the number of items in a tuple of floating points by counting semicolons. The technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the end of the loop is done with the `\use_none:n #1` construction.

```

23334 \cs_new:Npn \__fp_array_count:n #1
23335   { \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w {#1} ; }
23336 \cs_new:Npn \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w #1 ;
23337   {
23338     \int_value:w \__fp_int_eval:w 0
23339     \__fp_tuple_count_loop:Nw #1 { ? \prg_break: } ;
23340     \prg_break_point:
23341     \__fp_int_eval_end:
23342   }
23343 \cs_new:Npn \__fp_tuple_count_loop:Nw #1#2;
23344   { \use_none:n #1 + 1 \__fp_tuple_count_loop:Nw }

```

(End of definition for `__fp_tuple_count:w`, `__fp_array_count:n`, and `__fp_tuple_count_loop:Nw`.)

`__fp_if_type_fp:NTwFw` Used as `__fp_if_type_fp:NTwFw <marker> {<true code>} \s__fp {<false code>} \s__fp_stop`, this test whether the `<marker>` is `\s__fp` or not and runs the appropriate `<code>`. The very unusual syntax is for optimization purposes as that function is used for all floating point operations.

```

23345 \cs_new:Npn \__fp_if_type_fp:NTwFw #1 \s__fp #2 #3 \s__fp_stop {#2}

```

(End of definition for `__fp_if_type_fp:NTwFw`.)

`__fp_array_if_all_fp:nTF` True if all items are floating point numbers. Used for `min`.
`__fp_array_if_all_fp_loop:w`

```

23346 \cs_new:Npn \__fp_array_if_all_fp:nTF #1
23347   {
23348     \__fp_array_if_all_fp_loop:w #1 { \s__fp \prg_break: } ;
23349     \prg_break_point: \use_i:nn
23350   }
23351 \cs_new:Npn \__fp_array_if_all_fp_loop:w #1#2 ;
23352   {
23353     \__fp_if_type_fp:NTwFw
23354     #1 \__fp_array_if_all_fp_loop:w
23355     \s__fp { \prg_break:n \use_iii:nnn }
23356     \s__fp_stop
23357   }

```

(End of definition for `__fp_array_if_all_fp:nTF` and `__fp_array_if_all_fp_loop:w`.)

`__fp_type_from_scan:N` Used as `__fp_type_from_scan:N <token>`. Grabs the pieces of the stringified `<token>` which lies after the first `s__fp`. If the `<token>` does not contain that string, the result is `_?`.
`__fp_type_from_scan_other:N`
`__fp_type_from_scan:w`

```

23358 \cs_new:Npn \__fp_type_from_scan:N #1
23359   {
23360     \__fp_if_type_fp:NTwFw
23361     #1 { }
23362     \s__fp { \__fp_type_from_scan_other:N #1 }
23363     \s__fp_stop
23364   }
23365 \cs_new:Npe \__fp_type_from_scan_other:N #1
23366   {
23367     \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w

```

```

23368 \exp_not:N \token_to_str:N #1 \s__fp_mark
23369 \tl_to_str:n { s__fp ? } \s__fp_mark \s__fp_stop
23370 }
23371 \exp_last_unbraced:NNNNo
23372 \cs_new:Npn \__fp_type_from_scan:w #1
23373 { \tl_to_str:n { s__fp } } #2 \s__fp_mark #3 \s__fp_stop {#2}

```

(End of definition for `__fp_type_from_scan:N`, `__fp_type_from_scan_other:N`, and `__fp_type_from_scan:w`.)

`__fp_change_func_type:NNN` Arguments are `<type marker>` `<function>` `<recovery>`. This gives the function obtained by placing the type after `@@`. If the function is not defined then `<recovery>` `<function>` is used instead; however that test is not run when the `<type marker>` is `\s__fp`.

`__fp_change_func_type_aux:w`
`__fp_change_func_type_chk:NNN`

```

23374 \cs_new:Npn \__fp_change_func_type:NNN #1#2#3
23375 {
23376   \__fp_if_type_fp:NTwFw
23377   #1 #2
23378   \s__fp
23379   {
23380     \exp_after:wN \__fp_change_func_type_chk:NNN
23381     \cs:w
23382     __fp \__fp_type_from_scan_other:N #1
23383     \exp_after:wN \__fp_change_func_type_aux:w \token_to_str:N #2
23384     \cs_end:
23385     #2 #3
23386   }
23387   \s__fp_stop
23388 }
23389 \exp_last_unbraced:NNNNo
23390 \cs_new:Npn \__fp_change_func_type_aux:w #1 { \tl_to_str:n { __fp } } { }
23391 \cs_new:Npn \__fp_change_func_type_chk:NNN #1#2#3
23392 {
23393   \if_meaning:w \scan_stop: #1
23394   \exp_after:wN #3 \exp_after:wN #2
23395   \else:
23396   \exp_after:wN #1
23397   \fi:
23398 }

```

(End of definition for `__fp_change_func_type:NNN`, `__fp_change_func_type_aux:w`, and `__fp_change_func_type_chk:NNN`.)

`__fp_exp_after_any_f:Nnw`
`__fp_exp_after_any_f:nw`
`__fp_exp_after_expr_stop_f:nw`

The `Nnw` function simply dispatches to the appropriate `__fp_exp_after..._f:nw` with “...” (either empty or `<type>`) extracted from `#1`, which should start with `\s__fp`. If it doesn't start with `\s__fp` the function `__fp_exp_after_?_f:nw` defined in `l3fp-parse` gives an error; another special `<type>` is `stop`, useful for loops, see below. The `nw` function has an important optimization for floating points numbers; it also fetches its type marker `#2` from the floating point.

```

23399 \cs_new:Npn \__fp_exp_after_any_f:Nnw #1
23400 { \cs:w __fp_exp_after \__fp_type_from_scan_other:N #1 _f:nw \cs_end: }
23401 \cs_new:Npn \__fp_exp_after_any_f:nw #1#2
23402 {
23403   \__fp_if_type_fp:NTwFw
23404   #2 \__fp_exp_after_f:nw

```

```

23405     \s__fp { \__fp_exp_after_any_f:Nnw #2 }
23406     \s__fp_stop
23407     {#1} #2
23408   }
23409 \cs_new_eq:NN \__fp_exp_after_expr_stop_f:nw \use_none:nn

```

(End of definition for `__fp_exp_after_any_f:Nnw`, `__fp_exp_after_any_f:nw`, and `__fp_exp_after_expr_stop_f:nw`.)

`__fp_exp_after_tuple_o:w` The loop works by using the `n` argument of `__fp_exp_after_any_f:nw` to place the
`__fp_exp_after_tuple_f:nw` loop macro after the next item in the tuple and expand it.
`__fp_exp_after_array_f:w`

```

     \__fp_exp_after_array_f:w
     <fp1> ;
     ...
     <fpn> ;
     \s__fp_expr_stop

23410 \cs_new:Npn \__fp_exp_after_tuple_o:w
23411   { \__fp_exp_after_tuple_f:nw { \exp_after:wN \exp_stop_f: } }
23412 \cs_new:Npn \__fp_exp_after_tuple_f:nw
23413   #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
23414   {
23415     \exp_after:wN \s__fp_tuple
23416     \exp_after:wN \__fp_tuple_chk:w
23417     \exp_after:wN {
23418       \exp:w \exp_end_continue_f:w
23419       \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
23420     \exp_after:wN }
23421     \exp_after:wN ;
23422     \exp:w \exp_end_continue_f:w #1
23423   }
23424 \cs_new:Npn \__fp_exp_after_array_f:w
23425   { \__fp_exp_after_any_f:nw { \__fp_exp_after_array_f:w } }

```

(End of definition for `__fp_exp_after_tuple_o:w`, `__fp_exp_after_tuple_f:nw`, and `__fp_exp_after_array_f:w`.)

68.8 Packing digits

When a positive integer `#1` is known to be less than 10^8 , the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

```

\cs_new:Npn \pack:NNNNw #1 #2#3#4#5 #6; { {#2#3#4#5} {#6} }
\exp_after:wN \pack:NNNNw
  \__fp_int_value:w \__fp_int_eval:w 1 0000 0000 + #1 ;

```

The idea is that adding 10^8 to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by `TEX`'s integers). This method is very heavily relied upon in `l3fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 ; {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute 12345×66778899 . With simplified names, we would do

```

\exp_after:wN \post_processing:w
\__fp_int_value:w \__fp_int_eval:w - 5 0000
  \exp_after:wN \pack:NNNNNw
  \__fp_int_value:w \__fp_int_eval:w 4 9995 0000
    + 12345 * 6677
  \exp_after:wN \pack:NNNNNw
  \__fp_int_value:w \__fp_int_eval:w 5 0000 0000
    + 12345 * 8899 ;

```

The `\exp_after:wN` triggers `\int_value:w __fp_int_eval:w`, which starts a first computation, whose initial value is $-5\,0000$ (the “leading shift”). In that computation appears an `\exp_after:wN`, which triggers the nested computation `\int_value:w __fp_int_eval:w` with starting value $4\,9995\,0000$ (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation’s value is $5\,0000\,0000 + 12345 \times 8899$, which has 9 digits. Adding $5 \cdot 10^8$ to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The `pack` function puts the last 4 of those 9 digits into a brace group, moves the semi-colon delimiter, and inserts a `+`, which combines the carry with the previous computation. The shifts nicely combine into $5\,0000\,0000/10^4 + 4\,9995\,0000 = 5\,0000\,0000$. As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding `pack` function, expanded after the result is computed, braces the last 4 digits, and leaves `+ <5 digits>` for the initial computation. The “leading shift” cancels the combination of the other shifts, and the `\post_processing:w` takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making `l3fp` as fast as other pure `TeX` floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

```

\__fp_pack:NNNNNw This set of shifts allows for computations involving results in the range  $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$ .
\c_fp_trailing_shift_int Shifted values all have exactly 9 digits.
\c_fp_middle_shift_int
\c_fp_leading_shift_int
23426 \int_const:Nn \c_fp_leading_shift_int { - 5 0000 }
23427 \int_const:Nn \c_fp_middle_shift_int { 5 0000 * 9999 }
23428 \int_const:Nn \c_fp_trailing_shift_int { 5 0000 * 10000 }
23429 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }

```

(End of definition for `__fp_pack:NNNNNw` and others.)

```

\__fp_pack_big:NNNNNNw This set of shifts allows for computations involving results in the range  $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$ 
\c_fp_big_trailing_shift_int (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper
\c_fp_big_middle_shift_int bound is due to TeX’s limit of  $2^{31} - 1$  on integers. The shifts are chosen to be roughly
\c_fp_big_leading_shift_int the mid-point of  $10^9$  and  $2^{31}$ , the two bounds on 10-digit integers in TeX.
23430 \int_const:Nn \c_fp_big_leading_shift_int { - 15 2374 }
23431 \int_const:Nn \c_fp_big_middle_shift_int { 15 2374 * 9999 }
23432 \int_const:Nn \c_fp_big_trailing_shift_int { 15 2374 * 10000 }
23433 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
23434 { + #1#2#3#4#5#6 ; {#7} }

```

(End of definition for `__fp_pack_big:NNNNNNw` and others.)

`_fp_pack_Bigg:NNNNNNw` This set of shifts allows for computations with results in the range $[-1 \cdot 10^9, 147483647]$;
`\c_fp_Bigg_trailing_shift_int` the end-point is $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$. Shifted values all have exactly 10 digits.
`\c_fp_Bigg_middle_shift_int` 23435 `\int_const:Nn \c_fp_Bigg_leading_shift_int { - 20 0000 }`
`\c_fp_Bigg_leading_shift_int` 23436 `\int_const:Nn \c_fp_Bigg_middle_shift_int { 20 0000 * 9999 }`
23437 `\int_const:Nn \c_fp_Bigg_trailing_shift_int { 20 0000 * 10000 }`
23438 `\cs_new:Npn _fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;`
23439 `{ + #1#2#3#4#5#6 ; {#7} }`

(End of definition for _fp_pack_Bigg:NNNNNNw and others.)

`_fp_pack_twice_four:wNNNNNNNN` `_fp_pack_twice_four:wNNNNNNNN <tokens> ; <≥ 8 digits>`
Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.
23440 `\cs_new:Npn _fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9`
23441 `{ #1 {#2#3#4#5} {#6#7#8#9} ; }`

(End of definition for _fp_pack_twice_four:wNNNNNNNN.)

`_fp_pack_eight:wNNNNNNNN` `_fp_pack_eight:wNNNNNNNN <tokens> ; <≥ 8 digits>`
Grabs one set of 8 digits and places them before the semi-colon delimiter as a single group. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.
23442 `\cs_new:Npn _fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9`
23443 `{ #1 {#2#3#4#5#6#7#8#9} ; }`

(End of definition for _fp_pack_eight:wNNNNNNNN.)

`_fp_basics_pack_low:NNNNNNw` Addition and multiplication of significands are done in two steps: first compute a (more or
`_fp_basics_pack_high:NNNNNNw` less) exact result, then round and pack digits in the final (braced) form. These functions
`_fp_basics_pack_high_carry:w` take care of the packing, with special attention given to the case where rounding has caused a carry. Since rounding can only shift the final digit by 1, a carry always produces an exact power of 10. Thus, `_fp_basics_pack_high_carry:w` is always followed by four times `{0000}`.

This is used in l3fp-basics and l3fp-extended.

23444 `\cs_new:Npn _fp_basics_pack_low:NNNNNNw #1 #2#3#4#5 #6;`
23445 `{ + #1 - 1 ; {#2#3#4#5} {#6} ; }`
23446 `\cs_new:Npn _fp_basics_pack_high:NNNNNNw #1 #2#3#4#5 #6;`
23447 `{`
23448 `\if_meaning:w 2 #1`
23449 `_fp_basics_pack_high_carry:w`
23450 `\fi:`
23451 `; {#2#3#4#5} {#6}`
23452 `}`
23453 `\cs_new:Npn _fp_basics_pack_high_carry:w \fi: ; #1`
23454 `{ \fi: + 1 ; {1000} }`

(End of definition for _fp_basics_pack_low:NNNNNNw, _fp_basics_pack_high:NNNNNNw, and _fp_basics_pack_high_carry:w.)

`_fp_basics_pack_weird_low:NNNNw`
`_fp_basics_pack_weird_high:NNNNNNNNw`

This is used in `l3fp-basics` for additions and divisions. Their syntax is confusing, hence the name.

```

23455 \cs_new:Npn \_fp\_basics\_pack\_weird\_low:NNNNw #1 #2#3#4 #5;
23456 {
23457   \if_meaning:w 2 #1
23458     + 1
23459   \fi:
23460   \_fp\_int\_eval\_end:
23461   #2#3#4; {#5} ;
23462 }
23463 \cs_new:Npn \_fp\_basics\_pack\_weird\_high:NNNNNNNNw
23464 1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }

```

(End of definition for `_fp_basics_pack_weird_low:NNNNw` and `_fp_basics_pack_weird_high:NNNNNNNNw`.)

68.9 Decimate (dividing by a power of 10)

`_fp_decimate:nNnnnn`

`_fp_decimate:nNnnnn` $\langle shift \rangle$ $\langle f_1 \rangle$
 $\langle X_1 \rangle$ $\langle X_2 \rangle$ $\langle X_3 \rangle$ $\langle X_4 \rangle$

Each $\langle X_i \rangle$ consists in 4 digits exactly, and $1000 \leq \langle X_1 \rangle < 9999$. The first argument determines by how much we shift the digits. $\langle f_1 \rangle$ is called as follows:

$\langle f_1 \rangle$ $\langle rounding \rangle$ $\langle X'_1 \rangle$ $\langle X'_2 \rangle$ $\langle extra-digits \rangle$;

where $0 \leq \langle X'_i \rangle < 10^8 - 1$ are 8 digit integers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^4 \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle shift \rangle} \right) - (\langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16}) = 0 \cdot \langle extra-digits \rangle \cdot 10^{-16} \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The $\langle rounding \rangle$ digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly $0.5 \cdot 10^{-16}$. Otherwise, it is the (non-0, non-5) digit closest to 10^{17} times the difference. In particular, if the shift is 17 or more, all the digits are dropped, $\langle rounding \rangle$ is 1 (not 0), and $\langle X'_1 \rangle$ and $\langle X'_2 \rangle$ are both zero.

If the shift is 1, the $\langle rounding \rangle$ digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the $\langle rounding \rangle$ digit to be placed after the $\langle X'_i \rangle$, but the choice we make involves less reshuffling.

Note that this function treats negative $\langle shift \rangle$ as 0.

```

23465 \cs_new:Npn \_fp\_decimate:nNnnnn #1
23466 {
23467   \cs:w
23468     \_fp\_decimate\_
23469     \if_int_compare:w \_fp\_int\_eval:w #1 > \c\_fp\_prec\_int
23470       tiny
23471     \else:
23472       \_fp\_int\_to\_roman:w \_fp\_int\_eval:w #1
23473     \fi:
23474     :Nnnnn
23475   \cs_end:
23476 }

```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(End of definition for `_fp_decimate:nNnnnn`.)

If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```
\_fp\_decimate_:Nnnnn
\_fp\_decimate\_tiny:Nnnnn
23477 \cs\_new:Npn \_fp\_decimate_:Nnnnn #1 #2#3#4#5
23478 { #1 0 {#2#3} {#4#5} ; }
23479 \cs\_new:Npn \_fp\_decimate\_tiny:Nnnnn #1 #2#3#4#5
23480 { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }
```

(End of definition for `_fp_decimate_:Nnnnn` and `_fp_decimate_tiny:Nnnnn`.)

```
\_fp\_decimate\_auxi:Nnnnn
\_fp\_decimate\_auxii:Nnnnn
\_fp\_decimate\_auxiii:Nnnnn
\_fp\_decimate\_auxiv:Nnnnn
\_fp\_decimate\_auxv:Nnnnn
\_fp\_decimate\_auxvi:Nnnnn
\_fp\_decimate\_auxvii:Nnnnn
\_fp\_decimate\_auxviii:Nnnnn
\_fp\_decimate\_auxix:Nnnnn
\_fp\_decimate\_auxx:Nnnnn
\_fp\_decimate\_auxxi:Nnnnn
\_fp\_decimate\_auxxii:Nnnnn
\_fp\_decimate\_auxxiii:Nnnnn
\_fp\_decimate\_auxxiv:Nnnnn
\_fp\_decimate\_auxxv:Nnnnn
\_fp\_decimate\_auxxvi:Nnnnn
\_fp\_decimate\_auxi:Nnnnn \_fp\_decimate\_auxi:Nnnnn  $\langle f_1 \rangle$  { $\langle X_1 \rangle$ } { $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ }
Shifting happens in two steps: compute the  $\langle rounding \rangle$  digit, and repack digits into
two blocks of 8. The sixteen functions are very similar, and defined through \_fp\_tmp:w.
The arguments are as follows: #1 indicates which function is being defined; after one step
of expansion, #2 yields the “extra digits” which are then converted by \_fp\_round\_digit:Nw
to the  $\langle rounding \rangle$  digit (note the + separating blocks of digits to avoid overflowing
TeX’s integers). This triggers the f-expansion of \_fp\_decimate\_pack:nnnnnnnnnw,10
responsible for building two blocks of 8 digits, and removing the rest. For this to work,
#3 alternates between braced and unbraced blocks of 4 digits, in such a way that the 5
first and 5 next token groups yield the correct blocks of 8 digits.
23481 \cs\_new:Npn \_fp\_tmp:w #1 #2 #3
23482 {
23483 \cs\_new:cpn { \_fp\_decimate\_ #1 :Nnnnn } ##1 ##2##3##4##5
23484 {
23485 \exp\_after:wN ##1
23486 \int\_value:w
23487 \exp\_after:wN \_fp\_round\_digit:Nw #2 ;
23488 \_fp\_decimate\_pack:nnnnnnnnnw #3 ;
23489 }
23490 }
23491 \_fp\_tmp:w {i} {\use\_none:nnn #50}{ 0{#2}#3{#4}#5 }
23492 \_fp\_tmp:w {ii} {\use\_none:nn #5 }{ 00{#2}#3{#4}#5 }
23493 \_fp\_tmp:w {iii} {\use\_none:n #5 }{ 000{#2}#3{#4}#5 }
23494 \_fp\_tmp:w {iv} { #5 }{ {0000}#2{#3}#4 #5 }
23495 \_fp\_tmp:w {v} {\use\_none:nnn #4#5 }{ 0{0000}#2{#3}#4 #5 }
23496 \_fp\_tmp:w {vi} {\use\_none:nn #4#5 }{ 00{0000}#2{#3}#4 #5 }
23497 \_fp\_tmp:w {vii} {\use\_none:n #4#5 }{ 000{0000}#2{#3}#4 #5 }
23498 \_fp\_tmp:w {viii}{ #4#5 }{ {0000}0000{#2}#3 #4 #5 }
23499 \_fp\_tmp:w {ix} {\use\_none:nnn #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5 }
23500 \_fp\_tmp:w {x} {\use\_none:nn #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5 }
23501 \_fp\_tmp:w {xi} {\use\_none:n #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5 }
23502 \_fp\_tmp:w {xii} { #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5 }
23503 \_fp\_tmp:w {xiii}{\use\_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5 }
23504 \_fp\_tmp:w {xiv} {\use\_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5 }
23505 \_fp\_tmp:w {xv} {\use\_none:n #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5 }
23506 \_fp\_tmp:w {xvi} { #2#3+#4#5}{ {0000}0000{0000}0000 #2 #3 #4 #5 }
```

(End of definition for `_fp_decimate_auxi:Nnnnn` and others.)

¹⁰No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

`_fp_decimate_pack:nnnnnnnnnw`

The computation of the `<rounding>` digit leaves an unfinished `\int_value:w`, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```
23507 \cs_new:Npn \_fp\_decimate\_pack:nnnnnnnnnw #1#2#3#4#5
23508   { \_fp\_decimate\_pack:nnnnnw { #1#2#3#4#5 } }
23509 \cs_new:Npn \_fp\_decimate\_pack:nnnnnw #1 #2#3#4#5#6
23510   { {#1} {#2#3#4#5#6} }
```

(End of definition for `_fp_decimate_pack:nnnnnnnnnw`.)

68.10 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one `\fi:` as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in `l3fp` must perform tests on the type of floating points that they receive. This is often done in an `\if_case:w` statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

```
\if\_case:w <integer> \exp\_stop\_f:
  \_fp\_case\_return\_o:Nw <fp var>
\or: \_fp\_case\_use:nw {<some computation>}
\or: \_fp\_case\_return\_same\_o:w
\or: \_fp\_case\_return:nw {<something>}
\fi:
<junk>
<floating point>
```

In this example, the case 0 returns the floating point `<fp var>`, expanding once after that floating point. Case 1 does `<some computation>` using the `<floating point>` (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the `<floating point>` without modifying it, removing the `<junk>` and expanding once after. Case 3 closes the conditional, removes the `<junk>` and the `<floating point>`, and expands `<something>` next. In other cases, the “`<junk>`” is expanded, performing some other operation on the `<floating point>`. We provide similar functions with two trailing `<floating points>`.

`_fp_case_use:nw`

This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```
23511 \cs_new:Npn \_fp\_case\_use:nw #1#2 \fi: #3 \s\_fp { \fi: #1 \s\_fp }
```

(End of definition for `_fp_case_use:nw`.)

`__fp_case_return:nw` This function ends a TeX conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don't define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the *<junk>* may not contain semicolons.

```
23512 \cs_new:Npn \__fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }
```

(End of definition for __fp_case_return:nw.)

`__fp_case_return_o:Nw` This function ends a TeX conditional, removes junk and a floating point, and returns its first argument (an *<fp var>*) then expands once after it.

```
23513 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
23514 { \fi: \exp_after:wN #1 }
```

(End of definition for __fp_case_return_o:Nw.)

`__fp_case_return_same_o:w` This function ends a TeX conditional, removes junk, and returns the following floating point, expanding once after it.

```
23515 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
23516 { \fi: \__fp_exp_after_o:w \s__fp }
```

(End of definition for __fp_case_return_same_o:w.)

`__fp_case_return_o:Nww` Same as `__fp_case_return_o:Nw` but with two trailing floating points.

```
23517 \cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
23518 { \fi: \exp_after:wN #1 }
```

(End of definition for __fp_case_return_o:Nww.)

`__fp_case_return_i_o:ww` Similar to `__fp_case_return_same_o:w`, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

`__fp_case_return_ii_o:ww`

```
23519 \cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
23520 { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
23521 \cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
23522 { \fi: \__fp_exp_after_o:w }
```

(End of definition for __fp_case_return_i_o:ww and __fp_case_return_ii_o:ww.)

68.11 Integer floating points

`__fp_int_p:w` Tests if the floating point argument is an integer. For normal floating point numbers, `__fp_int:wTF` this holds if the rounding digit resulting from `__fp_decimate:nNnnnn` is 0.

```
23523 \prg_new_conditional:Npnn \__fp_int:w \s__fp \__fp_chk:w #1 #2 #3 #4;
23524 { TF , T , F , p }
23525 {
23526   \if_case:w #1 \exp_stop_f:
23527     \prg_return_true:
23528   \or:
23529     \if_charcode:w 0
23530     \__fp_decimate:nNnnnn { \c__fp_prec_int - #3 }
23531     \__fp_use_i_until_s:nw #4
23532     \prg_return_true:
23533   \else:
23534     \prg_return_false:
```

```

23535     \fi:
23536     \else: \prg_return_false:
23537     \fi:
23538 }

```

(End of definition for `__fp_int:wTF`.)

68.12 Small integer floating points

Tests if the floating point argument is an integer or $\pm\infty$. If so, it is clipped to an integer in the range $[-10^8, 10^8]$ and fed as a braced argument to the *<true code>*. Otherwise, the *<false code>* is performed.

First filter special cases: zeros and infinities are integers, nan is not. For normal numbers, decimate. If the rounding digit is not 0 run the *<false code>*. If it is, then the integer is #2 #3; use #3 if #2 vanishes and otherwise 10^8 .

```

23539 \cs_new:Npn \__fp_small_int:wTF \s_fp \__fp_chk:w #1#2
23540 {
23541   \if_case:w #1 \exp_stop_f:
23542     \__fp_case_return:nw { \__fp_small_int_true:wTF 0 ; }
23543   \or:   \exp_after:wN \__fp_small_int_normal:NnwTF
23544   \or:
23545     \__fp_case_return:nw
23546     {
23547       \exp_after:wN \__fp_small_int_true:wTF \int_value:w
23548       \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
23549     }
23550   \else: \__fp_case_return:nw \use_ii:nn
23551   \fi:
23552   #2
23553 }
23554 \cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
23555 \cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3;
23556 {
23557   \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
23558   \__fp_small_int_test:NnnwNw
23559   #3 #1
23560 }
23561 \cs_new:Npn \__fp_small_int_test:NnnwNw #1#2#3#4; #5
23562 {
23563   \if_meaning:w 0 #1
23564     \exp_after:wN \__fp_small_int_true:wTF
23565     \int_value:w \if_meaning:w 2 #5 - \fi:
23566     \if_int_compare:w #2 > \c_zero_int
23567     1 0000 0000
23568     \else:
23569     #3
23570     \fi:
23571     \exp_after:wN ;
23572   \else:
23573     \exp_after:wN \use_ii:nn
23574   \fi:
23575 }

```

(End of definition for `__fp_small_int:wTF` and others.)

68.13 Fast string comparison

`__fp_str_if_eq:nn` A private version of the low-level string comparison function.

```
23576 \cs_new_eq:NN \__fp_str_if_eq:nn \tex_strcmp:D
```

(End of definition for __fp_str_if_eq:nn.)

68.14 Name of a function from its l3fp-parse name

`__fp_func_to_name:N` The goal is to convert for instance `__fp_sin_o:w` to `sin`. This is used in error messages hence does not need to be fast.

```
23577 \cs_new:Npn \__fp_func_to_name:N #1
23578 {
23579   \exp_last_unbraced:Nf
23580   \__fp_func_to_name_aux:w { \cs_to_str:N #1 } X
23581 }
23582 \cs_set_protected:Npn \__fp_tmp:w #1 #2
23583 { \cs_new:Npn \__fp_func_to_name_aux:w ##1 #1 ##2 #2 ##3 X {##2} }
23584 \exp_args:Nff \__fp_tmp:w { \tl_to_str:n { __fp_ } }
23585 { \tl_to_str:n { _o: } }
```

(End of definition for __fp_func_to_name:N and __fp_func_to_name_aux:w.)

68.15 Messages

Using a floating point directly is an error.

```
23586 \msg_new:nnnn { fp } { misused }
23587 { A~floating~point~with~value~'#1'~was~misused. }
23588 {
23589   To~obtain~the~value~of~a~floating~point~variable,~use~
23590   '\token_to_str:N \fp_to_decimal:N',~
23591   '\token_to_str:N \fp_to_tl:N',~or~other~
23592   conversion~functions.
23593 }
23594 \prop_gput:Nnn \g_msg_module_name_prop { fp } { LaTeX }
23595 \prop_gput:Nnn \g_msg_module_type_prop { fp } { }
23596 </package>
```

Chapter 69

l3fp-traps implementation

23597 `*package`

23598 `\@@=fp`

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

69.1 Flags

Flags to denote exceptions.

`\l_fp_invalid_operation_flag`
`\l_fp_division_by_zero_flag`
`\l_fp_overflow_flag`
`\l_fp_underflow_flag`

23599 `\flag_new:N \l_fp_invalid_operation_flag`

23600 `\flag_new:N \l_fp_division_by_zero_flag`

23601 `\flag_new:N \l_fp_overflow_flag`

23602 `\flag_new:N \l_fp_underflow_flag`

(End of definition for `\l_fp_invalid_operation_flag` and others. These variables are documented on page 270.)

69.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the `inexact` exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `_fp_invalid_operation:nnw`,

- `__fp_invalid_operation_o:Nww,`
- `__fp_invalid_operation_tl_o:ff,`
- `__fp_division_by_zero_o:Nnw,`
- `__fp_division_by_zero_o:NNww,`
- `__fp_overflow:w,`
- `__fp_underflow:w.`

Rather than changing them directly, we provide a user interface as `\fp_trap:nn` $\langle\textit{exception}\rangle$ $\langle\textit{way of trapping}\rangle$, where the $\langle\textit{way of trapping}\rangle$ is one of `error`, `flag`, or `none`.

We also provide `__fp_invalid_operation_o:nw`, defined in terms of `__fp_invalid_operation:nnw`.

`\fp_trap:nn`

```

23603 \cs_new_protected:Npn \fp_trap:nn #1#2
23604   {
23605     \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
23606     {
23607       \clist_if_in:nnTF
23608         { invalid_operation , division_by_zero , overflow , underflow }
23609         {#1}
23610         {
23611           \msg_error:nnee { fp }
23612             { unknown-fpu-trap-type } {#1} {#2}
23613         }
23614         {
23615           \msg_error:nne
23616             { fp } { unknown-fpu-exception } {#1}
23617         }
23618     }
23619   }

```

(End of definition for `\fp_trap:nn`. This function is documented on page 270.)

`__fp_trap_invalid_operation_set_error:` We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.

```

\__fp_trap_invalid_operation_set_flag:
\__fp_trap_invalid_operation_set_none:
\__fp_trap_invalid_operation_set:N
23620 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_error:
23621   { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
23622 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_flag:
23623   { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
23624 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_none:
23625   { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
23626 \cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
23627   {
23628     \exp_args:Nno \use:n
23629     { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
23630     {
23631       #1
23632       \__fp_error:nnfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }

```

```

23633     \flag_ensure_raised:N \l_fp_invalid_operation_flag
23634     ##1
23635   }
23636 \exp_args:Nno \use:n
23637   { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
23638   {
23639     #1
23640     \__fp_error:nfn { invalid-ii }
23641     { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
23642     \flag_ensure_raised:N \l_fp_invalid_operation_flag
23643     \exp_after:wN \c_nan_fp
23644   }
23645 \exp_args:Nno \use:n
23646   { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
23647   {
23648     #1
23649     \__fp_error:nfn { invalid } {##1} {##2} { }
23650     \flag_ensure_raised:N \l_fp_invalid_operation_flag
23651     \exp_after:wN \c_nan_fp
23652   }
23653 }

```

(End of definition for __fp_trap_invalid_operation_set_error: and others.)

__fp_trap_division_by_zero_set_error: We provide three types of trapping for invalid operations and division by zero: either
 __fp_trap_division_by_zero_set_flag: produce an error and raise the relevant flag; or only raise the flag; or don't even raise the
 __fp_trap_division_by_zero_set_none: flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or
 __fp_trap_division_by_zero_set:N nan.

```

23654 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
23655   { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
23656 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
23657   { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
23658 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
23659   { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
23660 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
23661   {
23662     \exp_args:Nno \use:n
23663     { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
23664     {
23665       #1
23666       \__fp_error:nfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
23667       \flag_ensure_raised:N \l_fp_division_by_zero_flag
23668       \exp_after:wN ##1
23669     }
23670     \exp_args:Nno \use:n
23671     { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
23672     {
23673       #1
23674       \__fp_error:nfn { zero-div-ii }
23675       { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
23676       \flag_ensure_raised:N \l_fp_division_by_zero_flag
23677       \exp_after:wN ##1
23678     }
23679   }

```

(End of definition for `__fp_trap_division_by_zero_set_error:` and others.)

`__fp_trap_overflow_set_error:` Just as for invalid operations and division by zero, the three different behaviours are
`__fp_trap_overflow_set_flag:` obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an
`__fp_trap_overflow_set_none:` auxiliary, with a further auxiliary common to overflow and underflow functions. In most
`__fp_trap_underflow_set_error:` cases, the argument of the `__fp_overflow:w` and `__fp_underflow:w` functions will
`__fp_trap_underflow_set_flag:` be an (almost) normal number (with an exponent outside the allowed range), and the
`__fp_trap_underflow_set_none:` error message thus displays that number together with the result to which it overflowed
`__fp_trap_underflow_set:N` or underflowed. For extreme cases such as 10^{9999} , the exponent would be too
`__fp_trap_overflow_set:NnNn` large for T_EX, and `__fp_overflow:w` receives $\pm\infty$ (`__fp_underflow:w` would receive
 ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```

23680 \cs_new_protected:Npn \__fp_trap_overflow_set_error:
23681   { \__fp_trap_overflow_set:N \prg_do_nothing: }
23682 \cs_new_protected:Npn \__fp_trap_overflow_set_flag:
23683   { \__fp_trap_overflow_set:N \use_none:nnnnn }
23684 \cs_new_protected:Npn \__fp_trap_overflow_set_none:
23685   { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
23686 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
23687   { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
23688 \cs_new_protected:Npn \__fp_trap_underflow_set_error:
23689   { \__fp_trap_underflow_set:N \prg_do_nothing: }
23690 \cs_new_protected:Npn \__fp_trap_underflow_set_flag:
23691   { \__fp_trap_underflow_set:N \use_none:nnnnn }
23692 \cs_new_protected:Npn \__fp_trap_underflow_set_none:
23693   { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
23694 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
23695   { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
23696 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
23697   {
23698     \exp_args:Nno \use:n
23699     { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
23700     {
23701       #1
23702       \__fp_error:nffn
23703         { flow \if_meaning:w 1 ##1 -to \fi: }
23704         { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
23705         { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
23706         {#2}
23707         \flag_ensure_raised:c { l_fp_#2_flag }
23708         #3 ##2
23709       }
23710     }

```

(End of definition for `__fp_trap_overflow_set_error:` and others.)

`__fp_invalid_operation:nnw` Initialize the control sequences (to log properly their existence). Then set invalid opera-
`__fp_invalid_operation_o:Nnw` tions to trigger an error, and division by zero, overflow, and underflow to act silently on
`__fp_invalid_operation_tl_o:ff` their flag.
`__fp_division_by_zero_o:Nnw` 23711 \cs_new:Npn __fp_invalid_operation:nnw #1#2#3; { }
`__fp_division_by_zero_o:NNww` 23712 \cs_new:Npn __fp_invalid_operation_o:Nnw #1#2; #3; { }
`__fp_overflow:w` 23713 \cs_new:Npn __fp_invalid_operation_tl_o:ff #1 #2 { }
`__fp_underflow:w` 23714 \cs_new:Npn __fp_division_by_zero_o:Nnw #1#2#3; { }
23715 \cs_new:Npn __fp_division_by_zero_o:NNww #1#2#3; #4; { }

```

23716 \cs_new:Npn \__fp_overflow:w { }
23717 \cs_new:Npn \__fp_underflow:w { }
23718 \fp_trap:nn { invalid_operation } { error }
23719 \fp_trap:nn { division_by_zero } { flag }
23720 \fp_trap:nn { overflow } { flag }
23721 \fp_trap:nn { underflow } { flag }

```

(End of definition for __fp_invalid_operation:nw and others.)

__fp_invalid_operation_o:nw Convenient short-hands for returning \c_nan_fp for a unary or binary operation, and
 __fp_invalid_operation_o:fw expanding after.

```

23722 \cs_new:Npn \__fp_invalid_operation_o:nw
23723 { \__fp_invalid_operation:nw { \exp_after:wN \c_nan_fp } }
23724 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }

```

(End of definition for __fp_invalid_operation_o:nw.)

69.3 Errors

```

\__fp_error:nnnn
\__fp_error:nfn 23725 \cs_new:Npn \__fp_error:nnnn
\__fp_error:nffn 23726 { \msg_expandable_error:nnnnn { fp } }
\__fp_error:nfff 23727 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff , nfff }

```

(End of definition for __fp_error:nnnn.)

69.4 Messages

Some messages.

```

23728 \msg_new:nnnn { fp } { unknown-fpu-exception }
23729 {
23730   The-FPU-exception-~'#1'~is-not-known:~
23731   that-trap-will-never-be-triggered.
23732 }
23733 {
23734   The-only-exceptions~to-which~traps~can-be-attached-are \\
23735   \iow_indent:n
23736   {
23737     * ~ invalid_operation \\
23738     * ~ division_by_zero \\
23739     * ~ overflow \\
23740     * ~ underflow
23741   }
23742 }
23743 \msg_new:nnnn { fp } { unknown-fpu-trap-type }
23744 { The-FPU-trap-type-~'#2'~is-not-known. }
23745 {
23746   The-trap-type-must-be-one~of \\
23747   \iow_indent:n
23748   {
23749     * ~ error \\
23750     * ~ flag \\

```



```

23751         * ~ none
23752     }
23753 }
23754 \msg_new:nnn { fp } { flow }
23755   { An ~ #3 ~ occurred. }
23756 \msg_new:nnn { fp } { flow-to }
23757   { #1 ~ #3 ed ~ to ~ #2 . }
23758 \msg_new:nnn { fp } { zero-div }
23759   { Division-by-zero-in~ #1 (#2) }
23760 \msg_new:nnn { fp } { zero-div-ii }
23761   { Division-by-zero-in~ (#1) #3 (#2) }
23762 \msg_new:nnn { fp } { invalid }
23763   { Invalid-operation~ #1 (#2) }
23764 \msg_new:nnn { fp } { invalid-ii }
23765   { Invalid-operation~ (#1) #3 (#2) }
23766 \msg_new:nnn { fp } { unknown-type }
23767   { Unknown-type-for~'#1' }
23768 </package>

```

Chapter 70

13fp-round implementation

```
23769 (*package)
23770 (@@=fp)

\__fp_parse_word_trunc:N
\__fp_parse_word_floor:N
\__fp_parse_word_ceil:N
23771 \cs_new:Npn \__fp_parse_word_trunc:N
23772   { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
23773 \cs_new:Npn \__fp_parse_word_floor:N
23774   { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
23775 \cs_new:Npn \__fp_parse_word_ceil:N
23776   { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_pinf:NNN }

(End of definition for \__fp_parse_word_trunc:N, \__fp_parse_word_floor:N, and \__fp_parse_
word_ceil:N.)

\__fp_parse_word_round:N
\__fp_parse_round:Nw
23777 \cs_new:Npn \__fp_parse_word_round:N #1#2
23778   {
23779     \__fp_parse_function:NNN
23780     \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
23781     #2
23782   }
23783 \cs_new:Npn \__fp_parse_round:Nw #1 #2 \__fp_round_to_nearest:NNN #3#4
23784   { #2 #1 #3 }
23785

(End of definition for \__fp_parse_word_round:N and \__fp_parse_round:Nw.)
```

70.1 Rounding tools

`\c__fp_five_int` This is used as the half-point for which numbers are rounded up/down.

```
23786 \int_const:Nn \c__fp_five_int { 5 }
```

(End of definition for `\c__fp_five_int`.)

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in `l3fp` yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `__fp_round:NNN`, which expands to `0\exp_stop_f:` or `1\exp_stop_f:` depending on whether the final result should be rounded up or down.

- `__fp_round:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:.`
- `__fp_round_s:NNNw <sign> <digit1> <digit2> <more digits>;` can expand to `0\exp_stop_f:;` or `1\exp_stop_f:;`.
- `__fp_round_neg:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:.`

See implementation comments for details on the syntax.

```

\__fp_round:NNN
\__fp_round_to_nearest:NNN
  \__fp_round_to_nearest_ninf:NNN
  \__fp_round_to_nearest_zero:NNN
  \__fp_round_to_nearest_pinf:NNN
\__fp_round_to_ninf:NNN
\__fp_round_to_zero:NNN
\__fp_round_to_pinf:NNN

```

```
\__fp_round:NNN <final sign> <digit1> <digit2>
```

If rounding the number `<final sign><digit1>.<digit2>` to an integer rounds it towards zero (truncates it), this function expands to `0\exp_stop_f:`, and otherwise to `1\exp_stop_f:.` Typically used within the scope of an `__fp_int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that `<final sign>` be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that `<final sign>` is 0 for positive, and 2 for negative.

By default, the functions below return `0\exp_stop_f:`, but this is superseded by `__fp_round_return_one:`, which instead returns `1\exp_stop_f:`, expanding everything and removing `0\exp_stop_f:` in the process. In the case of rounding towards $\pm\infty$ or towards 0, this is not really useful, but it prepares us for the “round to nearest, ties to even” mode.

The “round to nearest” mode is the default. If the `<digit2>` is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that `<digit1>` plus the result is even. In other words, round up if `<digit1>` is odd.

The “round to nearest” mode has three variants, which differ in how ties are rounded: down towards $-\infty$, truncated towards 0, or up towards $+\infty$.

```

23787 \cs_new:Npn \__fp_round_return_one:
23788   { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
23789 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
23790   {
23791     \if_meaning:w 2 #1
23792     \if_int_compare:w #3 > \c_zero_int
23793     \__fp_round_return_one:
23794     \fi:
23795     \fi:
23796     \c_zero_int
23797   }
23798 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero_int }
23799 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
23800   {
23801     \if_meaning:w 0 #1
23802     \if_int_compare:w #3 > \c_zero_int
23803     \__fp_round_return_one:
23804     \fi:
23805     \fi:
23806     \c_zero_int
23807   }
23808 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
23809   {
23810     \if_int_compare:w #3 > \c__fp_five_int
23811     \__fp_round_return_one:
23812     \else:
23813     \if_meaning:w 5 #3
23814     \if_int_odd:w #2 \exp_stop_f:
23815     \__fp_round_return_one:
23816     \fi:
23817     \fi:
23818     \fi:
23819     \c_zero_int
23820   }
23821 \cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
23822   {
23823     \if_int_compare:w #3 > \c__fp_five_int
23824     \__fp_round_return_one:
23825     \else:
23826     \if_meaning:w 5 #3
23827     \if_meaning:w 2 #1
23828     \__fp_round_return_one:
23829     \fi:
23830     \fi:
23831     \fi:
23832     \c_zero_int
23833   }
23834 \cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
23835   {
23836     \if_int_compare:w #3 > \c__fp_five_int
23837     \__fp_round_return_one:
23838     \fi:
23839     \c_zero_int
23840   }

```

```

23841 \cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
23842 {
23843   \if_int_compare:w #3 > \c__fp_five_int
23844     \__fp_round_return_one:
23845   \else:
23846     \if_meaning:w 5 #3
23847     \if_meaning:w 0 #1
23848     \__fp_round_return_one:
23849   \fi:
23850   \fi:
23851   \fi:
23852   \c_zero_int
23853 }
23854 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN

```

(End of definition for __fp_round:NNN and others.)

__fp_round_s:NNNw

__fp_round_s:NNNw *<final sign>* *<digit>* *<more digits>* ;
 Similar to __fp_round:NNN, but with an extra semicolon, this function expands to 0\exp_stop_f:; if rounding *<final sign>**<digit>**<more digits>* to an integer truncates, and to 1\exp_stop_f:; otherwise. The *<more digits>* part must be a digit, followed by something that does not overflow a \int_use:N __fp_int_eval:w construction. The only relevant information about this piece is whether it is zero or not.

```

23855 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
23856 {
23857   \exp_after:wN \__fp_round:NNN
23858   \exp_after:wN #1
23859   \exp_after:wN #2
23860   \int_value:w \__fp_int_eval:w
23861   \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
23862   \if_meaning:w 5 #3 1 \fi:
23863   \exp_stop_f:
23864   \if_int_compare:w \__fp_int_eval:w #4 > \c_zero_int
23865     1 +
23866   \fi:
23867   \fi:
23868   #3
23869   ;
23870 }

```

(End of definition for __fp_round_s:NNNw.)

__fp_round_digit:Nw

\int_value:w __fp_round_digit:Nw *<digit>* *<int expr>* ;
 This function should always be called within an \int_value:w or __fp_int_eval:w expansion; it may add an extra __fp_int_eval:w, which means that the integer or integer expression should not be ended with a synonym of \relax, but with a semi-colon for instance.

```

23871 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
23872 {
23873   \if_int_odd:w \if_meaning:w 0 #1 1 \else:
23874     \if_meaning:w 5 #1 1 \else:
23875     0 \fi: \fi: \exp_stop_f:
23876   \if_int_compare:w \__fp_int_eval:w #2 > \c_zero_int
23877     \__fp_int_eval:w 1 +

```

```

23878     \fi:
23879     \fi:
23880     #1
23881   }

```

(End of definition for `_fp_round_digit:Nw`.)

```

\_fp_round_neg:NNN
\_fp_round_to_nearest_neg:NNN
\_fp_round_to_nearest_ninf_neg:NNN
\_fp_round_to_nearest_zero_neg:NNN
\_fp_round_to_nearest_pinf_neg:NNN
\_fp_round_to_ninf_neg:NNN
\_fp_round_to_zero_neg:NNN
\_fp_round_to_pinf_neg:NNN

```

```

\_fp_round_neg:NNN <final sign> <digit1> <digit2>

```

This expands to `0\exp_stop_f:` or `1\exp_stop_f:` after doing the following test. Starting from a number of the form `<final sign>0.<15 digits><digit1>` with exactly 15 (non-all-zero) digits before `<digit1>`, subtract from it `<final sign>0.0...0<digit2>`, where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns `1\exp_stop_f:`. Otherwise, *i.e.*, if the result is rounded back to the first operand, then this function returns `0\exp_stop_f:`.

It turns out that this negative “round to nearest” is identical to the positive one. And this is the default mode.

```

23882 \cs_new_eq:NN \_fp_round_to_ninf_neg:NNN \_fp_round_to_pinf:NNN
23883 \cs_new:Npn \_fp_round_to_zero_neg:NNN #1 #2 #3
23884   {
23885     \if_int_compare:w #3 > \c_zero_int
23886       \_fp_round_return_one:
23887     \fi:
23888     \c_zero_int
23889   }
23890 \cs_new_eq:NN \_fp_round_to_pinf_neg:NNN \_fp_round_to_ninf:NNN
23891 \cs_new_eq:NN \_fp_round_to_nearest_neg:NNN \_fp_round_to_nearest:NNN
23892 \cs_new_eq:NN \_fp_round_to_nearest_ninf_neg:NNN
23893   \_fp_round_to_nearest_pinf:NNN
23894 \cs_new:Npn \_fp_round_to_nearest_zero_neg:NNN #1 #2 #3
23895   {
23896     \if_int_compare:w #3 < \c__fp_five_int \else:
23897       \_fp_round_return_one:
23898     \fi:
23899     \c_zero_int
23900   }
23901 \cs_new_eq:NN \_fp_round_to_nearest_pinf_neg:NNN
23902   \_fp_round_to_nearest_ninf:NNN
23903 \cs_new_eq:NN \_fp_round_neg:NNN \_fp_round_to_nearest_neg:NNN

```

(End of definition for `_fp_round_neg:NNN` and others.)

70.2 The round function

```

\_fp_round_o:Nw
\_fp_round_aux_o:Nw

```

First check that all arguments are floating point numbers. The `trunc`, `ceil` and `floor` functions expect one or two arguments (the second is 0 by default), and the `round` function also accepts a third argument (`nan` by default), which changes `#1` from `_fp_round_to_nearest:NNN` to one of its analogues.

```

23904 \cs_new:Npn \_fp_round_o:Nw #1
23905   {
23906     \_fp_parse_function_all_fp_o:fnw
23907     { \_fp_round_name_from_cs:N #1 }
23908     { \_fp_round_aux_o:Nw #1 }

```

```

23909 }
23910 \cs_new:Npn \__fp_round_aux_o:Nw #1#2 @
23911 {
23912   \if_case:w
23913     \__fp_int_eval:w \__fp_array_count:n {#2} \__fp_int_eval_end:
23914     \__fp_round_no_arg_o:Nw #1 \exp:w
23915   \or: \__fp_round:Nwn #1 #2 {0} \exp:w
23916   \or: \__fp_round:Nww #1 #2 \exp:w
23917   \else: \__fp_round:Nwww #1 #2 @ \exp:w
23918   \fi:
23919   \exp_after:wN \exp_end:
23920 }

```

(End of definition for __fp_round_o:Nw and __fp_round_aux_o:Nw.)

__fp_round_no_arg_o:Nw

```

23921 \cs_new:Npn \__fp_round_no_arg_o:Nw #1
23922 {
23923   \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
23924     { \__fp_error:nnnn { num-args } { round () } { 1 } { 3 } }
23925     {
23926       \__fp_error:nfn { num-args }
23927       { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
23928     }
23929   \exp_after:wN \c_nan_fp
23930 }

```

(End of definition for __fp_round_no_arg_o:Nw.)

__fp_round:Nwww Having three arguments is only allowed for round, not trunc, ceil, floor, so check for that case. If all is well, construct one of __fp_round_to_nearest:NNN, __fp_round_to_nearest_zero:NNN, __fp_round_to_nearest_ninf:NNN, __fp_round_to_nearest_pinf:NNN and act accordingly.

```

23931 \cs_new:Npn \__fp_round:Nwww #1#2 ; #3 ; \s__fp \__fp_chk:w #4#5#6 ; #7 @
23932 {
23933   \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
23934     {
23935       \tl_if_empty:nTF {#7}
23936       {
23937         \exp_args:Nc \__fp_round:Nww
23938           {
23939             \__fp_round_to_nearest
23940             \if_meaning:w 0 #4 _zero \else:
23941             \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
23942             :NNN
23943           }
23944         #2 ; #3 ;
23945       }
23946       {
23947         \__fp_error:nnnn { num-args } { round () } { 1 } { 3 }
23948         \exp_after:wN \c_nan_fp
23949       }
23950     }
23951 }

```

```

23952     \_fp_error:nfn { num-args }
23953     { \_fp_round_name_from_cs:N #1 () } { 1 } { 2 }
23954     \exp_after:wN \c_nan_fp
23955   }
23956 }

```

(End of definition for _fp_round:Nwww.)

_fp_round_name_from_cs:N

```

23957 \cs_new:Npn \_fp_round_name_from_cs:N #1
23958 {
23959   \cs_if_eq:NNTF #1 \_fp_round_to_zero:NNN { trunc }
23960   {
23961     \cs_if_eq:NNTF #1 \_fp_round_to_ninf:NNN { floor }
23962     {
23963       \cs_if_eq:NNTF #1 \_fp_round_to_pinf:NNN { ceil }
23964       { round }
23965     }
23966   }
23967 }

```

(End of definition for _fp_round_name_from_cs:N.)

_fp_round:Nww

_fp_round:Nwn

_fp_round_normal:NwNNnw

_fp_round_normal:NnnwNNnn

_fp_round_pack:Nw

_fp_round_normal:NNwNnn

_fp_round_normal_end:wwNnn

_fp_round_special:NwwNnn

_fp_round_special_aux:Nw

```

23968 \cs_new:Npn \_fp_round:Nww #1#2 ; #3 ;
23969 {
23970   \_fp_small_int:wTF #3; { \_fp_round:Nwn #1#2; }
23971   {
23972     \if:w 3 \_fp_kind:w #3 ;
23973     \exp_after:wN \use_i:nn
23974     \else:
23975     \exp_after:wN \use_ii:nn
23976     \fi:
23977     { \exp_after:wN \c_nan_fp }
23978     {
23979       \_fp_invalid_operation_tl_o:ff
23980       { \_fp_round_name_from_cs:N #1 }
23981       { \_fp_array_to_clist:n { #2; #3; } }
23982     }
23983   }
23984 }
23985 \cs_new:Npn \_fp_round:Nwn #1 \s__fp \_fp_chk:w #2#3#4; #5
23986 {
23987   \if_meaning:w 1 #2
23988   \exp_after:wN \_fp_round_normal:NwNNnw
23989   \exp_after:wN #1
23990   \int_value:w #5
23991   \else:
23992   \exp_after:wN \_fp_exp_after_o:w
23993   \fi:
23994   \s__fp \_fp_chk:w #2#3#4;
23995 }

```

If the number of digits to round to is an integer or infinity all is good; if it is nan then just produce a nan; otherwise invalid as we have something like round(1,3.14) where the number of digits is not an integer.


```

23996 \cs_new:Npn \__fp_round_normal:NwNnNw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
23997 {
23998   \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 - #2 }
23999   \__fp_round_normal:NnnwNnNn #5 #1 #3 {#4} {#2}
24000 }
24001 \cs_new:Npn \__fp_round_normal:NnnwNnNn #1#2#3#4; #5#6
24002 {
24003   \exp_after:wN \__fp_round_normal:NNwNnn
24004   \int_value:w \__fp_int_eval:w
24005   \if_int_compare:w #2 > \c_zero_int
24006     1 \int_value:w #2
24007   \exp_after:wN \__fp_round_pack:Nw
24008   \int_value:w \__fp_int_eval:w 1#3 +
24009   \else:
24010     \if_int_compare:w #3 > \c_zero_int
24011       1 \int_value:w #3 +
24012     \fi:
24013   \fi:
24014   \exp_after:wN #5
24015   \exp_after:wN #6
24016   \use_none:nnnnnn #3
24017   #1
24018   \__fp_int_eval_end:
24019   0000 0000 0000 0000 ; #6
24020 }
24021 \cs_new:Npn \__fp_round_pack:Nw #1
24022 { \if_meaning:w 2 #1 + 1 \fi: \__fp_int_eval_end: }
24023 \cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
24024 {
24025   \if_meaning:w 0 #2
24026     \exp_after:wN \__fp_round_special:NwwNnn
24027     \exp_after:wN #1
24028     \fi:
24029     \__fp_pack_twice_four:wNNNNNNNN
24030     \__fp_pack_twice_four:wNNNNNNNN
24031     \__fp_round_normal_end:wwNnn
24032     ; #2
24033 }
24034 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
24035 {
24036   \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
24037   \__fp_sanitizew:Nw #3 #4 ; #1 ;
24038 }
24039 \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
24040 {
24041   \if_meaning:w 0 #1
24042     \__fp_case_return:nw
24043     { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
24044   \else:
24045     \exp_after:wN \__fp_round_special_aux:Nw
24046     \exp_after:wN #4
24047     \int_value:w \__fp_int_eval:w 1
24048     \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
24049   \fi:

```

```
24050     ;
24051   }
24052   \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
24053     {
24054       \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
24055       \__fp_sanitise:Nw #1#2; {1000}{0000}{0000}{0000};
24056     }
```

(End of definition for __fp_round:Nww and others.)

```
24057 </package>
```

Chapter 71

l3fp-parse implementation

24058 `*package`

24059 `\@@=fp`

71.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

In this file at least, a *floating point object* is a floating point number or tuple. This can be extended to anything that starts with `\s__fp` or `\s__fp_⟨type⟩` and ends with `;` with some internal structure that depends on the *⟨type⟩*.

`__fp_parse:n`

`__fp_parse:n {⟨fp expr⟩}`

Evaluates the *⟨fp expr⟩* and leaves the result in the input stream as a floating point object. This function forms the basis of almost all public l3fp functions. During evaluation, each token is fully *f*-expanded.

`__fp_parse_o:n` does the same but expands once after its result.

TeXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after *f*-expansion lead to unrecoverable low-level TeX errors.

(End of definition for __fp_parse:n.)

`\c__fp_prec_func_int`
`\c__fp_prec_hatii_int`
`\c__fp_prec_hat_int`
`\c__fp_prec_not_int`
`\c__fp_prec_juxt_int`
`\c__fp_prec_times_int`
`\c__fp_prec_plus_int`
`\c__fp_prec_comp_int`
`\c__fp_prec_and_int`
`\c__fp_prec_or_int`
`\c__fp_prec_quest_int`
`\c__fp_prec_colon_int`
`\c__fp_prec_comma_int`
`\c__fp_prec_tuple_int`
`\c__fp_prec_end_int`

Floating point expressions are composed of numbers, given in various forms, infix operators, such as `+`, `**`, or `,` (which joins two numbers into a list), and prefix operators, such as the unary `-`, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

16 Function calls.

13/14 Binary `**` and `^` (right to left).

12 Unary `+`, `-`, `!` (right to left).

11 Juxtaposition (implicit `*`) with no parenthesis.

- 10 Binary * and /.
- 9 Binary + and -.
- 7 Comparisons.
- 6 Logical and, denoted by &&.
- 5 Logical or, denoted by ||.
- 4 Ternary operator ?:, piece ?.
- 3 Ternary operator ?:, piece :.
- 2 Commas.
- 1 Place where a comma is allowed and generates a tuple.
- 0 Start and end of the expression.

```

24060 \int_const:Nn \c__fp_prec_func_int { 16 }
24061 \int_const:Nn \c__fp_prec_hatii_int { 14 }
24062 \int_const:Nn \c__fp_prec_hat_int { 13 }
24063 \int_const:Nn \c__fp_prec_not_int { 12 }
24064 \int_const:Nn \c__fp_prec_juxt_int { 11 }
24065 \int_const:Nn \c__fp_prec_times_int { 10 }
24066 \int_const:Nn \c__fp_prec_plus_int { 9 }
24067 \int_const:Nn \c__fp_prec_comp_int { 7 }
24068 \int_const:Nn \c__fp_prec_and_int { 6 }
24069 \int_const:Nn \c__fp_prec_or_int { 5 }
24070 \int_const:Nn \c__fp_prec_quest_int { 4 }
24071 \int_const:Nn \c__fp_prec_colon_int { 3 }
24072 \int_const:Nn \c__fp_prec_comma_int { 2 }
24073 \int_const:Nn \c__fp_prec_tuple_int { 1 }
24074 \int_const:Nn \c__fp_prec_end_int { 0 }

```

(End of definition for \c__fp_prec_func_int and others.)

71.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to f-expand tokens one by one in the expression (as `\int_eval:n` does), and with this approach, expanding the next unread token forces us to jump with `\exp_after:wN` over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the `\exp_after:wN` is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets

us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be better to avoid requiring the closing parenthesis to be present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored *before* the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

```
\exp_after:wN \add:ww \int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w (stuff)
```

One step of expansion expands `\exp_after:wN`, which triggers the primitive `\int_value:w`, which reads the five digits we have already found, 12345. This integer is unfinished, causing the second `\exp_after:wN` to expand, and to trigger the construction `\exp:w`, which expands `\operand:w`, defined to read what follows and make a number out of it, then leave `\exp_end:`, the number, and a semicolon in the input stream. Once `\operand:w` is done expanding, we obtain essentially

```
\exp_after:wN \add:ww \int_value:w 12345 ;
\exp:w \exp_end: 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `\int_value:w` has already seen 12345, and `\exp:w` is still looking for a number. It finds `\exp_end:`, hence expands to nothing. Now, `\int_value:w` sees the `;`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `\int_value:w 12345 \exp_after:wN ;` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `_fp_..._o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

71.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations: $1 + 2 \times 3 = 1 + (2 \times 3)$ because \times has a higher precedence than $+$. The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how we plan to do the calculation $41 - 2^3 * 4 + 5$. More precisely we describe how to perform the first operation in this expression. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer

constant (`\c__fp_prec_plus_int, ...`) equal to the precedence of the given operator, not directly the operator itself.

- Clean up 41 and find `-`. We call `\operand:Nw -` to find the second operand.
- Clean up 2 and find `^`.
- Compare the precedences of `-` and `^`. Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw ^`.
- Clean up 3 and find `*`.
- Compare the precedences of `^` and `*`. Since the former is higher, `\operand:Nw ^` has found the second operand of the exponentiation, which is computed: $2^3 = 8$.
- We now have `41-8*4+5`, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?
- Compare the precedences of `-` and `*`. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up 4, and find `+`.
- Compare the precedences of `*` and `+`. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed: $8 * 4 = 32$.
- We now have `41-32+5`, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of `-` and `+`. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed: $41 - 32 = 9$.
- We now have `9+5`.

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations are performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the `l3fp-parse` functions correspond to each step above.

First, `__fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by $\langle precedence \rangle$ the argument of `__fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `__fp_parse_one:Nw`. A first approximation of this function is that it reads one $\langle number \rangle$, performing no computation, and finds the following binary $\langle operator \rangle$. Then it expands to

```

⟨number⟩
  \_fp_parse_infix_⟨operator⟩:N ⟨precedence⟩

```

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as `1-2-3` being computed as `(1-2)-3`, but `2^3^4` should be evaluated as `2^(3^4)` instead. For this reason, and to support the equivalence between `**` and `^` more easily, each binary operator is converted to a control sequence `_fp_parse_infix_⟨operator⟩:N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the `⟨precedence⟩` (of the earlier operator) to the `infix` auxiliary for the following `⟨operator⟩`, to know whether to perform the computation of the `⟨operator⟩`. If it should not be performed, the `infix` auxiliary expands to

```

@ \use_none:n \_fp_parse_infix_⟨operator⟩:N

```

and otherwise it calls `_fp_parse_operand:Nw` with the precedence of the `⟨operator⟩` to find its second operand `⟨number2⟩` and the next `⟨operator2⟩`, and expands to

```

@ \_fp_parse_apply_binary:NwNwN
  ⟨operator⟩ ⟨number2⟩
@ \_fp_parse_infix_⟨operator2⟩:N

```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand `⟨number⟩` is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `_fp_parse_operand:Nw ⟨precedence⟩` with some of the expansion control removed is

```

\exp_after:wN \_fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\exp:w \exp_end_continue_f:w
  \_fp_parse_one:Nw ⟨precedence⟩

```

This expands `_fp_parse_one:Nw ⟨precedence⟩` completely, which finds a number, wraps the next `⟨operator⟩` into an `infix` function, feeds this function the `⟨precedence⟩`, and expands it, yielding either

```

\_fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\use_none:n \_fp_parse_infix_⟨operator⟩:N

```

or

```

\_fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\_fp_parse_apply_binary:NwNwN
  ⟨operator⟩ ⟨number2⟩
@ \_fp_parse_infix_⟨operator2⟩:N

```

The definition of `_fp_parse_continue:NwN` is then very simple:

```

\cs_new:Npn \_fp_parse_continue:NwN #1#2#3 { #3 #1 #2 @ }

```

In the first case, #3 is `\use_none:n`, yielding

```
\use_none:n <precedence> <number> @
\__fp_parse_infix_<operator>:N
```

then `<number> @ __fp_parse_infix_<operator>:N`. In the second case, #3 is `__fp_parse_apply_binary:NwNwN`, whose role is to compute `<number> <operator> <number_2>` and to prepare for the next comparison of precedences: first we get

```
\__fp_parse_apply_binary:NwNwN
<precedence> <number> @
<operator> <number_2>
@ \__fp_parse_infix_<operator_2>:N
```

then

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN <precedence>
\exp:w \exp_end_continue_f:w
\__fp_<operator>_o:ww <number> <number_2>
\exp:w \exp_end_continue_f:w
\__fp_parse_infix_<operator_2>:N <precedence>
```

where `__fp_<operator>_o:ww` computes `<number> <operator> <number_2>` and expands after the result, thus triggers the comparison of the precedence of the `<operator_2>` and the `<precedence>`, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

71.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary `-`, `+`, `!`) and parentheses are taken care of by the same mechanism, and functions (`sin`, `exp`, etc.) as well. Finding the argument of the unary `-`, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary `-`, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function `__fp_parse_one:Nw` earlier. This function is responsible for reading in the input stream the first possible `<number>` and the next infix `<operator>`. If what follows `__fp_parse_one:Nw <precedence>` is a prefix operator, then we must find the operand of this prefix operator through a nested call to `__fp_parse_operand:Nw` with the appropriate precedence, then apply the operator to the operand found to yield the result of `__fp_parse_one:Nw`. So far, all is simple.

The unary operators `+`, `-`, `!` complicate things a little bit: `-3**2` should be $-(3^2) = -9$, and not $(-3)^2 = 9$. This would easily be done by giving `-` a lower precedence, equal to that of the infix `+` and `-`. Unfortunately, this fails in cases such as `3**-2*4`, yielding $3^{-2 \times 4}$ instead of the correct $3^{-2} \times 4$. A second attempt would be to call `__fp_parse_operand:Nw` with the `<precedence>` of the previous operator, but `0>-2+3` is then

parsed as $0 > -(2+3)$: the addition is performed because it binds more tightly than the comparison which precedes $-$. The correct approach is for a unary $-$ to perform operations whose precedence is greater than both that of the previous operation, and that of the unary $-$ itself. The unary $-$ is given a precedence higher than multiplication and division. This does not lead to any surprising result, since $-(x/y) = (-x)/y$ and similarly for multiplication, and it reduces the number of nested calls to `_fp_parse_operand:Nw`.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous `<precedence>` to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

71.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form `<significand>e<exponent>`, where the `<significand>` is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “`e<exponent>`” is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs $+$ or $-$ and a non-empty string of decimal digits. The `<significand>` can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the `<exponent>` can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `_fp_parse_one:Nw` is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s__fp`, in which case our job is done, as what follows is an internal floating point number, or `\s__fp_expr_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from `c`-expansion.
- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the `<significand>` of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.
- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `_fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case `_fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `_fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s_fp_expr_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero_int`, with an implied multiplication.
- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.
- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `_fp_infix_⟨operator⟩:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero_int 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude 0, replace 9 by 10. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if `#1` lies in [65,90] (uppercase letters) or [97,112] (lowercase letters)

```
\if_int_compare:w \_fp_int_eval:w
  ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26 = 3 \exp_stop_f:
  is a letter
\else:
  not a letter
\fi:
```

At all steps, we try to accept all category codes: when `#1` is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes {3,6,7,8,11,12} should work without trouble, but not {1,2,4,10,13}, and of course {0,5,9} cannot become tokens.

Floating point expressions should behave as much as possible like ε -TeX-based integer expressions and dimension expressions. In particular, `f`-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop `f`-expansion: for instance, the macro `\X` below would not be expanded if we simply performed `f`-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back

in the input stream, then `f-expand` it. This is not a complete solution, since a macro's expansion could contain leading spaces which would stop the `f-expansion` before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s__fp ...` structure. The `f-expansion` is performed by `__fp_parse_expand:w`.

71.2 Main auxiliary functions

`__fp_parse_operand:Nw` `\exp:w __fp_parse_operand:Nw <precedence> __fp_parse_expand:w`
 Reads the "...", performing every computation with a precedence higher than `<precedence>`, then expands to

`<result> @ __fp_parse_infix_<operation>:N ...`

where the `<operation>` is the first operation with a lower precedence, possibly `end`, and the "..." start just after the `<operation>`.

(End of definition for __fp_parse_operand:Nw.)

`__fp_parse_infix_+:N` `__fp_parse_infix_+:N <precedence> ...`
 If `+` has a precedence higher than the `<precedence>`, cleans up a second `<operand>` and finds the `<operation2>` which follows, and expands to

`@ __fp_parse_apply_binary:NwNwN + <operand> @ __fp_parse_infix_<operation2>:N`
 ...

Otherwise expands to

`@ \use_none:n __fp_parse_infix_+:N ...`

A similar function exists for each infix operator.

(End of definition for __fp_parse_infix_+:N.)

`__fp_parse_one:Nw` `__fp_parse_one:Nw <precedence> ...`
 Cleans up one or two operands depending on how the precedence of the next operation compares to the `<precedence>`. If the following `<operation>` has a precedence higher than `<precedence>`, expands to

`<operand1> @ __fp_parse_apply_binary:NwNwN <operation> <operand2> @`
`__fp_parse_infix_<operation2>:N ...`

and otherwise expands to

`<operand> @ \use_none:n __fp_parse_infix_<operation>:N ...`

(End of definition for __fp_parse_one:Nw.)

71.3 Helpers

`__fp_parse_expand:w` `\exp:w __fp_parse_expand:w <tokens>`

This function must always come within a `\exp:w` expansion. The `<tokens>` should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

```
24075 \cs_new:Npn \__fp_parse_expand:w #1 { \exp_end_continue_f:w #1 }
```

(End of definition for __fp_parse_expand:w.)

`_fp_parse_return_semicolon:w` This very odd function swaps its position with the following `\fi:` and removes `__fp_parse_expand:w` normally responsible for expansion. That turns out to be useful.

```
24076 \cs_new:Npn \_fp_parse_return_semicolon:w
24077     #1 \fi: \__fp_parse_expand:w { \fi: ; #1 }
```

(End of definition for _fp_parse_return_semicolon:w.)

`__fp_parse_digits_vii:N` These functions must be called within an `\int_value:w` or `__fp_int_eval:w` construction. The first token which follows must be `f`-expanded prior to calling those functions. The functions read tokens one by one, and output digits into the input stream, until meeting a non-digit, or up to a number of digits equal to their index. The full expansion is

```
\__fp_parse_digits_vii:N
\__fp_parse_digits_vi:N
\__fp_parse_digits_v:N
\__fp_parse_digits_iv:N
\__fp_parse_digits_iii:N
\__fp_parse_digits_ii:N
\__fp_parse_digits_i:N
\__fp_parse_digits_:N
    <digits> ; <filling 0> ; <length>
```

where `<filling 0>` is a string of zeros such that `<digits> <filling 0>` has the length given by the index of the function, and `<length>` is the number of zeros in the `<filling 0>` string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through `\token_to_str:N` to normalize their category code.

```
24078 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
24079     {
24080     \cs_new:cpn { \__fp_parse_digits_ #1 :N } ##1
24081     {
24082     \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
24083     \token_to_str:N ##1 \exp_after:wN #2 \exp:w
24084     \else:
24085     \_fp_parse_return_semicolon:w #3 ##1
24086     \fi:
24087     \__fp_parse_expand:w
24088     }
24089     }
24090 \__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 000000 ; 7 }
24091 \__fp_tmp:w {vi} \__fp_parse_digits_v:N { 000000 ; 6 }
24092 \__fp_tmp:w {v} \__fp_parse_digits_iv:N { 00000 ; 5 }
24093 \__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 0000 ; 4 }
24094 \__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 000 ; 3 }
24095 \__fp_tmp:w {ii} \__fp_parse_digits_i:N { 00 ; 2 }
24096 \__fp_tmp:w {i} \__fp_parse_digits_:N { 0 ; 1 }
24097 \cs_new:Npn \__fp_parse_digits_:N { ; ; 0 }
```

(End of definition for __fp_parse_digits_vii:N and others.)

71.4 Parsing one number

`__fp_parse_one:Nw` This function finds one number, and packs the symbol which follows in an `__fp_parse_infix_...` csname. #1 is the previous `\precedence`, and #2 the first token of the operand. We distinguish four cases: #2 is equal to `\scan_stop:` in meaning, #2 is a different control sequence, #2 is a digit, and #2 is something else (this last case is split further later). Despite the earlier f-expansion, #2 may still be expandable if it was protected by `\exp_not:N`, as may happen with the L^AT_EX 2_ε command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `__fp_parse_one_fp:NN` which deals with it robustly.

```

24098 \cs_new:Npn \__fp_parse_one:Nw #1 #2
24099   {
24100     \if_catcode:w \scan_stop: \exp_not:N #2
24101     \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
24102     \exp_after:wN \reverse_if:N
24103     \fi:
24104     \if_meaning:w \scan_stop: #2
24105     \exp_after:wN \exp_after:wN
24106     \exp_after:wN \__fp_parse_one_fp:NN
24107     \else:
24108     \exp_after:wN \exp_after:wN
24109     \exp_after:wN \__fp_parse_one_register:NN
24110     \fi:
24111     \else:
24112     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
24113     \exp_after:wN \exp_after:wN
24114     \exp_after:wN \__fp_parse_one_digit:NN
24115     \else:
24116     \exp_after:wN \exp_after:wN
24117     \exp_after:wN \__fp_parse_one_other:NN
24118     \fi:
24119     \fi:
24120     #1 #2
24121   }

```

(End of definition for __fp_parse_one:Nw.)

`__fp_parse_one_fp:NN` This function receives a `\precedence` and a control sequence equal to `\scan_stop:` in meaning. There are three cases.

`_fp_exp_after_expr_mark_f:nw`
`__fp_exp_after_?_f:nw`

- `\s__fp` starts a floating point number, and we call `__fp_exp_after_f:nw`, which f-expands after the floating point.
- `\s__fp_expr_mark` is a premature end, we call `__fp_exp_after_expr_mark_f:nw`, which triggers an fp-early-end error.
- For a control sequence not containing `\s__fp`, we call `__fp_exp_after_?_f:nw`, causing a bad-variable error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form `\s__fp_⟨type⟩` and defining `__fp_exp_after_⟨type⟩_f:nw`. In all cases, we make sure that the second argument of `__fp_parse_infix:NN` is correctly expanded. A special case only enabled in L^AT_EX 2_ε is that if `\protect` is encountered then

the error message mentions the control sequence which follows it rather than `\protect` itself. The test for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$ uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop`: hence “does not exist”.

```

24122 \cs_new:Npn \__fp_parse_one_fp:NN #1
24123 {
24124   \__fp_exp_after_any_f:nw
24125   {
24126     \exp_after:wN \__fp_parse_infix:NN
24127     \exp_after:wN #1 \exp:w \__fp_parse_expand:w
24128   }
24129 }
24130 \cs_new:Npn \__fp_exp_after_expr_mark_f:nw #1
24131 {
24132   \int_case:nnF { \exp_after:wN \use_i:nnn \use_none:nnn #1 }
24133   {
24134     \c__fp_prec_comma_int { }
24135     \c__fp_prec_tuple_int { }
24136     \c__fp_prec_end_int
24137     {
24138       \exp_after:wN \c__fp_empty_tuple_fp
24139       \exp:w \exp_end_continue_f:w
24140     }
24141   }
24142   {
24143     \msg_expandable_error:nn { fp } { early-end }
24144     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
24145   }
24146   #1
24147 }
24148 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
24149 {
24150   \msg_expandable_error:nnn { kernel } { bad-variable }
24151   {#2}
24152   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
24153 }
24154 \cs_set_protected:Npn \__fp_tmp:w #1
24155 {
24156   \cs_if_exist:NT #1
24157   {
24158     \cs_gset:cpn { __fp_exp_after_?_f:nw } ##1##2
24159     {
24160       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1
24161       \str_if_eq:nnTF {##2} { \protect }
24162       {
24163         \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
24164         {
24165           \msg_expandable_error:nnn { fp }
24166           { robust-cmd }
24167         }
24168       }
24169     }
24170     \msg_expandable_error:nnn { kernel }
24171     { bad-variable } {##2}
24172   }

```

```

24173     }
24174   }
24175 }
24176 \exp_args:Nc \__fp_tmp:w { @unexpandable@protect }

```

(End of definition for `__fp_parse_one_fp:NN`, `__fp_exp_after_expr_mark_f:nw`, and `__fp_exp_after_?_f:nw`.)

```

\__fp_parse_one_register:NN
  \__fp_parse_one_register_aux:Nw
  \__fp_parse_one_register_auxii:wwwNw
  \__fp_parse_one_register_int:www
  \__fp_parse_one_register_mu:www
  \__fp_parse_one_register_dim:ww

```

This is called whenever #2 is a control sequence other than `\scan_stop`: in meaning. We special-case `\wd`, `\ht`, `\dp` (see later) and otherwise assume that it is a register, but carefully unpack it with `\tex_the:D` within braces. First, we find the exponent following #2. Then we unpack #2 with `\tex_the:D`, and the `auxii` auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of `pt`. For integers, simply convert $\langle value \rangle e \langle exponent \rangle$ to a floating point number with `__fp_parse:n` (this is somewhat wasteful). For other registers, the decimal rounding provided by T_EX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with `\int_value:w \dim_to_decimal_in_sp:n { \langle decimal value \rangle pt }`, and use an auxiliary of `\dim_to_fp:n`, which performs the multiplication by 2^{-16} , correctly rounded.

```

24177 \cs_new:Npn \__fp_parse_one_register:NN #1#2
24178 {
24179   \exp_after:wN \__fp_parse_infix_after_operand:NwN
24180   \exp_after:wN #1
24181   \exp:w \exp_end_continue_f:w
24182   \__fp_parse_one_register_special:N #2
24183   \exp_after:wN \__fp_parse_one_register_aux:Nw
24184   \exp_after:wN #2
24185   \int_value:w
24186   \exp_after:wN \__fp_parse_exponent:N
24187   \exp:w \__fp_parse_expand:w
24188 }
24189 \cs_new:Npe \__fp_parse_one_register_aux:Nw #1
24190 {
24191   \exp_not:n
24192   {
24193     \exp_after:wN \use:nn
24194     \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
24195   }
24196   \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
24197   ; \exp_not:N \__fp_parse_one_register_dim:ww
24198   \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
24199   . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
24200   \s__fp_stop
24201 }
24202 \exp_args:Nno \use:nn
24203 { \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw #1 . #2 }
24204 { \tl_to_str:n { pt } #3 ; #4#5 \s__fp_stop }
24205 { #4 #1.#2; }
24206 \exp_args:Nno \use:nn
24207 { \cs_new:Npn \__fp_parse_one_register_mu:www #1 }
24208 { \tl_to_str:n { mu } ; #2 ; }
24209 { \__fp_parse_one_register_dim:ww #1 ; }
24210 \cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
24211 { \__fp_parse:n { #1 e #3 } }

```

```

24212 \cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
24213 {
24214   \exp_after:wN \__fp_from_dim_test:ww
24215   \int_value:w #2 \exp_after:wN ,
24216   \int_value:w \dim_to_decimal_in_sp:n { #1 pt } ;
24217 }

```

(End of definition for __fp_parse_one_register:NN and others.)

```

\__fp_parse_one_register_special:N
\__fp_parse_one_register_math:NNw
  \__fp_parse_one_register_wd:w
  \__fp_parse_one_register_wd:Nw

```

The \wd, \dp, \ht primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker e. Once that “exponent” is found, use \tex_the:D to find the box dimension and then copy what we did for dimensions.

```

24218 \cs_new:Npn \__fp_parse_one_register_special:N #1
24219 {
24220   \if_meaning:w \box_wd:N #1 \__fp_parse_one_register_wd:w \fi:
24221   \if_meaning:w \box_ht:N #1 \__fp_parse_one_register_wd:w \fi:
24222   \if_meaning:w \box_dp:N #1 \__fp_parse_one_register_wd:w \fi:
24223   \if_meaning:w \infty #1
24224     \__fp_parse_one_register_math:NNw \infty #1
24225   \fi:
24226   \if_meaning:w \pi #1
24227     \__fp_parse_one_register_math:NNw \pi #1
24228   \fi:
24229 }
24230 \cs_new:Npn \__fp_parse_one_register_math:NNw
24231   #1#2#3#4 \__fp_parse_expand:w
24232 {
24233   #3
24234   \str_if_eq:nnTF {#1} {#2}
24235   {
24236     \msg_expandable_error:nnn
24237     { fp } { infinity-pi } {#1}
24238     \c_nan_fp
24239   }
24240   { #4 \__fp_parse_expand:w }
24241 }
24242 \cs_new:Npn \__fp_parse_one_register_wd:w
24243   #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
24244 {
24245   #1
24246   \exp_after:wN \__fp_parse_one_register_wd:Nw
24247   #4 \__fp_parse_expand:w e
24248 }
24249 \cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
24250 {
24251   \exp_after:wN \__fp_from_dim_test:ww
24252   \exp_after:wN 0 \exp_after:wN ,
24253   \int_value:w \dim_to_decimal_in_sp:n { #1 #2 } ;
24254 }

```

(End of definition for __fp_parse_one_register_special:N and others.)

```

\__fp_parse_one_digit:NN

```

A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with __fp_sanitize:wN,

then `__fp_parse_infix_after_operand:NwN` expands `__fp_parse_infix:NN` after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.

```

24255 \cs_new:Npn \__fp_parse_one_digit:NN #1
24256   {
24257     \exp_after:wN \__fp_parse_infix_after_operand:NwN
24258     \exp_after:wN #1
24259     \exp:w \exp_end_continue_f:w
24260     \exp_after:wN \__fp_sanitize:wN
24261     \int_value:w \__fp_int_eval:w 0 \__fp_parse_trim_zeros:N
24262   }

```

(End of definition for `__fp_parse_one_digit:NN`.)

`__fp_parse_one_other:NN` For this function, #2 is a character token which is not a digit. If it is an ASCII letter, `__fp_parse_letters:N` beyond this one and give the result to `__fp_parse_word:Nw`. Otherwise, the character is assumed to be a prefix operator, and we build `__fp_parse_prefix_{operator}:Nw`.

```

24263 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
24264   {
24265     \if_int_compare:w
24266       \__fp_int_eval:w
24267       ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
24268       = 3 \exp_stop_f:
24269     \exp_after:wN \__fp_parse_word:Nw
24270     \exp_after:wN #1
24271     \exp_after:wN #2
24272     \exp:w \exp_after:wN \__fp_parse_letters:N
24273     \exp:w
24274   \else:
24275     \exp_after:wN \__fp_parse_prefix:NNN
24276     \exp_after:wN #1
24277     \exp_after:wN #2
24278     \cs:w
24279     __fp_parse_prefix_ \token_to_str:N #2 :Nw
24280     \exp_after:wN
24281     \cs_end:
24282     \exp:w
24283   \fi:
24284   \__fp_parse_expand:w
24285   }

```

(End of definition for `__fp_parse_one_other:NN`.)

`__fp_parse_word:Nw` Finding letters is a simple recursion. Once `__fp_parse_letters:N` has done its job, `__fp_parse_letters:N` we try to build a control sequence from the word #2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield `\c_nan_fp`, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires “inf” and “infinity” and “nan” to be recognized regardless of case, but we probably don’t want to allow every `l3fp` word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```

24286 \cs_new:Npn \__fp_parse_word:Nw #1#2;
24287 {
24288   \cs_if_exist_use:cF { __fp_parse_word_#2:N }
24289   {
24290     \cs_if_exist_use:cF
24291     { __fp_parse_caseless_ \str_casefold:n {#2} :N }
24292     {
24293       \msg_expandable_error:nnn
24294       { fp } { unknown-fp-word } {#2}
24295       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
24296       \__fp_parse_infix:NN
24297     }
24298   }
24299   #1
24300 }
24301 \cs_new:Npn \__fp_parse_letters:N #1
24302 {
24303   \exp_end_continue_f:w
24304   \if_int_compare:w
24305   \if_catcode:w \scan_stop: \exp_not:N #1
24306   0
24307   \else:
24308   \__fp_int_eval:w
24309   ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26
24310   \fi:
24311   = 3 \exp_stop_f:
24312   \exp_after:wN #1
24313   \exp:w \exp_after:wN \__fp_parse_letters:N
24314   \exp:w
24315   \else:
24316   \__fp_parse_return_semicolon:w #1
24317   \fi:
24318   \__fp_parse_expand:w
24319 }

```

(End of definition for __fp_parse_word:Nw and __fp_parse_letters:N.)

__fp_parse_prefix:NNN For this function, #1 is the previous *precedence*, #2 is the operator just seen, and #3 is
 __fp_parse_prefix_unknown:NNN a control sequence which implements the operator if it is a known operator. If this control
 sequence is \scan_stop:, then the operator is in fact unknown. Either the expression is
 missing a number there (if the operator is valid as an infix operator), and we put nan,
 wrapping the infix operator in a csname as appropriate, or the character is simply invalid
 in floating point expressions, and we continue looking for a number, starting again from
 __fp_parse_one:Nw.

```

24320 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
24321 {
24322   \if_meaning:w \scan_stop: #3
24323   \exp_after:wN \__fp_parse_prefix_unknown:NNN
24324   \exp_after:wN #2
24325   \fi:
24326   #3 #1
24327 }
24328 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
24329 {

```

```

24330 \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
24331 {
24332   \msg_expandable_error:nnn
24333     { fp } { missing-number } {#1}
24334   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
24335   \__fp_parse_infix:NN #3 #1
24336 }
24337 {
24338   \msg_expandable_error:nnn
24339     { fp } { unknown-symbol } {#1}
24340   \__fp_parse_one:Nw #3
24341 }
24342 }

```

(End of definition for `__fp_parse_prefix:NNN` and `__fp_parse_prefix_unknown:NNN`.)

71.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand ≥ 1 with the set of functions `__fp_parse_large...`; if it is a period, the significand is < 1 ; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift $\langle \text{exp}_1 \rangle < 0$, then read the significand with the set of functions `__fp_parse_small...`. Once the significand is read, read the exponent if `e` is present.

`__fp_parse_trim_zeros:N` This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call `__fp_parse_large:N` (the significand is ≥ 1); if it is `.`, then continue trimming zeros with `__fp_parse_strim_zeros:N`; otherwise, our number is exactly zero, and we call `__fp_parse_zero:` to take care of that case.

```

24343 \cs_new:Npn \__fp_parse_trim_zeros:N #1
24344 {
24345   \if:w 0 \exp_not:N #1
24346     \exp_after:wN \__fp_parse_trim_zeros:N
24347     \exp:w
24348   \else:
24349     \if:w . \exp_not:N #1
24350       \exp_after:wN \__fp_parse_strim_zeros:N
24351       \exp:w
24352     \else:
24353       \__fp_parse_trim_end:w #1
24354     \fi:
24355   \fi:
24356   \__fp_parse_expand:w
24357 }
24358 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
24359 {
24360   \fi:
24361   \fi:
24362   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
24363     \exp_after:wN \__fp_parse_large:N
24364   \else:
24365     \exp_after:wN \__fp_parse_zero:

```

```

24366     \fi:
24367     #1
24368   }

```

(End of definition for `_fp_parse_trim_zeros:N` and `_fp_parse_trim_end:w`.)

```

\_fp_parse_strim_zeros:N
\_fp_parse_strim_end:w

```

If we have removed all digits until a period (or if the body started with a period), then enter the “`small_trim`” loop which outputs `-1` for each removed 0. Those `-1` are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call `_fp_parse_small:N` (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for “small trim”.

```

24369 \cs_new:Npn \_fp_parse_strim_zeros:N #1
24370   {
24371     \if:w 0 \exp_not:N #1
24372       - 1
24373     \exp_after:wN \_fp_parse_strim_zeros:N \exp:w
24374     \else:
24375       \_fp_parse_strim_end:w #1
24376     \fi:
24377     \_fp_parse_expand:w
24378   }
24379 \cs_new:Npn \_fp_parse_strim_end:w #1 \fi: \_fp_parse_expand:w
24380   {
24381     \fi:
24382     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
24383       \exp_after:wN \_fp_parse_small:N
24384     \else:
24385       \exp_after:wN \_fp_parse_zero:
24386     \fi:
24387     #1
24388   }

```

(End of definition for `_fp_parse_strim_zeros:N` and `_fp_parse_strim_end:w`.)

`_fp_parse_zero:` After reading a significand of 0, find any exponent, then put a sign of 1 for `_fp_sanitize:wN`, which removes everything and leaves an exact zero.

```

24389 \cs_new:Npn \_fp_parse_zero:
24390   {
24391     \exp_after:wN ; \exp_after:wN 1
24392     \int_value:w \_fp_parse_exponent:N
24393   }

```

(End of definition for `_fp_parse_zero:.`)

71.4.2 Number: small significand

```

\_fp_parse_small:N

```

This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can’t do that all at once, because `\int_value:w` (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since `#1` is a digit, read seven more digits using `_fp_parse_digits_vii:N`. The `small_leading` auxiliary leaves those digits in the `\int_value:w`, and grabs some more, or stops if there are no more digits. Then the

`pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```

24394 \cs_new:Npn \__fp_parse_small:N #1
24395 {
24396   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
24397   \int_value:w \__fp_int_eval:w 1 \token_to_str:N #1
24398   \exp_after:wN \__fp_parse_small_leading:wwNN
24399   \int_value:w 1
24400   \exp_after:wN \__fp_parse_digits_vii:N
24401   \exp:w \__fp_parse_expand:w
24402 }

```

(End of definition for `__fp_parse_small:N`.)

```

\__fp_parse_small_leading:wwNN   \__fp_parse_small_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>

```

We leave *<digits>* *<zeros>* in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of zero (this shift is used in the case of a large significand). If #4 is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significand, then look for an exponent.

```

24403 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
24404 {
24405   #1 #2
24406   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
24407   \exp_after:wN 0
24408   \int_value:w \__fp_int_eval:w 1
24409   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
24410     \token_to_str:N #4
24411     \exp_after:wN \__fp_parse_small_trailing:wwNN
24412     \int_value:w 1
24413     \exp_after:wN \__fp_parse_digits_vi:N
24414     \exp:w
24415   \else:
24416     0000 0000 \__fp_parse_exponent:Nw #4
24417   \fi:
24418   \__fp_parse_expand:w
24419 }

```

(End of definition for `__fp_parse_small_leading:wwNN`.)

```

\__fp_parse_small_trailing:wwNN   \__fp_parse_small_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>
                                   <next token>

```

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the *<next token>* is a digit, it is the 16th digit, we keep it, then the `small_round` auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing, to +0 or to +1. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

24420 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
24421 {
24422   #1 #2
24423   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
24424     \token_to_str:N #4
24425     \exp_after:wN \__fp_parse_small_round:NN

```

```

24426     \exp_after:wN #4
24427     \exp:w
24428   \else:
24429     0 \__fp_parse_exponent:Nw #4
24430   \fi:
24431   \__fp_parse_expand:w
24432 }

```

(End of definition for `__fp_parse_small_trailing:wwNN`.)

```

\__fp_parse_pack_trailing:NNNNNww
\__fp_parse_pack_leading:NNNNNww
\__fp_parse_pack_carry:w

```

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+1 in the code below). If the leading digits propagate this carry all the way up, the function `__fp_parse_pack_carry:w` increments the exponent, and changes the significand from 0000... to 1000...: this is simple because such a carry can only occur to give rise to a power of 10.

```

24433 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNww #1 #2 #3#4#5#6 #7; #8 ;
24434 {
24435   \if_meaning:w 2 #2 + 1 \fi:
24436   ; #8 + #1 ; {#3#4#5#6} {#7};
24437 }
24438 \cs_new:Npn \__fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
24439 {
24440   + #7
24441   \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
24442   ; 0 {#2#3#4#5} {#6}
24443 }
24444 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
24445 { \fi: + 1 ; 0 {1000} }

```

(End of definition for `__fp_parse_pack_trailing:NNNNNww`, `__fp_parse_pack_leading:NNNNNww`, and `__fp_parse_pack_carry:w`.)

71.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

```
\__fp_parse_large:N
```

This function is followed by the first non-zero digit of a “large” significand (≥ 1). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```

24446 \cs_new:Npn \__fp_parse_large:N #1
24447 {
24448   \exp_after:wN \__fp_parse_large_leading:wwNN
24449   \int_value:w 1 \token_to_str:N #1
24450   \exp_after:wN \__fp_parse_digits_vii:N
24451   \exp:w \__fp_parse_expand:w
24452 }

```

(End of definition for `_fp_parse_large:N`.)

```
\_fp_parse_large_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>  
<next token>
```

We shift the exponent by the number of digits in #1, namely the target number, 8, minus the *<number of zeros>* (number of digits missing). Then prepare to pack the 8 first digits. If the *<next token>* is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the *<zeros>* to complete the 8 first digits, insert 8 more, and look for an exponent.

```
24453 \cs_new:Npn \_fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4  
24454 {  
24455   + \c\_fp_half_prec_int - #3  
24456   \exp_after:wN \_fp_parse_pack_leading:NNNNNww  
24457   \int_value:w \_fp_int_eval:w 1 #1  
24458   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:  
24459     \exp_after:wN \_fp_parse_large_trailing:wwNN  
24460     \int_value:w 1 \token_to_str:N #4  
24461     \exp_after:wN \_fp_parse_digits_vi:N  
24462     \exp:w  
24463   \else:  
24464     \if:w . \exp_not:N #4  
24465     \exp_after:wN \_fp_parse_small_leading:wwNN  
24466     \int_value:w 1  
24467     \cs:w  
24468       \_fp_parse_digits_  
24469       \_fp_int_to_roman:w #3  
24470       :N \exp_after:wN  
24471     \cs_end:  
24472     \exp:w  
24473   \else:  
24474     #2  
24475     \exp_after:wN \_fp_parse_pack_trailing:NNNNNww  
24476     \exp_after:wN 0  
24477     \int_value:w 1 0000 0000  
24478     \_fp_parse_exponent:Nw #4  
24479   \fi:  
24480 \fi:  
24481 \_fp_parse_expand:w  
24482 }
```

(End of definition for `_fp_parse_large_leading:wwNN`.)

```
\_fp_parse_large_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>  
<next token>
```

We have just read 15 digits. If the *<next token>* is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the `pack_trailing` function. We keep the *<digits>* and this 16-th digit, and find how this should be rounded using `_fp_parse_large_round:NN`. Otherwise, the exponent shift is the number of *<digits>*, 7 minus the *<number of zeros>*, and we test for a decimal point. This case happens in 123451234512345.67 with exactly 15 digits before the decimal separator.

Then branch to the appropriate small auxiliary, grabbing a few more digits to complement the digits we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the `<zeros>` and providing a 16-th digit of 0.

```

24483 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
24484 {
24485   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
24486     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
24487     \exp_after:wN \c__fp_half_prec_int
24488     \int_value:w \__fp_int_eval:w 1 #1 \token_to_str:N #4
24489     \exp_after:wN \__fp_parse_large_round:NN
24490     \exp_after:wN #4
24491     \exp:w
24492   \else:
24493     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
24494     \int_value:w \__fp_int_eval:w 7 - #3 \exp_stop_f:
24495     \int_value:w \__fp_int_eval:w 1 #1
24496     \if:w . \exp_not:N #4
24497       \exp_after:wN \__fp_parse_small_trailing:wwNN
24498       \int_value:w 1
24499       \cs:w
24500         __fp_parse_digits_
24501         \__fp_int_to_roman:w #3
24502         :N \exp_after:wN
24503         \cs_end:
24504         \exp:w
24505     \else:
24506       #2 0 \__fp_parse_exponent:Nw #4
24507     \fi:
24508   \fi:
24509   \__fp_parse_expand:w
24510 }

```

(End of definition for `__fp_parse_large_trailing:wwNN`.)

71.4.4 Number: beyond 16 digits, rounding

`__fp_parse_round_loop:N` This loop is called when rounding a number (whether the mantissa is small or large).
`__fp_parse_round_up:N` It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by `;0`, otherwise by `;1`. This is done by switching the loop to `round_up` at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

```

24511 \cs_new:Npn \__fp_parse_round_loop:N #1
24512 {
24513   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
24514     + 1
24515     \if:w 0 \token_to_str:N #1
24516       \exp_after:wN \__fp_parse_round_loop:N
24517       \exp:w
24518     \else:
24519       \exp_after:wN \__fp_parse_round_up:N
24520       \exp:w
24521     \fi:

```



```

24522 \else:
24523 \__fp_parse_return_semicolon:w 0 #1
24524 \fi:
24525 \__fp_parse_expand:w
24526 }
24527 \cs_new:Npn \__fp_parse_round_up:N #1
24528 {
24529 \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
24530 + 1
24531 \exp_after:wN \__fp_parse_round_up:N
24532 \exp:w
24533 \else:
24534 \__fp_parse_return_semicolon:w 1 #1
24535 \fi:
24536 \__fp_parse_expand:w
24537 }

```

(End of definition for `__fp_parse_round_loop:N` and `__fp_parse_round_up:N`.)

`__fp_parse_round_after:wN` After the loop `__fp_parse_round_loop:N`, this function fetches an exponent with `__fp_parse_exponent:N`, and combines it with the number of digits counted by `__fp_parse_round_loop:N`. At the same time, the result 0 or 1 is added to the surrounding integer expression.

```

24538 \cs_new:Npn \__fp_parse_round_after:wN #1; #2
24539 {
24540 + #2 \exp_after:wN ;
24541 \int_value:w \__fp_int_eval:w #1 + \__fp_parse_exponent:N
24542 }

```

(End of definition for `__fp_parse_round_after:wN`.)

`__fp_parse_small_round:NN`
`__fp_parse_round_after:wN` Here, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If #2 is not a digit, then fetch an exponent and expand to `;<exponent>` only. Otherwise, we expand to `+0` or `+1`, then `;<exponent>`. To decide which, call `__fp_round_s:NNNw` to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit #1 to round, the first following digit #2, and either `+0` or `+1` depending on whether the following digits are all zero or not. This last argument is obtained by `__fp_parse_round_loop:N`, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by `__fp_parse_round_after:wN`.

```

24543 \cs_new:Npn \__fp_parse_small_round:NN #1#2
24544 {
24545 \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
24546 +
24547 \exp_after:wN \__fp_round_s:NNNw
24548 \exp_after:wN 0
24549 \exp_after:wN #1
24550 \exp_after:wN #2
24551 \int_value:w \__fp_int_eval:w
24552 \exp_after:wN \__fp_parse_round_after:wN
24553 \int_value:w \__fp_int_eval:w 0 * \__fp_int_eval:w 0
24554 \exp_after:wN \__fp_parse_round_loop:N
24555 \exp:w
24556 \else:

```

```

24557     \__fp_parse_exponent:Nw #2
24558     \fi:
24559     \__fp_parse_expand:w
24560   }

```

(End of definition for __fp_parse_small_round:NN and __fp_parse_round_after:wN.)

```

\__fp_parse_large_round:NN
  \__fp_parse_large_round_test:NN
  \__fp_parse_large_round_aux:wNN

```

Large numbers are harder to round, as there may be a period in the way. Again, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (#2 is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (#2 is a digit), then round, checking with __fp_parse_round_loop:N if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the aux function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```

24561 \cs_new:Npn \__fp_parse_large_round:NN #1#2
24562   {
24563     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
24564     +
24565     \exp_after:wN \__fp_round_s:NNNw
24566     \exp_after:wN 0
24567     \exp_after:wN #1
24568     \exp_after:wN #2
24569     \int_value:w \__fp_int_eval:w
24570     \exp_after:wN \__fp_parse_large_round_aux:wNN
24571     \int_value:w \__fp_int_eval:w 1
24572     \exp_after:wN \__fp_parse_round_loop:N
24573     \else: %^^A could be dot, or e, or other
24574     \exp_after:wN \__fp_parse_large_round_test:NN
24575     \exp_after:wN #1
24576     \exp_after:wN #2
24577     \fi:
24578   }
24579 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
24580   {
24581     \if:w . \exp_not:N #2
24582     \exp_after:wN \__fp_parse_small_round:NN
24583     \exp_after:wN #1
24584     \exp:w
24585     \else:
24586     \__fp_parse_exponent:Nw #2
24587     \fi:
24588     \__fp_parse_expand:w
24589   }
24590 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
24591   {
24592     + #2
24593     \exp_after:wN \__fp_parse_round_after:wN
24594     \int_value:w \__fp_int_eval:w #1
24595     \if:w . \exp_not:N #3
24596     + 0 * \__fp_int_eval:w 0
24597     \exp_after:wN \__fp_parse_round_loop:N
24598     \exp:w \exp_after:wN \__fp_parse_expand:w

```

```

24599     \else:
24600         \exp_after:wN ;
24601         \exp_after:wN 0
24602         \exp_after:wN #3
24603     \fi:
24604 }

```

(End of definition for `_fp_parse_large_round:NN`, `_fp_parse_large_round_test:NN`, and `_fp_parse_large_round_aux:wNN`.)

71.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```

\_fp_parse:n { 3.2 erf(0.1) }
\_fp_parse:n { 3.2 e\l_my_int }
\_fp_parse:n { 3.2 \c_pi_fp }

```

The first case indicates that just looking one character ahead for an “e” is not enough, since we would mistake the function `erf` for an exponent of “`rf`”. An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after `e`. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in the course of reading `3.2`, `\c_pi_fp` is expanded to `\s__fp_fp_chk:w 1 0 {-1} {3141}` `...`; and `\s__fp` stops the expansion. Expanding two tokens ahead would then force the expansion of `_fp_chk:w` (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as `TeX` does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is `e`. Then we expand (and possibly unpack) the second token.

`_fp_parse_exponent:Nw` This auxiliary is convenient to smuggle some material through `\fi:` ending conditional processing. We place those `\fi:` (argument #2) at a very odd place because this allows us to insert `_fp_int_eval:w ...` there if needed.

```

24605 \cs_new:Npn \_fp_parse_exponent:Nw #1 #2 \_fp_parse_expand:w
24606 {
24607     \exp_after:wN ;
24608     \int_value:w #2 \_fp_parse_exponent:N #1
24609 }

```

(End of definition for `_fp_parse_exponent:Nw`.)

`_fp_parse_exponent:N`
`_fp_parse_exponent_aux:NN` This function should be called within an `\int_value:w` expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no `e` (or `E`), leave an exponent of 0. If there is an `e` or `E`, expand the next token to run some tests on it. The first rough test is that if the character code of #1 is greater than that of 9 (largest code valid for an exponent, less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```

24610 \cs_new:Npn \_fp_parse_exponent:N #1

```

```

24611 {
24612 \if:w e \if:w E \exp_not:N #1 e \else: \exp_not:N #1 \fi:
24613 \exp_after:wN \__fp_parse_exponent_aux:NN
24614 \exp_after:wN #1
24615 \exp:w
24616 \else:
24617 0 \__fp_parse_return_semicolon:w #1
24618 \fi:
24619 \__fp_parse_expand:w
24620 }
24621 \cs_new:Npn \__fp_parse_exponent_aux:NN #1#2
24622 {
24623 \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #2
24624 0 \else: '#2 \fi: > '9 \exp_stop_f:
24625 0 \exp_after:wN ; \exp_after:wN #1
24626 \else:
24627 \exp_after:wN \__fp_parse_exponent_sign:N
24628 \fi:
24629 #2
24630 }

```

(End of definition for __fp_parse_exponent:N and __fp_parse_exponent_aux:NN.)

__fp_parse_exponent_sign:N Read signs one by one (if there is any).

```

24631 \cs_new:Npn \__fp_parse_exponent_sign:N #1
24632 {
24633 \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
24634 \exp_after:wN \__fp_parse_exponent_sign:N
24635 \exp:w \exp_after:wN \__fp_parse_expand:w
24636 \else:
24637 \exp_after:wN \__fp_parse_exponent_body:N
24638 \exp_after:wN #1
24639 \fi:
24640 }

```

(End of definition for __fp_parse_exponent_sign:N.)

__fp_parse_exponent_body:N An exponent can be an explicit integer (most common case), or various other things (most of which are invalid).

```

24641 \cs_new:Npn \__fp_parse_exponent_body:N #1
24642 {
24643 \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
24644 \token_to_str:N #1
24645 \exp_after:wN \__fp_parse_exponent_digits:N
24646 \exp:w
24647 \else:
24648 \__fp_parse_exponent_keep:NTF #1
24649 { \__fp_parse_return_semicolon:w #1 }
24650 {
24651 \exp_after:wN ;
24652 \exp:w
24653 }
24654 \fi:
24655 \__fp_parse_expand:w
24656 }

```

(End of definition for `__fp_parse_exponent_body:N`.)

`__fp_parse_exponent_digits:N` Read digits one by one, and leave them behind in the input stream. When finding a non-digit, stop, and insert a semicolon. Note that we do not check for overflow of the exponent, hence there can be a TeX error. It is mostly harmless, except when parsing `0e9876543210`, which should be a valid representation of 0, but is not.

```
24657 \cs_new:Npn \__fp_parse_exponent_digits:N #1
24658   {
24659     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
24660       \token_to_str:N #1
24661       \exp_after:wN \__fp_parse_exponent_digits:N
24662       \exp:w
24663     \else:
24664       \__fp_parse_return_semicolon:w #1
24665     \fi:
24666     \__fp_parse_expand:w
24667   }
```

(End of definition for `__fp_parse_exponent_digits:N`.)

`__fp_parse_exponent_keep:NTF` This is the last building block for parsing exponents. The argument `#1` is already fully expanded, and neither `+` nor `-` nor a digit. It can be:

- `\s__fp`, marking the start of an internal floating point, invalid here;
- another control sequence equal to `\relax`, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than `+`, `-` or digits, again, an error.

```
24668 \prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
24669   {
24670     \if_catcode:w \scan_stop: \exp_not:N #1
24671     \if_meaning:w \scan_stop: #1
24672       \if:w 0 \__fp_str_if_eq:nn { \s__fp } { \exp_not:N #1 }
24673       0
24674       \msg_expandable_error:nnn
24675         { fp } { after-e } { floating-point~ }
24676       \prg_return_true:
24677     \else:
24678       0
24679       \msg_expandable_error:nnn
24680         { kernel } { bad-variable } { #1 }
24681       \prg_return_false:
24682     \fi:
24683   \else:
24684     \if:w 0 \__fp_str_if_eq:nn { \int_value:w #1 } { \tex_the:D #1 }
24685     \int_value:w #1
24686   \else:
24687     0
24688     \msg_expandable_error:nnn
24689       { fp } { after-e } { dimension~#1 }
24690   \fi:
24691   \prg_return_false:
```

```

24692     \fi:
24693 \else:
24694     0
24695     \msg_expandable_error:nmn
24696     { fp } { missing } { exponent }
24697     \prg_return_true:
24698 \fi:
24699 }

```

(End of definition for `__fp_parse_exponent_keep:NTF`.)

71.5 Constants, functions and prefix operators

71.5.1 Prefix operators

`__fp_parse_prefix_+:Nw` A unary `+` does nothing: we should continue looking for a number.

```

24700 \cs_new_eq:cN { __fp_parse_prefix_+:Nw } \__fp_parse_one:Nw

```

(End of definition for `__fp_parse_prefix_+:Nw`.)

`__fp_parse_apply_function:NNwN` Here, `#1` is a precedence, `#2` is some extra data used by some functions, `#3` is *e.g.*, `__fp_sin_o:w`, and expands once after the calculation, `#4` is the operand, and `#5` is a `__fp_parse_infix_...:N` function. We feed the data `#2`, and the argument `#4`, to the function `#3`, which expands `\exp:w` thus the infix function `#5`.

```

24701 \cs_new:Npn \__fp_parse_apply_function:NNwN #1#2#3#4#5
24702 {
24703     #3 #2 #4 @
24704     \exp:w \exp_end_continue_f:w #5 #1
24705 }

```

(End of definition for `__fp_parse_apply_function:NNwN`.)

`__fp_parse_apply_unary:NNwN`
`__fp_parse_apply_unary_chk:NwNw`
`__fp_parse_apply_unary_chk:nNNw`
`__fp_parse_apply_unary_type:NNN`
`__fp_parse_apply_unary_error:NNw`

In contrast to `__fp_parse_apply_function:NNwN`, this checks that the operand `#4` is a single argument (namely there is a single `;`). We use the fact that any floating point starts with a “safe” token like `\s__fp`. If there is no argument produce the `fp-no-arg` error; if there are at least two produce `fp-multi-arg`. For the error message extract the mathematical function name (such as `sin`) from the `expl3` function that computes it, such as `__fp_sin_o:w`.

In addition, since there is a single argument we can dispatch on type and check that the resulting function exists. This catches things like `sin((1,2))` where it does not make sense to take the sine of a tuple.

```

24706 \cs_new:Npn \__fp_parse_apply_unary:NNwN #1#2#3#4#5
24707 {
24708     \__fp_parse_apply_unary_chk:NwNw #4 @ ; . \s__fp_stop
24709     \__fp_parse_apply_unary_type:NNN
24710     #3 #2 #4 @
24711     \exp:w \exp_end_continue_f:w #5 #1
24712 }
24713 \cs_new:Npn \__fp_parse_apply_unary_chk:NwNw #1#2 ; #3#4 \s__fp_stop
24714 {
24715     \if_meaning:w @ #3 \else:
24716         \token_if_eq_meaning:NNTF . #3
24717         { \__fp_parse_apply_unary_chk:nNNNw { no } }

```

```

24718         { \_fp_parse_apply_unary_chk:nNNNNw { multi } }
24719     \fi:
24720 }
24721 \cs_new:Npn \_fp_parse_apply_unary_chk:nNNNNw #1#2#3#4#5#6 @
24722 {
24723     #2
24724     \_fp_error:nffn { #1-arg } { \_fp_func_to_name:N #4 } { } { }
24725     \exp_after:wN #4 \exp_after:wN #5 \c_nan_fp @
24726 }
24727 \cs_new:Npn \_fp_parse_apply_unary_type:NNN #1#2#3
24728 {
24729     \_fp_change_func_type:NNN #3 #1 \_fp_parse_apply_unary_error:NNw
24730     #2 #3
24731 }
24732 \cs_new:Npn \_fp_parse_apply_unary_error:NNw #1#2#3 @
24733 { \_fp_invalid_operation_o:fw { \_fp_func_to_name:N #1 } #3 }

```

(End of definition for _fp_parse_apply_unary:NNNwN and others.)

_fp_parse_prefix_-:Nw
_fp_parse_prefix_!:Nw

The unary - and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence ##1 and the precedence \c__fp_prec_not_int of the unary operator, then call the appropriate _fp_<operation>_o:w function, where the <operation> is set_sign or not.

```

24734 \cs_set_protected:Npn \_fp_tmp:w #1#2#3#4
24735 {
24736     \cs_new:cpn { \_fp_parse_prefix_ #1 :Nw } ##1
24737     {
24738         \exp_after:wN \_fp_parse_apply_unary:NNNwN
24739         \exp_after:wN ##1
24740         \exp_after:wN #4
24741         \exp_after:wN #3
24742         \exp:w
24743         \if_int_compare:w #2 < ##1
24744             \_fp_parse_operand:Nw ##1
24745         \else:
24746             \_fp_parse_operand:Nw #2
24747         \fi:
24748         \_fp_parse_expand:w
24749     }
24750 }
24751 \_fp_tmp:w - \c__fp_prec_not_int \_fp_set_sign_o:w 2
24752 \_fp_tmp:w ! \c__fp_prec_not_int \_fp_not_o:w ?

```

(End of definition for _fp_parse_prefix_-:Nw and _fp_parse_prefix_!:Nw.)

_fp_parse_prefix_:Nw

Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to _fp_parse_one_digit:NN but calls _fp_parse_trim_zeros:N to trim zeros after the decimal point, rather than the trim_zeros function for zeros before the decimal point.

```

24753 \cs_new:cpn { \_fp_parse_prefix_:Nw } #1
24754 {
24755     \exp_after:wN \_fp_parse_infix_after_operand:NwN
24756     \exp_after:wN #1

```

```

24757 \exp:w \exp_end_continue_f:w
24758 \exp_after:wN \__fp_sanitize:wN
24759 \int_value:w \__fp_int_eval:w 0 \__fp_parse_strim_zeros:N
24760 }

```

(End of definition for __fp_parse_prefix_.:Nw.)

```

\__fp_parse_prefix_(:Nw
\__fp_parse_lparen_after:NwN

```

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. If the previous precedence is \c__fp_prec_func_int we are parsing arguments of a function and commas should not build tuples; otherwise commas should build tuples. We distinguish these cases by precedence: \c__fp_prec_comma_int for the case of arguments, \c__fp_prec_tuple_int for the case of tuples. Once the operand is found, the lparen_after auxiliary makes sure that there was a closing parenthesis (otherwise it complains), and leaves in the input stream an operand, fetching the following infix operator.

```

24761 \cs_new:cpn { \__fp_parse_prefix_(:Nw } #1
24762 {
24763   \exp_after:wN \__fp_parse_lparen_after:NwN
24764   \exp_after:wN #1
24765   \exp:w
24766   \if_int_compare:w #1 = \c__fp_prec_func_int
24767     \__fp_parse_operand:Nw \c__fp_prec_comma_int
24768   \else:
24769     \__fp_parse_operand:Nw \c__fp_prec_tuple_int
24770   \fi:
24771   \__fp_parse_expand:w
24772 }
24773 \cs_new:Npe \__fp_parse_lparen_after:NwN #1#2 @ #3
24774 {
24775   \exp_not:N \token_if_eq_meaning:NNTF #3
24776   \exp_not:c { \__fp_parse_infix_):N }
24777   {
24778     \exp_not:N \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
24779     \exp_not:N \exp_after:wN
24780     \exp_not:N \__fp_parse_infix_after_paren:NN
24781     \exp_not:N \exp_after:wN #1
24782     \exp_not:N \exp:w
24783     \exp_not:N \__fp_parse_expand:w
24784   }
24785   {
24786     \exp_not:N \msg_expandable_error:nnn
24787     { fp } { missing } { ) }
24788     \exp_not:N \tl_if_empty:nT {#2} \exp_not:N \c__fp_empty_tuple_fp
24789     #2 @
24790     \exp_not:N \use_none:n #3
24791   }
24792 }

```

(End of definition for __fp_parse_prefix_(:Nw and __fp_parse_lparen_after:NwN.)

```

\__fp_parse_prefix_):Nw

```

The right parenthesis can appear as a prefix in two similar cases: in an empty tuple or tuple ending with a comma, or in an empty argument list or argument list ending with a comma, such as in `max(1,2,)` or in `rand()`.

```

24793 \cs_new:cpn { \__fp_parse_prefix_):Nw } #1

```



```

24794 {
24795   \if_int_compare:w #1 = \c__fp_prec_comma_int
24796   \else:
24797     \if_int_compare:w #1 = \c__fp_prec_tuple_int
24798     \exp_after:wN \c__fp_empty_tuple_fp \exp:w
24799     \else:
24800       \msg_expandable_error:nnn
24801         { fp } { missing-number } { ) }
24802       \exp_after:wN \c_nan_fp \exp:w
24803       \fi:
24804       \exp_end_continue_f:w
24805     \fi:
24806     \__fp_parse_infix_after_paren:NN #1 )
24807   }

```

(End of definition for __fp_parse_prefix_):Nw.)

71.5.2 Constants

__fp_parse_word_inf:N Some words correspond to constant floating points. The floating point constant is left as a result of __fp_parse_one:Nw after expanding __fp_parse_infix:NN.

```

\__fp_parse_word_inf:N
\__fp_parse_word_nan:N
\__fp_parse_word_pi:N
\__fp_parse_word_deg:N
\__fp_parse_word_true:N
\__fp_parse_word_false:N
24808 \cs_set_protected:Npn \__fp_tmp:w #1 #2
24809 {
24810   \cs_new:cpn { __fp_parse_word_#1:N }
24811     { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
24812 }
24813 \__fp_tmp:w { inf } \c_inf_fp
24814 \__fp_tmp:w { nan } \c_nan_fp
24815 \__fp_tmp:w { pi } \c_pi_fp
24816 \__fp_tmp:w { deg } \c_one_degree_fp
24817 \__fp_tmp:w { true } \c_one_fp
24818 \__fp_tmp:w { false } \c_zero_fp

```

(End of definition for __fp_parse_word_inf:N and others.)

__fp_parse_caseless_inf:N Copies of __fp_parse_word_...:N commands, to allow arbitrary case as mandated by the standard.

```

\__fp_parse_caseless_inf:N
\__fp_parse_caseless_infinity:N
\__fp_parse_caseless_nan:N
24819 \cs_new_eq:NN \__fp_parse_caseless_inf:N \__fp_parse_word_inf:N
24820 \cs_new_eq:NN \__fp_parse_caseless_infinity:N \__fp_parse_word_inf:N
24821 \cs_new_eq:NN \__fp_parse_caseless_nan:N \__fp_parse_word_nan:N

```

(End of definition for __fp_parse_caseless_inf:N, __fp_parse_caseless_infinity:N, and __fp_parse_caseless_nan:N.)

__fp_parse_word_pt:N Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```

\__fp_parse_word_in:N
\__fp_parse_word_pc:N
\__fp_parse_word_cm:N
\__fp_parse_word_mm:N
\__fp_parse_word_dd:N
\__fp_parse_word_cc:N
\__fp_parse_word_nd:N
\__fp_parse_word_nc:N
\__fp_parse_word_bp:N
\__fp_parse_word_sp:N
24822 \cs_set_protected:Npn \__fp_tmp:w #1 #2
24823 {
24824   \cs_new:cpn { __fp_parse_word_#1:N }
24825     {
24826       \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
24827       \s__fp \__fp_chk:w 10 #2 ;
24828     }
24829 }

```

```

24830 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
24831 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
24832 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
24833 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
24834 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
24835 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
24836 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
24837 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
24838 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
24839 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
24840 \__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }

```

(End of definition for __fp_parse_word_pt:N and others.)

__fp_parse_word_em:N The font-dependent units em and ex must be evaluated on the fly. We reuse an auxiliary
 __fp_parse_word_ex:N of \dim_to_fp:n.

```

24841 \tl_map_inline:nn { {em} {ex} }
24842 {
24843   \cs_new:cpn { __fp_parse_word_#1:N }
24844   {
24845     \exp_after:wN \__fp_from_dim_test:ww
24846     \exp_after:wN 0 \exp_after:wN ,
24847     \int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN ;
24848     \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
24849   }
24850 }

```

(End of definition for __fp_parse_word_em:N and __fp_parse_word_ex:N.)

71.5.3 Functions

```

\__fp_parse_unary_function:NNN
\__fp_parse_function:NNN
24851 \cs_new:Npn \__fp_parse_unary_function:NNN #1#2#3
24852 {
24853   \exp_after:wN \__fp_parse_apply_unary:NNNwN
24854   \exp_after:wN #3
24855   \exp_after:wN #2
24856   \exp_after:wN #1
24857   \exp:w
24858   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
24859 }
24860 \cs_new:Npn \__fp_parse_function:NNN #1#2#3
24861 {
24862   \exp_after:wN \__fp_parse_apply_function:NNNwN
24863   \exp_after:wN #3
24864   \exp_after:wN #2
24865   \exp_after:wN #1
24866   \exp:w
24867   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
24868 }

```

(End of definition for __fp_parse_unary_function:NNN and __fp_parse_function:NNN.)

71.6 Main functions

`_fp_parse:n` Start an `\exp:w` expansion so that `_fp_parse:n` expands in two steps. The `_fp_parse_operand:Nw` function performs computations until reaching an operation with precedence `\c_fp_prec_end_int` or less, namely, the end of the expression. The marker `\s_fp_expr_mark` indicates that the next token is an already parsed version of an infix operator, and `_fp_parse_infix_end:N` has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with `\exp_end:.`

```

24869 \cs_new:Npn \_fp_parse:n #1
24870   {
24871     \exp:w
24872     \exp_after:wN \_fp_parse_after:ww
24873     \exp:w
24874     \_fp_parse_operand:Nw \c\_fp_prec_end_int
24875     \_fp_parse_expand:w #1
24876     \s\_fp_expr_mark \_fp_parse_infix_end:N
24877     \s\_fp_expr_stop
24878     \exp_end:
24879   }
24880 \cs_new:Npn \_fp_parse_after:ww
24881   #1@ \_fp_parse_infix_end:N \s\_fp_expr_stop #2 { #2 #1 }
24882 \cs_new:Npn \_fp_parse_o:n #1
24883   {
24884     \exp:w
24885     \exp_after:wN \_fp_parse_after:ww
24886     \exp:w
24887     \_fp_parse_operand:Nw \c\_fp_prec_end_int
24888     \_fp_parse_expand:w #1
24889     \s\_fp_expr_mark \_fp_parse_infix_end:N
24890     \s\_fp_expr_stop
24891     {
24892       \exp_end_continue_f:w
24893       \_fp_exp_after_any_f:nw { \exp_after:wN \exp_stop_f: }
24894     }
24895   }

```

(End of definition for `_fp_parse:n`, `_fp_parse_o:n`, and `_fp_parse_after:ww`.)

`_fp_parse_operand:Nw` This is just a shorthand which sets up both `_fp_parse_continue:NwN` and `_fp_parse_one:Nw` with the same precedence. Note the trailing `\exp:w`.

```

24896 \cs_new:Npn \_fp_parse_operand:Nw #1
24897   {
24898     \exp_end_continue_f:w
24899     \exp_after:wN \_fp_parse_continue:NwN
24900     \exp_after:wN #1
24901     \exp:w \exp_end_continue_f:w
24902     \exp_after:wN \_fp_parse_one:Nw
24903     \exp_after:wN #1
24904     \exp:w
24905   }
24906 \cs_new:Npn \_fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }

```

(End of definition for `_fp_parse_operand:Nw` and `_fp_parse_continue:NwN`.)

`_fp_parse_apply_binary:NwNwN` Receives $\langle precedence \rangle \langle operand_1 \rangle @ \langle operation \rangle \langle operand_2 \rangle @ \langle infix command \rangle$.
`_fp_parse_apply_binary_chk:NN` Builds the appropriate call to the $\langle operation \rangle$ #3, dispatching on both types. If the
`_fp_parse_apply_binary_error:NNN` resulting control sequence does not exist, the operation is not allowed.

This is redefined in l3fp-extras.

```

24907 \cs_new:Npn \_fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
24908 {
24909   \exp_after:wN \_fp_parse_continue:NwN
24910   \exp_after:wN #1
24911   \exp:w \exp_end_continue_f:w
24912   \exp_after:wN \_fp_parse_apply_binary_chk:NN
24913   \cs:w
24914     __fp
24915     \_fp_type_from_scan:N #2
24916     _#4
24917     \_fp_type_from_scan:N #5
24918     _o:ww
24919   \cs_end:
24920   #4
24921   #2#3 #5#6
24922   \exp:w \exp_end_continue_f:w #7 #1
24923 }
24924 \cs_new:Npn \_fp_parse_apply_binary_chk:NN #1#2
24925 {
24926   \if_meaning:w \scan_stop: #1
24927   \_fp_parse_apply_binary_error:NNN #2
24928   \fi:
24929   #1
24930 }
24931 \cs_new:Npn \_fp_parse_apply_binary_error:NNN #1#2#3
24932 {
24933   #2
24934   \_fp_invalid_operation_o:Nww #1
24935 }
  
```

(End of definition for `_fp_parse_apply_binary:NwNwN`, `_fp_parse_apply_binary_chk:NN`, and `_fp_parse_apply_binary_error:NNN`.)

`_fp_binary_type_o:Nww` Applies the operator #1 to its two arguments, dispatching according to their types, and
`_fp_binary_rev_type_o:Nww` expands once after the result. The rev version swaps its arguments before doing this.

```

24936 \cs_new:Npn \_fp_binary_type_o:Nww #1 #2#3 ; #4
24937 {
24938   \exp_after:wN \_fp_parse_apply_binary_chk:NN
24939   \cs:w
24940     __fp
24941     \_fp_type_from_scan:N #2
24942     _#1
24943     \_fp_type_from_scan:N #4
24944     _o:ww
24945   \cs_end:
24946   #1
24947   #2 #3 ; #4
24948 }
24949 \cs_new:Npn \_fp_binary_rev_type_o:Nww #1 #2#3 ; #4#5 ;
24950 {
  
```

```

24951 \exp_after:wN \_fp_parse_apply_binary_chk:NN
24952 \cs:w
24953 \_fp
24954 \_fp_type_from_scan:N #4
24955 - #1
24956 \_fp_type_from_scan:N #2
24957 _o:ww
24958 \cs_end:
24959 #1
24960 #4 #5 ; #2 #3 ;
24961 }

```

(End of definition for _fp_binary_type_o:Nww and _fp_binary_rev_type_o:Nww.)

71.7 Infix operators

_fp_parse_infix_after_operand:NwN

```

24962 \cs_new:Npn \_fp_parse_infix_after_operand:NwN #1 #2;
24963 {
24964 \_fp_exp_after_f:nw { \_fp_parse_infix:NN #1 }
24965 #2;
24966 }
24967 \cs_new:Npn \_fp_parse_infix:NN #1 #2
24968 {
24969 \if_catcode:w \scan_stop: \exp_not:N #2
24970 \if:w 0 \_fp_str_if_eq:nn { \s_fp_expr_mark } { \exp_not:N #2 }
24971 \exp_after:wN \exp_after:wN
24972 \exp_after:wN \_fp_parse_infix_mark:NNN
24973 \else:
24974 \exp_after:wN \exp_after:wN
24975 \exp_after:wN \_fp_parse_infix_juxt:N
24976 \fi:
24977 \else:
24978 \if_int_compare:w
24979 \_fp_int_eval:w
24980 ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
24981 = 3 \exp_stop_f:
24982 \exp_after:wN \exp_after:wN
24983 \exp_after:wN \_fp_parse_infix_juxt:N
24984 \else:
24985 \exp_after:wN \_fp_parse_infix_check:NNN
24986 \cs:w
24987 \_fp_parse_infix_ \token_to_str:N #2 :N
24988 \exp_after:wN \exp_after:wN \exp_after:wN
24989 \cs_end:
24990 \fi:
24991 \fi:
24992 #1
24993 #2
24994 }
24995 \cs_new:Npn \_fp_parse_infix_check:NNN #1#2#3
24996 {
24997 \if_meaning:w \scan_stop: #1

```

```

24998     \msg_expandable_error:nnn
24999     { fp } { missing } { * }
25000     \exp_after:wN \__fp_parse_infix_mul:N
25001     \exp_after:wN #2
25002     \exp_after:wN #3
25003     \else:
25004     \exp_after:wN #1
25005     \exp_after:wN #2
25006     \exp:w \exp_after:wN \__fp_parse_expand:w
25007     \fi:
25008     }

```

(End of definition for __fp_parse_infix_after_operand:NwN.)

__fp_parse_infix_after_paren:NN Variant of __fp_parse_infix:NN for use after a closing parenthesis. The only difference is that __fp_parse_infix_juxt:N is replaced by __fp_parse_infix_mul:N.

```

25009 \cs_new:Npn \__fp_parse_infix_after_paren:NN #1 #2
25010 {
25011   \if_catcode:w \scan_stop: \exp_not:N #2
25012   \if:w 0 \__fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
25013   \exp_after:wN \exp_after:wN
25014   \exp_after:wN \__fp_parse_infix_mark:NNN
25015   \else:
25016   \exp_after:wN \exp_after:wN
25017   \exp_after:wN \__fp_parse_infix_mul:N
25018   \fi:
25019   \else:
25020   \if_int_compare:w
25021     \__fp_int_eval:w
25022     ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
25023     = 3 \exp_stop_f:
25024     \exp_after:wN \exp_after:wN
25025     \exp_after:wN \__fp_parse_infix_mul:N
25026   \else:
25027     \exp_after:wN \__fp_parse_infix_check:NNN
25028     \cs:w
25029     __fp_parse_infix_ \token_to_str:N #2 :N
25030     \exp_after:wN \exp_after:wN \exp_after:wN
25031     \cs_end:
25032     \fi:
25033   \fi:
25034   #1
25035   #2
25036 }

```

(End of definition for __fp_parse_infix_after_paren:NN.)

71.7.1 Closing parentheses and commas

__fp_parse_infix_mark:NNN As an infix operator, \s__fp_expr_mark means that the next token (#3) has already gone through __fp_parse_infix:NN and should be provided the precedence #1. The scan mark #2 is discarded.

```

25037 \cs_new:Npn \__fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }

```

(End of definition for `__fp_parse_infix_mark:NNN`.)

`__fp_parse_infix_end:N` This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```
25038 \cs_new:Npn \__fp_parse_infix_end:N #1
25039   { @ \use_none:n \__fp_parse_infix_end:N }
```

(End of definition for `__fp_parse_infix_end:N`.)

`__fp_parse_infix_):N` This is very similar to `__fp_parse_infix_end:N`, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression, with precedence `\c__fp_prec_end_int`.

```
25040 \cs_set_protected:Npn \__fp_tmp:w #1
25041   {
25042     \cs_new:Npn #1 ##1
25043       {
25044         \if_int_compare:w ##1 > \c__fp_prec_end_int
25045           \exp_after:wN @
25046           \exp_after:wN \use_none:n
25047           \exp_after:wN #1
25048         \else:
25049           \msg_expandable_error:nnn { fp } { extra } { ) }
25050           \exp_after:wN \__fp_parse_infix:NN
25051           \exp_after:wN ##1
25052           \exp:w \exp_after:wN \__fp_parse_expand:w
25053         \fi:
25054       }
25055   }
25056 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_):N }
```

(End of definition for `__fp_parse_infix_):N`.)

`__fp_parse_infix_,:N` As for other infix operations, if the previous operations has higher precedence the comma waits. Otherwise we call `__fp_parse_operand:Nw` to read more comma-delimited arguments that `__fp_parse_infix_comma:w` simply concatenates into a `@`-delimited array. `__fp_parse_infix_comma:w` The first comma in a tuple that is not a function argument is distinguished: in that case call `__fp_parse_apply_comma:NwNwN` whose job is to convert the first item of the tuple and an array of the remaining items into a tuple. In contrast to `__fp_parse_apply_binary:NwNwN` this function's operands are not single-object arrays.

```
25057 \cs_set_protected:Npn \__fp_tmp:w #1
25058   {
25059     \cs_new:Npn #1 ##1
25060       {
25061         \if_int_compare:w ##1 > \c__fp_prec_comma_int
25062           \exp_after:wN @
25063           \exp_after:wN \use_none:n
25064           \exp_after:wN #1
25065         \else:
25066           \if_int_compare:w ##1 < \c__fp_prec_comma_int
25067             \exp_after:wN @
25068             \exp_after:wN \__fp_parse_apply_comma:NwNwN
25069             \exp_after:wN ,
25070             \exp:w
25071           \else:
```

```

25072         \exp_after:wN \__fp_parse_infix_comma:w
25073         \exp:w
25074         \fi:
25075         \__fp_parse_operand:Nw \c__fp_prec_comma_int
25076         \exp_after:wN \__fp_parse_expand:w
25077     \fi:
25078 }
25079 }
25080 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_,:N }
25081 \cs_new:Npn \__fp_parse_infix_comma:w #1 @
25082 { #1 @ \use_none:n }
25083 \cs_new:Npn \__fp_parse_apply_comma:NwNwN #1 #2@ #3 #4@ #5
25084 {
25085     \exp_after:wN \__fp_parse_continue:NwN
25086     \exp_after:wN #1
25087     \exp:w \exp_end_continue_f:w
25088     \__fp_exp_after_tuple_f:nw { }
25089     \s__fp_tuple \__fp_tuple_chk:w { #2 #4 } ;
25090     #5 #1
25091 }

```

(End of definition for __fp_parse_infix_,:N, __fp_parse_infix_comma:w, and __fp_parse_apply_comma:NwNwN.)

71.7.2 Usual infix operators

```

\__fp_parse_infix_+:N As described in the “work plan”, each infix operator has an associated \..._infix...
\__fp_parse_infix_-:N function, a computing function, and precedence, given as arguments to \__fp_tmp:w.
\__fp_parse_infix_juxt:N Using the general mechanism for arithmetic operations. The power operation must be
\__fp_parse_infix_/:N associative in the opposite order from all others. For this, we use two distinct precedences.
\__fp_parse_infix_mul:N
\__fp_parse_infix_and:N
\__fp_parse_infix_or:N
\__fp_parse_infix_^:N
25092 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
25093 {
25094     \cs_new:Npn #1 ##1
25095     {
25096         \if_int_compare:w ##1 < #3
25097         \exp_after:wN @
25098         \exp_after:wN \__fp_parse_apply_binary:NwNwN
25099         \exp_after:wN #2
25100         \exp:w
25101         \__fp_parse_operand:Nw #4
25102         \exp_after:wN \__fp_parse_expand:w
25103     \else:
25104         \exp_after:wN @
25105         \exp_after:wN \use_none:n
25106         \exp_after:wN #1
25107     \fi:
25108 }
25109 }
25110 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_^:N } ^
25111 \c__fp_prec_hatii_int \c__fp_prec_hat_int
25112 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_juxt:N } *
25113 \c__fp_prec_juxt_int \c__fp_prec_juxt_int
25114 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_/:N } /
25115 \c__fp_prec_times_int \c__fp_prec_times_int

```



```

25116 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_mul:N } *
25117 \c__fp_prec_times_int \c__fp_prec_times_int
25118 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_minus:N } -
25119 \c__fp_prec_plus_int \c__fp_prec_plus_int
25120 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_plus:N } +
25121 \c__fp_prec_plus_int \c__fp_prec_plus_int
25122 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_and:N } &
25123 \c__fp_prec_and_int \c__fp_prec_and_int
25124 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_or:N } |
25125 \c__fp_prec_or_int \c__fp_prec_or_int

```

(End of definition for __fp_parse_infix_+:N and others.)

71.7.3 Juxtaposition

__fp_parse_infix_(:N) When an opening parenthesis appears where we expect an infix operator, we compute the product of the previous operand and the contents of the parentheses using __fp_parse_infix_mul:N.

```

25126 \cs_new:cpn { __fp_parse_infix_(:N) } #1
25127 { \__fp_parse_infix_mul:N #1 ( }

```

(End of definition for __fp_parse_infix_(:N).)

71.7.4 Multi-character cases

__fp_parse_infix_*:N

```

25128 \cs_set_protected:Npn \__fp_tmp:w #1
25129 {
25130   \cs_new:cpn { __fp_parse_infix_*:N } ##1##2
25131   {
25132     \if:w * \exp_not:N ##2
25133       \exp_after:wN #1
25134       \exp_after:wN ##1
25135     \else:
25136       \exp_after:wN \__fp_parse_infix_mul:N
25137       \exp_after:wN ##1
25138       \exp_after:wN ##2
25139     \fi:
25140   }
25141 }
25142 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix^:N }

```

(End of definition for __fp_parse_infix_*:N.)

__fp_parse_infix_|:Nw

__fp_parse_infix_&:Nw

```

25143 \cs_set_protected:Npn \__fp_tmp:w #1#2#3
25144 {
25145   \cs_new:Npn #1 ##1##2
25146   {
25147     \if:w #2 \exp_not:N ##2
25148       \exp_after:wN #1
25149       \exp_after:wN ##1
25150       \exp:w \exp_after:wN \__fp_parse_expand:w
25151     \else:

```

```

25152         \exp_after:wN #3
25153         \exp_after:wN ##1
25154         \exp_after:wN ##2
25155     \fi:
25156 }
25157 }
25158 \exp_args:Nc \_fp_tmp:w { \_fp_parse_infix_|:N } | \_fp_parse_infix_or:N
25159 \exp_args:Nc \_fp_tmp:w { \_fp_parse_infix_&:N } & \_fp_parse_infix_and:N

```

(End of definition for _fp_parse_infix_|:Nw and _fp_parse_infix_&:Nw.)

71.7.5 Ternary operator

```

\_fp_parse_infix_?:N
\_fp_parse_infix_::N
25160 \cs_set_protected:Npn \_fp_tmp:w #1#2#3#4
25161 {
25162     \cs_new:Npn #1 ##1
25163     {
25164         \if_int_compare:w ##1 < \c__fp_prec_quest_int
25165             #4
25166             \exp_after:wN @
25167             \exp_after:wN #2
25168             \exp:w
25169             \_fp_parse_operand:Nw #3
25170             \exp_after:wN \_fp_parse_expand:w
25171         \else:
25172             \exp_after:wN @
25173             \exp_after:wN \use_none:n
25174             \exp_after:wN #1
25175         \fi:
25176     }
25177 }
25178 \exp_args:Nc \_fp_tmp:w { \_fp_parse_infix_?:N }
25179 \_fp_ternary:NwwN \c__fp_prec_quest_int { }
25180 \exp_args:Nc \_fp_tmp:w { \_fp_parse_infix_::N }
25181 \_fp_ternary_auxii:NwwN \c__fp_prec_colon_int
25182 {
25183     \msg_expandable_error:nnnn
25184     { fp } { missing } { ? } { ~for~?: }
25185 }

```

(End of definition for _fp_parse_infix_?:N and _fp_parse_infix_::N.)

71.7.6 Comparisons

```

\_fp_parse_infix_<:N
\_fp_parse_infix_=:N
\_fp_parse_infix_>:N
\_fp_parse_infix_!:N
\_fp_parse_excl_error:
\_fp_parse_compare:NNNNNNN
\_fp_parse_compare_auxi:NNNNNN
\_fp_parse_compare_auxii:NNNN
\_fp_parse_compare_end:NNNNw
\_fp_compare:wNNNNw
25186 \cs_new:cpn { \_fp_parse_infix_<:N } #1
25187 { \_fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
25188 \cs_new:cpn { \_fp_parse_infix_=:N } #1
25189 { \_fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
25190 \cs_new:cpn { \_fp_parse_infix_>:N } #1
25191 { \_fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
25192 \cs_new:cpn { \_fp_parse_infix_!:N } #1
25193 {

```

```

25194 \exp_after:wN \__fp_parse_compare:NNNNNNN
25195 \exp_after:wN #1
25196 \exp_after:wN 0
25197 \exp_after:wN 1
25198 \exp_after:wN 1
25199 \exp_after:wN 1
25200 \exp_after:wN 1
25201 }
25202 \cs_new:Npn \__fp_parse_excl_error:
25203 {
25204 \msg_expandable_error:nnnn
25205 { fp } { missing } { = } { ~after~!. }
25206 }
25207 \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
25208 {
25209 \if_int_compare:w #1 < \c__fp_prec_comp_int
25210 \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
25211 \exp_after:wN \__fp_parse_excl_error:
25212 \else:
25213 \exp_after:wN @
25214 \exp_after:wN \use_none:n
25215 \exp_after:wN \__fp_parse_compare:NNNNNNN
25216 \fi:
25217 }
25218 \cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
25219 {
25220 \if_case:w
25221 \__fp_int_eval:w \exp_after:wN ‘ \token_to_str:N #7 - ‘<
25222 \__fp_int_eval_end:
25223 \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
25224 \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
25225 \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
25226 \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
25227 \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
25228 \fi:
25229 }
25230 \cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
25231 {
25232 \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
25233 \exp_after:wN \prg_do_nothing:
25234 \exp_after:wN #1
25235 \exp_after:wN #2
25236 \exp_after:wN #3
25237 \exp_after:wN #4
25238 \exp_after:wN #5
25239 \exp:w \exp_after:wN \__fp_parse_expand:w
25240 }
25241 \cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
25242 {
25243 \fi:
25244 \exp_after:wN @
25245 \exp_after:wN \__fp_parse_apply_compare:NwNNNNNwN
25246 \exp_after:wN \c_one_fp
25247 \exp_after:wN #1

```

```

25248 \exp_after:wN #2
25249 \exp_after:wN #3
25250 \exp_after:wN #4
25251 \exp:w
25252 \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
25253 }
25254 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNNwN
25255 #1 #2@ #3 #4#5#6#7 #8@ #9
25256 {
25257 \if_int_odd:w
25258 \if_meaning:w \c_zero_fp #3
25259 0
25260 \else:
25261 \if_case:w \__fp_compare_back_any:ww #8 #2 \exp_stop_f:
25262 #5 \or: #6 \or: #7 \else: #4
25263 \fi:
25264 \fi:
25265 \exp_stop_f:
25266 \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
25267 \exp_after:wN \c_one_fp
25268 \else:
25269 \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
25270 \exp_after:wN \c_zero_fp
25271 \fi:
25272 #1 #8 #9
25273 }
25274 \cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
25275 {
25276 \if_meaning:w \__fp_parse_compare:NNNNNNN #4
25277 \exp_after:wN \__fp_parse_continue_compare:NNwNN
25278 \exp_after:wN #1
25279 \exp_after:wN #2
25280 \exp:w \exp_end_continue_f:w
25281 \__fp_exp_after_o:w #3;
25282 \exp:w \exp_end_continue_f:w
25283 \else:
25284 \exp_after:wN \__fp_parse_continue:NwN
25285 \exp_after:wN #2
25286 \exp:w \exp_end_continue_f:w
25287 \exp_after:wN #1
25288 \exp:w \exp_end_continue_f:w
25289 \fi:
25290 #4 #2
25291 }
25292 \cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
25293 { #4 #2 #3@ #1 }

```

(End of definition for __fp_parse_infix_<:N and others.)

71.8 Tools for functions

__fp_parse_function_all_fp_o:fnw Followed by $\{\langle function\ name\rangle\} \{\langle code\rangle\} \langle float\ array\rangle @$ this checks all floats are floating point numbers (no tuples).

```

25294 \cs_new:Npn \__fp_parse_function_all_fp_o:fnw #1#2#3 @
25295 {
25296   \__fp_array_if_all_fp:nTF {#3}
25297   { #2 #3 @ }
25298   {
25299     \__fp_error:nffn { bad-args }
25300     {#1}
25301     { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#3} ; } }
25302     { }
25303     \exp_after:wN \c_nan_fp
25304   }
25305 }

```

(End of definition for __fp_parse_function_all_fp_o:fnw.)

__fp_parse_function_one_two:nnw
 __fp_parse_function_one_two_error_o:w
 __fp_parse_function_one_two_aux:nnw
 __fp_parse_function_one_two_auxii:nnw

This is followed by $\langle\text{function name}\rangle$ $\langle\text{code}\rangle$ $\langle\text{float array}\rangle$ @. It checks that the $\langle\text{float array}\rangle$ consists of one or two floating point numbers (not tuples), then leaves the $\langle\text{code}\rangle$ (if there is one float) or its tail (if there are two floats) followed by the $\langle\text{float array}\rangle$. The $\langle\text{code}\rangle$ should start with a single token such as $\backslash_\text{fp_atan_default:w}$ that deals with the single-float case.

The first $\backslash_\text{fp_if_type_fp:NTwFw}$ test catches the case of no argument and the case of a tuple argument. The next one distinguishes the case of a single argument (no error, just add $\backslash\text{c_one_fp}$) from a tuple second argument. Finally check there is no further argument.

```

25306 \cs_new:Npn \__fp_parse_function_one_two:nnw #1#2#3
25307 {
25308   \__fp_if_type_fp:NTwFw
25309   #3 { } \s__fp \__fp_parse_function_one_two_error_o:w \s__fp_stop
25310   \__fp_parse_function_one_two_aux:nnw {#1} {#2} #3
25311 }
25312 \cs_new:Npn \__fp_parse_function_one_two_error_o:w #1#2#3#4 @
25313 {
25314   \__fp_error:nffn { bad-args }
25315   {#2}
25316   { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#4} ; } }
25317   { }
25318   \exp_after:wN \c_nan_fp
25319 }
25320 \cs_new:Npn \__fp_parse_function_one_two_aux:nnw #1#2 #3; #4
25321 {
25322   \__fp_if_type_fp:NTwFw
25323   #4 { }
25324   \s__fp
25325   {
25326     \if_meaning:w @ #4
25327     \exp_after:wN \use_iv:nnnn
25328     \fi:
25329     \__fp_parse_function_one_two_error_o:w
25330   }
25331   \s__fp_stop
25332   \__fp_parse_function_one_two_auxii:nnw {#1} {#2} #3; #4
25333 }
25334 \cs_new:Npn \__fp_parse_function_one_two_auxii:nnw #1#2#3; #4; #5
25335 {

```

```

25336 \if_meaning:w @ #5 \else:
25337 \exp_after:wN \__fp_parse_function_one_two_error_o:w
25338 \fi:
25339 \use_ii:nn {#1} { \use_none:n #2 } #3; #4; #5
25340 }

```

(End of definition for __fp_parse_function_one_two:nw and others.)

__fp_tuple_map_o:nw Apply #1 to all items in the following tuple and expand once afterwards. The code #1
 __fp_tuple_map_loop_o:nw should itself expand once after its result.

```

25341 \cs_new:Npn \__fp_tuple_map_o:nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
25342 {
25343 \exp_after:wN \s__fp_tuple
25344 \exp_after:wN \__fp_tuple_chk:w
25345 \exp_after:wN {
25346 \exp:w \exp_end_continue_f:w
25347 \__fp_tuple_map_loop_o:nw {#1} #2
25348 { \s__fp \prg_break: } ;
25349 \prg_break_point:
25350 \exp_after:wN } \exp_after:wN ;
25351 }
25352 \cs_new:Npn \__fp_tuple_map_loop_o:nw #1#2#3 ;
25353 {
25354 \use_none:n #2
25355 #1 #2 #3 ;
25356 \exp:w \exp_end_continue_f:w
25357 \__fp_tuple_map_loop_o:nw {#1}
25358 }

```

(End of definition for __fp_tuple_map_o:nw and __fp_tuple_map_loop_o:nw.)

__fp_tuple_mapthread_o:nw Apply #1 to pairs of items in the two following tuples and expand once afterwards.

```

\__fp_tuple_mapthread_loop_o:nw
25359 \cs_new:Npn \__fp_tuple_mapthread_o:nw #1
25360 \s__fp_tuple \__fp_tuple_chk:w #2 ;
25361 \s__fp_tuple \__fp_tuple_chk:w #3 ;
25362 {
25363 \exp_after:wN \s__fp_tuple
25364 \exp_after:wN \__fp_tuple_chk:w
25365 \exp_after:wN {
25366 \exp:w \exp_end_continue_f:w
25367 \__fp_tuple_mapthread_loop_o:nw {#1}
25368 #2 { \s__fp \prg_break: } ; @
25369 #3 { \s__fp \prg_break: } ;
25370 \prg_break_point:
25371 \exp_after:wN } \exp_after:wN ;
25372 }
25373 \cs_new:Npn \__fp_tuple_mapthread_loop_o:nw #1#2#3 ; #4 @ #5#6 ;
25374 {
25375 \use_none:n #2
25376 \use_none:n #5
25377 #1 #2 #3 ; #5 #6 ;
25378 \exp:w \exp_end_continue_f:w
25379 \__fp_tuple_mapthread_loop_o:nw {#1} #4 @
25380 }

```

(End of definition for __fp_tuple_mapthread_o:nw and __fp_tuple_mapthread_loop_o:nw.)

71.9 Messages

```
25381 \msg_new:nnn { fp } { deprecated }
25382   { '#1'~deprecated;~use~'#2' }
25383 \msg_new:nnn { fp } { unknown-fp-word }
25384   { Unknown~fp~word~#1. }
25385 \msg_new:nnn { fp } { missing }
25386   { Missing~#1~inserted #2. }
25387 \msg_new:nnn { fp } { extra }
25388   { Extra~#1~ignored. }
25389 \msg_new:nnn { fp } { early-end }
25390   { Premature~end~in~fp~expression. }
25391 \msg_new:nnn { fp } { after-e }
25392   { Cannot~use~#1 after~'e'. }
25393 \msg_new:nnn { fp } { missing-number }
25394   { Missing~number~before~'#1'. }
25395 \msg_new:nnn { fp } { unknown-symbol }
25396   { Unknown~symbol~#1~ignored. }
25397 \msg_new:nnn { fp } { extra-comma }
25398   { Unexpected~comma~turned~to~nan~result.}
25399 \msg_new:nnn { fp } { no-arg }
25400   { #1~got~no~argument;~used~nan. }
25401 \msg_new:nnn { fp } { multi-arg }
25402   { #1~got~more~than~one~argument;~used~nan. }
25403 \msg_new:nnn { fp } { num-args }
25404   { #1~expects~between~#2~and~#3~arguments. }
25405 \msg_new:nnn { fp } { bad-args }
25406   { Arguments~in~#1#2~are~invalid. }
25407 \msg_new:nnn { fp } { infity-pi }
25408   { Math~command~#1 is~not~an~fp }
25409 \cs_if_exist:cT { @unexpandable@protect }
25410   {
25411     \msg_new:nnn { fp } { robust-cmd }
25412     { Robust~command~#1 invalid~in~fp~expression! }
25413   }
25414 </package>
```

Chapter 72

l3fp-assign implementation

```
25415 (*package)
25416 (@@=fp)
```

72.1 Assigning values

\fp_new:N Floating point variables are initialized to be +0.

```
25417 \cs_new_protected:Npn \fp_new:N #1
25418   { \cs_new_eq:NN #1 \c_zero_fp }
25419 \cs_generate_variant:Nn \fp_new:N {c}
```

(End of definition for \fp_new:N. This function is documented on page 260.)

\fp_set:Nn Simply use `__fp_parse:n` within various f-expanding assignments.

```
\fp_set:cn 25420 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_gset:Nn 25421   { \__kernel_tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cn 25422 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_const:Nn 25423   { \__kernel_tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_const:cn 25424 \cs_new_protected:Npn \fp_const:Nn #1#2
25425   { \tl_const:Ne #1 { \exp_not:f { \__fp_parse:n {#2} } } }
25426 \cs_generate_variant:Nn \fp_set:Nn {c}
25427 \cs_generate_variant:Nn \fp_gset:Nn {c}
25428 \cs_generate_variant:Nn \fp_const:Nn {c}
```

(End of definition for \fp_set:Nn, \fp_gset:Nn, and \fp_const:Nn. These functions are documented on page 260.)

\fp_set_eq:NN Copying a floating point is the same as copying the underlying token list.

```
\fp_set_eq:cN 25429 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 25430 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 25431 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 25432 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
\fp_gset_eq:cN
\fp_gset_eq:Nc
\fp_gset_eq:cc
```

(End of definition for \fp_set_eq:NN and \fp_gset_eq:NN. These functions are documented on page 260.)


```

\fp_zero:N Setting a floating point to zero: copy \c_zero_fp.
\fp_zero:c 25433 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 25434 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 25435 \cs_generate_variant:Nn \fp_zero:N { c }
25436 \cs_generate_variant:Nn \fp_gzero:N { c }

```

(End of definition for `\fp_zero:N` and `\fp_gzero:N`. These functions are documented on page 260.)

```

\fp_zero_new:N Set the floating point to zero, or define it if needed.
\fp_zero_new:c 25437 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 25438 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 25439 \cs_new_protected:Npn \fp_gzero_new:N #1
25440 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
25441 \cs_generate_variant:Nn \fp_zero_new:N { c }
25442 \cs_generate_variant:Nn \fp_gzero_new:N { c }

```

(End of definition for `\fp_zero_new:N` and `\fp_gzero_new:N`. These functions are documented on page 260.)

72.2 Updating values

These match the equivalent functions in `l3int` and `l3skip`.

```

\fp_add:Nn For the sake of error recovery we should not simply set #1 to #1 ± (#2): for instance, if #2
\fp_add:cn is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at
\fp_gadd:Nn the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting
\fp_gadd:cn parentheses. As an optimization we use \__fp_parse:n rather than \fp_eval:n, which
\fp_sub:Nn would convert the result away from the internal representation and back.
\fp_sub:cn 25443 \cs_new_protected:Npn \fp_add:Nn { \__fp_add:NNNn \fp_set:Nn + }
\fp_gsub:Nn 25444 \cs_new_protected:Npn \fp_gadd:Nn { \__fp_add:NNNn \fp_gset:Nn + }
\fp_gsub:cn 25445 \cs_new_protected:Npn \fp_sub:Nn { \__fp_add:NNNn \fp_set:Nn - }
\__fp_add:NNNn 25446 \cs_new_protected:Npn \fp_gsub:Nn { \__fp_add:NNNn \fp_gset:Nn - }
25447 \cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4
25448 { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
25449 \cs_generate_variant:Nn \fp_add:Nn { c }
25450 \cs_generate_variant:Nn \fp_gadd:Nn { c }
25451 \cs_generate_variant:Nn \fp_sub:Nn { c }
25452 \cs_generate_variant:Nn \fp_gsub:Nn { c }

```

(End of definition for `\fp_add:Nn` and others. These functions are documented on page 260.)

72.3 Showing values

```

\fp_show:N This shows the result of computing its argument by passing the right data to \tl_show:n
\fp_show:c or \tl_log:n.
\fp_log:N 25453 \cs_new_protected:Npn \fp_show:N { \__fp_show:NN \tl_show:n }
\fp_log:c 25454 \cs_generate_variant:Nn \fp_show:N { c }
\__fp_show:NN 25455 \cs_new_protected:Npn \fp_log:N { \__fp_show:NN \tl_log:n }
25456 \cs_generate_variant:Nn \fp_log:N { c }
25457 \cs_new_protected:Npn \__fp_show:NN #1#2
25458 {
25459 \__kernel_chk_tl_type:NnnT #2 { fp }

```

```

25460     { \exp_args:No \__fp_show_validate:n #2 }
25461     { \exp_args:Ne #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }
25462   }

```

(End of definition for `\fp_show:N`, `\fp_log:N`, and `__fp_show:NN`. These functions are documented on page 271.)

To support symbolic expression, validation has to be done recursively. Two `\@@_show_validate:nn` wrappers are used to distinguish between initial and recursive calls, in which the former provides a demo of possible forms a fp variable would have.

```

\__fp_show_validate:n
\__fp_show_validate_aux:n
\__fp_show_validate:nn
\__fp_show_validate:w
\__fp_tuple_show_validate:w
\__fp_symbolic_show_validate:w
25463 \cs_new:Npn \__fp_show_validate:n #1
25464 {
25465   \__fp_show_validate:nn { #1 }
25466   {
25467     \s__fp \__fp_chk:w ??? ;~ or \iow_newline:
25468     \s__fp_tuple \__fp_tuple_chk:w ? ;~ or \iow_newline:
25469     \s__fp_symbolic \__fp_symbolic_chk:w ? , ? ;
25470   }
25471 }
25472 \cs_new:Npn \__fp_show_validate_aux:n #1
25473 {
25474   \__fp_show_validate:nn { #1 } { }
25475 }
25476 \cs_new:Npn \__fp_show_validate:nn #1#2
25477 {
25478   \tl_if_empty:nF { #1 }
25479   {
25480     \str_case:enF { \tl_head:n { #1 } }
25481     {
25482       { \s__fp }
25483       {
25484         \__fp_show_validate:w #1 \s__fp
25485         \__fp_chk:w ??? ; \s__fp_stop
25486       }
25487       { \s__fp_tuple }
25488       {
25489         \__fp_tuple_show_validate:w #1
25490         \s__fp_tuple \__fp_tuple_chk:w ?? ; \s__fp_stop
25491       }
25492       { \s__fp_symbolic }
25493       {
25494         \__fp_symbolic_show_validate:w #1
25495         \s__fp_symbolic \__fp_symbolic_chk:w ? , ?? ; \s__fp_stop
25496       }
25497     }
25498     { #2 }
25499   }
25500 }
25501 \cs_new:Npn \__fp_show_validate:w
25502 #1 \s__fp \__fp_chk:w #2#3#4#5 ; #6 \s__fp_stop
25503 {
25504   \str_if_eq:nnF { #2 } {?}
25505   {
25506     \token_if_eq_meaning:NNTF #2 1

```

```

25507         { \s__fp \__fp_chk:w #2 #3 { #4 } #5 ; }
25508         { \s__fp \__fp_chk:w #2 #3 #4 #5 ; }
25509         \__fp_show_validate_aux:n { #6 }
25510     }
25511 }
25512 \cs_new:Npn \__fp_tuple_show_validate:w
25513     #1 \s__fp_tuple \__fp_tuple_chk:w #2#3 ; #4 \s__fp_stop
25514     {
25515     \str_if_eq:nnF { #2 } {?}
25516     { \s__fp_tuple \__fp_tuple_chk:w { \__fp_show_validate_aux:n { #2 } } ; }
25517     }
25518 \cs_new:Npn \__fp_symbolic_show_validate:w
25519     #1 \s__fp_symbolic \__fp_symbolic_chk:w #2 , #3#4 ; #5 \s__fp_stop
25520     {
25521     \str_if_eq:nnF { #2 } {?}
25522     {
25523     \s__fp_symbolic \__fp_symbolic_chk:w \exp_not:n { #2 } ,
25524     { \__fp_show_validate_aux:n { #3 } };
25525     \__fp_show_validate_aux:n { #5 }
25526     }
25527     }

```

(End of definition for `__fp_show_validate:n` and others.)

`\fp_show:n` Use general tools.

```

\fp_log:n 25528 \cs_new_protected:Npn \fp_show:n
25529     { \__kernel_msg_show_eval:Nn \fp_to_tl:n }
25530 \cs_new_protected:Npn \fp_log:n
25531     { \__kernel_msg_log_eval:Nn \fp_to_tl:n }

```

(End of definition for `\fp_show:n` and `\fp_log:n`. These functions are documented on page 271.)

72.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.

```

\c_e_fp 25532 \fp_const:Nn \c_e_fp          { 2.718 2818 2845 9045 }
25533 \fp_const:Nn \c_one_fp          { 1 }

```

(End of definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 269.)

`\c_pi_fp` We simply round π to and $\pi/180$ to 16 significant digits.

```

\c_one_degree_fp 25534 \fp_const:Nn \c_pi_fp          { 3.141 5926 5358 9793 }
25535 \fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }

```

(End of definition for `\c_pi_fp` and `\c_one_degree_fp`. These variables are documented on page 269.)

`\l_tmpa_fp` Scratch variables are simply initialized there.

```

\l_tmpb_fp 25536 \fp_new:N \l_tmpa_fp
\g_tmpa_fp 25537 \fp_new:N \l_tmpb_fp
\g_tmpb_fp 25538 \fp_new:N \g_tmpa_fp
25539 \fp_new:N \g_tmpb_fp

```

(End of definition for `\l_tmpa_fp` and others. These variables are documented on page 269.)

```

25540 </package>

```

Chapter 73

l3fp-logic implementation

```
25541 (*package)
25542 (@@=fp)
__fp_parse_word_max:N Those functions may receive a variable number of arguments.
__fp_parse_word_min:N
25543 \cs_new:Npn \__fp_parse_word_max:N
25544   { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
25545 \cs_new:Npn \__fp_parse_word_min:N
25546   { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }
(End of definition for \__fp_parse_word_max:N and \__fp_parse_word_min:N.)
```

73.1 Syntax of internal functions

- `__fp_compare_npos:nww {<expo1>} <body1> ; {<expo2>} <body2> ;`
- `__fp_minmax_o:Nw <sign> <floating point array>`
- `__fp_not_o:w ? <floating point array>` (with one floating point number only)
- `__fp_&_o:ww <floating point> <floating point>`
- `__fp_|_o:ww <floating point> <floating point>`
- `__fp_ternary:NwwN, __fp_ternary_auxi:NwwN, __fp_ternary_auxii:NwwN` have to be understood.

73.2 Tests

```
\fp_if_exist_p:N Copies of the cs functions defined in l3basics.
\fp_if_exist_p:c 25547 \prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
\fp_if_exist:NTF 25548 \prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }
\fp_if_exist:cTF
(End of definition for \fp_if_exist:NTF. This function is documented on page 262.)
```

`\fp_if_nan_p:n` Evaluate and check if the result is a floating point of the same kind as `nan`.
`\fp_if_nan:nTF`

```

25549 \prg_new_conditional:Npnn \fp_if_nan:n #1 { TF , T , F , p }
25550 {
25551   \if:w 3 \exp_last_unbraced:Nf \__fp_kind:w { \__fp_parse:n {#1} }
25552   \prg_return_true:
25553   \else:
25554     \prg_return_false:
25555   \fi:
25556 }

```

(End of definition for `\fp_if_nan:nTF`. This function is documented on page 264.)

73.3 Comparison

`\fp_compare_p:n` Within floating point expressions, comparison operators are treated as operations, so we
`\fp_compare:nTF` evaluate `#1`, then compare with ± 0 . Tuples are true.

```

\__fp_compare_return:w 25557 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
25558 {
25559   \exp_after:wN \__fp_compare_return:w
25560   \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
25561 }
25562 \cs_new:Npn \__fp_compare_return:w #1#2#3;
25563 {
25564   \if_charcode:w 0
25565     \__fp_if_type_fp:NTwFw
25566     #1 { \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
25567     \s__fp 1 \s__fp_stop
25568   \prg_return_false:
25569   \else:
25570     \prg_return_true:
25571   \fi:
25572 }

```

(End of definition for `\fp_compare:nTF` and `__fp_compare_return:w`. This function is documented on page 264.)

`\fp_compare_p:nNn` Evaluate `#1` and `#3`, using an auxiliary to expand both, and feed the two floating point
`\fp_compare:nNnTF` numbers swapped to `__fp_compare_back_any:ww`, defined below. Compare the result
`__fp_compare_aux:wn` with `'#2-'`, which is -1 for $<$, 0 for $=$, 1 for $>$ and $?$.

```

25573 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
25574 {
25575   \if_int_compare:w
25576     \exp_after:wN \__fp_compare_aux:wn
25577     \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
25578     = \__fp_int_eval:w '#2 - ' = \__fp_int_eval_end:
25579   \prg_return_true:
25580   \else:
25581     \prg_return_false:
25582   \fi:
25583 }
25584 \cs_new:Npn \__fp_compare_aux:wn #1; #2
25585 {
25586   \exp_after:wN \__fp_compare_back_any:ww

```

```

25587     \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1;
25588   }

```

(End of definition for `\fp_compare:nNnTF` and `__fp_compare_aux:wn`. This function is documented on page 263.)

```

\__fp_compare_back:ww
  \__fp_bcmp:ww
\__fp_compare_back_any:ww
  \__fp_compare_nan:w

```

`__fp_compare_back_any:ww` $\langle y \rangle$; $\langle x \rangle$;
 Expands (in the same way as `\int_eval:n`) to -1 if $x < y$, 0 if $x = y$, 1 if $x > y$, and 2 otherwise (denoted as $x?y$). If either operand is `nan`, stop the comparison with `__fp_compare_nan:w` returning 2 . If x is negative, swap the outputs 1 and -1 (i.e., $>$ and $<$); we can henceforth assume that $x \geq 0$. If $y \geq 0$, and they have the same type, either they are normal and we compare them with `__fp_compare_npos:nwnw`, or they are equal. If $y \geq 0$, but of a different type, the highest type is a larger number. Finally, if $y \leq 0$, then $x > y$, unless both are zero.

```

25589 \cs_new:Npn \__fp_compare_back:ww #1#2; #3#4;
25590   {
25591     \cs:w
25592       __fp
25593       \__fp_type_from_scan:N #1
25594       _bcmp
25595       \__fp_type_from_scan:N #3
25596       :ww
25597     \cs_end:
25598     #1#2; #3#4;
25599   }
26000 \cs_new:Npn \__fp_compare_back_any:ww #1#2; #3
26001   {
26002     \__fp_if_type_fp:NTwFw
26003     #1 { \__fp_if_type_fp:NTwFw #3 \use_i:nn \s__fp \use_ii:nn \s__fp_stop }
26004     \s__fp \use_ii:nn \s__fp_stop
26005     \__fp_compare_back:ww
26006     {
26007       \cs:w
26008         __fp
26009         \__fp_type_from_scan:N #1
26010         _compare_back
26011         \__fp_type_from_scan:N #3
26012         :ww
26013       \cs_end:
26014     }
26015     #1#2 ; #3
26016   }
26017 \cs_new:Npn \__fp_bcmp:ww
26018   \s__fp \__fp_chk:w #1 #2 #3;
26019   \s__fp \__fp_chk:w #4 #5 #6;
26020   {
26021     \int_value:w
26022     \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
26023     \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
26024     \if_meaning:w 2 #5 - \fi:
26025     \if_meaning:w #2 #5
26026     \if_meaning:w #1 #4
26027     \if_meaning:w 1 #1
26028     \__fp_compare_npos:nwnw #6; #3;

```

```

25629         \else:
25630             0
25631         \fi:
25632     \else:
25633         \if_int_compare:w #4 < #1 - \fi: 1
25634     \fi:
25635 \else:
25636     \if_int_compare:w #1#4 = \c_zero_int
25637         0
25638     \else:
25639         1
25640     \fi:
25641 \fi:
25642 \exp_stop_f:
25643 }
25644 \cs_new:Npn \__fp_compare_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }

```

(End of definition for `__fp_compare_back:ww` and others.)

`__fp_compare_back_tuple:ww` Tuple and floating point numbers are not comparable so return 2 in mixed cases or
`__fp_tuple_compare_back:ww` when tuples have a different number of items. Otherwise compare pairs of items with
`__fp_tuple_compare_back_tuple:ww` `__fp_compare_back_any:ww` and if any don't match return 2 (as `\int_value:w 02`
`__fp_tuple_compare_back_loop:w` `\exp_stop_f:`).

```

25645 \cs_new:Npn \__fp_compare_back_tuple:ww #1; #2; { 2 }
25646 \cs_new:Npn \__fp_tuple_compare_back:ww #1; #2; { 2 }
25647 \cs_new:Npn \__fp_tuple_compare_back_tuple:ww
25648     \s__fp_tuple \__fp_tuple_chk:w #1;
25649     \s__fp_tuple \__fp_tuple_chk:w #2;
25650 {
25651     \int_compare:nNnTF { \__fp_array_count:n {#1} } =
25652     { \__fp_array_count:n {#2} }
25653     {
25654         \int_value:w 0
25655         \__fp_tuple_compare_back_loop:w
25656             #1 { \s__fp \prg_break: } ; @
25657             #2 { \s__fp \prg_break: } ;
25658         \prg_break_point:
25659         \exp_stop_f:
25660     }
25661     { 2 }
25662 }
25663 \cs_new:Npn \__fp_tuple_compare_back_loop:w #1#2 ; #3 @ #4#5 ;
25664 {
25665     \use_none:n #1
25666     \use_none:n #4
25667     \if_int_compare:w
25668         \__fp_compare_back_any:ww #1 #2 ; #4 #5 ; = \c_zero_int
25669     \else:
25670         2 \exp_after:wN \prg_break:
25671     \fi:
25672     \__fp_tuple_compare_back_loop:w #3 @
25673 }

```

(End of definition for `__fp_compare_back_tuple:ww` and others.)

```
\_fp_compare_npos:nwnw
\_fp_compare_significand:nnnnnnnn
```

`_fp_compare_npos:nwnw {<expo1>} <body1> ; {<expo2>} <body2> ;`
 Within an `\int_value:w ... \exp_stop_f:` construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```
25674 \cs_new:Npn \_fp_compare_npos:nwnw #1#2; #3#4;
25675 {
25676   \if_int_compare:w #1 = #3 \exp_stop_f:
25677     \_fp_compare_significand:nnnnnnnn #2 #4
25678   \else:
25679     \if_int_compare:w #1 < #3 - \fi: 1
25680   \fi:
25681 }
25682 \cs_new:Npn \_fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
25683 {
25684   \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
25685   \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
25686   0
25687   \else:
25688     \if_int_compare:w #3#4 < #7#8 - \fi: 1
25689   \fi:
25690   \else:
25691     \if_int_compare:w #1#2 < #5#6 - \fi: 1
25692   \fi:
25693 }
```

(End of definition for `_fp_compare_npos:nwnw` and `_fp_compare_significand:nnnnnnnn`.)

73.4 Floating point expression loops

`\fp_do_until:nn` These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```
\fp_do_while:nn
\fp_until_do:nn
\fp_while_do:nn
25694 \cs_new:Npn \fp_do_until:nn #1#2
25695 {
25696   #2
25697   \fp_compare:nF {#1}
25698   { \fp_do_until:nn {#1} {#2} }
25699 }
25700 \cs_new:Npn \fp_do_while:nn #1#2
25701 {
25702   #2
25703   \fp_compare:nT {#1}
25704   { \fp_do_while:nn {#1} {#2} }
25705 }
25706 \cs_new:Npn \fp_until_do:nn #1#2
25707 {
25708   \fp_compare:nF {#1}
25709   {
25710     #2
25711     \fp_until_do:nn {#1} {#2}
```



```

25712     }
25713   }
25714 \cs_new:Npn \fp_while_do:nn #1#2
25715   {
25716     \fp_compare:nT {#1}
25717     {
25718       #2
25719       \fp_while_do:nn {#1} {#2}
25720     }
25721   }

```

(End of definition for `\fp_do_until:nn` and others. These functions are documented on page 265.)

`\fp_do_until:nNnn` As above but not using the `nNn` syntax.

```

\fp_do_while:nNnn
\fp_until_do:nNnn
\fp_while_do:nNnn
25722 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
25723   {
25724     #4
25725     \fp_compare:nNnF {#1} #2 {#3}
25726     { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
25727   }
25728 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
25729   {
25730     #4
25731     \fp_compare:nNnT {#1} #2 {#3}
25732     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
25733   }
25734 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
25735   {
25736     \fp_compare:nNnF {#1} #2 {#3}
25737     {
25738       #4
25739       \fp_until_do:nNnn {#1} #2 {#3} {#4}
25740     }
25741   }
25742 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
25743   {
25744     \fp_compare:nNnT {#1} #2 {#3}
25745     {
25746       #4
25747       \fp_while_do:nNnn {#1} #2 {#3} {#4}
25748     }
25749   }

```

(End of definition for `\fp_do_until:nNnn` and others. These functions are documented on page 264.)

`\fp_step_function:nnnN` The approach here is somewhat similar to `\int_step_function:nnnN`. There are two subtleties: we use the internal parser `__fp_parse:n` to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

```

\fp_step_function:nnnc
  \__fp_step:wwwN
  \__fp_step_fp:wwwN
  \__fp_step:NnnnnN
  \__fp_step:NfnnnN
25750 \cs_new:Npn \fp_step_function:nnnN #1#2#3
25751   {
25752     \exp_after:wN \__fp_step:wwwN
25753     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}
25754     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}

```

```

25755     \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
25756   }
25757 \cs_generate_variant:Nn \fp_step_function:nnnN { nnc }

```

Only floating point numbers (not tuples) are allowed arguments. Only “normal” floating points (not ± 0 , $\pm \text{inf}$, nan) can be used as step; if positive, call `__fp_step:NnnnnN` with argument `>` otherwise `<`. This function has one more argument than its integer counterpart, namely the previous value, to catch the case where the loop has made no progress. Conversion to decimal is done just before calling the user’s function.

```

25758 \cs_new:Npn \__fp_step:wwwN #1#2; #3#4; #5#6; #7
25759 {
25760   \__fp_if_type_fp:NTwFw #1 { } \s__fp \prg_break: \s__fp_stop
25761   \__fp_if_type_fp:NTwFw #3 { } \s__fp \prg_break: \s__fp_stop
25762   \__fp_if_type_fp:NTwFw #5 { } \s__fp \prg_break: \s__fp_stop
25763   \use_i:nnnn { \__fp_step_fp:wwwN #1#2; #3#4; #5#6; #7 }
25764   \prg_break_point:
25765   \use:n
25766     {
25767       \__fp_error:nfff { step-tuple } { \fp_to_tl:n { #1#2 ; } }
25768       { \fp_to_tl:n { #3#4 ; } } { \fp_to_tl:n { #5#6 ; } }
25769     }
25770 }
25771 \cs_new:Npn \__fp_step_fp:wwwN #1 ; \s__fp \__fp_chk:w #2#3#4 ; #5; #6
25772 {
25773   \token_if_eq_meaning:NNTF #2 1
25774   {
25775     \token_if_eq_meaning:NNTF #3 0
25776     { \__fp_step:NnnnnN > }
25777     { \__fp_step:NnnnnN < }
25778   }
25779   {
25780     \token_if_eq_meaning:NNTF #2 0
25781     {
25782       \msg_expandable_error:nnn { kernel }
25783       { zero-step } {#6}
25784     }
25785     {
25786       \__fp_error:nnfn { bad-step } { }
25787       { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 ; } } {#6}
25788     }
25789     \use_none:nnnnn
25790   }
25791   { #1 ; } { \c_nan_fp } { \s__fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6
25792 }
25793 \cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
25794 {
25795   \fp_compare:nNnTF {#2} = {#3}
25796   {
25797     \__fp_error:nffn { tiny-step }
25798     { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
25799   }
25800   {
25801     \fp_compare:nNnF {#2} #1 {#5}
25802     {

```

```

25803         \exp_args:Nf #6 { \__fp_to_decimal_dispatch:w #2 }
25804         \__fp_step:NfnnnN
25805             #1 { \__fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
25806     }
25807 }
25808 }
25809 \cs_generate_variant:Nn \__fp_step:NnnnnN { Nf }

```

(End of definition for `\fp_step_function:nnnN` and others. This function is documented on page 266.)

`\fp_step_inline:nnnn` As for `\int_step_inline:nnnn`, create a global function and apply it, following up with
`\fp_step_variable:nnnNn` a break point.

```

\__fp_step:NNnnnn
25810 \cs_new_protected:Npn \fp_step_inline:nnnn
25811 {
25812     \int_gincr:N \g__kernel_prg_map_int
25813     \exp_args:NNc \__fp_step:NNnnnn
25814     \cs_gset_protected:Npn
25815         { \__fp_map_ \int_use:N \g__kernel_prg_map_int :w }
25816 }
25817 \cs_new_protected:Npn \fp_step_variable:nnnNn #1#2#3#4#5
25818 {
25819     \int_gincr:N \g__kernel_prg_map_int
25820     \exp_args:NNc \__fp_step:NNnnnn
25821     \cs_gset_protected:Npe
25822         { \__fp_map_ \int_use:N \g__kernel_prg_map_int :w }
25823     {#1} {#2} {#3}
25824     {
25825         \tl_set:Nn \exp_not:N #4 {##1}
25826         \exp_not:n {#5}
25827     }
25828 }
25829 \cs_new_protected:Npn \__fp_step:NNnnnn #1#2#3#4#5#6
25830 {
25831     #1 #2 ##1 {#6}
25832     \fp_step_function:nnnN {#3} {#4} {#5} #2
25833     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
25834 }

```

(End of definition for `\fp_step_inline:nnnn`, `\fp_step_variable:nnnNn`, and `__fp_step:NNnnnn`. These functions are documented on page 266.)

```

25835 \msg_new:nnn { fp } { step-tuple }
25836 { Tuple~argument~in~fp_step...~{#1}{#2}{#3}. }
25837 \msg_new:nnn { fp } { bad-step }
25838 { Invalid~step~size~#2~for~function~#3. }
25839 \msg_new:nnn { fp } { tiny-step }
25840 { Tiny~step~size~(#1+#2=#1)~for~function~#3. }

```

73.5 Extrema

`__fp_minmax_o:Nw` First check all operands are floating point numbers. The argument #1 is 2 to find the
`__fp_minmax_aux_o:Nw` maximum of an array #2 of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance ± 0), the first is kept. We append $-\infty$ (∞), for the case

of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of `max()` and `min()` with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `__fp_minmax_loop:Nww`.

```

25841 \cs_new:Npn \__fp_minmax_o:Nw #1
25842 {
25843   \__fp_parse_function_all_fp_o:fnw
25844   { \token_if_eq_meaning:NNTF 0 #1 { min } { max } }
25845   { \__fp_minmax_aux_o:Nw #1 }
25846 }
25847 \cs_new:Npn \__fp_minmax_aux_o:Nw #1#2 @
25848 {
25849   \if_meaning:w 0 #1
25850   \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
25851   \else:
25852   \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
25853   \fi:
25854   #2
25855   \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
25856   \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
25857 }

```

(End of definition for `__fp_minmax_o:Nw` and `__fp_minmax_aux_o:Nw`.)

`__fp_minmax_loop:Nww` The first argument is `-` or `+` to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is `nan`, keep that as the extremum, unless that extremum is already a `nan`. Otherwise, compare the two numbers. If the new number is larger (in the case of `max`) or smaller (in the case of `min`), the test yields `true`, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

```

25858 \cs_new:Npn \__fp_minmax_loop:Nww
25859   #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
25860 {
25861   \if_meaning:w 3 #4
25862   \if_meaning:w 3 #2
25863   \__fp_minmax_auxi:ww
25864   \else:
25865   \__fp_minmax_auxii:ww
25866   \fi:
25867   \else:
25868   \if_int_compare:w
25869   \__fp_compare_back:ww
25870   \s__fp \__fp_chk:w #4#5;
25871   \s__fp \__fp_chk:w #2#3;
25872   = #1 1 \exp_stop_f:
25873   \__fp_minmax_auxii:ww
25874   \else:
25875   \__fp_minmax_auxi:ww
25876   \fi:
25877   \fi:
25878   \__fp_minmax_loop:Nww #1
25879   \s__fp \__fp_chk:w #2#3;

```

```

25880     \s__fp \__fp_chk:w #4#5;
25881 }

```

(End of definition for __fp_minmax_loop:Nww.)

```

\__fp_minmax_auxi:ww  Keep the first/second number, and remove the other.
\__fp_minmax_auxii:ww
25882 \cs_new:Npn \__fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
25883 { \fi: \fi: #2 \s__fp #3 ; }
25884 \cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
25885 { \fi: \fi: #2 }

```

(End of definition for __fp_minmax_auxi:ww and __fp_minmax_auxii:ww.)

```

\__fp_minmax_break_o:w  This function is called from within an \if_meaning:w test. Skip to the end of the tests,
                        close the current test with \fi:, clean up, and return the appropriate number with one
                        post-expansion.

```

```

25886 \cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
25887 { \fi: \__fp_exp_after_o:w \s__fp #3; }

```

(End of definition for __fp_minmax_break_o:w.)

73.6 Boolean operations

```

\__fp_not_o:w          Return true or false, with two expansions, one to exit the conditional, and one to please
\__fp_tuple_not_o:w   l3fp-parse. The first argument is provided by l3fp-parse and is ignored.

```

```

25888 \cs_new:Npn \__fp_not_o:w #1 \s__fp \__fp_chk:w #2#3; @
25889 {
25890     \if_meaning:w 0 #2
25891     \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
25892     \else:
25893     \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
25894     \fi:
25895 }
25896 \cs_new:Npn \__fp_tuple_not_o:w #1 @ { \exp_after:wN \c_zero_fp }

```

(End of definition for __fp_not_o:w and __fp_tuple_not_o:w.)

```

\__fp_&_o:ww          For and, if the first number is zero, return it (with the same sign). Otherwise, return
\__fp_tuple_&_o:ww   the second one. For or, the logic is reversed: if the first number is non-zero, return
\__fp_&_tuple_o:ww   it, otherwise return the second number: we achieve that by hi-jacking \__fp_&_o:ww,
\__fp_tuple_&_tuple_o:ww inserting an extra argument, \else:, before \s__fp. In all cases, expand after the
\__fp_|_o:ww          floating point number.

```

```

\__fp_tuple_|_o:ww
\__fp_|_tuple_o:ww
\__fp_tuple_|_tuple_o:ww
\__fp_and_return:wNw
25897 \group_begin:
25898 \char_set_catcode_letter:N &
25899 \char_set_catcode_letter:N |
25900 \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
25901 {
25902     \if_meaning:w 0 #2 #1
25903     \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
25904     \fi:
25905     \__fp_exp_after_o:w
25906 }
25907 \cs_new:Npn \__fp_&_tuple_o:ww #1 \s__fp \__fp_chk:w #2#3;

```

```

25908 {
25909   \if_meaning:w 0 #2 #1
25910   \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
25911   \fi:
25912   \__fp_exp_after_tuple_o:w
25913 }
25914 \cs_new:Npn \__fp_tuple_&_o:ww #1; { \__fp_exp_after_o:w }
25915 \cs_new:Npn \__fp_tuple_&_tuple_o:ww #1; { \__fp_exp_after_tuple_o:w }
25916 \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
25917 \cs_new:Npn \__fp_|_tuple_o:ww { \__fp_&_tuple_o:ww \else: }
25918 \cs_new:Npn \__fp_tuple_|_o:ww #1; #2; { \__fp_exp_after_tuple_o:w #1; }
25919 \cs_new:Npn \__fp_tuple_|_tuple_o:ww #1; #2;
25920   { \__fp_exp_after_tuple_o:w #1; }
25921 \group_end:
25922 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2;
25923   { \fi: \__fp_exp_after_o:w #1; }

```

(End of definition for __fp_&_o:ww and others.)

73.7 Ternary operator

__fp_ternary:NwwN The first function receives the test and the true branch of the ?: ternary operator. __fp_ternary_auxi:NwwN It calls __fp_ternary_auxii:NwwN if the test branch is a floating point number ± 0 , and otherwise calls __fp_ternary_auxi:NwwN. These functions select one of their two arguments.

```

25924 \cs_new:Npn \__fp_ternary:NwwN #1 #2#3@ #4@ #5
25925 {
25926   \if_meaning:w \__fp_parse_infix_:N #5
25927   \if_charcode:w 0
25928     \__fp_if_type_fp:NTwFw
25929     #2 { \use_i:nn \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
25930     \s__fp 1 \s__fp_stop
25931     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxii:NwwN
25932   \else:
25933     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxi:NwwN
25934   \fi:
25935   \exp_after:wN #1
25936   \exp:w \exp_end_continue_f:w
25937   \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
25938   \exp_after:wN @
25939   \exp:w
25940     \__fp_parse_operand:Nw \c__fp_prec_colon_int
25941     \__fp_parse_expand:w
25942   \else:
25943     \msg_expandable_error:nmnn
25944     { fp } { missing } { : } { -for~?: }
25945     \exp_after:wN \__fp_parse_continue:NwN
25946     \exp_after:wN #1
25947     \exp:w \exp_end_continue_f:w
25948     \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
25949     \exp_after:wN #5
25950     \exp_after:wN #1
25951   \fi:

```

```

25952 }
25953 \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
25954 {
25955   \exp_after:wN \__fp_parse_continue:NwN
25956   \exp_after:wN #1
25957   \exp:w \exp_end_continue_f:w
25958   \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
25959   #4 #1
25960 }
25961 \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
25962 {
25963   \exp_after:wN \__fp_parse_continue:NwN
25964   \exp_after:wN #1
25965   \exp:w \exp_end_continue_f:w
25966   \__fp_exp_after_array_f:w #3 \s__fp_expr_stop
25967   #4 #1
25968 }

```

(End of definition for __fp_ternary:NwwN, __fp_ternary_auxi:NwwN, and __fp_ternary_auxii:NwwN.)

```

25969 </package>

```

Chapter 74

l3fp-basics implementation

```
25970 (*package)
25971 (@@=fp)
```

The `l3fp-basics` module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in “What Every Computer Scientist Should Know About Floating Point Arithmetic”, by David Goldberg, which can be found at <http://cr.yp.to/2005-590/goldberg.pdf>.

Unary functions.

```

\__fp_parse_word_abs:N
\__fp_parse_word_logb:N
\__fp_parse_word_sign:N
\__fp_parse_word_sqrt:N
25972 \cs_new:Npn \__fp_parse_word_abs:N
25973   { \__fp_parse_unary_function:NNN \__fp_set_sign_o:w 0 }
25974 \cs_new:Npn \__fp_parse_word_logb:N
25975   { \__fp_parse_unary_function:NNN \__fp_logb_o:w ? }
25976 \cs_new:Npn \__fp_parse_word_sign:N
25977   { \__fp_parse_unary_function:NNN \__fp_sign_o:w ? }
25978 \cs_new:Npn \__fp_parse_word_sqrt:N
25979   { \__fp_parse_unary_function:NNN \__fp_sqrt_o:w ? }
```

(End of definition for __fp_parse_word_abs:N and others.)

74.1 Addition and subtraction

We define here two functions, `__fp-_o:ww` and `__fp+_o:ww`, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, `__fp_add_big_i_o:wNww`, is used in `l3fp-expo`.

The logic goes as follows:

- `__fp-_o:ww` calls `__fp+_o:ww` to do the work, with the sign of the second operand flipped;
- `__fp+_o:ww` dispatches depending on the type of floating point, calling specialized auxiliaries;

- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of $\infty - \infty$;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the `__fp_basics_pack...` functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.

74.1.1 Sign, exponent, and special numbers

`__fp_-_o:ww` The `__fp+_o:ww` auxiliary has a hook: it takes one argument between the first `\s__fp` and `__fp_chk:w`, which is applied to the sign of the second operand. Positioning the hook there means that `__fp+_o:ww` can still perform the sanity check that it was followed by `\s__fp`.

```

25980 \cs_new:cpe { __fp_-_o:ww } \s__fp
25981   {
25982     \exp_not:c { __fp+_o:ww }
25983     \exp_not:n { \s__fp \__fp_neg_sign:N }
25984   }

```

(End of definition for `__fp_-_o:ww`.)

`__fp+_o:ww` This function is either called directly with an empty #1 to compute an addition, or it is called by `__fp_-_o:ww` with `__fp_neg_sign:N` as #1 to compute a subtraction, in which case the second operand's sign should be changed. If the `<types>` #2 and #4 are the same, dispatch to case #2 (0, 1, 2, or 3), where we call specialized functions: thanks to `\int_value:w`, those receive the tweaked `<sign2>` (expansion of #1#5) as an argument. If the `<types>` are distinct, the result is simply the floating point number with the highest `<type>`. Since case 3 (used for two nan) also picks the first operand, we can also use it when `<type1>` is greater than `<type2>`. Also note that we don't need to worry about `<sign2>` in that case since the second operand is discarded.

```

25985 \cs_new:cpn { __fp+_o:ww }
25986   \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
25987   {
25988     \if_case:w
25989       \if_meaning:w #2 #4
25990         #2
25991       \else:
25992         \if_int_compare:w #2 > #4 \exp_stop_f:
25993           3
25994         \else:
25995           4
25996       \fi:
25997     \fi:
25998     \exp_stop_f:
25999     \exp_after:wN \__fp_add_zeros_o:Nww \int_value:w
26000   \or:   \exp_after:wN \__fp_add_normal_o:Nww \int_value:w

```

```

26001 \or: \exp_after:wN \__fp_add_inf_o:Nww \int_value:w
26002 \or: \__fp_case_return_i_o:ww
26003 \else: \exp_after:wN \__fp_add_return_ii_o:Nww \int_value:w
26004 \fi:
26005 #1 #5
26006 \s__fp \__fp_chk:w #2 #3 ;
26007 \s__fp \__fp_chk:w #4 #5
26008 }

```

(End of definition for __fp_+_o:ww.)

__fp_add_return_ii_o:Nww Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

```

26009 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
26010 { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }

```

(End of definition for __fp_add_return_ii_o:Nww.)

__fp_add_zeros_o:Nww Adding two zeros yields \c_zero_fp, except if both zeros were -0 .

```

26011 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
26012 {
26013   \if_int_compare:w #2 #1 = 20 \exp_stop_f:
26014   \exp_after:wN \__fp_add_return_ii_o:Nww
26015   \else:
26016     \__fp_case_return_i_o:ww
26017   \fi:
26018   #1
26019   \s__fp \__fp_chk:w 0 #2
26020 }

```

(End of definition for __fp_add_zeros_o:Nww.)

__fp_add_inf_o:Nww If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked $\langle sign_2 \rangle$ (#1) and the $\langle sign_2 \rangle$ (#4) are identical.

```

26021 \cs_new:Npn \__fp_add_inf_o:Nww
26022   #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
26023   {
26024     \if_meaning:w #1 #2
26025     \__fp_case_return_i_o:ww
26026     \else:
26027       \__fp_case_use:nw
26028       {
26029         \exp_last_unbraced:Nf \__fp_invalid_operation_o:Nww
26030         { \token_if_eq_meaning:NNTF #1 #4 + - }
26031       }
26032     \fi:
26033     \s__fp \__fp_chk:w 2 #2 #3;
26034     \s__fp \__fp_chk:w 2 #4
26035   }

```

(End of definition for __fp_add_inf_o:Nww.)

```

\__fp_add_normal_o:Nww      \__fp_add_normal_o:Nww <sign2> \s__fp \__fp_chk:w 1 <sign1> <exp1>
                             <body1> ; \s__fp \__fp_chk:w 1 <initial sign2> <exp2> <body2> ;

```

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

26036 \cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
26037   {
26038     \if_meaning:w #1#2
26039       \exp_after:wN \__fp_add_npos_o:NnwNnw
26040     \else:
26041       \exp_after:wN \__fp_sub_npos_o:NnwNnw
26042     \fi:
26043     #2
26044   }

```

(End of definition for __fp_add_normal_o:Nww.)

74.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```

\__fp_add_npos_o:NnwNnw      \__fp_add_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s__fp \__fp_chk:w 1
                             <initial sign2> <exp2> <body2> ;

```

Since we are doing an addition, the final sign is $\langle sign_1 \rangle$. Start an `__fp_int_eval:w`, responsible for computing the exponent: the result, and the $\langle final\ sign \rangle$ are then given to `__fp_sanitize:Nw` which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by `__fp_add_big_i:wNww` or `__fp_add_big_ii:wNww`. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```

26045 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
26046   {
26047     \exp_after:wN \__fp_sanitize:Nw
26048     \exp_after:wN #1
26049     \int_value:w \__fp_int_eval:w
26050     \if_int_compare:w #2 > #5 \exp_stop_f:
26051       #2
26052     \exp_after:wN \__fp_add_big_i_o:wNww \int_value:w -
26053     \else:
26054       #5
26055     \exp_after:wN \__fp_add_big_ii_o:wNww \int_value:w
26056     \fi:
26057     \__fp_int_eval:w #5 - #2 ; #1 #3;
26058   }

```

(End of definition for __fp_add_npos_o:NnwNnw.)

```

\__fp_add_big_i_o:wNww      \__fp_add_big_i_o:wNww <shift> ; <final sign> <body1> ; <body2> ;
\__fp_add_big_ii_o:wNww

```

Used in `l3fp-expo`. Shift the significand of the small number, then add with `__fp_add_significand_o:NnnwnnnnN`.

```

26059 \cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
26060   {
26061     \__fp_decimate:nNnnnn {#1}
26062     \__fp_add_significand_o:NnnwnnnnN

```

```

26063     #4
26064     #3
26065     #2
26066   }
26067 \cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
26068   {
26069     \__fp_decimate:nNnnnn {#1}
26070     \__fp_add_significand_o:NnnwnnnnN
26071     #3
26072     #4
26073     #2
26074   }

```

(End of definition for `__fp_add_big_i_o:wNww` and `__fp_add_big_ii_o:wNww`.)

```

\__fp_add_significand_o:NnnwnnnnN   \__fp_add_significand_o:NnnwnnnnN <rounding digit> {\langle Y'_1 \rangle} {\langle Y'_2 \rangle}
\__fp_add_significand_pack:NNNNNNN   <extra-digits> ; {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} <final sign>
\__fp_add_significand_test_o:N

```

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99\dots95 \rightarrow 1.00\dots0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

26075 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
26076   {
26077     \exp_after:wN \__fp_add_significand_test_o:N
26078     \int_value:w \__fp_int_eval:w 1#5#6 + #2
26079     \exp_after:wN \__fp_add_significand_pack:NNNNNNN
26080     \int_value:w \__fp_int_eval:w 1#7#8 + #3 ; #1
26081   }
26082 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
26083   {
26084     \if_meaning:w 2 #1
26085       + 1
26086     \fi:
26087     ; #2 #3 #4 #5 #6 #7 ;
26088   }
26089 \cs_new:Npn \__fp_add_significand_test_o:N #1
26090   {
26091     \if_meaning:w 2 #1
26092       \exp_after:wN \__fp_add_significand_carry_o:wwwNN
26093     \else:
26094       \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
26095     \fi:
26096   }

```

(End of definition for `__fp_add_significand_o:NnnwnnnnN`, `__fp_add_significand_pack:NNNNNNN`, and `__fp_add_significand_test_o:N`.)

```

\__fp_add_significand_no_carry_o:wwwNN   \__fp_add_significand_no_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

If there's no carry, grab all the digits again and round. The packing function `__fp_basics_pack_high:NNNNNw` takes care of the case where rounding brings a carry.

```

26097 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
26098     #1; #2; #3#4 ; #5#6

```

```

26099 {
26100 \exp_after:wN \_fp_basics_pack_high:NNNNw
26101 \int_value:w \_fp_int_eval:w 1 #1
26102 \exp_after:wN \_fp_basics_pack_low:NNNNw
26103 \int_value:w \_fp_int_eval:w 1 #2 #3#4
26104 + \_fp_round:NNN #6 #4 #5
26105 \exp_after:wN ;
26106 }

```

(End of definition for `_fp_add_significand_no_carry_o:wwwNN`.)

```

\_fp_add_significand_carry_o:wwwNN \_fp_add_significand_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```

26107 \cs_new:Npn \_fp_add_significand_carry_o:wwwNN
26108 #1; #2; #3#4; #5#6
26109 {
26110 + 1
26111 \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNNw
26112 \int_value:w \_fp_int_eval:w 1 1 #1
26113 \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
26114 \int_value:w \_fp_int_eval:w 1 #2#3 +
26115 \exp_after:wN \_fp_round:NNN
26116 \exp_after:wN #6
26117 \exp_after:wN #3
26118 \int_value:w \_fp_round_digit:Nw #4 #5 ;
26119 \exp_after:wN ;
26120 }

```

(End of definition for `_fp_add_significand_carry_o:wwwNN`.)

74.1.3 Absolute subtraction

```

\_fp_sub_npos_o:NnwNnw \_fp_sub_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s_fp \_fp_chk:w 1
\_fp_sub_eq_o:Nnwnw <initial sign2> <exp2> <body2> ;
\_fp_sub_npos_ii_o:Nnwnw

```

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call `_fp_sub_npos_i_o:Nnwnw` with the opposite of `<sign1>`.

```

26121 \cs_new:Npn \_fp_sub_npos_o:NnwNnw #1#2#3; \s_fp \_fp_chk:w 1 #4#5#6;
26122 {
26123 \if_case:w \_fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
26124 \exp_after:wN \_fp_sub_eq_o:Nnwnw
26125 \or:
26126 \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
26127 \else:
26128 \exp_after:wN \_fp_sub_npos_ii_o:Nnwnw
26129 \fi:
26130 #1 {#2} #3; {#5} #6;
26131 }
26132 \cs_new:Npn \_fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
26133 \cs_new:Npn \_fp_sub_npos_ii_o:Nnwnw #1 #2; #3;

```

```

26134 {
26135   \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
26136   \int_value:w \_fp_neg_sign:N #1
26137   #3; #2;
26138 }

```

(End of definition for _fp_sub_npos_o:NnwNnw, _fp_sub_eq_o:Nnwnw, and _fp_sub_npos_ii_o:Nnwnw.)

_fp_sub_npos_i_o:Nnwnw

After the computation is done, _fp_sanitize:Nw checks for overflow/underflow. It expects the $\langle final\ sign \rangle$ and the $\langle exponent \rangle$ (delimited by ;). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the *near* auxiliary. Otherwise, decimate y , then call the *far* auxiliary to evaluate the difference between the two significands. Note that we decimate by 1 less than one could expect.

```

26139 \cs_new:Npn \_fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
26140 {
26141   \exp_after:wN \_fp_sanitize:Nw
26142   \exp_after:wN #1
26143   \int_value:w \_fp_int_eval:w
26144   #2
26145   \if_int_compare:w #2 = #4 \exp_stop_f:
26146   \exp_after:wN \_fp_sub_back_near_o:nnnnnnnnN
26147   \else:
26148   \exp_after:wN \_fp_decimate:nNnnnn \exp_after:wN
26149   { \int_value:w \_fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
26150   \exp_after:wN \_fp_sub_back_far_o:NnwnnnnnN
26151   \fi:
26152   #5
26153   #3
26154   #1
26155 }

```

(End of definition for _fp_sub_npos_i_o:Nnwnw.)

_fp_sub_back_near_o:nnnnnnnnN
_fp_sub_back_near_pack:NNNNNNw
_fp_sub_back_near_after:wNNNNw

_fp_sub_back_near_o:nnnnnnnnN $\{\langle Y_1 \rangle\}$ $\{\langle Y_2 \rangle\}$ $\{\langle Y_3 \rangle\}$ $\{\langle Y_4 \rangle\}$ $\{\langle X_1 \rangle\}$
 $\{\langle X_2 \rangle\}$ $\{\langle X_3 \rangle\}$ $\{\langle X_4 \rangle\}$ $\langle final\ sign \rangle$

In this case, the subtraction is exact, so we discard the $\langle final\ sign \rangle$ #9. The very large shifts of 10^9 and $1.1 \cdot 10^9$ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```

26156 \cs_new:Npn \_fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
26157 {
26158   \exp_after:wN \_fp_sub_back_near_after:wNNNNw
26159   \int_value:w \_fp_int_eval:w 10#5#6 - #1#2 - 11
26160   \exp_after:wN \_fp_sub_back_near_pack:NNNNNNw
26161   \int_value:w \_fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
26162 }
26163 \cs_new:Npn \_fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
26164 { + #1#2 ; {#3#4#5#6} {#7} ; }
26165 \cs_new:Npn \_fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
26166 {
26167   \if_meaning:w 0 #1

```

```

26168     \exp_after:wN \__fp_sub_back_shift:wnnnn
26169     \fi:
26170     ; {#1#2#3#4} {#5}
26171     }

```

(End of definition for __fp_sub_back_near_o:nnnnnnnN, __fp_sub_back_near_pack:NNNNNNw, and __fp_sub_back_near_after:wNNNNw.)

```

\__fp_sub_back_shift:wnnnn
\__fp_sub_back_shift_ii:ww
\__fp_sub_back_shift_iii:NNNNNNNNw
\__fp_sub_back_shift_iv:nnnnw

```

__fp_sub_back_shift:wnnnn ; {⟨Z₁⟩} {⟨Z₂⟩} {⟨Z₃⟩} {⟨Z₄⟩} ;
This function is called with ⟨Z₁⟩ ≤ 999. Act with \number to trim leading zeros from ⟨Z₁⟩ ⟨Z₂⟩ (we don't do all four blocks at once, since non-zero blocks would then overflow T_EX's integers). If the first two blocks are zero, the auxiliary receives an empty #1 and trims #2#30 from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from #1 alone, yielding a result between 1 and 6. Once the exponent is taken care of, trim leading zeros from #1#2#3 (when #1 is empty, the space before #2#3 is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```

26172 \cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
26173 {
26174     \exp_after:wN \__fp_sub_back_shift_ii:ww
26175     \int_value:w #1 #2 0 ;
26176 }
26177 \cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
26178 {
26179     \if_meaning:w @ #1 @
26180     - 7
26181     - \exp_after:wN \use_i:nnn
26182     \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
26183     \int_value:w #2#3 0 ~ 123456789;
26184     \else:
26185     - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
26186     \fi:
26187     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
26188     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
26189     \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
26190     \exp_after:wN ;
26191     \int_value:w
26192     #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
26193 }
26194 \cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
26195 \cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }

```

(End of definition for __fp_sub_back_shift:wnnnn and others.)

```

\__fp_sub_back_far_o:NnnwnnnnN

```

__fp_sub_back_far_o:NnnwnnnnN ⟨rounding⟩ {⟨Y'₁⟩} {⟨Y'₂⟩}
⟨extra-digits⟩ ; {⟨X₁⟩} {⟨X₂⟩} {⟨X₃⟩} {⟨X₄⟩} ⟨final sign⟩
If the difference is greater than 10^(expo_x), call the very_far auxiliary. If the result is less than 10^(expo_x), call the not_far auxiliary. If it is too close a call to know yet, namely if 1⟨Y'₁⟩⟨Y'₂⟩ = ⟨X₁⟩⟨X₂⟩⟨X₃⟩⟨X₄⟩0, then call the quite_far auxiliary. We use the odd combination of space and semi-colon delimiters to allow the not_far auxiliary to grab each piece individually, the very_far auxiliary to use __fp_pack_eight:wNNNNNNNN, and the quite_far to ignore the significands easily (using the ; delimiter).

```

26196 \cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8

```

```

26197 {
26198   \if_case:w
26199     \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
26200     \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
26201       0
26202     \else:
26203       \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
26204     \fi:
26205   \else:
26206     \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
26207   \fi:
26208   \exp_stop_f:
26209     \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
26210 \or:   \exp_after:wN \__fp_sub_back_very_far_o:wwwNN
26211 \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwNN
26212 \fi:
26213 #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
26214 }

```

(End of definition for __fp_sub_back_far_o:NnnwnnnN.)

__fp_sub_back_quite_far_o:wwNN
 __fp_sub_back_quite_far_ii:NN

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y . Then the *<rounding>* #3 and the *<final sign>* #4 control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we get 1 whenever the *<rounding>* digit is less than or equal to 5 (remember that the *<rounding>* digit is only equal to 5 if there was no further non-zero digit).

```

26215 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
26216 {
26217   \exp_after:wN \__fp_sub_back_quite_far_ii:NN
26218   \exp_after:wN #3
26219   \exp_after:wN #4
26220 }
26221 \cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
26222 {
26223   \if_case:w \__fp_round_neg:NNN #2 0 #1
26224     \exp_after:wN \use_i:nn
26225   \else:
26226     \exp_after:wN \use_ii:nn
26227   \fi:
26228   { ; {1000} {0000} {0000} {0000} ; }
26229   { - 1 ; {9999} {9999} {9999} {9999} ; }
26230 }

```

(End of definition for __fp_sub_back_quite_far_o:wwNN and __fp_sub_back_quite_far_ii:NN.)

__fp_sub_back_not_far_o:wwwNN

In the present case, x and y have different exponents, but y is large enough that $x - y$ has a smaller exponent than x . Decrement the exponent (with -1). Then proceed in a way similar to the *near* auxiliaries seen earlier, but multiplying x by 10 (#30 and #40 below), and with the added quirk that the *<rounding>* digit has to be taken into account. Namely, we may have to decrease the result by one unit if __fp_round_neg:NNN returns 1. This function expects the *<final sign>* #6, the last digit of 1100000000+#40-#2, and the *<rounding>* digit. Instead of redoing the computation for the second argument, we note that __fp_round_neg:NNN only cares about its parity, which is identical to that of the last digit of #2.


```

26231 \cs_new:Npn \__fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
26232 {
26233   - 1
26234   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
26235   \int_value:w \__fp_int_eval:w 1#30 - #1 - 11
26236   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
26237   \int_value:w \__fp_int_eval:w 11 0000 0000 + #40 - #2
26238   - \exp_after:wN \__fp_round_neg:NNN
26239   \exp_after:wN #6
26240   \use_none:nnnnnn #2 #5
26241   \exp_after:wN ;
26242 }

```

(End of definition for __fp_sub_back_not_far_o:wwwNN.)

__fp_sub_back_very_far_o:wwwNN
 __fp_sub_back_very_far_ii_o:nnNwwNN

The case where $x - y$ and x have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the y significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two (*rounding*) digits #3 and #6 (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```

26243 \cs_new:Npn \__fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
26244 {
26245   \__fp_pack_eight:wNNNNNNNN
26246   \__fp_sub_back_very_far_ii_o:nnNwwNN
26247   { 0 #1#2#3 #4#5#6#7 }
26248   ;
26249 }
26250 \cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
26251 {
26252   \exp_after:wN \__fp_basics_pack_high:NNNNNw
26253   \int_value:w \__fp_int_eval:w 1#4 - #1 - 1
26254   \exp_after:wN \__fp_basics_pack_low:NNNNNw
26255   \int_value:w \__fp_int_eval:w 2#5 - #2
26256   - \exp_after:wN \__fp_round_neg:NNN
26257   \exp_after:wN #7
26258   \int_value:w
26259   \if_int_odd:w \__fp_int_eval:w #5 - #2 \__fp_int_eval_end:
26260     1 \else: 2 \fi:
26261   \int_value:w \__fp_round_digit:Nw #3 #6 ;
26262   \exp_after:wN ;
26263 }

```

(End of definition for __fp_sub_back_very_far_o:wwwNN and __fp_sub_back_very_far_ii_o:nnNwwNN.)

74.2 Multiplication

74.2.1 Signs, and special numbers

__fp*_o:ww

We go through an auxiliary, which is common with `__fp/_o:ww`. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The

third is the operation for normal floating points. The fourth is there for extra cases needed in `_fp_o:ww`.

```

26264 \cs_new:cpn { \_fp*_o:ww }
26265 {
26266   \_fp_mul_cases_o:NnNnw
26267   *
26268   { - 2 + }
26269   \_fp_mul_npos_o:Nww
26270   { }
26271 }

```

(End of definition for `_fp*_o:ww`.)

`_fp_mul_cases_o:nNnw` Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call `_fp_mul_npos_o:Nww` to do the work. If the first operand is `nan`, go to case 2, in which the second operand is discarded; if the second operand is `nan`, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$ to case 4 or 5 depending on the combined sign, the products $0 \times \infty$ and $\infty \times 0$ to case 6 or 7 (invalid operation), and the products $1 \times \infty = \infty \times 1 = \infty \times \infty = \infty$ to cases 8 and 9. Note that the code for these two cases (which return $\pm\infty$) is inserted as argument #4, because it differs in the case of divisions.

```

26272 \cs_new:Npn \_fp_mul_cases_o:NnNnw
26273   #1#2#3#4 \s_fp \_fp_chk:w #5#6#7; \s_fp \_fp_chk:w #8#9
26274 {
26275   \if_case:w \_fp_int_eval:w
26276     \if_int_compare:w #5 #8 = 11 ~
26277     1
26278   \else:
26279     \if_meaning:w 3 #8
26280     3
26281   \else:
26282     \if_meaning:w 3 #5
26283     2
26284   \else:
26285     \if_int_compare:w #5 #8 = 10 ~
26286     9 #2 - 2
26287   \else:
26288     (#5 #2 #8) / 2 * 2 + 7
26289   \fi:
26290   \fi:
26291   \fi:
26292   \fi:
26293   \if_meaning:w #6 #9 - 1 \fi:
26294   \_fp_int_eval_end:
26295   \_fp_case_use:nw { #3 0 }
26296 \or: \_fp_case_use:nw { #3 2 }
26297 \or: \_fp_case_return_i_o:ww
26298 \or: \_fp_case_return_ii_o:ww
26299 \or: \_fp_case_return_o:Nww \c_zero_fp
26300 \or: \_fp_case_return_o:Nww \c_minus_zero_fp

```

```

26301 \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
26302 \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
26303 \or: \__fp_case_return_o:Nww \c_inf_fp
26304 \or: \__fp_case_return_o:Nww \c_minus_inf_fp
26305 #4
26306 \fi:
26307 \s__fp \__fp_chk:w #5 #6 #7;
26308 \s__fp \__fp_chk:w #8 #9
26309 }

```

(End of definition for __fp_mul_cases_o:nNnnww.)

74.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

```

\__fp_mul_npos_o:Nww \__fp_mul_npos_o:Nww <final sign> \s__fp \__fp_chk:w 1 <sign1> {<exp1>}
<body1> ; \s__fp \__fp_chk:w 1 <sign2> {<exp2>} <body2> ;

```

After the computation, __fp_sanitize:Nw checks for overflow or underflow. As we did for addition, __fp_int_eval:w computes the exponent, catching any shift coming from the computation in the significand. The <final sign> is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by __fp_mul_significand_o:nnnnNnnnn.

This is also used in l3fp-convert.

```

26310 \cs_new:Npn \__fp_mul_npos_o:Nww
26311 #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
26312 {
26313 \exp_after:wN \__fp_sanitize:Nw
26314 \exp_after:wN #1
26315 \int_value:w \__fp_int_eval:w
26316 #4 + #8
26317 \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
26318 }

```

(End of definition for __fp_mul_npos_o:Nww.)

```

\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_o:nnnnNnnnn {<X1>} {<X2>} {<X3>} {<X4>} <sign>
\__fp_mul_significand_drop:NNNNWw {<Y1>} {<Y2>} {<Y3>} {<Y4>}
\__fp_mul_significand_keep:NNNNWw

```

Note the three semicolons at the end of the definition. One is for the last __fp_mul_significand_drop:NNNNWw; one is for __fp_round_digit:Nw later on; and one, preceded by \exp_after:wN, which is correctly expanded (within an __fp_int_eval:w), is used by __fp_basics_pack_low:NNNNWw.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of __fp_int_eval:w.

```

26319 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
26320 {
26321 \exp_after:wN \__fp_mul_significand_test_f:NNN
26322 \exp_after:wN #5
26323 \int_value:w \__fp_int_eval:w 99990000 + #1*#6 +

```

```

26324 \exp_after:wN \_fp_mul_significand_keep:NNNNw
26325 \int_value:w \_fp_int_eval:w 99990000 + #1*#7 + #2*#6 +
26326 \exp_after:wN \_fp_mul_significand_keep:NNNNw
26327 \int_value:w \_fp_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
26328 \exp_after:wN \_fp_mul_significand_drop:NNNNw
26329 \int_value:w \_fp_int_eval:w 99990000 + #1*#9 + #2*#8 +
26330 #3*#7 + #4*#6 +
26331 \exp_after:wN \_fp_mul_significand_drop:NNNNw
26332 \int_value:w \_fp_int_eval:w 99990000 + #2*#9 + #3*#8 +
26333 #4*#7 +
26334 \exp_after:wN \_fp_mul_significand_drop:NNNNw
26335 \int_value:w \_fp_int_eval:w 99990000 + #3*#9 + #4*#8 +
26336 \exp_after:wN \_fp_mul_significand_drop:NNNNw
26337 \int_value:w \_fp_int_eval:w 10000000 + #4*#9 ;
26338 ; \exp_after:wN ;
26339 }
26340 \cs_new:Npn \_fp_mul_significand_drop:NNNNw #1#2#3#4#5 #6;
26341 { #1#2#3#4#5 ; + #6 }
26342 \cs_new:Npn \_fp_mul_significand_keep:NNNNw #1#2#3#4#5 #6;
26343 { #1#2#3#4#5 ; #6 ; }

```

(End of definition for `_fp_mul_significand_o:nnnnNnnn`, `_fp_mul_significand_drop:NNNNw`, and `_fp_mul_significand_keep:NNNNw`.)

```

\_fp_mul_significand_test_f:NNN \_fp_mul_significand_test_f:NNN <sign> 1 <digits 1-8> ; <digits
9-12> ; <digits 13-16> ; + <digits 17-20> + <digits 21-24> + <digits
25-28> + <digits 29-32> ; \exp_after:wN ;

```

If the *<digit 1>* is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if *<digit 1>* is zero, we care about digits 17 and 18, and whether further digits are zero.

```

26344 \cs_new:Npn \_fp_mul_significand_test_f:NNN #1 #2 #3
26345 {
26346 \if_meaning:w 0 #3
26347 \exp_after:wN \_fp_mul_significand_small_f:NNwwwN
26348 \else:
26349 \exp_after:wN \_fp_mul_significand_large_f:NwwNNNN
26350 \fi:
26351 #1 #3
26352 }

```

(End of definition for `_fp_mul_significand_test_f:NNN`.)

`_fp_mul_significand_large_f:NwwNNNN` In this branch, *<digit 1>* is non-zero. The result is thus *<digits 1-16>*, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, `_fp_round_digit:Nw` takes digits 17 and further (as an integer expression), and replaces it by a *<rounding digit>*, suitable for `_fp_round:NNN`.

```

26353 \cs_new:Npn \_fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
26354 {
26355 \exp_after:wN \_fp_basics_pack_high:NNNNw
26356 \int_value:w \_fp_int_eval:w 1#2
26357 \exp_after:wN \_fp_basics_pack_low:NNNNw
26358 \int_value:w \_fp_int_eval:w 1#3#4#5#6#7
26359 + \exp_after:wN \_fp_round:NNN
26360 \exp_after:wN #1

```

```

26361         \exp_after:wN #7
26362         \int_value:w \__fp_round_digit:Nw
26363     }

```

(End of definition for `__fp_mul_significand_large_f:NwwNNNN`.)

`__fp_mul_significand_small_f:NNwwN`

In this branch, $\langle \text{digit } 1 \rangle$ is zero. Our result is thus $\langle \text{digits } 2\text{-}17 \rangle$, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits `1#3` are followed, after expansion of the `small_pack` auxiliary, by the next digit, to form a 9 digit number.

```

26364 \cs_new:Npn \__fp_mul_significand_small_f:NNwwN #1 #2#3; #4#5; #6; + #7
26365 {
26366     - 1
26367     \exp_after:wN \__fp_basics_pack_high:NNNNNw
26368     \int_value:w \__fp_int_eval:w 1#3#4
26369     \exp_after:wN \__fp_basics_pack_low:NNNNNw
26370     \int_value:w \__fp_int_eval:w 1#5#6#7
26371     + \exp_after:wN \__fp_round:NNN
26372     \exp_after:wN #1
26373     \exp_after:wN #7
26374     \int_value:w \__fp_round_digit:Nw
26375 }

```

(End of definition for `__fp_mul_significand_small_f:NNwwN`.)

74.3 Division

74.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

`__fp/_o:ww`

Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display `/` rather than `*`. In the formula for dispatch, we replace `- 2 +` by `-`. The case of normal numbers is treated using `__fp_div_npos_o:Nww` rather than `__fp_mul_npos_o:Nww`. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the `\if_case:w` construction in `__fp_mul_cases_o:NnNww` are provided as the fourth argument here.

```

26376 \cs_new:cpn { \__fp/_o:ww }
26377 {
26378     \__fp_mul_cases_o:NnNww
26379     /
26380     { - }
26381     \__fp_div_npos_o:Nww
26382     {
26383         \or:
26384         \__fp_case_use:nw
26385         { \__fp_division_by_zero_o:NNww \c_inf_fp / }
26386         \or:
26387         \__fp_case_use:nw
26388         { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
26389     }
26390 }

```

(End of definition for `_fp_/_o:ww`.)

```
\_fp_div_npos_o:Nww \_fp_div_npos_o:Nww <final sign> \s\_fp \_fp_chk:w 1 <sign_A> {\exp
A} {\A_1} {\A_2} {\A_3} {\A_4} ; \s\_fp \_fp_chk:w 1 <sign_Z> {\exp
Z} {\Z_1} {\Z_2} {\Z_3} {\Z_4} ;
```

We want to compute A/Z . As for multiplication, `_fp_sanitize:Nw` checks for overflow or underflow; we provide it with the `<final sign>`, and an integer expression in which we compute the exponent. We set up the arguments of `_fp_div_significand_i_o:wnnw`, namely an integer $\langle y \rangle$ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four $\{A_i\}$, then the four $\{Z_i\}$, a semi-colon, and the `<final sign>`, used for rounding at the end.

```
26391 \cs_new:Npn \_fp_div_npos_o:Nww
26392   #1 \s\_fp \_fp_chk:w 1 #2 #3 #4 ; \s\_fp \_fp_chk:w 1 #5 #6 #7#8#9;
26393   {
26394     \exp_after:wN \_fp_sanitize:Nw
26395     \exp_after:wN #1
26396     \int_value:w \_fp_int_eval:w
26397     #3 - #6
26398     \exp_after:wN \_fp_div_significand_i_o:wnnw
26399     \int_value:w \_fp_int_eval:w #7 \use_i:nnnn #8 + 1 ;
26400     #4
26401     {\#7}{\#8}\#9 ;
26402     #1
26403   }
```

(End of definition for `_fp_div_npos_o:Nww`.)

74.3.2 Work plan

In this subsection, we explain how to avoid overflowing $\text{T}_{\text{E}}\text{X}$'s integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z , we proceed as follows.

- Find an integer $Q_A \simeq 10^4 A/Z$.
- Replace A by $B = 10^4 A - Q_A Z$.
- Find an integer $Q_B \simeq 10^4 B/Z$.
- Replace B by $C = 10^4 B - Q_B Z$.
- Find an integer $Q_C \simeq 10^4 C/Z$.
- Replace C by $D = 10^4 C - Q_C Z$.
- Find an integer $Q_D \simeq 10^4 D/Z$.
- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the Q_i are integers, B , C , D , and E are all exact multiples of 10^{-16} , in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general B , C , D , and E may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $Q_A \leq 10^4 A/Z$ etc. A reasonable attempt would be to define Q_A as

$$\backslash\text{int_eval:n} \left\{ \frac{A_1 A_2}{Z_1 + 1} - 1 \right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that $\varepsilon\text{-TeX}$'s $\backslash_fp_int_eval:w$ rounds divisions instead of truncating (really, $1/2$ would be sufficient, but we work with integers). We add 1 to Z_1 because $Z_1 \leq 10^4 Z < Z_1 + 1$ and we need Q_A to be an underestimate. However, we are now underestimating Q_A too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then B could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since TeX can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \backslash\text{int_eval:n} \left\{ \frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1 \right\}.$$

This is always less than $10^9 A / (10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that $\varepsilon\text{-TeX}$ rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1) / 10^9 \simeq 2.147 \dots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B :

$$\begin{aligned} 10^5 B &= A_1 A_2 0 + 10 \cdot 0.A_3 A_4 - 10 \cdot Z_1.Z_2 Z_3 Z_4 \cdot Q_A \\ &< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2 Z_3 Z_4}{y} \right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2 Z_3 Z_4 + 10 \\ &\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1.Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\ &\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned} 10^5 B &< 10^9 A/y + 1.6y, \\ 10^5 C &< 10^9 B/y + 1.6y, \\ 10^5 D &< 10^9 C/y + 1.6y, \\ 10^5 E &< 10^9 D/y + 1.6y. \end{aligned}$$

The goal is now to prove that none of B , C , D , and E can go beyond $(2^{31} - 1)/10^9 = 2.147\dots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned} 10^5 B &< 10^9/y + 1.6y, \\ 10^5 C &< 10^{13}/y^2 + 1.6(y + 10^4), \\ 10^5 D &< 10^{17}/y^3 + 1.6(y + 10^4 + 10^8/y), \\ 10^5 E &< 10^{21}/y^4 + 1.6(y + 10^4 + 10^8/y + 10^{12}/y^2). \end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned} 10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\ 10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\ 10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\ 10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5). \end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within \TeX 's bounds in all cases!

We later need to have a bound on the Q_i . Their definitions imply that $Q_A < 10^9 A/y - 1/2 < 10^5 A$ and similarly for the other Q_i . Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^4 (10^{-4i} Q_i) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result is in $[0.1, 10)$, hence will be rounded to a multiple of 10^{-16} or of 10^{-15} , so we only need to know the integer part of E/Z , and a ‘‘rounding’’ digit encoding the rest. Equivalently, we need to find the integer part of $2E/Z$, and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2 \frac{10^5 E}{10^5 Z} \leq 2 \frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let ε -TeX round

$$P = \text{\int_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

($1/2$ comes from ε -TeX's rounding) because each absolute value is less than 10^{-7} . Thus P is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, $P = 2E/Z$. We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in ((P-1)/2, P/2)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in (P/2, (P+1)/2)$. In each case, we know how to round to an integer, depending on the parity of P , and the rounding mode.

74.3.3 Implementing the significand division

`_fp_div_significand_i_o:wmmw`

```
\_fp\_div\_significand\_i\_o:wmmw <y> ; <A1> <A2> <A3> <A4>
<Z1> <Z2> <Z3> <Z4> ; <sign>
```

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ arguments for 4 consecutive calls to `_fp_div_significand_calc:wmmmmmm`. Each of these calls needs $\langle y \rangle$ (#1), and it turns out that we need post-expansion there, hence the `\int_value:w`. Here, #4 is six brace groups, which give the six first n-type arguments of the `calc` function.

```
26404 \cs_new:Npn \_fp\_div\_significand\_i\_o:wmmw #1 ; #2#3 #4 ;
26405   {
26406     \exp_after:wN \_fp\_div\_significand\_test\_o:w
26407     \int\_value:w \_fp\_int\_eval:w
26408     \exp_after:wN \_fp\_div\_significand\_calc:wmmmmmm
26409     \int\_value:w \_fp\_int\_eval:w 999999 + #2 #3 0 / #1 ;
26410     #2 #3 ;
26411     #4
26412     { \exp_after:wN \_fp\_div\_significand\_ii:wmm \int\_value:w #1 }
26413     { \exp_after:wN \_fp\_div\_significand\_ii:wmm \int\_value:w #1 }
26414     { \exp_after:wN \_fp\_div\_significand\_ii:wmm \int\_value:w #1 }
26415     { \exp_after:wN \_fp\_div\_significand\_iii:wmmmmmm \int\_value:w #1 }
26416   }
```

(End of definition for `_fp_div_significand_i_o:wmmw`.)

`_fp_div_significand_calc:wmmmmmm`
`_fp_div_significand_calc_i:wmmmmmm`
`_fp_div_significand_calc_ii:wmmmmmm`

```
\_fp\_div\_significand\_calc:wmmmmmm <106 + QA> ; <A1> <A2> ; <A3>
<A4> <Z1> <Z2> <Z3> <Z4> <continuation>
expands to
```

```
<106 + QA> <continuation> ; <B1> <B2> ; <B3> <B4> <Z1> <Z2> <Z3>
<Z4>
```

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C , D , E (with the input shifted accordingly), and is used in `l3fp-expo`.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within TeX's bounds. However, it is a little bit too large for our purposes: we would not be able to

use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on Q_A , implies that $10^6 + Q_A$ starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a `<continuation>`, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),$$

where $\langle i \rangle$ stands for the 10^5 digit of Q_A , which is 0 or 1, and $\#1$, $\#2$, *etc.* are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9 . Indeed, for the auxiliary with $\langle i \rangle = 1$, $\#1$ is at most 80000, leading to contributions of at worst $-8 \cdot 10^8$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i \rangle = 0$, $\#1$ can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with \TeX 's limits once more.

```

26417 \cs_new:Npn \__fp_div_significand_calc:wnnnnnnn 1#1
26418 {
26419   \if_meaning:w 1 #1
26420     \exp_after:wN \__fp_div_significand_calc_i:wnnnnnnn
26421   \else:
26422     \exp_after:wN \__fp_div_significand_calc_ii:wnnnnnnn
26423   \fi:
26424 }
26425 \cs_new:Npn \__fp_div_significand_calc_i:wnnnnnnn
26426 #1; #2;#3#4 #5#6#7#8 #9
26427 {
26428   1 1 #1
26429   #9 \exp_after:wN ;
26430   \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
26431   + #2 - #1 * #5 - #5#60
26432   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
26433   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
26434   + #3 - #1 * #6 - #70
26435   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
26436   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
26437   + #4 - #1 * #7 - #80
26438   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
26439   \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
26440   - #1 * #8 ;
26441   {#5}{#6}{#7}{#8}
26442 }
26443 \cs_new:Npn \__fp_div_significand_calc_ii:wnnnnnnn
26444 #1; #2;#3#4 #5#6#7#8 #9
26445 {
26446   1 0 #1

```

```

26447 #9 \exp_after:wN ;
26448 \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
26449 + #2 - #1 * #5
26450 \exp_after:wN \__fp_pack_Bigg:NNNNNNw
26451 \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
26452 + #3 - #1 * #6
26453 \exp_after:wN \__fp_pack_Bigg:NNNNNNw
26454 \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
26455 + #4 - #1 * #7
26456 \exp_after:wN \__fp_pack_Bigg:NNNNNNw
26457 \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
26458 - #1 * #8 ;
26459 {#5}{#6}{#7}{#8}
26460 }

```

(End of definition for `__fp_div_significand_calc:w`, `__fp_div_significand_calc_i:w`, and `__fp_div_significand_calc_ii:w`.)

```

\__fp_div_significand_ii:w
\__fp_div_significand_ii:w \langle y \rangle ; \langle B_1 \rangle ; \{ \langle B_2 \rangle \} \{ \langle B_3 \rangle \} \{ \langle B_4 \rangle \} \{ \langle Z_1 \rangle \}
\{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} \langle continuations \rangle \langle sign \rangle

```

Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle 0 / y - 1$. The result is output to the left, in an `__fp_int_eval:w` which we start now. Once that is evaluated (and the other Q_i also, since later expansions are triggered by this one), a packing auxiliary takes care of placing the digits of Q_B in an appropriate way for the final addition to obtain Q . This auxiliary is also used to compute Q_C and Q_D with the inputs C and D instead of B .

```

26461 \cs_new:Npn \__fp_div_significand_ii:w #1; #2; #3
26462 {
26463 \exp_after:wN \__fp_div_significand_pack:NNN
26464 \int_value:w \__fp_int_eval:w
26465 \exp_after:wN \__fp_div_significand_calc:w
26466 \int_value:w \__fp_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
26467 }

```

(End of definition for `__fp_div_significand_ii:w`.)

```

\__fp_div_significand_iii:w
\__fp_div_significand_iii:w \langle y \rangle ; \langle E_1 \rangle ; \{ \langle E_2 \rangle \} \{ \langle E_3 \rangle \} \{ \langle E_4 \rangle \}
\{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} \langle sign \rangle

```

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we later add (roughly) $5 \cdot P$, which amounts to adding $P/2 \simeq E/Z$ to Q_D , the appropriate correction from a hypothetical Q_E .

```

26468 \cs_new:Npn \__fp_div_significand_iii:w #1; #2; #3 #4 #5 #6 #7
26469 {
26470 0
26471 \exp_after:wN \__fp_div_significand_iv:w
26472 \int_value:w \__fp_int_eval:w ( 2 * #2 #3 ) / #6 #7 ; % <- P
26473 #2 ; {#3} {#4} {#5}
26474 {#6} {#7}
26475 }

```

(End of definition for `__fp_div_significand_iii:w`.)

```

\__fp_div_significand_iv:w
\__fp_div_significand_iv:w \langle P \rangle ; \langle E_1 \rangle ; \{ \langle E_2 \rangle \} \{ \langle E_3 \rangle \} \{ \langle E_4 \rangle \}
\{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} \langle sign \rangle
\__fp_div_significand_v:NNw
\__fp_div_significand_vi:Nw

```

This adds to the current expression ($10^7 + 10 \cdot Q_D$) a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D , with an extra *rounding* digit. This *rounding* digit is 0 or 5 if T does not contribute, *i.e.*, if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8 , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P , which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use $+$ as a separator (ending integer expressions explicitly). T is negative if the first character is $-$, it is positive if the first character is neither 0 nor $-$. It is also positive if the first character is 0 and second argument of `_fp_div_significand_vi:Nw`, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```

26476 \cs_new:Npn \_fp_div_significand_iv:wmmmmmmn #1; #2;#3#4#5 #6#7#8#9
26477 {
26478   + 5 * #1
26479   \exp_after:wN \_fp_div_significand_vi:Nw
26480   \int_value:w \_fp_int_eval:w -50 + 2*#2#3 - #1*#6#7 +
26481   \exp_after:wN \_fp_div_significand_v:NN
26482   \int_value:w \_fp_int_eval:w 499950 + 2*#4 - #1*#8 +
26483   \exp_after:wN \_fp_div_significand_v:NN
26484   \int_value:w \_fp_int_eval:w 500000 + 2*#5 - #1*#9 ;
26485 }
26486 \cs_new:Npn \_fp_div_significand_v:NN #1#2 { #1#2 \_fp_int_eval_end: + }
26487 \cs_new:Npn \_fp_div_significand_vi:Nw #1#2;
26488 {
26489   \if_meaning:w 0 #1
26490   \if_int_compare:w \_fp_int_eval:w #2 > 0 + 1 \fi:
26491   \else:
26492   \if_meaning:w - #1 - \else: + \fi: 1
26493   \fi:
26494   ;
26495 }

```

(End of definition for `_fp_div_significand_iv:wmmmmmmn`, `_fp_div_significand_v:NNw`, and `_fp_div_significand_vi:Nw`.)

`_fp_div_significand_pack:NNN`

At this stage, we are in the following situation: \TeX is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

```

    \_fp_div_significand_test_o:w 106 + QA \_fp_div_significand_
    pack:NNN 106 + QB \_fp_div_significand_pack:NNN 106 + QC \_fp_
    div_significand_pack:NNN 107 + 10 · QD + 5 · P + ε ; <sign>

```

Here, $\varepsilon = \text{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, *i.e.*, P was an overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5 's and the 10^4 's).

```

26496 \cs_new:Npn \_fp_div_significand_pack:NNN #1 #2 { + #1 #2 ; }

```

(End of definition for `_fp_div_significand_pack:NNN`.)

```

\_fp_div_significand_test_o:w \_fp_div_significand_test_o:w 1 0 <5d> ; <4d> ; <4d> ; <5d> ; <sign>

```

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence \widetilde{Q}_A (the tilde denoting the contribution from the other Q_i) is at most 99999, and $10^6 + \widetilde{Q}_A = 10 \dots$.

It is now time to round. This depends on how many digits the final result will have.

```

26497 \cs_new:Npn \_fp_div_significand_test_o:w 10 #1
26498 {
26499   \if_meaning:w 0 #1
26500     \exp_after:wN \_fp_div_significand_small_o:wwwNNNNwN
26501   \else:
26502     \exp_after:wN \_fp_div_significand_large_o:wwwNNNNwN
26503   \fi:
26504   #1
26505 }

```

(End of definition for `_fp_div_significand_test_o:w`.)

```

\_fp_div_significand_small_o:wwwNNNNwN \_fp_div_significand_small_o:wwwNNNNwN 0 <4d> ; <4d> ; <4d> ; <5d>
; <final sign>

```

Standard use of the functions `_fp_basics_pack_low:NNNNw` and `_fp_basics_pack_high:NNNNw`. We finally get to use the `<final sign>` which has been sitting there for a while.

```

26506 \cs_new:Npn \_fp_div_significand_small_o:wwwNNNNwN
26507   0 #1; #2; #3; #4#5#6#7#8; #9
26508 {
26509   \exp_after:wN \_fp_basics_pack_high:NNNNw
26510   \int_value:w \_fp_int_eval:w 1 #1#2
26511   \exp_after:wN \_fp_basics_pack_low:NNNNw
26512   \int_value:w \_fp_int_eval:w 1 #3#4#5#6#7
26513   + \_fp_round:NNN #9 #7 #8
26514   \exp_after:wN ;
26515 }

```

(End of definition for `_fp_div_significand_small_o:wwwNNNNwN`.)

```

\_fp_div_significand_large_o:wwwNNNNwN \_fp_div_significand_large_o:wwwNNNNwN <5d> ; <4d> ; <4d> ; <5d> ;
<sign>

```

We know that the final result cannot reach 10, hence `1#1#2`, together with contributions from the level below, cannot reach $2 \cdot 10^9$. For rounding, we build the `<rounding digit>` from the last two of our 18 digits.

```

26516 \cs_new:Npn \_fp_div_significand_large_o:wwwNNNNwN

```

```

26517 #1; #2; #3; #4#5#6#7#8; #9
26518 {
26519 + 1
26520 \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNw
26521 \int_value:w \_fp_int_eval:w 1 #1 #2
26522 \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
26523 \int_value:w \_fp_int_eval:w 1 #3 #4 #5 #6 +
26524 \exp_after:wN \_fp_round:NNN
26525 \exp_after:wN #9
26526 \exp_after:wN #6
26527 \int_value:w \_fp_round_digit:Nw #7 #8 ;
26528 \exp_after:wN ;
26529 }

```

(End of definition for `_fp_div_significand_large_o:wwwNNNNwN`.)

74.4 Square root

`_fp_sqrt_o:w` Zeros are unchanged: $\sqrt{-0} = -0$ and $\sqrt{+0} = +0$. Negative numbers (other than -0) have no real square root. Positive infinity, and `nan`, are unchanged. Finally, for normal positive numbers, there is some work to do.

```

26530 \cs_new:Npn \_fp_sqrt_o:w #1 \s_fpf \_fp_chk:w #2#3#4; @
26531 {
26532 \if_meaning:w 0 #2 \_fp_case_return_same_o:w \fi:
26533 \if_meaning:w 2 #3
26534 \_fp_case_use:nw { \_fp_invalid_operation_o:nw { sqrt } }
26535 \fi:
26536 \if_meaning:w 1 #2 \else: \_fp_case_return_same_o:w \fi:
26537 \_fp_sqrt_npos_o:w
26538 \s_fpf \_fp_chk:w #2 #3 #4;
26539 }

```

(End of definition for `_fp_sqrt_o:w`.)

`_fp_sqrt_npos_o:w` Prepare `_fp_sanitize:Nw` to receive the final sign 0 (the result is always positive) and the exponent, equal to half of the exponent #1 of the argument. If the exponent #1 is even, find a first approximation of the square root of the significand $10^8 a_1 + a_2 = 10^8 \#2\#3 + \#4\#5$ through Newton's method, starting at $x = 57234133 \simeq 10^{7.75}$. Otherwise, first shift the significand of the argument by one digit, getting $a'_1 \in [10^6, 10^7)$ instead of $[10^7, 10^8)$, then use Newton's method starting at $17782794 \simeq 10^{7.25}$.

```

26540 \cs_new:Npn \_fp_sqrt_npos_o:w \s_fpf \_fp_chk:w 1 0 #1#2#3#4#5;
26541 {
26542 \exp_after:wN \_fp_sanitize:Nw
26543 \exp_after:wN 0
26544 \int_value:w \_fp_int_eval:w
26545 \if_int_odd:w #1 \exp_stop_f:
26546 \exp_after:wN \_fp_sqrt_npos_auxi_o:wwnnN
26547 \fi:
26548 #1 / 2
26549 \_fp_sqrt_Newton_o:wwn 56234133; 0; {#2#3} {#4#5} 0
26550 }
26551 \cs_new:Npn \_fp_sqrt_npos_auxi_o:wwnnN #1 / 2 #2; 0; #3#4#5
26552 {

```

```

26553      ( #1 + 1 ) / 2
26554      \__fp_pack_eight:wNNNNNNNN
26555      \__fp_sqrt_npos_auxii_o:wNNNNNNNN
26556      ;
26557      0 #3 #4
26558  }
26559 \cs_new:Npn \__fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
26560 { \__fp_sqrt_Newton_o:wwn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }

```

(End of definition for `__fp_sqrt_npos_o:w`, `__fp_sqrt_npos_auxi_o:w`, and `__fp_sqrt_npos_auxii_o:w`.)

`__fp_sqrt_Newton_o:w`

Newton's method maps $x \mapsto [(x + [10^8 a_1/x])/2]$ in each iteration, where $[b/c]$ denotes ε -TeX's division. This division rounds the real number b/c to the closest integer, rounding ties away from zero, hence when c is even, $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$. For all c , $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$.

Let us prove that the method converges when implemented with ε -TeX integer division, for any $10^6 \leq a_1 < 10^8$ and starting value $10^6 \leq x < 10^8$. Using the inequalities above and the arithmetic–geometric inequality $(x + t)/2 \geq \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left[\frac{x + [10^8 a_1/x]}{2} \right] \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}.$$

For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \leq 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8 a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for x may contain two integers, hence x might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1/x] \leq 2x - 2$$

hence $10^8 a_1/x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1/(x - 1)] \geq 2x - 1$$

hence $10^8 a_1/(x - 1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges to a single integer x . To stop the iteration when two consecutive results are equal, the function `__fp_sqrt_Newton_o:w` receives the newly computed result as `#1`, the previous result as `#2`, and a_1 as `#3`. Note that ε -TeX combines the computation of a multiplication and a following division, thus avoiding overflow in `#3 * 10000000 / #1`. In any case, the result is within $[10^7, 10^8]$.

```

26561 \cs_new:Npn \__fp_sqrt_Newton_o:wnn #1; #2; #3
26562 {
26563   \if_int_compare:w #1 = #2 \exp_stop_f:
26564     \exp_after:wN \__fp_sqrt_auxi_o:NNNNwnnN
26565     \int_value:w \__fp_int_eval:w 9999 9999 +
26566     \exp_after:wN \__fp_use_none_until_s:w
26567   \fi:
26568   \exp_after:wN \__fp_sqrt_Newton_o:wnn
26569   \int_value:w \__fp_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
26570   #1; {#3}
26571 }

```

(End of definition for __fp_sqrt_Newton_o:wnn.)

__fp_sqrt_auxi_o:NNNNwnnN This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then ; $\{a_1\}$ $\{a_2\}$ $\{a'\}$. Here, $x \simeq \sqrt{10^8 a_1}$ and we want to estimate the square root of $a = 10^{-8}a_1 + 10^{-16}a_2 + 10^{-17}a'$. We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}.$$

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8}a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8}a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8}a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, __fp_sqrt_auxii_o:NnnnnnnnN is called several times to get closer and closer underestimates of \sqrt{a} . By construction, the underestimates y are always increasing, $a - y^2 < 3.2 \cdot 10^{-8}$ for all. Also, $y < 1$.

```

26572 \cs_new:Npn \__fp_sqrt_auxi_o:NNNNwnnN 1 #1#2#3#4#5;
26573 {
26574   \__fp_sqrt_auxii_o:NnnnnnnnN
26575   \__fp_sqrt_auxiii_o:wnnnnnnnn
26576   {#1#2#3#4} {#5} {2499} {9988} {7500}
26577 }

```

(End of definition for __fp_sqrt_auxi_o:NNNNwnnN.)

__fp_sqrt_auxii_o:NnnnnnnnN This receives a continuation function #1, then five blocks of 4 digits for y , then two 8-digit blocks and a single digit for a . A common estimate of $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$ is $(a - y^2)/(2y)$, which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then computes the integer (with ε -TeX's rounding division)

$$10^{4j}z = \left[(10^{4j}(a - y^2)) - 257 \right] \cdot (0.5 \cdot 10^8) / \lfloor 10^8 y + 1 \rfloor.$$

The choice of j ensures that $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8 / 10^7 = 10^9$, thus $10^9 + 10^{4j}z$ has exactly 10 digits, does not overflow TeX's integer range, and starts with 1. Incidentally, since all $a - y^2 \leq 3.2 \cdot 10^{-8}$, we know that $j \geq 3$.

Let us show that z is an underestimate of $\sqrt{a} - y$. On the one hand, $\sqrt{a} - y \leq 16 \cdot 10^{-8}$ because this holds for the initial y and values of y can only increase. On the other hand, the choice of j implies that $\sqrt{a} - y \leq 5(\sqrt{a} + y)(\sqrt{a} - y) = 5(a - y^2) < 10^{9-4j}$. For $j = 3$,

the first bound is better, while for larger j , the second bound is better. For all $j \in [3, 6]$, we find $\sqrt{a} - y < 16 \cdot 10^{-2j}$. From this, we deduce that

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}(a - y^2 - (\sqrt{a} - y)^2)}{2y} \geq \frac{\lfloor 10^{4j}(a - y^2) \rfloor - 257}{2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/(2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor) \geq 1/2$. Given that ε -TeX's integer division obeys $\lfloor b/c \rfloor \leq b/c + 1/2$, we deduce that $10^{4j}z \leq 10^{4j}(\sqrt{a} - y)$, hence $y + z \leq \sqrt{a}$ is an underestimate of \sqrt{a} , as claimed. One implementation detail: because the computation involves $-4 \cdot 4 - 2 \cdot 3 \cdot 5 - 2 \cdot 2 \cdot 6$ which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the big shifts.

```

26578 \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnN #1 #2#3#4#5#6 #7#8#9
26579 {
26580   \exp_after:wN #1
26581   \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
26582     + #7 - #2 * #2
26583   \exp_after:wN \__fp_pack_big:NNNNNNw
26584   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
26585     - 2 * #2 * #3
26586   \exp_after:wN \__fp_pack_big:NNNNNNw
26587   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
26588     + #8 - #3 * #3 - 2 * #2 * #4
26589   \exp_after:wN \__fp_pack_big:NNNNNNw
26590   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
26591     - 2 * #3 * #4 - 2 * #2 * #5
26592   \exp_after:wN \__fp_pack_big:NNNNNNw
26593   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
26594     + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
26595   \exp_after:wN \__fp_pack_big:NNNNNNw
26596   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
26597     - 2 * #4 * #5 - 2 * #3 * #6
26598   \exp_after:wN \__fp_pack_big:NNNNNNw
26599   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
26600     - #5 * #5 - 2 * #4 * #6
26601   \exp_after:wN \__fp_pack_big:NNNNNNw
26602   \int_value:w \__fp_int_eval:w
26603     \c__fp_big_middle_shift_int
26604     - 2 * #5 * #6
26605   \exp_after:wN \__fp_pack_big:NNNNNNw
26606   \int_value:w \__fp_int_eval:w
26607     \c__fp_big_trailing_shift_int
26608     - #6 * #6 ;
26609   % (
26610   - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
26611   {#2}{#3}{#4}{#5}{#6} {#7}{#8}{#9
26612 }

```

(End of definition for `__fp_sqrt_auxii_o:NnnnnnnN`.)

```

\__fp_sqrt_auxiii_o:wnnnnnnn
\__fp_sqrt_auxiv_o:NNNNNw
\__fp_sqrt_auxv_o:NNNNNw
\__fp_sqrt_auxvi_o:NNNNNw
\__fp_sqrt_auxvii_o:NNNNNw

```

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2 \rangle$; $\{\langle d_3 \rangle\} \dots \{\langle d_{10} \rangle\}$, where each block has 4 digits, except $\langle d_2 \rangle$. This function finds the largest $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then leaves an open parenthesis and the integer $\lfloor 10^{4j}(a - y^2) \rfloor$

in an integer expression. The closing parenthesis is provided by the caller `_fp_sqrt_`
`auxii_o:NnnnnnnnN`, which completes the expression

$$10^{4j}z = \left[([10^{4j}(a - y^2)] - 257) \cdot (0.5 \cdot 10^8) / [10^8y + 1] \right]$$

for an estimate of $10^{4j}(\sqrt{a} - y)$. If $d_2 \geq 2$, $j = 3$ and the `auxiv` auxiliary receives $10^{12}z$. If $d_2 \leq 1$ but $10^4d_2 + d_3 \geq 2$, $j = 4$ and the `auxv` auxiliary is called, and receives $10^{16}z$, and so on. In all those cases, the `auxviii` auxiliary is set up to add z to y , then go back to the `auxii` step with continuation `auxiii` (the function we are currently describing). The maximum value of j is 6, regardless of whether $10^{12}d_2 + 10^8d_3 + 10^4d_4 + d_5 \geq 1$. In this last case, we detect when $10^{24}z < 10^7$, which essentially means $\sqrt{a} - y \lesssim 10^{-17}$: once this threshold is reached, there is enough information to find the correctly rounded \sqrt{a} with only one more call to `_fp_sqrt_auxii_o:NnnnnnnnN`. Note that the iteration cannot be stuck before reaching $j = 6$, because for $j < 6$, one has $2 \cdot 10^8 \leq 10^{4(j+1)}(a - y^2)$, hence

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{[10^8y + 1]} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

26613 \cs_new:Npn \_fp_sqrt_auxiii_o:wnnnnnnnn
26614   #1; #2#3#4#5#6#7#8#9
26615   {
26616     \if_int_compare:w #1 > \c_one_int
26617       \exp_after:wN \_fp_sqrt_auxiv_o:NNNNNw
26618       \int_value:w \_fp_int_eval:w (#1#2 %)
26619     \else:
26620       \if_int_compare:w #1#2 > \c_one_int
26621         \exp_after:wN \_fp_sqrt_auxv_o:NNNNNw
26622         \int_value:w \_fp_int_eval:w (#1#2#3 %)
26623       \else:
26624         \if_int_compare:w #1#2#3 > \c_one_int
26625           \exp_after:wN \_fp_sqrt_auxvi_o:NNNNNw
26626           \int_value:w \_fp_int_eval:w (#1#2#3#4 %)
26627         \else:
26628           \exp_after:wN \_fp_sqrt_auxvii_o:NNNNNw
26629           \int_value:w \_fp_int_eval:w (#1#2#3#4#5 %)
26630         \fi:
26631       \fi:
26632     \fi:
26633   }
26634 \cs_new:Npn \_fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
26635   { \_fp_sqrt_auxviii_o:nnnnnnn {#1#2#3#4#5#6} {00000000} }
26636 \cs_new:Npn \_fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
26637   { \_fp_sqrt_auxviii_o:nnnnnnn {000#1#2#3#4#5} {#60000} }
26638 \cs_new:Npn \_fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
26639   { \_fp_sqrt_auxviii_o:nnnnnnn {0000000#1} {#2#3#4#5#6} }
26640 \cs_new:Npn \_fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
26641   {
26642     \if_int_compare:w #1#2 = \c_zero_int
26643       \exp_after:wN \_fp_sqrt_auxx_o:Nnnnnnnn
26644     \fi:
26645     \_fp_sqrt_auxviii_o:nnnnnnn {00000000} {000#1#2#3#4#5}
26646   }

```

(End of definition for `_fp_sqrt_auxiii_o:wnnnnnnnn` and others.)

`__fp_sqrt_auxviii_o:nnnnnnn` Simply add the two 8-digit blocks of z , aligned to the last four of the five 4-digit blocks
`__fp_sqrt_auxix_o:wnnnw` of y , then call the `auxii` auxiliary to evaluate $y'^2 = (y + z)^2$.

```

26647 \cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnn #1#2 #3#4#5#6#7
26648 {
26649   \exp_after:wN \__fp_sqrt_auxix_o:wnnnw
26650   \int_value:w \__fp_int_eval:w #3
26651   \exp_after:wN \__fp_basics_pack_low:NNNNNw
26652   \int_value:w \__fp_int_eval:w #1 + 1#4#5
26653   \exp_after:wN \__fp_basics_pack_low:NNNNNw
26654   \int_value:w \__fp_int_eval:w #2 + 1#6#7 ;
26655 }
26656 \cs_new:Npn \__fp_sqrt_auxix_o:wnnnw #1; #2#3; #4#5;
26657 {
26658   \__fp_sqrt_auxii_o:NnnnnnnnN
26659   \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
26660 }

```

(End of definition for `__fp_sqrt_auxviii_o:nnnnnnn` and `__fp_sqrt_auxix_o:wnnnw`.)

`__fp_sqrt_auxx_o:Nnnnnnnn` At this stage, $j = 6$ and $10^{24}z < 10^7$, hence
`__fp_sqrt_auxxi_o:wnnnN`

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) / (10^8y + 1),$$

then $10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$, and

$$10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a}),$$

which finally implies $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$. In particular, y is an underestimate of \sqrt{a} and $y + 0.5 \cdot 10^{-16}$ is a (strict) overestimate. There is at exactly one multiple m of $0.5 \cdot 10^{-16}$ in the interval $[y, y + 0.5 \cdot 10^{-16})$. If $m^2 > a$, then the square root is inexact and is obtained by rounding $m - \epsilon$ to a multiple of 10^{-16} (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly m , and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments `#1`, `#2`, `#3`, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y + 0.5 \cdot 10^{-16})$; rather, only the last 4 digits `#8` of y are considered, and we do not perform any carry yet. The `auxxi` auxiliary sets up `auxii` with a continuation function `auxxii` instead of `auxiii` as before. To prevent `auxii` from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$ and $a \leq 1 - 10^{-16}$.

```

26661 \cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
26662 {
26663   \exp_after:wN \__fp_sqrt_auxxi_o:wnnnN
26664   \int_value:w \__fp_int_eval:w
26665   (#8 + 2499) / 5000 * 5000 ;
26666   {#4} {#5} {#6} {#7} ;
26667 }
26668 \cs_new:Npn \__fp_sqrt_auxxi_o:wnnnN #1; #2; #3#4#5
26669 {
26670   \__fp_sqrt_auxii_o:NnnnnnnnN
26671   \__fp_sqrt_auxxii_o:nnnnnnnnw
26672   #2 {#1}
26673   {#3} { #4 + 1 } #5
26674 }

```

(End of definition for `_fp_sqrt_auxx_o:Nnnnnnnn` and `_fp_sqrt_auxxi_o:wwnnN`.)

`_fp_sqrt_auxxii_o:nnnnnnnw`
`_fp_sqrt_auxxiii_o:w`

The difference $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$ was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess m is an overestimate if $a + 10^{-16} - m^2 < 10^{-16}$, that is, #1#2 vanishes. Otherwise it is an underestimate, unless $a + 10^{-16} - m^2 = 10^{-16}$ exactly. For an underestimate, call the `auxxiv` function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```

26675 \cs_new:Npn \_fp_sqrt_auxxii_o:nnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
26676 {
26677   \if_int_compare:w #1#2 > \c_zero_int
26678     \if_int_compare:w #1#2 = \c_one_int
26679       \if_int_compare:w #3#4 = \c_zero_int
26680         \if_int_compare:w #5#6 = \c_zero_int
26681           \if_int_compare:w #7#8 = \c_zero_int
26682             \_fp_sqrt_auxxiii_o:w
26683           \fi:
26684         \fi:
26685       \fi:
26686     \fi:
26687     \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
26688     \int_value:w 9998
26689   \else:
26690     \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
26691     \int_value:w 10000
26692   \fi:
26693 ;
26694 }
26695 \cs_new:Npn \_fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
26696 {
26697   \fi: \fi: \fi: \fi: \fi:
26698   \_fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
26699 }

```

(End of definition for `_fp_sqrt_auxxii_o:nnnnnnnw` and `_fp_sqrt_auxxiii_o:w`.)

`_fp_sqrt_auxxiv_o:wnnnnnnnN`

This receives 9998, 9999 or 10000 as #1 when m is an underestimate, exact, or an overestimate, respectively. Then comes m as five blocks of 4 digits, but where the last block #6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless m is an overestimate (#1 is then 10000). Then comes a as three arguments. Rounding is done by `_fp_round:NNN`, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are “rounded to even”, and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by `_fp_round_digit:Nw`, which receives (after removal of the 10000’s digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```

26700 \cs_new:Npn \_fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
26701 {
26702   \exp_after:wN \_fp_basics_pack_high:NNNNNw
26703   \int_value:w \_fp_int_eval:w 1 0000 0000 + #2#3

```

```

26704      \exp_after:wN \__fp_basics_pack_low:NNNNNw
26705      \int_value:w \__fp_int_eval:w 1 0000 0000
26706      + #4#5
26707      \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
26708      + \exp_after:wN \__fp_round:NNN
26709      \exp_after:wN 0
26710      \exp_after:wN 0
26711      \int_value:w
26712      \exp_after:wN \use_i:nn
26713      \exp_after:wN \__fp_round_digit:Nw
26714      \int_value:w \__fp_int_eval:w #6 + 19999 - #1 ;
26715      \exp_after:wN ;
26716      }

```

(End of definition for __fp_sqrt_auxxiv_o:wnnnnnnnN.)

74.5 About the sign and exponent

```

\__fp_logb_o:w The exponent of a normal number is its <exponent> minus one.
\__fp_logb_aux_o:w
26717 \cs_new:Npn \__fp_logb_o:w ? \s__fp \__fp_chk:w #1#2; @
26718 {
26719     \if_case:w #1 \exp_stop_f:
26720         \__fp_case_use:nw
26721         { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { logb } }
26722     \or: \exp_after:wN \__fp_logb_aux_o:w
26723     \or: \__fp_case_return_o:Nw \c_inf_fp
26724     \else: \__fp_case_return_same_o:w
26725     \fi:
26726     \s__fp \__fp_chk:w #1 #2;
26727 }
26728 \cs_new:Npn \__fp_logb_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 #4 ;
26729 {
26730     \exp_after:wN \__fp_parse:n \exp_after:wN
26731     { \int_value:w \int_eval:w #3 - 1 \exp_after:wN }
26732 }

```

(End of definition for __fp_logb_o:w and __fp_logb_aux_o:w.)

```

\__fp_sign_o:w Find the sign of the floating point: nan, +0, -0, +1 or -1.
\__fp_sign_aux_o:w
26733 \cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1#2; @
26734 {
26735     \if_case:w #1 \exp_stop_f:
26736         \__fp_case_return_same_o:w
26737     \or: \exp_after:wN \__fp_sign_aux_o:w
26738     \or: \exp_after:wN \__fp_sign_aux_o:w
26739     \else: \__fp_case_return_same_o:w
26740     \fi:
26741     \s__fp \__fp_chk:w #1 #2;
26742 }
26743 \cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 ;
26744 { \exp_after:wN \__fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ }

```

(End of definition for __fp_sign_o:w and __fp_sign_aux_o:w.)

`_fp_set_sign_o:w` This function is used for the unary minus and for `abs`. It leaves the sign of `nan` invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on #1. It also expands after itself in the input stream, just like `_fp+_o:ww`.

```

26745 \cs_new:Npn \_fp_set_sign_o:w #1 \s__fp \_fp_chk:w #2#3#4; @
26746 {
26747   \exp_after:wN \_fp_exp_after_o:w
26748   \exp_after:wN \s__fp
26749   \exp_after:wN \_fp_chk:w
26750   \exp_after:wN #2
26751   \int_value:w
26752   \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
26753   #4;
26754 }

```

(End of definition for `_fp_set_sign_o:w`.)

74.6 Operations on tuples

`_fp_tuple_set_sign_o:w` Two cases: `abs(<tuple>)` for which #1 is 0 (invalid for tuples) and `-<tuple>` for which
`_fp_tuple_set_sign_aux_o:Nnw` #1 is 2. In that case, map over all items in the tuple an auxiliary that dispatches to the
`_fp_tuple_set_sign_aux_o:w` type-appropriate sign-flipping function.

```

26755 \cs_new:Npn \_fp_tuple_set_sign_o:w #1#2 @
26756 {
26757   \if_meaning:w 2 #1
26758     \exp_after:wN \_fp_tuple_set_sign_aux_o:Nnw
26759     \fi:
26760     \_fp_invalid_operation_o:nw { abs }
26761     #2
26762   }
26763 \cs_new:Npn \_fp_tuple_set_sign_aux_o:Nnw #1#2
26764 { \_fp_tuple_map_o:nw \_fp_tuple_set_sign_aux_o:w }
26765 \cs_new:Npn \_fp_tuple_set_sign_aux_o:w #1#2 ;
26766 {
26767   \_fp_change_func_type:NNN #1 \_fp_set_sign_o:w
26768   \_fp_parse_apply_unary_error:NNw
26769   2 #1 #2 ; @
26770 }

```

(End of definition for `_fp_tuple_set_sign_o:w`, `_fp_tuple_set_sign_aux_o:Nnw`, and `_fp_tuple_set_sign_aux_o:w`.)

`_fp*_tuple_o:ww` For `<number>*<tuple>` and `<tuple>*<number>` and `<tuple>/<number>`, loop through the
`_fp_tuple*_o:ww` `<tuple>` some code that multiplies or divides by the appropriate `<number>`. Importantly
`_fp_tuple/_o:ww` we need to dispatch according to the type, and we make sure to apply the operator in the correct order.

```

26771 \cs_new:cpn { \_fp*_tuple_o:ww } #1 ;
26772 { \_fp_tuple_map_o:nw { \_fp_binary_type_o:Nww * #1 ; } }
26773 \cs_new:cpn { \_fp_tuple*_o:ww } #1 ; #2 ;
26774 { \_fp_tuple_map_o:nw { \_fp_binary_rev_type_o:Nww * #2 ; } #1 ; }
26775 \cs_new:cpn { \_fp_tuple/_o:ww } #1 ; #2 ;
26776 { \_fp_tuple_map_o:nw { \_fp_binary_rev_type_o:Nww / #2 ; } #1 ; }

```

(End of definition for `__fp*_tuple_o:ww`, `__fp_tuple*_o:ww`, and `__fp_tuple_/o:ww`.)

`__fp_tuple+_tuple_o:ww` Check the two tuples have the same number of items and map through these a helper
`__fp_tuple-_tuple_o:ww` that dispatches appropriately depending on the types. This means $(1,2)+((1,1),2)$
gives $(\text{nan},4)$.

```
26777 \cs_set_protected:Npn \__fp_tmp:w #1
26778   {
26779     \cs_new:cpn { __fp_tuple_#1_tuple_o:ww }
26780       \s__fp_tuple \__fp_tuple_chk:w ##1 ;
26781       \s__fp_tuple \__fp_tuple_chk:w ##2 ;
26782     {
26783       \int_compare:nNnTF
26784         { \__fp_array_count:n {##1} } = { \__fp_array_count:n {##2} }
26785         { \__fp_tuple_mapthread_o:nww { \__fp_binary_type_o:Nww #1 } }
26786         { \__fp_invalid_operation_o:nww #1 }
26787       \s__fp_tuple \__fp_tuple_chk:w {##1} ;
26788       \s__fp_tuple \__fp_tuple_chk:w {##2} ;
26789     }
26790   }
26791 \__fp_tmp:w +
26792 \__fp_tmp:w -
```

(End of definition for `__fp_tuple+_tuple_o:ww` and `__fp_tuple-_tuple_o:ww`.)

```
26793 </package>
```

Chapter 75

l3fp-extended implementation

```
26794 <*package>
```

```
26795 <@@=fp>
```

75.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

$$\{ \langle a_1 \rangle \} \{ \langle a_2 \rangle \} \{ \langle a_3 \rangle \} \{ \langle a_4 \rangle \} \{ \langle a_5 \rangle \} \{ \langle a_6 \rangle \} ;$$

where each $\langle a_i \rangle$ is exactly 4 digits (ranging from 0000 to 9999), except $\langle a_1 \rangle$, which may be any “not-too-large” non-negative integer, with or without leading zeros. Here, “not-too-large” depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number a corresponding to the representation above is $a = \sum_{i=1}^6 \langle a_i \rangle \cdot 10^{-4i}$.

Most functions we define here have the form

```
\_fp_fixed_<calculation>:wnn <operand1> ; <operand2> ; {<continuation>}
```

They perform the $\langle \text{calculation} \rangle$ on the two $\langle \text{operands} \rangle$, then feed the result (6 brace groups followed by a semicolon) to the $\langle \text{continuation} \rangle$, responsible for the next step of the calculation. Some functions only accept an N-type $\langle \text{continuation} \rangle$. This allows constructions such as

```
\_fp_fixed_add:wnn <X1> ; <X2> ;  
\_fp_fixed_mul:wnn <X3> ;  
\_fp_fixed_add:wnn <X4> ;
```


to compute $(X_1 + X_2) \cdot X_3 + X_4$. This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using `__fp_fixed_to_float_o:wN`. This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

75.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_t1` The fixed-point number 1, used in `l3fp-expo`.

```
26796 \t1_const:Nn \c__fp_one_fixed_t1
26797   { {10000} {0000} {0000} {0000} {0000} {0000} ; }
```

(End of definition for `\c__fp_one_fixed_t1`.)

`__fp_fixed_continue:wn` This function simply calls the next function.

```
26798 \cs_new:Npn \__fp_fixed_continue:wn #1; #2 { #2 #1; }
```

(End of definition for `__fp_fixed_continue:wn`.)

`__fp_fixed_add_one:wN` `__fp_fixed_add_one:wN <a> ; <continuation>`

This function adds 1 to the fixed point `<a>`, by changing a_1 to $10000 + a_1$, then calls the `<continuation>`. This requires $a_1 + 10000 < 2^{31}$.

```
26799 \cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
26800   {
26801     \exp_after:wN #3 \exp_after:wN
26802     { \int_value:w \__fp_int_eval:w \c__fp_myriad_int + #1 } #2 ;
26803   }
```

(End of definition for `__fp_fixed_add_one:wN`.)

`__fp_fixed_div_myriad:wn` Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group `#1` may have any number of digits, and we must split `#1` into the new first group and a second group of exactly 4 digits. The choice of shifts allows `#1` to be in the range $[0, 5 \cdot 10^8 - 1]$.

```
26804 \cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
26805   {
26806     \exp_after:wN \__fp_fixed_mul_after:wN
26807     \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
26808     \exp_after:wN \__fp_pack:NNNNw
26809     \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
26810     + #1 ; {#2}{#3}{#4}{#5};
26811   }
```

(End of definition for `__fp_fixed_div_myriad:wn`.)

`__fp_fixed_mul_after:wN` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the `<continuation>` `#3` in front.

```
26812 \cs_new:Npn \__fp_fixed_mul_after:wN #1; #2; #3 { #3 {#1} #2; }
```

(End of definition for `__fp_fixed_mul_after:wN`.)

75.3 Multiplying a fixed point number by a short one

`_fp_fixed_mul_short:wnn` `_fp_fixed_mul_short:wnn`
`{\langle a_1 \rangle} {\langle a_2 \rangle} {\langle a_3 \rangle} {\langle a_4 \rangle} {\langle a_5 \rangle} {\langle a_6 \rangle} ;`
`{\langle b_0 \rangle} {\langle b_1 \rangle} {\langle b_2 \rangle} ; {\langle continuation \rangle}`
 Computes the product $c = ab$ of $a = \sum_i \langle a_i \rangle 10^{-4i}$ and $b = \sum_i \langle b_i \rangle 10^{-4i}$, rounds it to the closest multiple of 10^{-24} , and leaves `\langle continuation \rangle` `{\langle c_1 \rangle} ... {\langle c_6 \rangle}`; in the input stream, where each of the `\langle c_i \rangle` are blocks of 4 digits, except `\langle c_1 \rangle`, which is any TeX integer. Note that indices for `\langle b \rangle` start at 0: for instance a second operand of `{0001}{0000}{0000}` leaves the first operand unchanged (rather than dividing it by 10^4 , as `_fp_fixed_mul:wnn` would).

```

26813 \cs_new:Npn \_fp_fixed_mul_short:wnn #1#2#3#4#5#6; #7#8#9;
26814 {
26815   \exp_after:wN \_fp_fixed_mul_after:wnn
26816   \int_value:w \_fp_int_eval:w \c_fp_leading_shift_int
26817     + #1*#7
26818   \exp_after:wN \_fp_pack:NNNNw
26819   \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
26820     + #1*#8 + #2*#7
26821   \exp_after:wN \_fp_pack:NNNNw
26822   \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
26823     + #1*#9 + #2*#8 + #3*#7
26824   \exp_after:wN \_fp_pack:NNNNw
26825   \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
26826     + #2*#9 + #3*#8 + #4*#7
26827   \exp_after:wN \_fp_pack:NNNNw
26828   \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
26829     + #3*#9 + #4*#8 + #5*#7
26830   \exp_after:wN \_fp_pack:NNNNw
26831   \int_value:w \_fp_int_eval:w \c_fp_trailing_shift_int
26832     + #4*#9 + #5*#8 + #6*#7
26833     + ( #5*#9 + #6*#8 + #6*#9 / \c_fp_myriad_int )
26834     / \c_fp_myriad_int ; ;
26835 }

```

(End of definition for `_fp_fixed_mul_short:wnn`.)

75.4 Dividing a fixed point number by a small integer

`_fp_fixed_div_int:wnN` `_fp_fixed_div_int:wnN` `\langle a \rangle ; \langle n \rangle ; \langle continuation \rangle`
`_fp_fixed_div_int:wnN` Divides the fixed point number `\langle a \rangle` by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds
`_fp_fixed_div_int_auxi:wnn` the result to the `\langle continuation \rangle`. There is no bound on a_1 .
`_fp_fixed_div_int_auxii:wnn` The arguments of the `i` auxiliary are 1: one of the a_i , 2: n , 3: the `ii` or the `iii`
`_fp_fixed_div_int_pack:Nw` auxiliary. It computes a (somewhat tight) lower bound Q_i for the ratio a_i/n .
`_fp_fixed_div_int_after:Nw` The `ii` auxiliary receives Q_i , n , and a_i as arguments. It adds Q_i to a surrounding
 integer expression, and starts a new one with the initial value 9999, which ensures that
 the result of this expression has 5 digits. The auxiliary also computes $a_i - n \cdot Q_i$, placing
 the result in front of the 4 digits of a_{i+1} . The resulting $a'_{i+1} = 10^4(a_i - n \cdot Q_i) + a_{i+1}$
 serves as the first argument for a new call to the `i` auxiliary.

When the `iii` auxiliary is called, the situation looks like this:

```

\__fp_fixed_div_int_after:Nw <continuation>
-1 + Q1
\__fp_fixed_div_int_pack:Nw 9999 + Q2
\__fp_fixed_div_int_pack:Nw 9999 + Q3
\__fp_fixed_div_int_pack:Nw 9999 + Q4
\__fp_fixed_div_int_pack:Nw 9999 + Q5
\__fp_fixed_div_int_pack:Nw 9999
\__fp_fixed_div_int_auxii:wnn Q6 ; {<n>} {<a6

```

where expansion is happening from the last line up. The `iii` auxiliary adds $Q_6 + 2 \simeq a_6/n + 1$ to the last 9999, giving the integer closest to $10000 + a_6/n$.

Each `pack` auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the `pack` auxiliary thus produces one brace group. The last brace group is produced by the `after` auxiliary, which places the `<continuation>` as appropriate.

```

26836 \cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
26837   {
26838     \exp_after:wN \__fp_fixed_div_int_after:Nw
26839     \exp_after:wN #8
26840     \int_value:w \__fp_int_eval:w - 1
26841     \__fp_fixed_div_int:wnN
26842     #1; {#7} \__fp_fixed_div_int_auxi:wnn
26843     #2; {#7} \__fp_fixed_div_int_auxi:wnn
26844     #3; {#7} \__fp_fixed_div_int_auxi:wnn
26845     #4; {#7} \__fp_fixed_div_int_auxi:wnn
26846     #5; {#7} \__fp_fixed_div_int_auxi:wnn
26847     #6; {#7} \__fp_fixed_div_int_auxii:wnn ;
26848   }
26849 \cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
26850   {
26851     \exp_after:wN #3
26852     \int_value:w \__fp_int_eval:w #1 / #2 - 1 ;
26853     {#2}
26854     {#1}
26855   }
26856 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
26857   {
26858     + #1
26859     \exp_after:wN \__fp_fixed_div_int_pack:Nw
26860     \int_value:w \__fp_int_eval:w 9999
26861     \exp_after:wN \__fp_fixed_div_int:wnN
26862     \int_value:w \__fp_int_eval:w #3 - #1*#2 \__fp_int_eval_end:
26863   }
26864 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + 2 ; }
26865 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
26866 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }

```

(End of definition for `__fp_fixed_div_int:wwN` and others.)

75.5 Adding and subtracting fixed points

```

\__fp_fixed_add:wnn          \__fp_fixed_add:wnn <a> ; <b> ; {<continuation>}
\__fp_fixed_sub:wnn
\__fp_fixed_add:Nnnnnwnn
\__fp_fixed_add:nnNnnnwn
\__fp_fixed_add_pack:NNNNNwn
\__fp_fixed_add_after:NNNNNwn

```

Computes $a+b$ (resp. $a-b$) and feeds the result to the $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 \leq 114748$, its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, a_1, \dots, a_4 , the rest of a , and b_1 and b_2 . The second auxiliary receives the rest of a , the sign multiplying b , the rest of b , and the $\langle continuation \rangle$ as arguments. After going down through the various level, we go back up, packing digits and bringing the $\langle continuation \rangle$ (#8, then #7) from the end of the argument list to its start.

```

26867 \cs_new:Npn \__fp_fixed_add:wnn { \__fp_fixed_add:Nnnnnwnn + }
26868 \cs_new:Npn \__fp_fixed_sub:wnn { \__fp_fixed_add:Nnnnnwnn - }
26869 \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
26870 {
26871   \exp_after:wN \__fp_fixed_add_after:NNNNNwn
26872   \int_value:w \__fp_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
26873   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
26874   \int_value:w \__fp_int_eval:w 1 9999 9998 + #4#5
26875   \__fp_fixed_add:nnNnnwn #6 #1
26876 }
26877 \cs_new:Npn \__fp_fixed_add:nnNnnwn #1#2 #3 #4#5 #6#7 ; #8
26878 {
26879   #3 #4#5
26880   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
26881   \int_value:w \__fp_int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
26882 }
26883 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
26884 { + #1 ; {#7} {#2#3#4#5} {#6} }
26885 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
26886 { #7 {#1#2#3#4#5} {#6} }

```

(End of definition for `__fp_fixed_add:wnn` and others.)

75.6 Multiplying fixed points

```

\__fp_fixed_mul:wnn
\__fp_fixed_mul:nnnnnnnw

```

`__fp_fixed_mul:wnn` $\langle a \rangle$; $\langle b \rangle$; $\langle continuation \rangle$
 Computes $a \times b$ and feeds the result to $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for \TeX macros. Note that we don't need to obtain an exact rounding, contrarily to the $*$ operator, so things could be harder. We wish to perform carries in

$$\begin{aligned}
 a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
 & + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
 & + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
 & + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
 & + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
 & \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
 & \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24} + O(10^{-24}),
 \end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on a_1, \dots, a_4 and b_1, \dots, b_4 , while the last 6 terms only depend on a_1, a_2, a_5, a_6 , and the corresponding parts of b . Hence, the first function grabs a_1, \dots, a_4 , the rest of a , and b_1, \dots, b_4 , and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives $a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6$ and finally the $\langle continuation \rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The $\langle continuation \rangle$ is finally placed in front of the 6 brace groups by `_fp_fixed_mul_after:wwn`.

```

26887 \cs_new:Npn \_fp_fixed_mul:wwn #1#2#3#4 #5; #6#7#8#9
26888 {
26889   \exp_after:wN \_fp_fixed_mul_after:wwn
26890   \int_value:w \_fp_int_eval:w \c__fp_leading_shift_int
26891   \exp_after:wN \_fp_pack:NNNNNw
26892   \int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
26893   + #1*#6
26894   \exp_after:wN \_fp_pack:NNNNNw
26895   \int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
26896   + #1*#7 + #2*#6
26897   \exp_after:wN \_fp_pack:NNNNNw
26898   \int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
26899   + #1*#8 + #2*#7 + #3*#6
26900   \exp_after:wN \_fp_pack:NNNNNw
26901   \int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
26902   + #1*#9 + #2*#8 + #3*#7 + #4*#6
26903   \exp_after:wN \_fp_pack:NNNNNw
26904   \int_value:w \_fp_int_eval:w \c__fp_trailing_shift_int
26905   + #2*#9 + #3*#8 + #4*#7
26906   + ( #3*#9 + #4*#8
26907     + \_fp_fixed_mul:nnnnnnw #5 {#6}{#7} {#1}{#2}
26908   )
26909 \cs_new:Npn \_fp_fixed_mul:nnnnnnw #1#2 #3#4 #5#6 #7#8 ;
26910 {
26911   #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c__fp_myriad_int
26912   + #1*#3 + #5*#7 ; ;
26913 }

```

(End of definition for `_fp_fixed_mul:wwn` and `_fp_fixed_mul:nnnnnnw`.)

75.7 Combining product and sum of fixed points

```

\_fp_fixed_mul_add:wwwn \_fp_fixed_mul_add:wwwn <a> ; <b> ; <c> ; {<continuation>}
\_fp_fixed_mul_sub_back:wwwn \_fp_fixed_mul_sub_back:wwwn <a> ; <b> ; <c> ; {<continuation>}
\_fp_fixed_mul_one_minus_mul:wwn \_fp_fixed_mul_one_minus_mul:wwn <a> ; <b> ; {<continuation>}

```

Sometimes called FMA (fused multiply-add), these functions compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the $\langle continuation \rangle$. Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We perform carries in

$$\begin{aligned}
a \times b + c = & (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where $c_1 c_2$, $c_3 c_4$, $c_5 c_6$ denote the 8-digit number obtained by juxtaposing the two blocks of digits of c , and \cdot denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to 10^{-4} , is empty), with $c_1 c_2$ in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing `+ c_5 c_6 ; {continuation}`; `.`. The `+ c_5 c_6` piece, which is omitted for `_fp_fixed_one_minus_mul:wwn`, is taken in the integer expression for the 10^{-24} level.

```

26914 \cs_new:Npn \_fp\_fixed\_mul\_add:wwwn #1; #2; #3#4#5#6#7#8;
26915 {
26916   \exp\_after:wN \_fp\_fixed\_mul\_after:wwn
26917   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
26918   \exp\_after:wN \_fp\_pack\_big:NNNNNNw
26919   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int + #3 #4
26920   \_fp\_fixed\_mul\_add:Nwnnnwnnn +
26921   + #5 #6 ; #2 ; #1 ; #2 ; +
26922   + #7 #8 ; ;
26923 }
26924 \cs_new:Npn \_fp\_fixed\_mul\_sub\_back:wwwn #1; #2; #3#4#5#6#7#8;
26925 {
26926   \exp\_after:wN \_fp\_fixed\_mul\_after:wwn
26927   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
26928   \exp\_after:wN \_fp\_pack\_big:NNNNNNw
26929   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int + #3 #4
26930   \_fp\_fixed\_mul\_add:Nwnnnwnnn -
26931   + #5 #6 ; #2 ; #1 ; #2 ; -
26932   + #7 #8 ; ;
26933 }
26934 \cs_new:Npn \_fp\_fixed\_one\_minus\_mul:wwn #1; #2;
26935 {
26936   \exp\_after:wN \_fp\_fixed\_mul\_after:wwn
26937   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
26938   \exp\_after:wN \_fp\_pack\_big:NNNNNNw
26939   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int +
26940   1 0000 0000
26941   \_fp\_fixed\_mul\_add:Nwnnnwnnn -
26942   ; #2 ; #1 ; #2 ; -
26943   ; ;
26944 }

```

(End of definition for `_fp_fixed_mul_add:wwwn`, `_fp_fixed_mul_sub_back:wwwn`, and `_fp_fixed_mul_one_minus_mul:wwn`.)

```
\_fp\_fixed\_mul\_add:Nwnnnwnnn
    \_fp\_fixed\_mul\_add:Nwnnnwnnn <op> + <c3> <c4> ;
    <b> ; <a> ; <b> ; <op>
    + <c5> <c6> ;
```

Here, `<op>` is either + or -. Arguments #3, #4, #5 are `<b1>`, `<b2>`, `<b3>`; arguments #7, #8, #9 are `<a1>`, `<a2>`, `<a3>`. We can build three levels: $a_1 \cdot b_1$ for 10^{-8} , $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12} , and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16} . The a - b products use the sign #1. Note that #2 is empty for `_fp_fixed_one_minus_mul:wwn`. We call the `ii` auxiliary for levels 10^{-20} and 10^{-24} , keeping the pieces of `<a>` we've read, but not ``, since there is another copy later in the input stream.

```
26945 \cs_new:Npn \_fp\_fixed\_mul\_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
26946 {
26947   #1 #7*#3
26948   \exp_after:wN \_fp\_pack\_big:NNNNNNw
26949   \int_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int
26950   #1 #7*#4 #1 #8*#3
26951   \exp_after:wN \_fp\_pack\_big:NNNNNNw
26952   \int_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int
26953   #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
26954   \exp_after:wN \_fp\_pack\_big:NNNNNNw
26955   \int_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int
26956   #1 \_fp\_fixed\_mul\_add:nnnnwnnn {#7}{#8}{#9}
26957 }
```

(End of definition for `_fp_fixed_mul_add:Nwnnnwnnn`.)

```
\_fp\_fixed\_mul\_add:nnnnwnnn
    \_fp\_fixed\_mul\_add:nnnnwnnn <a> ; <b> ; <op>
    + <c5> <c6> ;
```

Level 10^{-20} is $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$, multiplied by the sign, which was inserted by the `i` auxiliary. Then we prepare level 10^{-24} . We don't have access to all parts of `<a>` and `` needed to make all products. Instead, we prepare the partial expressions

$$b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$

$$b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.$$

Obviously, those expressions make no mathematical sense: we complete them with $a_5 \cdot$ and $\cdot b_5$, and with $a_6 \cdot b_1 + a_5 \cdot$ and $\cdot b_5 + a_1 \cdot b_6$, and of course with the trailing $+ c_5 c_6$. To do all this, we keep a_1, a_5, a_6 , and the corresponding pieces of ``.

```
26958 \cs_new:Npn \_fp\_fixed\_mul\_add:nnnnwnnn #1#2#3#4#5; #6#7#8#9
26959 {
26960   ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
26961   \exp_after:wN \_fp\_pack\_big:NNNNNNw
26962   \int_value:w \_fp\_int\_eval:w \c\_fp\_big\_trailing\_shift\_int
26963   \_fp\_fixed\_mul\_add:nnnnwnnn
26964   { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
26965   { #7 + #4*#8 + #3*#9 + #2 }
26966   {#1} #5;
26967   {#6}
26968 }
```

(End of definition for `_fp_fixed_mul_add:nnnnwnnnn`.)

```
\_fp_fixed_mul_add:nnnnwnnwN {<partial1>} {<partial2>}
  {<a1>} {<a5>} {<a6>} ; {<b1>} {<b5>} {<b6>} ;
  <op> + <c5> <c6> ;
```

Complete the `<partial1>` and `<partial2>` expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28} . The trailing $+c_5c_6$ is taken into the expression for level 10^{-24} . Note that the total of level 10^{-24} is in the interval $[-5 \cdot 10^8, 6 \cdot 10^8]$ (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See `l3fp-aux` for the definition of the shifts and packing auxiliaries.

```
26969 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
26970   {
26971     #9 (#4* #1 *#7)
26972     #9 (#5*#6+#4* #2 *#7+#3*#8) / \c__fp_myriad_int
26973   }
```

(End of definition for `_fp_fixed_mul_add:nnnnwnnwN`.)

75.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures (“extended-precision” numbers, in short, “ep”), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is $0.\langle digits \rangle \cdot 10^{\langle exponent \rangle}$. This convention differs from floating points.

Converts an extended-precision number with an exponent at most 4 and a first block less than 10^8 to a fixed point number whose first block has 12 digits, hopefully starting with many zeros.

```
\_fp_ep_to_fixed:wwn
\_fp_ep_to_fixed_auxi:www
\_fp_ep_to_fixed_auxii:nnnnnnwn

26974 \cs_new:Npn \_fp_ep_to_fixed:wwn #1,#2
26975   {
26976     \exp_after:wN \_fp_ep_to_fixed_auxi:www
26977     \int_value:w \_fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
26978     \exp:w \exp_end_continue_f:w
26979     \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
26980   }
26981 \cs_new:Npn \_fp_ep_to_fixed_auxi:www #1#; #2; #3#4#5#6#7;
26982   {
26983     \_fp_pack_eight:wNNNNNNNN
26984     \_fp_pack_twice_four:wNNNNNNNN
26985     \_fp_pack_twice_four:wNNNNNNNN
26986     \_fp_pack_twice_four:wNNNNNNNN
26987     \_fp_ep_to_fixed_auxii:nnnnnnwn ;
26988     #2 #1#3#4#5#6#7 0000 !
26989   }
26990 \cs_new:Npn \_fp_ep_to_fixed_auxii:nnnnnnwn #1#2#3#4#5#6#7; #8! #9
26991   { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }
```

(End of definition for `_fp_ep_to_fixed:wwn`, `_fp_ep_to_fixed_auxi:www`, and `_fp_ep_to_fixed_auxii:nnnnnnwn`.)


```

\__fp_ep_to_ep:wwN
\__fp_ep_to_ep_loop:N
\__fp_ep_to_ep_end:www
\__fp_ep_to_ep_zero:ww

```

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation #8 is placed before the resulting exponent-mantissa pair. The input exponent may in fact be given as an integer expression. The loop auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the end auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with __fp_use_i:ww any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```

26992 \cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
26993 {
26994   \exp_after:wN #8
26995   \int_value:w \__fp_int_eval:w #1 + 4
26996   \exp_after:wN \use_i:nn
26997   \exp_after:wN \__fp_ep_to_ep_loop:N
26998   \int_value:w \__fp_int_eval:w 1 0000 0000 + #2 \__fp_int_eval_end:
26999   #3#4#5#6#7 ; ; !
27000 }
27001 \cs_new:Npn \__fp_ep_to_ep_loop:N #1
27002 {
27003   \if_meaning:w 0 #1
27004   - 1
27005   \else:
27006     \__fp_ep_to_ep_end:www #1
27007   \fi:
27008   \__fp_ep_to_ep_loop:N
27009 }
27010 \cs_new:Npn \__fp_ep_to_ep_end:www
27011 #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
27012 {
27013   \fi:
27014   \if_meaning:w ; #1
27015   - 2 * \c__fp_max_exponent_int
27016   \__fp_ep_to_ep_zero:ww
27017   \fi:
27018   \__fp_pack_twice_four:wNNNNNNNN
27019   \__fp_pack_twice_four:wNNNNNNNN
27020   \__fp_pack_twice_four:wNNNNNNNN
27021   \__fp_use_i:ww , ;
27022   #1 #2 0000 0000 0000 0000 0000 0000 ;
27023 }
27024 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
27025 { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }

```

(End of definition for __fp_ep_to_ep:wwN and others.)

```

\__fp_ep_compare:www
\__fp_ep_compare_aux:www

```

In l3fp-trig we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in [1000, 9999].

```

27026 \cs_new:Npn \__fp_ep_compare:www #1,#2#3#4#5#6#7;

```

```

27027 { \_fp_ep_compare_aux:www #1}{#2}{#3}{#4}{#5}; #6#7; }
27028 \cs_new:Npn \_fp_ep_compare_aux:www #1;#2;#3,#4#5#6#7#8#9;
27029 {
27030   \if_case:w
27031     \_fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
27032     \if_int_compare:w #2 = #8#9 \exp_stop_f:
27033       0
27034     \else:
27035       \if_int_compare:w #2 < #8#9 - \fi: 1
27036     \fi:
27037   \or: 1
27038   \else: -1
27039   \fi:
27040 }

```

(End of definition for _fp_ep_compare:www and _fp_ep_compare_aux:www.)

_fp_ep_mul:wwwN Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas #2 and #4 as fixed point numbers, and sum the exponents #1 and #3. The result's first block is in [100,9999].

```

27041 \cs_new:Npn \_fp_ep_mul:wwwN #1,#2; #3,#4;
27042 {
27043   \_fp_ep_to_ep:wwN #3,#4;
27044   \_fp_fixed_continue:wn
27045   {
27046     \_fp_ep_to_ep:wwN #1,#2;
27047     \_fp_ep_mul_raw:wwwN
27048   }
27049   \_fp_fixed_continue:wn
27050 }
27051 \cs_new:Npn \_fp_ep_mul_raw:wwwN #1,#2; #3,#4; #5
27052 {
27053   \_fp_fixed_mul:wn #2; #4;
27054   { \exp_after:wN #5 \int_value:w \_fp_int_eval:w #1 + #3 , }
27055 }

```

(End of definition for _fp_ep_mul:wwwN and _fp_ep_mul_raw:wwwN.)

75.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in l3fp-basics for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call $\langle n \rangle$ the numerator and $\langle d \rangle$ the denominator. After a simple normalization step, we can assume that $\langle n \rangle \in [0.1, 1)$ and $\langle d \rangle \in [0.1, 1)$, and compute $\langle n \rangle / (10 \langle d \rangle) \in (0.01, 1)$. In terms of the 6 blocks of digits $\langle n_1 \rangle \cdots \langle n_6 \rangle$ and the 6 blocks $\langle d_1 \rangle \cdots \langle d_6 \rangle$, the condition translates to $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$.

We first find an integer estimate $a \simeq 10^8/\langle d \rangle$ by computing

$$\begin{aligned}\alpha &= \left[\frac{10^9}{\langle d_1 \rangle + 1} \right] \\ \beta &= \left[\frac{10^9}{\langle d_1 \rangle} \right] \\ a &= 10^3\alpha + (\beta - \alpha) \cdot \left(10^3 - \left[\frac{\langle d_2 \rangle}{10} \right] \right) - 1250,\end{aligned}$$

where $\left[\frac{\bullet}{\bullet} \right]$ denotes ε -TeX's rounding division, which rounds ties away from zero. The idea is to interpolate between $10^3\alpha$ and $10^3\beta$ with a parameter $\langle d_2 \rangle/10^4$, so that when $\langle d_2 \rangle = 0$ one gets $a = 10^3\beta - 1250 \simeq 10^{12}/\langle d_1 \rangle \simeq 10^8/\langle d \rangle$, while when $\langle d_2 \rangle = 9999$ one gets $a = 10^3\alpha - 1250 \simeq 10^{12}/(\langle d_1 \rangle + 1) \simeq 10^8/\langle d \rangle$. The shift by 1250 helps to ensure that a is an underestimate of the correct value. We shall prove that

$$1 - 1.755 \cdot 10^{-5} < \frac{\langle d \rangle a}{10^8} < 1.$$

We can then compute the inverse of $\langle d \rangle a/10^8 = 1 - \epsilon$ using the relation $1/(1 - \epsilon) \simeq (1 + \epsilon)(1 + \epsilon^2) + \epsilon^4$, which is correct up to a relative error of $\epsilon^5 < 1.6 \cdot 10^{-24}$. This allows us to find the desired ratio as

$$\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} ((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4).$$

Let us prove the upper bound first (multiplied by 10^{15}). Note that $10^7\langle d \rangle < 10^3\langle d_1 \rangle + 10^{-1}(\langle d_2 \rangle + 1)$, and that ε -TeX's division $\left[\frac{\langle d_2 \rangle}{10} \right]$ underestimates $10^{-1}(\langle d_2 \rangle + 1)$ by 0.5 at most, as can be checked for each possible last digit of $\langle d_2 \rangle$. Then,

$$10^7\langle d \rangle a < \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \left(\left(10^3 - \left[\frac{\langle d_2 \rangle}{10} \right] \right) \beta + \left[\frac{\langle d_2 \rangle}{10} \right] \alpha - 1250 \right) \quad (1)$$

$$< \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \quad (2)$$

$$\left(\left(10^3 - \left[\frac{\langle d_2 \rangle}{10} \right] \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left[\frac{\langle d_2 \rangle}{10} \right] \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left[\frac{\langle d_2 \rangle}{10} \right] \frac{10^9}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in $[\langle d_2 \rangle/10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2 \rangle/10] + a)(b - c[\langle d_2 \rangle/10]) \leq (b + ca)^2/(4c)$. Hence,

$$10^7\langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} \left(\langle d_1 \rangle + \frac{1}{2} + \frac{1}{4}10^{-3} - \frac{3}{8} \cdot 10^{-9}\langle d_1 \rangle(\langle d_1 \rangle + 1) \right)^2$$

Since $\langle d_1 \rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1 \rangle(\langle d_1 \rangle + 1)$, hence $10^7\langle d \rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller numbers, but nothing less than $0.374563\dots$, and going back through the derivation of

the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10} \right] - \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left[\frac{\langle d_2 \rangle}{10} \right] \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values $[y/10] = 0$ or $[y/10] = 100$, and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8 / \langle d \rangle \leq 10^9$, hence we can compute a safely as a $\text{T}_{\text{E}}\text{X}$ integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`_fp_ep_div:wwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the $\langle \text{continuation} \rangle$ once we are done. First normalize the inputs so that both first block lie in $[1000, 9999]$, then call `_fp_ep_div_esti:wwwn` $\langle \text{denominator} \rangle$ $\langle \text{numerator} \rangle$, responsible for estimating the inverse of the denominator.

```

27056 \cs_new:Npn \_fp\_ep\_div:wwwn #1,#2; #3,#4;
27057   {
27058     \_fp\_ep\_to\_ep:wwN #1,#2;
27059     \_fp\_fixed\_continue:wn
27060     {
27061       \_fp\_ep\_to\_ep:wwN #3,#4;
27062       \_fp\_ep\_div\_esti:wwwn
27063     }
27064   }

```

(End of definition for `_fp_ep_div:wwwn`.)

`_fp_ep_div_esti:wwwn` The `esti` function evaluates $\alpha = 10^9 / (\langle d_1 \rangle + 1)$, which is used twice in the expression for a , and combines the exponents `#1` and `#4` (with a shift by 1 because we later compute $\langle n \rangle / (10 \langle d \rangle)$). Then the `estii` function evaluates $10^9 + a$, and puts the exponent `#2` after the continuation `#7`: from there on we can forget exponents and focus on the mantissa. The `estiii` function multiplies the denominator `#7` by $10^{-8}a$ (obtained as a split into the single digit `#1` and two blocks of 4 digits, `#2#3#4#5` and `#6`). The result $10^{-8}a \langle d \rangle = (1 - \epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to `_fp_ep_div_epsi:wnNNNn`, which computes $10^{-9}a / (1 - \epsilon)$, that is, $1 / (10 \langle d \rangle)$ and we finally multiply this by the numerator `#8`.

```

27065 \cs_new:Npn \_fp\_ep\_div\_esti:wwwn #1,#2#3; #4,
27066   {
27067     \exp\_after:wN \_fp\_ep\_div\_estii:wwnnwwn
27068     \int\_value:w \_fp\_int\_eval:w 10 0000 0000 / ( #2 + 1 )
27069     \exp\_after:wN ;
27070     \int\_value:w \_fp\_int\_eval:w #4 - #1 + 1 ,
27071     {#2} #3;
27072   }
27073 \cs_new:Npn \_fp\_ep\_div\_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
27074   {

```

```

27075 \exp_after:wN \_fp_ep_div_estiii:NNNNNwwwn
27076 \int_value:w \_fp_int_eval:w 10 0000 0000 - 1750
27077 + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
27078 {#3}{#4}#5; #6; { #7 #2, }
27079 }
27080 \cs_new:Npn \_fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
27081 {
27082 \_fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
27083 \_fp_ep_div_epsi:wnNNNNNn {#1#2#3#4}#5#6
27084 \_fp_fixed_mul:wwn
27085 }

```

(End of definition for `_fp_ep_div_esti:wwwn`, `_fp_ep_div_estii:wwnwwn`, and `_fp_ep_div_estiii:NNNNNwwwn`.)

```

\_fp_ep_div_epsi:wnNNNNNn
\_fp_ep_div_eps_pack:NNNNNw
\_fp_ep_div_epsi:wnNNNNNn

```

The bounds shown above imply that the `epsi` function's first operand is $(1 - \epsilon)$ with $\epsilon \in [0, 1.755 \cdot 10^{-5}]$. The `epsi` function computes ϵ as $1 - (1 - \epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use `#1` (which is 9999). Then `epsi` evaluates $10^{-9}a/(1 - \epsilon)$ as $(1 + \epsilon^2)(1 + \epsilon)(10^{-9}a\epsilon) + 10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by ϵ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of `short_mul` and `div_myriad` is both faster and more precise than a simple `mul`.

```

27086 \cs_new:Npn \_fp_ep_div_epsi:wnNNNNNn #1#2#3#4#5#6;
27087 {
27088 \exp_after:wN \_fp_ep_div_epsi:wnNNNNNn
27089 \int_value:w \_fp_int_eval:w 1 9998 - #2
27090 \exp_after:wN \_fp_ep_div_eps_pack:NNNNNw
27091 \int_value:w \_fp_int_eval:w 1 9999 9998 - #3#4
27092 \exp_after:wN \_fp_ep_div_eps_pack:NNNNNw
27093 \int_value:w \_fp_int_eval:w 2 0000 0000 - #5#6 ; ;
27094 }
27095 \cs_new:Npn \_fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
27096 { + #1 ; {#2#3#4#5} {#6} }
27097 \cs_new:Npn \_fp_ep_div_epsi:wnNNNNNn 1#1; #2; #3#4#5#6#7#8
27098 {
27099 \_fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
27100 \_fp_fixed_add_one:wN
27101 \_fp_fixed_mul:wwn {10000} {#1} #2 ;
27102 {
27103 \_fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
27104 \_fp_fixed_div_myriad:wn
27105 \_fp_fixed_mul:wwn
27106 }
27107 \_fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
27108 }

```

(End of definition for `_fp_ep_div_epsi:wnNNNNNn`, `_fp_ep_div_eps_pack:NNNNNw`, and `_fp_ep_div_epsi:wnNNNNNn`.)

75.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have $x \in [0.01, 1)$. Then find an integer approximation $r \in [101, 1003]$ of $10^2/\sqrt{x}$, as the fixed point of iterations of the Newton method: essentially $r \mapsto (r + 10^8/(x_1 r))/2$, starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace 10^8 by a slightly larger number which ensures that $r^2 x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r , we set $y = 10^{-4} r^2 x$ (or rather, the correct power of 10 to get $y \simeq 1$) and compute $y^{-1/2}$ through another application of Newton's method. This time, the starting value is $z = 1$, each step maps $z \mapsto z(1.5 - 0.5yz^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2} r y^{-1/2}$.

First normalize the input, then check the parity of the exponent #1. If it is even, the result's exponent will be $-\#1/2$, otherwise it will be $(\#1 - 1)/2$ (except in the case where the input was an exact power of 100). The `auxii` function receives as #1 the result's exponent just computed, as #2 the starting value for the iteration giving r (the values 168 and 535 lead to the least number of iterations before convergence, on average), as #3 and #4 one empty argument and one 0, depending on the parity of the original exponent, as #5 and #6 the normalized mantissa ($\#5 \in [1000, 9999]$), and as #7 the continuation. It sets up the iteration giving r : the `esti` function thus receives the initial two guesses #2 and 0, an approximation #5 of $10^4 x$ (its first block of digits), and the empty/zero arguments #3 and #4, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```

27109 \cs_new:Npn \__fp_ep_isqrt:wwn #1,#2;
27110 {
27111   \__fp_ep_to_ep:wwN #1,#2;
27112   \__fp_ep_isqrt_auxi:wwn
27113 }
27114 \cs_new:Npn \__fp_ep_isqrt_auxi:wwn #1,
27115 {
27116   \exp_after:wN \__fp_ep_isqrt_auxii:wwnnwn
27117   \int_value:w \__fp_int_eval:w
27118   \int_if_odd:nTF {#1}
27119     { (1 - #1) / 2 , 535 , { 0 } { } }
27120     { 1 - #1 / 2 , 168 , { } { 0 } }
27121 }
27122 \cs_new:Npn \__fp_ep_isqrt_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
27123 {
27124   \__fp_ep_isqrt_esti:wwnnwn #2, 0, #5, {#3} {#4}
27125   {#5} #6 ; { #7 #1 , }
27126 }

```

(End of definition for `__fp_ep_isqrt:wwn`, `__fp_ep_isqrt_aux:wwn`, and `__fp_ep_isqrt_auxii:wwnnwn`.)

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate $(\langle prev \rangle + 1.005 \cdot 10^8 \text{ or } 9 / (\langle prev \rangle \cdot x))/2$, as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can check

by brute force that if #4 is empty (the original exponent was even), the process computes an integer slightly larger than $100/\sqrt{x}$, while if #4 is 0 (the original exponent was odd), the result is an integer slightly larger than $100/\sqrt{x/10}$. Once we are done, we evaluate $100r^2/2$ or $10r^2/2$ (when the exponent is even or odd, respectively) and feed that to `esti`. This third auxiliary finds $y_{\text{even}}/2 = 10^{-4}r^2x/2$ or $y_{\text{odd}}/2 = 10^{-5}r^2x/2$ (again, depending on earlier parity). A simple program shows that $y \in [1, 1.0201]$. The number $y/2$ is fed to `__fp_ep_isqrt_epsilon`, which computes $1/\sqrt{y}$, and we finally multiply the result by r .

```

27127 \cs_new:Npn \__fp_ep_isqrt_esti:wwnwn #1, #2, #3, #4
27128 {
27129   \if_int_compare:w #1 = #2 \exp_stop_f:
27130     \exp_after:wN \__fp_ep_isqrt_estii:wwnwn
27131   \fi:
27132   \exp_after:wN \__fp_ep_isqrt_esti:wwnwn
27133   \int_value:w \__fp_int_eval:w
27134     (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
27135   #1, #3, {#4}
27136 }
27137 \cs_new:Npn \__fp_ep_isqrt_estii:wwnwn #1, #2, #3, #4#5
27138 {
27139   \exp_after:wN \__fp_ep_isqrt_estiii:NNNNwewn
27140   \int_value:w \__fp_int_eval:w 1000 0000 + #2 * #2 #5 * 5
27141   \exp_after:wN , \int_value:w \__fp_int_eval:w 10000 + #2 ;
27142 }
27143 \cs_new:Npn \__fp_ep_isqrt_estiii:NNNNwewn 1#1#2#3#4#5#6, 1#7#8; #9;
27144 {
27145   \__fp_fixed_mul_short:wew #9; {#1} {#2#3#4#5} {#600} ;
27146   \__fp_ep_isqrt_epsilon:wN
27147   \__fp_fixed_mul_short:wew {#7} {#80} {0000} ;
27148 }

```

(End of definition for `__fp_ep_isqrt_esti:wwnwn`, `__fp_ep_isqrt_estii:wwnwn`, and `__fp_ep_isqrt_estiii:NNNNwewn`.)

`__fp_ep_isqrt_epsilon` Here, we receive a fixed point number $y/2$ with $y \in [1, 1.0201]$. Starting from $z = 1$ we iterate $z \mapsto z(3/2 - z^2y/2)$. In fact, we start from the first iteration $z = 3/2 - y/2$ to avoid useless multiplications. The `epsii` auxiliary receives z as #1 and y as #2.

```

27149 \cs_new:Npn \__fp_ep_isqrt_epsilon:wN #1;
27150 {
27151   \__fp_fixed_sub:wew {15000}{0000}{0000}{0000}{0000}{0000}; #1;
27152   \__fp_ep_isqrt_epsii:wew #1;
27153   \__fp_ep_isqrt_epsii:wew #1;
27154   \__fp_ep_isqrt_epsii:wew #1;
27155 }
27156 \cs_new:Npn \__fp_ep_isqrt_epsii:wew #1; #2;
27157 {
27158   \__fp_fixed_mul:wew #1; #1;
27159   \__fp_fixed_mul_sub_back:wew #2;
27160   {15000}{0000}{0000}{0000}{0000}{0000};
27161   \__fp_fixed_mul:wew #1;
27162 }

```

(End of definition for `__fp_ep_isqrt_epsilon:wN` and `__fp_ep_isqrt_epsii:wew`.)

75.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here should be called within an integer expression for the overall exponent of the floating point.

`_fp_ep_to_float_o:wwN`
`_fp_ep_inv_to_float_o:wwN` An extended-precision number is simply a comma-delimited exponent followed by a fixed point number. Leave the exponent in the current integer expression then convert the fixed point number.

```
27163 \cs_new:Npn \_fp_ep_to_float_o:wwN #1,
27164   { + \_fp_int_eval:w #1 \_fp_fixed_to_float_o:wN }
27165 \cs_new:Npn \_fp_ep_inv_to_float_o:wwN #1,#2;
27166   {
27167     \_fp_ep_div:wwwn 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
27168     \_fp_ep_to_float_o:wwN
27169   }
```

(End of definition for `_fp_ep_to_float_o:wwN` and `_fp_ep_inv_to_float_o:wwN`.)

`_fp_fixed_inv_to_float_o:wN` Another function which reduces to converting an extended precision number to a float.

```
27170 \cs_new:Npn \_fp_fixed_inv_to_float_o:wN
27171   { \_fp_ep_inv_to_float_o:wwN 0, }
```

(End of definition for `_fp_fixed_inv_to_float_o:wN`.)

`_fp_fixed_to_float_rad_o:wN` Converts the fixed point number #1 from degrees to radians then to a floating point number. This could perhaps remain in `l3fp-trig`.

```
27172 \cs_new:Npn \_fp_fixed_to_float_rad_o:wN #1;
27173   {
27174     \_fp_fixed_mul:wN #1; {5729}{5779}{5130}{8232}{0876}{7981};
27175     { \_fp_ep_to_float_o:wwN 2, }
27176   }
```

(End of definition for `_fp_fixed_to_float_rad_o:wN`.)

`_fp_fixed_to_float_o:wN`
`_fp_fixed_to_float_o:Nw` ... `_fp_int_eval:w` $\langle exponent \rangle$ `_fp_fixed_to_float_o:wN` $\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \{\langle a_3 \rangle\} \{\langle a_4 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\}$; $\langle sign \rangle$
yields

$\langle exponent' \rangle$; $\{\langle a'_1 \rangle\} \{\langle a'_2 \rangle\} \{\langle a'_3 \rangle\} \{\langle a'_4 \rangle\}$;

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \leq \langle a'_1 \rangle \leq 9999$. At this stage, we know that $\langle a_1 \rangle$ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8 .¹¹

```
27177 \cs_new:Npn \_fp_fixed_to_float_o:Nw #1#2;
27178   { \_fp_fixed_to_float_o:wN #2; #1 }
27179 \cs_new:Npn \_fp_fixed_to_float_o:wN #1#2#3#4#5#6; #7
27180   { % for the 8-digit-at-the-start thing
27181     + \_fp_int_eval:w \_fp_block_int
27182     \exp_after:wN \exp_after:wN
27183     \exp_after:wN \_fp_fixed_to_loop:N
```

¹¹Bruno: I must double check this assumption.


```

27184 \exp_after:wN \use_none:n
27185 \int_value:w \__fp_int_eval:w
27186 1 0000 0000 + #1 \exp_after:wN \__fp_use_none_stop_f:n
27187 \int_value:w 1#2 \exp_after:wN \__fp_use_none_stop_f:n
27188 \int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
27189 \int_value:w 1#5#6
27190 \exp_after:wN ;
27191 \exp_after:wN ;
27192 }
27193 \cs_new:Npn \__fp_fixed_to_loop:N #1
27194 {
27195 \if_meaning:w 0 #1
27196 - 1
27197 \exp_after:wN \__fp_fixed_to_loop:N
27198 \else:
27199 \exp_after:wN \__fp_fixed_to_loop_end:w
27200 \exp_after:wN #1
27201 \fi:
27202 }
27203 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
27204 {
27205 \if_meaning:w ; #1
27206 \exp_after:wN \__fp_fixed_to_float_zero:w
27207 \else:
27208 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
27209 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
27210 \exp_after:wN \__fp_fixed_to_float_pack:ww
27211 \exp_after:wN ;
27212 \fi:
27213 #1 #2 0000 0000 0000 0000 ;
27214 }
27215 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
27216 {
27217 - 2 * \c__fp_max_exponent_int ;
27218 {0000} {0000} {0000} {0000} ;
27219 }
27220 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
27221 {
27222 \if_int_compare:w #2 > 4 \exp_stop_f:
27223 \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
27224 \fi:
27225 ; #1 ;
27226 }
27227 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
27228 {
27229 \exp_after:wN \__fp_basics_pack_high:NNNNNw
27230 \int_value:w \__fp_int_eval:w 1 #1#2
27231 \exp_after:wN \__fp_basics_pack_low:NNNNNw
27232 \int_value:w \__fp_int_eval:w 1 #3#4 + 1 ;
27233 }

```

(End of definition for __fp_fixed_to_float_o:wN and __fp_fixed_to_float_o:Nw.)

```

27234 \</package>

```

Chapter 76

l3fp-expo implementation

```
27235 (*package)
27236 (@@=fp)

Unary functions.
\__fp_parse_word_exp:N
\__fp_parse_word_ln:N
\__fp_parse_word_fact:N
27237 \cs_new:Npn \__fp_parse_word_exp:N
27238   { \__fp_parse_unary_function:NNN \__fp_exp_o:w ? }
27239 \cs_new:Npn \__fp_parse_word_ln:N
27240   { \__fp_parse_unary_function:NNN \__fp_ln_o:w ? }
27241 \cs_new:Npn \__fp_parse_word_fact:N
27242   { \__fp_parse_unary_function:NNN \__fp_fact_o:w ? }

(End of definition for \__fp_parse_word_exp:N, \__fp_parse_word_ln:N, and \__fp_parse_word_fact:N.)
```

76.1 Logarithm

76.1.1 Work plan

As for many other functions, we filter out special cases in `__fp_ln_o:w`. Then `__fp_ln_npos_o:w` receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in [0.1, 1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code, $c \in [1, 10]$ is such that $0.7 \leq ac < 1.4$.

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1, 1)$. To compute its logarithm, we find a small integer $5 \leq c < 50$ such that $0.91 \leq ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of \ln near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \dots,$$

is not too difficult to compute.

76.1.2 Some constants

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of $\ln(5)$.

```

\c__fp_ln_i_fixed_tl 27243 \tl_const:Nn \c__fp_ln_i_fixed_tl { {0000}{0000}{0000}{0000}{0000}{0000};}
\c__fp_ln_ii_fixed_tl 27244 \tl_const:Nn \c__fp_ln_ii_fixed_tl { {6931}{4718}{0559}{9453}{0941}{7232};}
\c__fp_ln_iii_fixed_tl 27245 \tl_const:Nn \c__fp_ln_iii_fixed_tl {{{10986}{1228}{8668}{1096}{9139}{5245};}
\c__fp_ln_iv_fixed_tl 27246 \tl_const:Nn \c__fp_ln_iv_fixed_tl {{{13862}{9436}{1119}{8906}{1883}{4464};}
\c__fp_ln_vii_fixed_tl 27247 \tl_const:Nn \c__fp_ln_vii_fixed_tl {{{17917}{5946}{9228}{0550}{0081}{2477};}
\c__fp_ln_viii_fixed_tl 27248 \tl_const:Nn \c__fp_ln_viii_fixed_tl {{{19459}{1014}{9055}{3133}{0510}{5353};}
\c__fp_ln_ix_fixed_tl 27249 \tl_const:Nn \c__fp_ln_ix_fixed_tl {{{20794}{4154}{1679}{8359}{2825}{1696};}
\c__fp_ln_x_fixed_tl 27250 \tl_const:Nn \c__fp_ln_ix_fixed_tl {{{21972}{2457}{7336}{2193}{8279}{0490};}
27251 \tl_const:Nn \c__fp_ln_x_fixed_tl {{{23025}{8509}{2994}{0456}{8401}{7991};}

```

(End of definition for `\c__fp_ln_i_fixed_tl` and others.)

76.1.3 Sign, exponent, and special numbers

The logarithm of negative numbers (including $-\infty$ and -0) raises the “invalid” exception. The logarithm of $+0$ is $-\infty$, raising a division by zero exception. The logarithm of $+\infty$ or a nan is itself. Positive normal numbers call `__fp_ln_npos_o:w`.

```

27252 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
27253 {
27254   \if_meaning:w 2 #3
27255     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
27256   \fi:
27257   \if_case:w #2 \exp_stop_f:
27258     \__fp_case_use:nw
27259     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
27260   \or:
27261   \else:
27262     \__fp_case_return_same_o:w
27263   \fi:
27264   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
27265 }

```

(End of definition for `__fp_ln_o:w`.)

76.1.4 Absolute ln

We catch the case of a significant very close to 0.1 or to 1. In all other cases, the final result is at least 10^{-4} , and then an error of $0.5 \cdot 10^{-20}$ is acceptable.

```

27266 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
27267 { %^A todo: ln(1) should be "exact zero", not "underflow"
27268   \exp_after:wN \__fp_sanitize:Nw

```

```

27269 \int_value:w % for the overall sign
27270 \if_int_compare:w #1 < \c_one_int
27271 2
27272 \else:
27273 0
27274 \fi:
27275 \exp_after:wN \exp_stop_f:
27276 \int_value:w \__fp_int_eval:w % for the exponent
27277 \__fp_ln_significand:NNNNnnnN #2#3
27278 \__fp_ln_exponent:wn {#1}
27279 }

```

(End of definition for __fp_ln_npos_o:w.)

__fp_ln_significand:NNNNnnnN __fp_ln_significand:NNNNnnnN $\langle X_1 \rangle$ $\langle X_2 \rangle$ $\langle X_3 \rangle$ $\langle X_4 \rangle$ *(continuation)*
This function expands to

$\langle continuation \rangle$ $\langle Y_1 \rangle$ $\langle Y_2 \rangle$ $\langle Y_3 \rangle$ $\langle Y_4 \rangle$ $\langle Y_5 \rangle$ $\langle Y_6 \rangle$;

where $Y = -\ln(X)$ as an extended fixed point.

```

27280 \cs_new:Npn \__fp_ln_significand:NNNNnnnN #1#2#3#4
27281 {
27282 \exp_after:wN \__fp_ln_x_ii:wnnnn
27283 \int_value:w
27284 \if_case:w #1 \exp_stop_f:
27285 \or:
27286 \if_int_compare:w #2 < 4 \exp_stop_f:
27287 \__fp_int_eval:w 10 - #2
27288 \else:
27289 6
27290 \fi:
27291 \or: 4
27292 \or: 3
27293 \or: 2
27294 \or: 2
27295 \or: 2
27296 \else: 1
27297 \fi:
27298 ; { #1 #2 #3 #4 }
27299 }

```

(End of definition for __fp_ln_significand:NNNNnnnN.)

__fp_ln_x_ii:wnnnn We have thus found $c \in [1, 10]$ such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```

27300 \cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
27301 {
27302 \exp_after:wN \__fp_ln_div_after:Nw
27303 \cs:w c__fp_ln_ \__fp_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
27304 \int_value:w
27305 \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
27306 \int_value:w \__fp_int_eval:w
27307 \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
27308 \int_value:w \__fp_int_eval:w 9999 9990 + #1*#2#3 +
27309 \exp_after:wN \__fp_ln_x_iii:NNNNNNw

```

```

27310         \int_value:w \_fp_int_eval:w 10 0000 0000 + #1*#4#5 ;
27311         {20000} {0000} {0000} {0000}
27312     } %^A todo: reoptimize (a generalization attempt failed).
27313 \cs_new:Npn \_fp_ln_x_iii:NNNNNw #1#2 #3#4#5#6 #7;
27314     { #1#2; {#3#4#5#6} {#7} }
27315 \cs_new:Npn \_fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
27316     {
27317         #1#2#3#4#5 + 1 ;
27318         {#1#2#3#4#5} {#6}
27319     }

```

The Taylor series to be used is expressed in terms of $t = (x - 1)/(x + 1) = 1 - 2/(x + 1)$. We now compute the quotient with extended precision, reusing some code from `_fp_/_o:ww`. Note that $1 + x$ is known exactly.

To reuse notations from `l3fp-basics`, we want to compute A/Z with $A = 2$ and $Z = x + 1$. In `l3fp-basics`, we considered the case where both A and Z are arbitrary, in the range $[0.1, 1)$, and we had to monitor the growth of the sequence of remainders A, B, C , etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly $10^9 \cdot A/10^5 \cdot Z$: then A was bound to be below $2.147 \dots$, and this limit was never far.

In our case, we can simply work with $10^8 \cdot A$ and $10^4 \cdot Z$, because our reason to work with higher powers has gone: we needed the integer $y \simeq 10^5 \cdot Z$ to be at least 10^4 , and now, the definition $y \simeq 10^4 \cdot Z$ suffices.

Let us thus define $y = \lfloor 10^4 \cdot Z \rfloor + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4]$, and

$$Q_1 = \left\lfloor \frac{\lfloor 10^8 \cdot A \rfloor}{y} - \frac{1}{2} \right\rfloor.$$

(The $1/2$ comes from how ε -TeX rounds.) As for division, it is easy to see that $Q_1 \leq 10^4 A/Z$, *i.e.*, Q_1 is an underestimate.

Exactly as we did for division, we set $B = 10^4 A - Q_1 Z$. Then

$$\begin{aligned}
 10^4 B &\leq A_1 A_2 \cdot A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\
 &\leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\
 &\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y
 \end{aligned}$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$\begin{aligned}
 10^4 A &= 2 \cdot 10^4 \\
 10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\
 10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\
 10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\
 10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\
 10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4
 \end{aligned}$$

Note that we compute more steps than for division: since t is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).

`__fp_ln_x_iv:wnnnnnnnn <1 or 2> <8d> ; <{4d}> <{4d}> <fixed-t1>`

The number is x . Compute y by adding 1 to the five first digits.

```

27320 \cs_new:Npn __fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
27321 {
27322   \exp_after:wN __fp_div_significand_pack:NNN
27323   \int_value:w __fp_int_eval:w
27324   __fp_ln_div_i:w #1 ;
27325   #6 #7 ; {#8} {#9}
27326   {#2} {#3} {#4} {#5}
27327   { \exp_after:wN __fp_ln_div_ii:wN \int_value:w #1 }
27328   { \exp_after:wN __fp_ln_div_ii:wN \int_value:w #1 }
27329   { \exp_after:wN __fp_ln_div_ii:wN \int_value:w #1 }
27330   { \exp_after:wN __fp_ln_div_ii:wN \int_value:w #1 }
27331   { \exp_after:wN __fp_ln_div_vi:wN \int_value:w #1 }
27332 }
27333 \cs_new:Npn __fp_ln_div_i:w #1;
27334 {
27335   \exp_after:wN __fp_div_significand_calc:wnnnnnnnn
27336   \int_value:w __fp_int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
27337 }
27338 \cs_new:Npn __fp_ln_div_ii:wN #1; #2;#3 % y; B1;B2 <- for k=1
27339 {
27340   \exp_after:wN __fp_div_significand_pack:NNN
27341   \int_value:w __fp_int_eval:w
27342   \exp_after:wN __fp_div_significand_calc:wnnnnnnnn
27343   \int_value:w __fp_int_eval:w 999999 + #2 #3 / #1 ; % Q2
27344   #2 #3 ;
27345 }
27346 \cs_new:Npn __fp_ln_div_vi:wN #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
27347 {
27348   \exp_after:wN __fp_div_significand_pack:NNN

```

```

27349     \int_value:w \_fp_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
27350   }

```

We now have essentially

```

\_fp_ln_div_after:Nw <fixed t1>
\_fp_div_significand_pack:NNN 106 + Q1
\_fp_div_significand_pack:NNN 106 + Q2
\_fp_div_significand_pack:NNN 106 + Q3
\_fp_div_significand_pack:NNN 106 + Q4
\_fp_div_significand_pack:NNN 106 + Q5
\_fp_div_significand_pack:NNN 106 + Q6 ;
<exponent> ; <continuation>

```

where $\langle \text{fixed } t1 \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle \text{exponent} \rangle$ is the exponent. Also, the expansion is done backwards. Then `_fp_div_significand_pack:NNN` puts things in the correct order to add the Q_i together and put semicolons between each piece. Once those have been expanded, we get

```

\_fp_ln_div_after:Nw <fixed-t1> <1d> ; <4d> ; <4d> ;
<4d> ; <4d> ; <4d> ; <4d> ; <exponent> ;

```

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division $2/(x+1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x+1)$, after testing whether $2/(x+1)$ is greater than or smaller than 1.

```

27351 \cs_new:Npn \_fp_ln_div_after:Nw #1#2;
27352   {
27353     \if_meaning:w 0 #2
27354     \exp_after:wN \_fp_ln_t_small:Nw
27355     \else:
27356     \exp_after:wN \_fp_ln_t_large:NNw
27357     \exp_after:wN -
27358     \fi:
27359     #1
27360   }
27361 \cs_new:Npn \_fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
27362   {
27363     \exp_after:wN \_fp_ln_t_large:NNw
27364     \exp_after:wN + % <sign>
27365     \exp_after:wN #1
27366     \int_value:w \_fp_int_eval:w 9999 - #2 \exp_after:wN ;
27367     \int_value:w \_fp_int_eval:w 9999 - #3 \exp_after:wN ;
27368     \int_value:w \_fp_int_eval:w 9999 - #4 \exp_after:wN ;
27369     \int_value:w \_fp_int_eval:w 9999 - #5 \exp_after:wN ;
27370     \int_value:w \_fp_int_eval:w 9999 - #6 \exp_after:wN ;
27371     \int_value:w \_fp_int_eval:w 1 0000 - #7 ;
27372   }

```

```

\_fp_ln_t_large:NNw <sign> <fixed t1>
<t1>; <t2>; <t3>; <t4>; <t5>; <t6>;
<exponent> ; <continuation>

```

Compute the square t^2 , and keep t at the end with its sign. We know that $t < 0.1765$, so every piece has at most 4 digits. However, since we were not careful in `_fp_ln_t_small:w`, they can have less than 4 digits.

```

27373 \cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
27374 {
27375   \exp_after:wN \__fp_ln_square_t_after:w
27376   \int_value:w \__fp_int_eval:w 9999 0000 + #3*#3
27377   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
27378   \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#4
27379   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
27380   \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
27381   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
27382   \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
27383   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
27384   \int_value:w \__fp_int_eval:w
27385     1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
27386     + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
27387     % ; ; ;
27388   \exp_after:wN \__fp_ln_twice_t_after:w
27389   \int_value:w \__fp_int_eval:w -1 + 2*#3
27390   \exp_after:wN \__fp_ln_twice_t_pack:Nw
27391   \int_value:w \__fp_int_eval:w 9999 + 2*#4
27392   \exp_after:wN \__fp_ln_twice_t_pack:Nw
27393   \int_value:w \__fp_int_eval:w 9999 + 2*#5
27394   \exp_after:wN \__fp_ln_twice_t_pack:Nw
27395   \int_value:w \__fp_int_eval:w 9999 + 2*#6
27396   \exp_after:wN \__fp_ln_twice_t_pack:Nw
27397   \int_value:w \__fp_int_eval:w 9999 + 2*#7
27398   \exp_after:wN \__fp_ln_twice_t_pack:Nw
27399   \int_value:w \__fp_int_eval:w 10000 + 2*#8 ; ;
27400   { \__fp_ln_c:NwNw #1 }
27401   #2
27402 }
27403 \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
27404 \cs_new:Npn \__fp_ln_twice_t_after:w #1; { ; ; ; {#1} }
27405 \cs_new:Npn \__fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
27406   { + #1#2#3#4#5 ; {#6} }
27407 \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
27408   { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

```

(End of definition for __fp_ln_x_ii:wnnnn.)

__fp_ln_Taylor:wwNw Denoting $T = t^2$, we get

```

\__fp_ln_Taylor:wwNw
{<T1>} {<T2>} {<T3>} {<T4>} {<T5>} {<T6>} ; ;
{<(2t)1>} {<(2t)2>} {<(2t)3>} {<(2t)4>} {<(2t)5>} {<(2t)6>} ;
{ \__fp_ln_c:NwNw <sign> }
<fixed t1> <exponent> ; <continuation>

```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T \left(\frac{1}{3} + T \left(\frac{1}{5} + T \left(\frac{1}{7} + T \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

The process looks as follows


```

\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;

```

This uses the routine for dividing a number by a small integer ($< 10^4$).

```

27409 \cs_new:Npn \__fp_ln_Taylor:wwNw
27410 { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
27411 \cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
27412 {
27413   \if_int_compare:w #1 = \c_one_int
27414     \__fp_ln_Taylor_break:w
27415   \fi:
27416   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_t1 #1;
27417   \__fp_fixed_add:wwn #2;
27418   \__fp_fixed_mul:wwn #3;
27419   {
27420     \exp_after:wN \__fp_ln_Taylor_loop:www
27421     \int_value:w \__fp_int_eval:w #1 - 2 ;
27422   }
27423   #3;
27424 }
27425 \cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;;
27426 {
27427   \fi:
27428   \exp_after:wN \__fp_fixed_mul:wwn
27429   \exp_after:wN { \int_value:w \__fp_int_eval:w 10000 + #2 } #3;
27430 }

```

(End of definition for `__fp_ln_Taylor:wwNw`.)

```

\__fp_ln_c:NwNw \__fp_ln_c:NwNw <sign>
{\langle r_1 \rangle} {\langle r_2 \rangle} {\langle r_3 \rangle} {\langle r_4 \rangle} {\langle r_5 \rangle} {\langle r_6 \rangle} ;
<fixed t1> <exponent> ; <continuation>

```

We are now reduced to finding $\ln(c)$ and $\langle \text{exponent} \rangle \ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get $\ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result is at least $\ln(10/7) \simeq 0.35$.

```

27431 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
27432 {
27433   \if_meaning:w + #1
27434     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
27435   \else:
27436     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
27437   \fi:
27438   #3 #2 ;
27439 }

```

(End of definition for `__fp_ln_c:NwNw`.)

```

\__fp_ln_exponent:wn
\__fp_ln_exponent:wn
  \__fp_ln_exponent:wn
    {\s1} {\s2} {\s3} {\s4} {\s5} {\s6} ;
    {\exponent}

```

Compute $\langle \text{exponent} \rangle$ times $\ln(10)$. Apart from the cases where $\langle \text{exponent} \rangle$ is 0 or 1, the result is necessarily at least $\ln(10) \simeq 2.3$ in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order 10^4 . Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```

27440 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
27441   {
27442     \if_case:w #2 \exp_stop_f:
27443     0 \__fp_case_return:nw { \__fp_fixed_to_float_o:Nw 2 }
27444     \or:
27445     \exp_after:wN \__fp_ln_exponent_one:ww \int_value:w
27446     \else:
27447     \if_int_compare:w #2 > \c_zero_int
27448     \exp_after:wN \__fp_ln_exponent_small:NNww
27449     \exp_after:wN 0
27450     \exp_after:wN \__fp_fixed_sub:wwn \int_value:w
27451     \else:
27452     \exp_after:wN \__fp_ln_exponent_small:NNww
27453     \exp_after:wN 2
27454     \exp_after:wN \__fp_fixed_add:wwn \int_value:w -
27455     \fi:
27456     \fi:
27457     #2; #1;
27458   }

```

Now we painfully write all the cases.¹² No overflow nor underflow can happen, except when computing $\ln(1)$.

```

27459 \cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
27460   {
27461     0
27462     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl #1;
27463     \__fp_fixed_to_float_o:wN 0
27464   }

```

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```

27465 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
27466   {
27467     4
27468     \exp_after:wN \__fp_fixed_mul:wwn
27469     \c__fp_ln_x_fixed_tl
27470     {\#3}{0000}{0000}{0000}{0000}{0000} ;
27471     #2
27472     {0000}{#4}{#5}{#6}{#7}{#8};
27473     \__fp_fixed_to_float_o:wN #1
27474   }

```

¹²Bruno: do rounding.

(End of definition for `_fp_ln_exponent:w`.)

76.2 Exponential

76.2.1 Sign, exponent, and special numbers

`_fp_exp_o:w`

```
27475 \cs_new:Npn \_fp_exp_o:w #1 \s__fp \_fp_chk:w #2#3#4; @
27476 {
27477   \if_case:w #2 \exp_stop_f:
27478     \_fp_case_return_o:Nw \c_one_fp
27479   \or:
27480     \exp_after:wN \_fp_exp_normal_o:w
27481   \or:
27482     \if_meaning:w 0 #3
27483       \exp_after:wN \_fp_case_return_o:Nw
27484       \exp_after:wN \c_inf_fp
27485     \else:
27486       \exp_after:wN \_fp_case_return_o:Nw
27487       \exp_after:wN \c_zero_fp
27488     \fi:
27489   \or:
27490     \_fp_case_return_same_o:w
27491   \fi:
27492   \s__fp \_fp_chk:w #2#3#4;
27493 }
```

(End of definition for `_fp_exp_o:w`.)

`_fp_exp_normal_o:w`

`_fp_exp_pos_o:NNwnw`

`_fp_exp_overflow:NN`

```
27494 \cs_new:Npn \_fp_exp_normal_o:w \s__fp \_fp_chk:w 1#1
27495 {
27496   \if_meaning:w 0 #1
27497     \_fp_exp_pos_o:NNwnw + \_fp_fixed_to_float_o:wN
27498   \else:
27499     \_fp_exp_pos_o:NNwnw - \_fp_fixed_inv_to_float_o:wN
27500   \fi:
27501 }
27502 \cs_new:Npn \_fp_exp_pos_o:NNwnw #1#2#3 \fi: #4#5;
27503 {
27504   \fi:
27505   \if_int_compare:w #4 > \c__fp_max_exp_exponent_int
27506     \token_if_eq_charcode:NNTF + #1
27507     { \_fp_exp_overflow:NN \_fp_overflow:w \c_inf_fp }
27508     { \_fp_exp_overflow:NN \_fp_underflow:w \c_zero_fp }
27509   \exp:w
27510   \else:
27511     \exp_after:wN \_fp_sanitize:Nw
27512     \exp_after:wN 0
27513     \int_value:w #1 \_fp_int_eval:w
27514     \if_int_compare:w #4 < \c_zero_int
27515       \exp_after:wN \use_i:nn
27516     \else:
```

```

27517         \exp_after:wN \use_ii:nn
27518     \fi:
27519     {
27520         0
27521         \__fp_decimate:nNnnnn { - #4 }
27522         \__fp_exp_Taylor:Nnnwn
27523     }
27524     {
27525         \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 }
27526         \__fp_exp_pos_large:NnnNwn
27527     }
27528     #5
27529     {#4}
27530     #1 #2 0
27531     \exp:w
27532 \fi:
27533 \exp_after:wN \exp_end:
27534 }
27535 \cs_new:Npn \__fp_exp_overflow:NN #1#2
27536 {
27537     \exp_after:wN \exp_after:wN
27538     \exp_after:wN #1
27539     \exp_after:wN #2
27540 }

```

(End of definition for __fp_exp_normal_o:w, __fp_exp_pos_o:Nnnww, and __fp_exp_overflow:NN.)

```

\__fp_exp_Taylor:Nnnwn
\__fp_exp_Taylor_loop:www
\__fp_exp_Taylor_break:Nww

```

This function is called for numbers in the range $[10^{-9}, 10^{-1}]$. We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

```

27541 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
27542 {
27543     #6
27544     \__fp_pack_twice_four:wNNNNNNNN
27545     \__fp_pack_twice_four:wNNNNNNNN
27546     \__fp_pack_twice_four:wNNNNNNNN
27547     \__fp_exp_Taylor_ii:ww
27548     ; #2#3#4 0000 0000 ;
27549 }
27550 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
27551 { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s__fp_stop }
27552 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
27553 {
27554     \if_int_compare:w #1 = \c_one_int
27555         \exp_after:wN \__fp_exp_Taylor_break:Nww
27556     \fi:
27557     \__fp_fixed_div_int:wwN #3 ; #1 ;
27558     \__fp_fixed_add_one:wN
27559     \__fp_fixed_mul:wN #2 ;
27560     {
27561         \exp_after:wN \__fp_exp_Taylor_loop:www
27562         \int_value:w \__fp_int_eval:w #1 - 1 ;
27563         #2 ;

```

```

27564     }
27565   }
27566   \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s__fp_stop
27567     { \__fp_fixed_add_one:wN #2 ; }

```

(End of definition for `__fp_exp_Taylor:Nnnwn`, `__fp_exp_Taylor_loop:www`, and `__fp_exp_Taylor_break:Nww`.)

`\c__fp_exp_intarray` The integer array has $6 \times 9 \times 4 = 216$ items encoding the values of $\exp(j \times 10^i)$ for $j = 1, \dots, 9$ and $i = -1, \dots, 4$. Each value is expressed as $\simeq 10^p \times 0.m_1m_2m_3$ with three 8-digit blocks m_1, m_2, m_3 and an integer exponent p (one more than the scientific exponent), and these are stored in the integer array as four items: $p, 10^8 + m_1, 10^8 + m_2, 10^8 + m_3$. The various exponentials are stored in increasing order of $j \times 10^i$.

Storing this data in an integer array makes it slightly harder to access (slower, too), but uses 16 bytes of memory per exponential stored, while storing as tokens used around 40 tokens; tokens have an especially large footprint in Unicode-aware engines.

```

27568   \intarray_const_from_clist:Nn \c__fp_exp_intarray
27569     {
27570         1 , 1 1105 1709 , 1 1807 5647 , 1 6248 1171 ,
27571         1 , 1 1221 4027 , 1 5816 0169 , 1 8339 2107 ,
27572         1 , 1 1349 8588 , 1 0757 6003 , 1 1039 8374 ,
27573         1 , 1 1491 8246 , 1 9764 1270 , 1 3178 2485 ,
27574         1 , 1 1648 7212 , 1 7070 0128 , 1 1468 4865 ,
27575         1 , 1 1822 1188 , 1 0039 0508 , 1 9748 7537 ,
27576         1 , 1 2013 7527 , 1 0747 0476 , 1 5216 2455 ,
27577         1 , 1 2225 5409 , 1 2849 2467 , 1 6045 7954 ,
27578         1 , 1 2459 6031 , 1 1115 6949 , 1 6638 0013 ,
27579         1 , 1 2718 2818 , 1 2845 9045 , 1 2353 6029 ,
27580         1 , 1 7389 0560 , 1 9893 0650 , 1 2272 3043 ,
27581         2 , 1 2008 5536 , 1 9231 8766 , 1 7740 9285 ,
27582         2 , 1 5459 8150 , 1 0331 4423 , 1 9078 1103 ,
27583         3 , 1 1484 1315 , 1 9102 5766 , 1 0342 1116 ,
27584         3 , 1 4034 2879 , 1 3492 7351 , 1 2260 8387 ,
27585         4 , 1 1096 6331 , 1 5842 8458 , 1 5992 6372 ,
27586         4 , 1 2980 9579 , 1 8704 1728 , 1 2747 4359 ,
27587         4 , 1 8103 0839 , 1 2757 5384 , 1 0077 1000 ,
27588         5 , 1 2202 6465 , 1 7948 0671 , 1 6516 9579 ,
27589         9 , 1 4851 6519 , 1 5409 7902 , 1 7796 9107 ,
27590         14 , 1 1068 6474 , 1 5815 2446 , 1 2146 9905 ,
27591         18 , 1 2353 8526 , 1 6837 0199 , 1 8540 7900 ,
27592         22 , 1 5184 7055 , 1 2858 7072 , 1 4640 8745 ,
27593         27 , 1 1142 0073 , 1 8981 5684 , 1 2836 6296 ,
27594         31 , 1 2515 4386 , 1 7091 9167 , 1 0062 6578 ,
27595         35 , 1 5540 6223 , 1 8439 3510 , 1 0525 7117 ,
27596         40 , 1 1220 4032 , 1 9431 7840 , 1 8020 0271 ,
27597         44 , 1 2688 1171 , 1 4181 6135 , 1 4484 1263 ,
27598         87 , 1 7225 9737 , 1 6812 5749 , 1 2581 7748 ,
27599         131 , 1 1942 4263 , 1 9524 1255 , 1 9365 8421 ,
27600         174 , 1 5221 4696 , 1 8976 4143 , 1 9505 8876 ,
27601         218 , 1 1403 5922 , 1 1785 2837 , 1 4107 3977 ,
27602         261 , 1 3773 0203 , 1 0092 9939 , 1 8234 0143 ,
27603         305 , 1 1014 2320 , 1 5473 5004 , 1 5094 5533 ,
27604         348 , 1 2726 3745 , 1 7211 2566 , 1 5673 6478 ,
27605         391 , 1 7328 8142 , 1 2230 7421 , 1 7051 8866 ,

```

```

27606      435 , 1 1970 0711 , 1 1401 7046 , 1 9938 8888 ,
27607      869 , 1 3881 1801 , 1 9428 4368 , 1 5764 8232 ,
27608      1303 , 1 7646 2009 , 1 8905 4704 , 1 8893 1073 ,
27609      1738 , 1 1506 3559 , 1 7005 0524 , 1 9009 7592 ,
27610      2172 , 1 2967 6283 , 1 8402 3667 , 1 0689 6630 ,
27611      2606 , 1 5846 4389 , 1 5650 2114 , 1 7278 5046 ,
27612      3041 , 1 1151 7900 , 1 5080 6878 , 1 2914 4154 ,
27613      3475 , 1 2269 1083 , 1 0850 6857 , 1 8724 4002 ,
27614      3909 , 1 4470 3047 , 1 3316 5442 , 1 6408 6591 ,
27615      4343 , 1 8806 8182 , 1 2566 2921 , 1 5872 6150 ,
27616      8686 , 1 7756 0047 , 1 2598 6861 , 1 0458 3204 ,
27617      13029 , 1 6830 5723 , 1 7791 4884 , 1 1932 7351 ,
27618      17372 , 1 6015 5609 , 1 3095 3052 , 1 3494 7574 ,
27619      21715 , 1 5297 7951 , 1 6443 0315 , 1 3251 3576 ,
27620      26058 , 1 4665 6719 , 1 0099 3379 , 1 5527 2929 ,
27621      30401 , 1 4108 9724 , 1 3326 3186 , 1 5271 5665 ,
27622      34744 , 1 3618 6973 , 1 3140 0875 , 1 3856 4102 ,
27623      39087 , 1 3186 9209 , 1 6113 3900 , 1 6705 9685 ,
27624    }

```

(End of definition for \c__fp_exp_intarray.)

```

\__fp_exp_pos_large:NnnNwn
\__fp_exp_large_after:wnn
  \__fp_exp_large:NwN
  \__fp_exp_intarray:w
  \__fp_exp_intarray_aux:w

```

The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros). The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range [0, 5]. Remove leading zeros from the integer part: putting #4 in there too ensures that an integer part of 0 is also removed. Then read digits one by one, looking up $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$ in a table, and multiplying that to the current total. The loop is done by __fp_exp_large:NwN, whose #1 is the $\langle exponent \rangle$, #2 is the current mantissa, and #3 is the $\langle digit \rangle$. At the end, __fp_exp_large_after:wnn moves on to the Taylor series, eventually multiplied with the mantissa that we have just computed.

```

27625 \cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
27626 {
27627   \exp_after:wN \exp_after:wN \exp_after:wN \__fp_exp_large:NwN
27628   \exp_after:wN \exp_after:wN \exp_after:wN #6
27629   \exp_after:wN \c__fp_one_fixed_tl
27630   \int_value:w #3 #4 \exp_stop_f:
27631   #5 00000 ;
27632 }
27633 \cs_new:Npn \__fp_exp_large:NwN #1#2; #3
27634 {
27635   \if_case:w #3 ~
27636     \exp_after:wN \__fp_fixed_continue:wn
27637   \else:
27638     \exp_after:wN \__fp_exp_intarray:w
27639     \int_value:w \__fp_int_eval:w 36 * #1 + 4 * #3 \exp_after:wN ;
27640   \fi:
27641   #2;
27642   {
27643     \if_meaning:w 0 #1
27644       \exp_after:wN \__fp_exp_large_after:wnn
27645     \else:
27646       \exp_after:wN \__fp_exp_large:NwN
27647       \int_value:w \__fp_int_eval:w #1 - 1 \exp_after:wN \scan_stop:

```

```

27648     \fi:
27649   }
27650 }
27651 \cs_new:Npn \__fp_exp_intarray:w #1 ;
27652 {
27653   +
27654   \__kernel_intarray_item:Nn \c__fp_exp_intarray
27655   { \__fp_int_eval:w #1 - 3 \scan_stop: }
27656   \exp_after:wN \use_i:nnn
27657   \exp_after:wN \__fp_fixed_mul:wwn
27658   \int_value:w 0
27659   \exp_after:wN \__fp_exp_intarray_aux:w
27660   \int_value:w \__kernel_intarray_item:Nn
27661     \c__fp_exp_intarray { \__fp_int_eval:w #1 - 2 }
27662   \exp_after:wN \__fp_exp_intarray_aux:w
27663   \int_value:w \__kernel_intarray_item:Nn
27664     \c__fp_exp_intarray { \__fp_int_eval:w #1 - 1 }
27665   \exp_after:wN \__fp_exp_intarray_aux:w
27666   \int_value:w \__kernel_intarray_item:Nn \c__fp_exp_intarray {#1} ; ;
27667 }
27668 \cs_new:Npn \__fp_exp_intarray_aux:w 1 #1#2#3#4#5 ; { ; {#1#2#3#4} {#5} }
27669 \cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
27670 {
27671   \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; { } #3
27672   \__fp_fixed_mul:wwn #1;
27673 }

```

(End of definition for `__fp_exp_pos_large:NnnNwn` and others.)

76.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$(-\infty, -0)$	-integer	± 0	+integer	$(0, \infty)$	$+\infty$	nan
$+\infty$	+0		+0	+1	$+\infty$	$+\infty$	$+\infty$	nan
$(1, \infty)$	+0		$+ a ^b$	+1	$+ a ^b$	$+\infty$	$+\infty$	nan
+1	+1		+1	+1	+1	+1	+1	+1
$(0, 1)$	$+\infty$		$+ a ^b$	+1	$+ a ^b$	$+\infty$	$+\infty$	nan
+0	$+\infty$		$+\infty$	+1	+0	$+\infty$	$+\infty$	nan
-0	$+\infty$	nan	$(-1)^b \infty$	+1	$(-1)^b 0$	+0	+0	nan
$(-1, 0)$	$+\infty$	nan	$(-1)^b a ^b$	+1	$(-1)^b a ^b$	nan	+0	nan
-1	+1	nan	$(-1)^b$	+1	$(-1)^b$	nan	+1	nan
$(-\infty, -1)$	+0	nan	$(-1)^b a ^b$	+1	$(-1)^b a ^b$	nan	$+\infty$	nan
$-\infty$	+0	+0	$(-1)^b 0$	+1	$(-1)^b \infty$	nan	$+\infty$	nan
nan	nan	nan	nan	+1	nan	nan	nan	nan

We distinguished in this table the cases of finite (positive or negative) integer exponents, as $(-1)^b$ is defined in that case. One peculiarity of this operation is that $\text{nan}^0 = 1^{\text{nan}} = 1$, because this relation is obeyed for any number, even $\pm\infty$.

`__fp^_o:ww` We cram most of the tests into a single function to save csnames. First treat the case $b = 0$: $a^0 = 1$ for any a , even nan. Then test the sign of a .

- If it is positive, and a is a normal number, call `__fp_pow_normal_o:ww` followed by the two `fp` a and b . For $a = +0$ or $+\infty$, call `__fp_pow_zero_or_inf:ww` instead, to return either $+0$ or $+\infty$ as appropriate.
- If a is a `nan`, then skip to the next semicolon (which happens to be conveniently the end of b) and return `nan`.
- Finally, if a is negative, compute a^b (`__fp_pow_normal_o:ww` which ignores the sign of its first operand), and keep an extra copy of a and b (the second brace group, containing `{ b a }`, is inserted between a and b). Then do some tests to find the final sign of the result if it exists.

```

27674 \cs_new:cpn { __fp_ \iow_char:N \^_o:ww }
27675   \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
27676   {
27677     \if_meaning:w 0 #4
27678     \__fp_case_return_o:Nw \c_one_fp
27679     \fi:
27680     \if_case:w #2 \exp_stop_f:
27681     \exp_after:wN \use_i:nn
27682     \or:
27683     \__fp_case_return_o:Nw \c_nan_fp
27684     \else:
27685     \exp_after:wN \__fp_pow_neg:www
27686     \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
27687     \fi:
27688     {
27689       \if_meaning:w 1 #1
27690       \exp_after:wN \__fp_pow_normal_o:ww
27691       \else:
27692       \exp_after:wN \__fp_pow_zero_or_inf:ww
27693       \fi:
27694       \s__fp \__fp_chk:w #1#2#3;
27695     }
27696     { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
27697     \s__fp \__fp_chk:w #4#5#6;
27698   }

```

(End of definition for `__fp_^o:ww`.)

`__fp_pow_zero_or_inf:ww` Raising -0 or $-\infty$ to `nan` yields `nan`. For other powers, the result is $+0$ if 0 is raised to a positive power or ∞ to a negative power, and $+\infty$ otherwise. Thus, if the type of a and the sign of b coincide, the result is 0 , since those conveniently take the same possible values, 0 and 2 . Otherwise, either $a = \pm\infty$ and $b > 0$ and the result is $+\infty$, or $a = \pm 0$ with $b < 0$ and we have a division by zero unless $b = -\infty$.

```

27699 \cs_new:Npn \__fp_pow_zero_or_inf:ww
27700   \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
27701   {
27702     \if_meaning:w 1 #4
27703     \__fp_case_return_same_o:w
27704     \fi:
27705     \if_meaning:w #1 #4
27706     \__fp_case_return_o:Nw \c_zero_fp
27707     \fi:

```



```

27708 \if_meaning:w 2 #1
27709 \__fp_case_return_o:Nw \c_inf_fp
27710 \fi:
27711 \if_meaning:w 2 #3
27712 \__fp_case_return_o:Nw \c_inf_fp
27713 \else:
27714 \__fp_case_use:nw
27715 {
27716 \__fp_division_by_zero_o:NNww \c_inf_fp ^
27717 \s__fp \__fp_chk:w #1 #2 ;
27718 }
27719 \fi:
27720 \s__fp \__fp_chk:w #3#4
27721 }

```

(End of definition for `__fp_pow_zero_or_inf:ww`.)

`__fp_pow_normal_o:ww` We have in front of us a , and $b \neq 0$, we know that a is a normal number, and we wish to compute $|a|^b$. If $|a| = 1$, we return 1, unless $a = -1$ and b is `nan`. Indeed, returning 1 at this point would wrongly raise “invalid” when the sign is considered. If $|a| \neq 1$, test the type of b :

- 0 Impossible, we already filtered $b = \pm 0$.
- 1 Call `__fp_pow_npos_o:Nww`.
- 2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.
- 3 Return b .

```

27722 \cs_new:Npn \__fp_pow_normal_o:ww
27723 \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
27724 {
27725 \if:w 0 \__fp_str_if_eq:nn { #2 #3 } { 1 {1000} {0000} {0000} {0000} }
27726 \if_int_compare:w #4 #1 = 32 \exp_stop_f:
27727 \exp_after:wN \__fp_case_return_ii_o:ww
27728 \fi:
27729 \__fp_case_return_o:Nww \c_one_fp
27730 \fi:
27731 \if_case:w #4 \exp_stop_f:
27732 \or:
27733 \exp_after:wN \__fp_pow_npos_o:Nww
27734 \exp_after:wN #5
27735 \or:
27736 \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
27737 \if_int_compare:w #2 > \c_zero_int
27738 \exp_after:wN \__fp_case_return_o:Nww
27739 \exp_after:wN \c_inf_fp
27740 \else:
27741 \exp_after:wN \__fp_case_return_o:Nww
27742 \exp_after:wN \c_zero_fp
27743 \fi:
27744 \or:
27745 \__fp_case_return_ii_o:ww

```

```

27746 \fi:
27747 \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
27748 \s__fp \__fp_chk:w #4 #5
27749 }

```

(End of definition for __fp_pow_normal_o:ww.)

__fp_pow_npos_o:Nww We now know that $a \neq \pm 1$ is a normal number, and b is a normal number too. We want to compute $|a|^b = (|x| \cdot 10^n)^{y \cdot 10^p} = \exp((\ln|x| + n \ln(10)) \cdot y \cdot 10^p) = \exp(z)$. To compute the exponential accurately, we need to know the digits of z up to the 16-th position. Since the exponential of 10^5 is infinite, we only need at most 21 digits, hence the fixed point result of __fp_ln_o:w is precise enough for our needs. Start an integer expression for the decimal exponent of $e^{|z|}$. If z is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```

27750 \cs_new:Npn \__fp_pow_npos_o:Nww #1 \s__fp \__fp_chk:w 1#2#3
27751 {
27752   \exp_after:wN \__fp_sanitize:Nw
27753   \exp_after:wN 0
27754   \int_value:w
27755   \if:w #1 \if_int_compare:w #3 > \c_zero_int 0 \else: 2 \fi:
27756   \exp_after:wN \__fp_pow_npos_aux:NNnw
27757   \exp_after:wN +
27758   \exp_after:wN \__fp_fixed_to_float_o:wN
27759   \else:
27760   \exp_after:wN \__fp_pow_npos_aux:NNnw
27761   \exp_after:wN -
27762   \exp_after:wN \__fp_fixed_inv_to_float_o:wN
27763   \fi:
27764   {#3}
27765 }

```

(End of definition for __fp_pow_npos_o:Nww.)

__fp_pow_npos_aux:NNnw The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of x , followed by b . Compute $-\ln(x)$.

```

27766 \cs_new:Npn \__fp_pow_npos_aux:NNnw #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
27767 {
27768   #1
27769   \__fp_int_eval:w
27770   \__fp_ln_significand:NNNNnnnN #4#5
27771   \__fp_pow_exponent:wnN {#3}
27772   \__fp_fixed_mul:wwn #8 {0000}{0000} ;
27773   \__fp_pow_B:wwN #7;
27774   #1 #2 0 % fixed_to_float_o:wN
27775 }
27776 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
27777 {
27778   \if_int_compare:w #2 > \c_zero_int
27779   \exp_after:wN \__fp_pow_exponent:Nwnnnnw % n\ln(10) - (-\ln(x))
27780   \exp_after:wN +
27781   \else:
27782   \exp_after:wN \__fp_pow_exponent:Nwnnnnw % -(\ln|\ln(10) + (-\ln(x)))
27783   \exp_after:wN -
27784   \fi:

```

```

27785     #2; #1;
27786   }
27787 \cs_new:Npn \__fp_pow_exponent:Nwnnnnw #1#2; #3#4#5#6#7#8;
27788 { %A todo: use that in ln.
27789   \exp_after:wN \__fp_fixed_mul_after:wN
27790   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
27791   \exp_after:wN \__fp_pack:NNNNNw
27792   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27793   #1#2*23025 - #1 #3
27794   \exp_after:wN \__fp_pack:NNNNNw
27795   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27796   #1 #2*8509 - #1 #4
27797   \exp_after:wN \__fp_pack:NNNNNw
27798   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27799   #1 #2*2994 - #1 #5
27800   \exp_after:wN \__fp_pack:NNNNNw
27801   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27802   #1 #2*0456 - #1 #6
27803   \exp_after:wN \__fp_pack:NNNNNw
27804   \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
27805   #1 #2*8401 - #1 #7
27806   #1 ( #2*7991 - #8 ) / 1 0000 ; ;
27807 }
27808 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
27809 {
27810   \if_int_compare:w #7 < \c_zero_int
27811     \exp_after:wN \__fp_pow_C_neg:w \int_value:w -
27812   \else:
27813     \if_int_compare:w #7 < 22 \exp_stop_f:
27814     \exp_after:wN \__fp_pow_C_pos:w \int_value:w
27815   \else:
27816     \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
27817   \fi:
27818   \fi:
27819   #7 \exp_after:wN ;
27820   \int_value:w \__fp_int_eval:w 10 0000 + #1 \__fp_int_eval_end:
27821   #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %A todo: how many 0?
27822 }
27823 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
27824 {
27825   + 2 * \c__fp_max_exponent_int
27826   \exp_after:wN \__fp_fixed_continue:wN \c__fp_one_fixed_t1
27827 }
27828 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
27829 {
27830   \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
27831   \prg_replicate:mn {#1} {0}
27832 }
27833 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
27834 { \__fp_pow_C_pos_loop:wN #1; }
27835 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
27836 {
27837   \if_meaning:w 0 #1
27838     \exp_after:wN \__fp_pow_C_pack:w

```

```

27839     \exp_after:wN #2
27840 \else:
27841     \if_meaning:w 0 #2
27842     \exp_after:wN \__fp_pow_C_pos_loop:wN \int_value:w
27843 \else:
27844     \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
27845 \fi:
27846 \__fp_int_eval:w #1 - 1 \exp_after:wN ;
27847 \fi:
27848 }
27849 \cs_new:Npn \__fp_pow_C_pack:w
27850 {
27851     \exp_after:wN \__fp_exp_large:NwN
27852     \exp_after:wN 5
27853     \c__fp_one_fixed_tl
27854 }

```

(End of definition for `__fp_pow_npos_aux:NNnww`.)

`__fp_pow_neg:www`
`__fp_pow_neg_aux:wNN`

This function is followed by three floating point numbers: a^b , $a \in [-\infty, -0]$, and b . If b is an even integer (case -1), $a^b = a^b$. If b is an odd integer (case 0), $a^b = -a^b$, obtained by a call to `__fp_pow_neg_aux:wNN`. Otherwise, the sign is undefined. This is invalid, unless a^b turns out to be $+0$ or `nan`, in which case we return that as a^b . In particular, since the underflow detection occurs before `__fp_pow_neg:www` is called, `(-0.1)**(12345.67)` gives $+0$ rather than complaining that the sign is not defined.

```

27855 \cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
27856 {
27857     \if_case:w \__fp_pow_neg_case:w #4 ;
27858     \exp_after:wN \__fp_pow_neg_aux:wNN
27859 \or:
27860     \if_int_compare:w \__fp_int_eval:w #1 / 2 = \c_one_int
27861     \__fp_invalid_operation_o:Nww ^ #3; #4;
27862     \exp:w \exp_end_continue_f:w
27863     \exp_after:wN \exp_after:wN
27864     \exp_after:wN \__fp_use_none_until_s:w
27865 \fi:
27866 \fi:
27867 \__fp_exp_after_o:w
27868 \s__fp \__fp_chk:w #1#2;
27869 }
27870 \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
27871 {
27872     \exp_after:wN \__fp_exp_after_o:w
27873     \exp_after:wN \s__fp
27874     \exp_after:wN \__fp_chk:w
27875     \exp_after:wN #2
27876     \int_value:w \__fp_int_eval:w 2 - #3 \__fp_int_eval_end:
27877 }

```

(End of definition for `__fp_pow_neg:www` and `__fp_pow_neg_aux:wNN`.)

`__fp_pow_neg_case:w`
`__fp_pow_neg_case_aux:nnnnn`
`__fp_pow_neg_case_aux:Nnnw`

This function expects a floating point number, and determines its “parity”. It should be used after `\if_case:w` or in an integer expression. It gives -1 if the number is an even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and $\pm\infty$ are even

(because very large finite floating points are even), while `nan` is a non-integer. The sign of normal numbers is irrelevant to parity. After `__fp_decimate:nNnnnn` the argument #1 of `__fp_pow_neg_case_aux:Nnnw` is a rounding digit, 0 if and only if the number was an integer, and #3 is the 8 least significant digits of that integer.

```

27878 \cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
27879 {
27880   \if_case:w #1 \exp_stop_f:
27881     -1
27882   \or:   \__fp_pow_neg_case_aux:nnnn #3
27883   \or:   -1
27884   \else: 1
27885   \fi:
27886   \exp_stop_f:
27887 }
27888 \cs_new:Npn \__fp_pow_neg_case_aux:nnnn #1#2#3#4#5
27889 {
27890   \if_int_compare:w #1 > \c__fp_prec_int
27891     -1
27892   \else:
27893     \__fp_decimate:nNnnnn { \c__fp_prec_int - #1 }
27894     \__fp_pow_neg_case_aux:Nnnw
27895     {#2} {#3} {#4} {#5}
27896   \fi:
27897 }
27898 \cs_new:Npn \__fp_pow_neg_case_aux:Nnnw #1#2#3#4 ;
27899 {
27900   \if_meaning:w 0 #1
27901     \if_int_odd:w #3 \exp_stop_f:
27902     0
27903   \else:
27904     -1
27905   \fi:
27906   \else:
27907     1
27908   \fi:
27909 }

```

(End of definition for `__fp_pow_neg_case:w`, `__fp_pow_neg_case_aux:nnnn`, and `__fp_pow_neg_case_aux:Nnnw`.)

76.4 Factorial

`\c__fp_fact_max_arg_int` The maximum integer whose factorial fits in the exponent range is 3248, as $3249! \sim 10^{10000.8}$

```

27910 \int_const:Nn \c__fp_fact_max_arg_int { 3248 }

```

(End of definition for `\c__fp_fact_max_arg_int`.)

`__fp_fact_o:w` First detect ± 0 and $+\infty$ and `nan`. Then note that factorial of anything with a negative sign (except -0) is undefined. Then call `__fp_small_int:wTF` to get an integer as the argument, and start a loop. This is not the most efficient way of computing the factorial, but it works all right. Of course we work with 24 digits instead of 16. It is easy to check that computing factorials with this precision is enough.

```

27911 \cs_new:Npn \__fp_fact_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
27912 {
27913   \if_case:w #2 \exp_stop_f:
27914     \__fp_case_return_o:Nw \c_one_fp
27915   \or:
27916   \or:
27917     \if_meaning:w 0 #3
27918     \exp_after:wN \__fp_case_return_same_o:w
27919   \fi:
27920   \or:
27921     \__fp_case_return_same_o:w
27922   \fi:
27923   \if_meaning:w 2 #3
27924     \__fp_case_use:nw { \__fp_invalid_operation_o:fw { fact } }
27925   \fi:
27926   \__fp_fact_pos_o:w
27927   \s__fp \__fp_chk:w #2 #3 #4 ;
27928 }

```

(End of definition for __fp_fact_o:w.)

__fp_fact_pos_o:w Then check the input is an integer, and call __fp_facorial_int_o:n with that int as
 __fp_fact_int_o:w an argument. If it's too big the factorial overflows. Otherwise call __fp_sanitize:Nw
 with a positive sign marker 0 and an integer expression that will mop up any exponent
 in the calculation.

```

27929 \cs_new:Npn \__fp_fact_pos_o:w #1;
27930 {
27931   \__fp_small_int:wTF #1;
27932   { \__fp_fact_int_o:n }
27933   { \__fp_invalid_operation_o:fw { fact } #1; }
27934 }
27935 \cs_new:Npn \__fp_fact_int_o:n #1
27936 {
27937   \if_int_compare:w #1 > \c__fp_fact_max_arg_int
27938     \__fp_case_return:nw
27939     {
27940       \exp_after:wN \exp_after:wN \exp_after:wN \__fp_overflow:w
27941       \exp_after:wN \c_inf_fp
27942     }
27943   \fi:
27944   \exp_after:wN \__fp_sanitize:Nw
27945   \exp_after:wN 0
27946   \int_value:w \__fp_int_eval:w
27947   \__fp_fact_loop_o:w #1 . 4 , { 1 } { } { } { } { } { } ;
27948 }

```

(End of definition for __fp_fact_pos_o:w and __fp_fact_int_o:w.)

__fp_fact_loop_o:w The loop receives an integer #1 whose factorial we want to compute, which we progres-
 sively decrement, and the result so far as an extended-precision number #2 in the form
 <exponent>, <mantissa>;. The loop goes in steps of two because we compute #1*#1-1
 as an integer expression (it must fit since #1 is at most 3248), then multiply with the
 result so far. We don't need to fill in most of the mantissa with zeros because __fp_-
 ep_mul:wwwn first normalizes the extended precision number to avoid loss of precision.

When reaching a small enough number simply use a table of factorials less than 10^8 . This limit is chosen because the normalization step cannot deal with larger integers.

```

27949 \cs_new:Npn \__fp_fact_loop_o:w #1 . #2 ;
27950 {
27951   \if_int_compare:w #1 < 12 \exp_stop_f:
27952     \__fp_fact_small_o:w #1
27953     \fi:
27954   \exp_after:wN \__fp_ep_mul:wwwwn
27955   \exp_after:wN 4 \exp_after:wN ,
27956   \exp_after:wN { \int_value:w \__fp_int_eval:w #1 * (#1 - 1) }
27957   { } { } { } { } { } { } ;
27958   #2 ;
27959   {
27960     \exp_after:wN \__fp_fact_loop_o:w
27961     \int_value:w \__fp_int_eval:w #1 - 2 .
27962   }
27963 }
27964 \cs_new:Npn \__fp_fact_small_o:w #1 \fi: #2 ; #3 ; #4
27965 {
27966   \fi:
27967   \exp_after:wN \__fp_ep_mul:wwwwn
27968   \exp_after:wN 4 \exp_after:wN ,
27969   \exp_after:wN
27970   {
27971     \int_value:w
27972     \if_case:w #1 \exp_stop_f:
27973     1 \or: 1 \or: 2 \or: 6 \or: 24 \or: 120 \or: 720 \or: 5040
27974     \or: 40320 \or: 362880 \or: 3628800 \or: 39916800
27975     \fi:
27976   } { } { } { } { } { } { } ;
27977   #3 ;
27978   \__fp_ep_to_float_o:wwN 0
27979 }

```

(End of definition for __fp_fact_loop_o:w.)

```

27980 </package>

```

Chapter 77

l3fp-trig implementation

```
27981 (*package)
27982 (@@=fp)

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_asecd:N
\__fp_parse_word_asin:N
\__fp_parse_word_asind:N
\__fp_parse_word_cos:N
\__fp_parse_word_cosd:N
\__fp_parse_word_cot:N
\__fp_parse_word_cotd:N
\__fp_parse_word_csc:N
\__fp_parse_word_cscd:N
\__fp_parse_word_sec:N
\__fp_parse_word_secd:N
\__fp_parse_word_sin:N
\__fp_parse_word_sind:N
\__fp_parse_word_tan:N
\__fp_parse_word_tand:N

Unary functions.
27983 \tl_map_inline:nn
27984 {
27985   {acos} {acsc} {asec} {asin}
27986   {cos} {cot} {csc} {sec} {sin} {tan}
27987 }
27988 {
27989   \cs_new:cpe { __fp_parse_word_#1:N }
27990   {
27991     \exp_not:N \__fp_parse_unary_function:NNN
27992     \exp_not:c { __fp_#1_o:w }
27993     \exp_not:N \use_i:nn
27994   }
27995   \cs_new:cpe { __fp_parse_word_#1d:N }
27996   {
27997     \exp_not:N \__fp_parse_unary_function:NNN
27998     \exp_not:c { __fp_#1_o:w }
27999     \exp_not:N \use_ii:nn
28000   }
28001 }

(End of definition for \__fp_parse_word_acos:N and others.)

\__fp_parse_word_acot:N
\__fp_parse_word_acotd:N
\__fp_parse_word_atan:N
\__fp_parse_word_atand:N

Those functions may receive a variable number of arguments.
28002 \cs_new:Npn \__fp_parse_word_acot:N
28003 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
28004 \cs_new:Npn \__fp_parse_word_acotd:N
28005 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
28006 \cs_new:Npn \__fp_parse_word_atan:N
28007 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
28008 \cs_new:Npn \__fp_parse_word_atand:N
28009 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }

(End of definition for \__fp_parse_word_acot:N and others.)
```


77.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant), with arguments given in radians or in degrees, is the same.

- Filter out special cases (± 0 , $\pm \text{inf}$ and **nan**).
- Keep the sign for later, and work with the absolute value $|x|$ of the argument.
- Small numbers ($|x| < 1$ in radians, $|x| < 10$ in degrees) are converted to fixed point numbers (and to radians if $|x|$ is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of $\pi/2$ (in degrees, 90) to bring the number to the range to $[0, \pi/2)$ (in degrees, $[0, 90)$).
- Reduce further to $[0, \pi/4]$ (in degrees, $[0, 45]$) using $\sin x = \cos(\pi/2 - x)$, and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant $\lfloor \frac{x}{\pi/4} \rfloor \bmod 8$ (in degrees, the same formula with $\pi/4 \rightarrow 45$), the sign, and the function to compute.

77.1.1 Filtering special cases

`__fp_sin_o:w` This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of ± 0 or **nan** is the same float. The sine of $\pm\infty$ raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians. Then, `__fp_sin_series_o:NNwww` is called to compute the Taylor series: this function receives a sign `#3`, an initial octant of 0, and the function `__fp_ep_to_float_o:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```

28010 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
28011   {
28012     \if_case:w #2 \exp_stop_f:
28013       \__fp_case_return_same_o:w
28014     \or: \__fp_case_use:nw
28015       {
28016         \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
28017         \__fp_ep_to_float_o:wwN #3 0
28018       }
28019     \or: \__fp_case_use:nw
28020       { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
28021     \else: \__fp_case_return_same_o:w
28022     \fi:
28023     \s__fp \__fp_chk:w #2 #3 #4;
28024   }

```

(End of definition for `__fp_sin_o:w`.)

`__fp_cos_o:w` The cosine of ± 0 is 1. The cosine of $\pm\infty$ raises an invalid operation exception. The cosine of **nan** is itself. Otherwise, the `trig` function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as

for sine, but using a positive sign 0 regardless of the sign of x , and with an initial octant of 2, because $\cos(x) = +\sin(\pi/2 + |x|)$.

```

28025 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
28026 {
28027   \if_case:w #2 \exp_stop_f:
28028     \__fp_case_return_o:Nw \c_one_fp
28029   \or: \__fp_case_use:nw
28030     {
28031       \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
28032       \__fp_ep_to_float_o:wwN 0 2
28033     }
28034   \or: \__fp_case_use:nw
28035     { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
28036   \else: \__fp_case_return_same_o:w
28037   \fi:
28038   \s__fp \__fp_chk:w #2 #3;
28039 }

```

(End of definition for __fp_cos_o:w.)

`__fp_csc_o:w` The cosecant of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `__fp_cot_zero_o:Nfw` defined below), which requires the function name. The cosecant of $\pm\infty$ raises an invalid operation exception. The cosecant of `nan` is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign #3, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because $\csc(x) = \#3(\sin|x|)^{-1}$.

```

28040 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
28041 {
28042   \if_case:w #2 \exp_stop_f:
28043     \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
28044   \or: \__fp_case_use:nw
28045     {
28046       \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
28047       \__fp_ep_inv_to_float_o:wwN #3 0
28048     }
28049   \or: \__fp_case_use:nw
28050     { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
28051   \else: \__fp_case_return_same_o:w
28052   \fi:
28053   \s__fp \__fp_chk:w #2 #3 #4;
28054 }

```

(End of definition for __fp_csc_o:w.)

`__fp_sec_o:w` The secant of ± 0 is 1. The secant of $\pm\infty$ raises an invalid operation exception. The secant of `nan` is itself. Otherwise, the `trig` function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

28055 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
28056 {
28057   \if_case:w #2 \exp_stop_f:
28058     \__fp_case_return_o:Nw \c_one_fp

```

```

28059 \or: \__fp_case_use:nw
28060 {
28061 \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
28062 \__fp_ep_inv_to_float_o:wwN 0 2
28063 }
28064 \or: \__fp_case_use:nw
28065 { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
28066 \else: \__fp_case_return_same_o:w
28067 \fi:
28068 \s__fp \__fp_chk:w #2 #3;
28069 }

```

(End of definition for __fp_sec_o:w.)

__fp_tan_o:w The tangent of ± 0 or `nan` is the same floating point number. The tangent of $\pm\infty$ raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling `__fp_tan_series_o:NNwwww`, with a sign `#3` and an initial octant of 1 (this shift is somewhat arbitrary). See `__fp_cot_o:w` for an explanation of the 0 argument.

```

28070 \cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
28071 {
28072 \if_case:w #2 \exp_stop_f:
28073 \__fp_case_return_same_o:w
28074 \or: \__fp_case_use:nw
28075 {
28076 \__fp_trig:NNNNNwn #1
28077 \__fp_tan_series_o:NNwwww 0 #3 1
28078 }
28079 \or: \__fp_case_use:nw
28080 { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
28081 \else: \__fp_case_return_same_o:w
28082 \fi:
28083 \s__fp \__fp_chk:w #2 #3 #4;
28084 }

```

(End of definition for __fp_tan_o:w.)

__fp_cot_o:w The cotangent of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `__fp_cot_zero_o:Nfw`). The cotangent of $\pm\infty$ raises an invalid operation exception. The cotangent of `nan` is itself. We use $\cot x = -\tan(\pi/2 + x)$, and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding `__fp_tan_series_o:NNwwww` two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```

28085 \cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
28086 {
28087 \if_case:w #2 \exp_stop_f:
28088 \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
28089 \or: \__fp_case_use:nw
28090 {
28091 \__fp_trig:NNNNNwn #1
28092 \__fp_tan_series_o:NNwwww 2 #3 3
28093 }
28094 \or: \__fp_case_use:nw

```

```

28095         { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
28096 \else: \__fp_case_return_same_o:w
28097 \fi:
28098 \s__fp \__fp_chk:w #2 #3 #4;
28099 }
28100 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
28101 {
28102 \fi:
28103 \token_if_eq_meaning:NNTF 0 #1
28104 { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
28105 { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
28106 {#2}
28107 }

```

(End of definition for __fp_cot_o:w and __fp_cot_zero_o:Nfw.)

77.1.2 Distinguishing small and large arguments

__fp_trig:NNNNNwn The first argument is \use_i:nn if the operand is in radians and \use_ii:nn if it is in degrees. Arguments #2 to #5 control what trigonometric function we compute, and #6 to #8 are pieces of a normal floating point number. Call the `_series` function #2, with arguments #3, either a conversion function (`__fp_ep_to_float_o:wN` or `__fp_ep_inv_to_float_o:wN`) or a sign 0 or 2 when computing tangent or cotangent; #4, a sign 0 or 2; the octant, computed in an integer expression starting with #5 and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four `\exp_after:wN` are expanded, the expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining `\exp_after:wN` hits #1, which picks the `trig` or `trigd` function in whichever branch of the conditional was taken. The final `\exp_after:wN` closes the conditional. At the end of the day, a number is `large` if it is ≥ 1 in radians or ≥ 10 in degrees, and `small` otherwise. All four `trig/trigd` auxiliaries receive the operand as an extended-precision number.

```

28108 \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
28109 {
28110 \exp_after:wN #2
28111 \exp_after:wN #3
28112 \exp_after:wN #4
28113 \int_value:w \__fp_int_eval:w #5
28114 \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
28115 \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
28116 #1 \__fp_trig_large:ww \__fp_trigd_large:ww
28117 \else:
28118 #1 \__fp_trig_small:ww \__fp_trigd_small:ww
28119 \fi:
28120 #7,#8{0000}{0000};
28121 }

```

(End of definition for __fp_trig:NNNNNwn.)

77.1.3 Small arguments

`__fp_trig_small:ww` This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

```
28122 \cs_new:Npn \__fp_trig_small:ww #1,#2;
28123   { \__fp_ep_to_fixed:wwn #1,#2; . #1,#2; }
```

(End of definition for `__fp_trig_small:ww`.)

`__fp_trigd_small:ww` Convert the extended-precision number to radians, then call `__fp_trig_small:ww` to massage it in the form appropriate for the `_series` auxiliary.

```
28124 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
28125   {
28126     \__fp_ep_mul_raw:wwwN
28127     -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
28128     \__fp_trig_small:ww
28129   }
```

(End of definition for `__fp_trigd_small:ww`.)

77.1.4 Argument reduction in degrees

`__fp_trigd_large:ww`
`__fp_trigd_large_auxi:nnnwNNNN`
`__fp_trigd_large_auxii:wNw`
`__fp_trigd_large_auxiii:www`

Note that $25 \times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent `#1` is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times \langle block_1 \rangle \cdots \langle block_3 \rangle . \langle block_4 \rangle \cdots \langle block_7 \rangle$, or is equal to it modulo 360 if the exponent `#1` is very large. The first auxiliary finds $\langle block_1 \rangle + \langle block_2 \rangle \pmod{9}$, a single digit, and prepends it to the 4 digits of $\langle block_3 \rangle$. It also unpacks $\langle block_4 \rangle$ and grabs the 4 digits of $\langle block_7 \rangle$. The second auxiliary grabs the $\langle block_3 \rangle$ plus any contribution from the first two blocks as `#1`, the first digit of $\langle block_4 \rangle$ (just after the decimal point in hundreds of degrees) as `#2`, and the three other digits as `#3`. It finds the quotient and remainder of `#1#2` modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before `#3` to form a new $\langle block_4 \rangle$. The resulting fixed point number is $x \in [0, 0.9]$. If $x \geq 0.45$, we add 1 to the octant and feed $0.9 - x$ with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `__fp_trigd_small:ww`. Otherwise, we feed it x with an exponent of 2. The third auxiliary also discards digits which were not packed into the various $\langle blocks \rangle$. Since the original exponent `#1` is at least 2, those are all 0 and no precision is lost (`#6` and `#7` are four 0 each).

```
28130 \cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7;
28131   {
28132     \exp_after:wN \__fp_pack_eight:wNNNNNNNN
28133     \exp_after:wN \__fp_pack_eight:wNNNNNNNN
28134     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
28135     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
28136     \exp_after:wN \__fp_trigd_large_auxi:nnnwNNNN
28137     \exp_after:wN ;
```

```

28138 \exp:w \exp_end_continue_f:w
28139 \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
28140 #2#3#4#5#6#7 0000 0000 0000 !
28141 }
28142 \cs_new:Npn \__fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
28143 {
28144 \exp_after:wN \__fp_trigd_large_auxii:wNw
28145 \int_value:w \__fp_int_eval:w #1 + #2
28146 - (#1 + #2 - 4) / 9 * 9 \__fp_int_eval_end:
28147 #3;
28148 #4; #5{#6#7#8#9};
28149 }
28150 \cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
28151 {
28152 + (#1#2 - 4) / 9 * 2
28153 \exp_after:wN \__fp_trigd_large_auxiii:www
28154 \int_value:w \__fp_int_eval:w #1#2
28155 - (#1#2 - 4) / 9 * 9 \__fp_int_eval_end: #3 ;
28156 }
28157 \cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
28158 {
28159 \if_int_compare:w #1 < 4500 \exp_stop_f:
28160 \exp_after:wN \__fp_use_i_until_s:nw
28161 \exp_after:wN \__fp_fixed_continue:wn
28162 \else:
28163 + 1
28164 \fi:
28165 \__fp_fixed_sub:wnn {9000}{0000}{0000}{0000}{0000}{0000};
28166 {#1}#2{0000}{0000};
28167 { \__fp_trigd_small:ww 2, }
28168 }

```

(End of definition for `__fp_trigd_large:ww` and others.)

77.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range $[0, 2\pi]$ by subtracting multiples of 2π , then to the smaller range $[0, \pi/2]$ by subtracting multiples of $\pi/2$ (keeping track of how many times $\pi/2$ is subtracted), then to $[0, \pi/4]$ by mapping $x \rightarrow \pi/2 - x$ if appropriate. When the argument is very large, say, 10^{100} , an equally large multiple of 2π must be subtracted, hence we must work with a very good approximation of 2π in order to get a sensible remainder modulo 2π .

Specifically, we multiply the argument by an approximation of $1/(2\pi)$ with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of $x/(2\pi)$ we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or -8 (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by $\pi/4$ to convert back to a value in radians in $[0, \pi/4]$.

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are

affected by carries from computations which are not done, hence we are left with at least $52 - 24 - 5 = 23$ significant digits, enough to round correctly up to $0.6 \cdot \text{ulp}$ in all cases.

`\c__fp_trig_intarray` This integer array stores blocks of 8 decimals of $10^{-16}/(2\pi)$. Each entry is 10^8 plus an 8 digit number storing 8 decimals. In total we store 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 ($4 - 1$ groups of 4 digits). The memory footprint (1/2 byte per digit) is the same as an earlier method of storing the data as a control sequence name, but the major advantage is that we can unpack specific subsets of the digits without unpacking the 10112 decimals.

```

28169 \intarray_const_from_clist:Nn \c__fp_trig_intarray
28170   {
28171     100000000, 100000000, 115915494, 130918953, 135768883, 176337251,
28172     143620344, 159645740, 145644874, 176673440, 158896797, 163422653,
28173     150901138, 102766253, 108595607, 128427267, 157958036, 189291184,
28174     161145786, 152877967, 141073169, 198392292, 139966937, 140907757,
28175     130777463, 196925307, 168871739, 128962173, 197661693, 136239024,
28176     117236290, 111832380, 111422269, 197557159, 140461890, 108690267,
28177     139561204, 189410936, 193784408, 155287230, 199946443, 140024867,
28178     123477394, 159610898, 132309678, 130749061, 166986462, 180469944,
28179     186521878, 181574786, 156696424, 110389958, 174139348, 160998386,
28180     180991999, 162442875, 158517117, 188584311, 117518767, 116054654,
28181     175369880, 109739460, 136475933, 137680593, 102494496, 163530532,
28182     171567755, 103220324, 177781639, 171660229, 146748119, 159816584,
28183     106060168, 103035998, 113391198, 174988327, 186654435, 127975507,
28184     100162406, 177564388, 184957131, 108801221, 199376147, 168137776,
28185     147378906, 133068046, 145797848, 117613124, 127314069, 196077502,
28186     145002977, 159857089, 105690279, 167851315, 125210016, 131774602,
28187     109248116, 106240561, 145620314, 164840892, 148459191, 143521157,
28188     154075562, 100871526, 160680221, 171591407, 157474582, 172259774,
28189     162853998, 175155329, 139081398, 117724093, 158254797, 107332871,
28190     190406999, 175907657, 170784934, 170393589, 182808717, 134256403,
28191     166895116, 162545705, 194332763, 112686500, 126122717, 197115321,
28192     112599504, 138667945, 103762556, 108363171, 116952597, 158128224,
28193     194162333, 143145106, 112353687, 185631136, 136692167, 114206974,
28194     169601292, 150578336, 105311960, 185945098, 139556718, 170995474,
28195     165104316, 123815517, 158083944, 129799709, 199505254, 138756612,
28196     194458833, 106846050, 178529151, 151410404, 189298850, 163881607,
28197     176196993, 107341038, 199957869, 118905980, 193737772, 106187543,
28198     122271893, 101366255, 126123878, 103875388, 181106814, 106765434,
28199     108282785, 126933426, 179955607, 107903860, 160352738, 199624512,
28200     159957492, 176297023, 159409558, 143011648, 129641185, 157771240,
28201     157544494, 157021789, 176979240, 194903272, 194770216, 164960356,
28202     153181535, 144003840, 168987471, 176915887, 163190966, 150696440,
28203     147769706, 187683656, 177810477, 197954503, 153395758, 130188183,
28204     186879377, 166124814, 195305996, 155802190, 183598751, 103512712,
28205     190432315, 180498719, 168687775, 194656634, 162210342, 104440855,
28206     149785037, 192738694, 129353661, 193778292, 187359378, 143470323,
28207     102371458, 137923557, 111863634, 119294601, 183182291, 196416500,
28208     187830793, 131353497, 179099745, 186492902, 167450609, 189368909,
28209     145883050, 133703053, 180547312, 132158094, 131976760, 132283131,
28210     141898097, 149822438, 133517435, 169898475, 101039500, 168388003,
28211     197867235, 199608024, 100273901, 108749548, 154787923, 156826113,

```

28212 199489032, 168997427, 108349611, 149208289, 103776784, 174303550,
 28213 145684560, 183671479, 130845672, 133270354, 185392556, 120208683,
 28214 193240995, 162211753, 131839402, 109707935, 170774965, 149880868,
 28215 160663609, 168661967, 103747454, 121028312, 119251846, 122483499,
 28216 111611495, 166556037, 196967613, 199312829, 196077608, 127799010,
 28217 107830360, 102338272, 198790854, 102387615, 157445430, 192601191,
 28218 100543379, 198389046, 154921248, 129516070, 172853005, 122721023,
 28219 160175233, 113173179, 175931105, 103281551, 109373913, 163964530,
 28220 157926071, 180083617, 195487672, 146459804, 173977292, 144810920,
 28221 109371257, 186918332, 189588628, 139904358, 168666639, 175673445,
 28222 114095036, 137327191, 174311388, 106638307, 125923027, 159734506,
 28223 105482127, 178037065, 133778303, 121709877, 134966568, 149080032,
 28224 169885067, 141791464, 168350828, 116168533, 114336160, 173099514,
 28225 198531198, 119733758, 144420984, 116559541, 152250643, 139431286,
 28226 144403838, 183561508, 179771645, 101706470, 167518774, 156059160,
 28227 187168578, 157939226, 123475633, 117111329, 198655941, 159689071,
 28228 198506887, 144230057, 151919770, 156900382, 118392562, 120338742,
 28229 135362568, 108354156, 151729710, 188117217, 195936832, 156488518,
 28230 174997487, 108553116, 159830610, 113921445, 144601614, 188452770,
 28231 125114110, 170248521, 173974510, 138667364, 103872860, 109967489,
 28232 131735618, 112071174, 104788993, 168886556, 192307848, 150230570,
 28233 157144063, 163863202, 136852010, 174100574, 185922811, 115721968,
 28234 100397824, 175953001, 166958522, 112303464, 118773650, 143546764,
 28235 164565659, 171901123, 108476709, 193097085, 191283646, 166919177,
 28236 169387914, 133315566, 150669813, 121641521, 100895711, 172862384,
 28237 126070678, 145176011, 113450800, 169947684, 122356989, 162488051,
 28238 157759809, 153397080, 185475059, 175362656, 149034394, 145420581,
 28239 178864356, 183042000, 131509559, 147434392, 152544850, 167491429,
 28240 108647514, 142303321, 133245695, 111634945, 167753939, 142403609,
 28241 105438335, 152829243, 142203494, 184366151, 146632286, 102477666,
 28242 166049531, 140657343, 157553014, 109082798, 180914786, 169343492,
 28243 127376026, 134997829, 195701816, 119643212, 133140475, 176289748,
 28244 140828911, 174097478, 126378991, 181699939, 148749771, 151989818,
 28245 172666294, 160183053, 195832752, 109236350, 168538892, 128468247,
 28246 125997252, 183007668, 156937583, 165972291, 198244297, 147406163,
 28247 181831139, 158306744, 134851692, 185973832, 137392662, 140243450,
 28248 119978099, 140402189, 161348342, 173613676, 144991382, 171541660,
 28249 163424829, 136374185, 106122610, 186132119, 198633462, 184709941,
 28250 183994274, 129559156, 128333990, 148038211, 175011612, 111667205,
 28251 119125793, 103552929, 124113440, 131161341, 112495318, 138592695,
 28252 184904438, 146807849, 109739828, 108855297, 104515305, 139914009,
 28253 188698840, 188365483, 166522246, 168624087, 125401404, 100911787,
 28254 142122045, 123075334, 173972538, 114940388, 141905868, 142311594,
 28255 163227443, 139066125, 116239310, 162831953, 123883392, 113153455,
 28256 163815117, 152035108, 174595582, 101123754, 135976815, 153401874,
 28257 107394340, 136339780, 138817210, 104531691, 182951948, 179591767,
 28258 139541778, 179243527, 161740724, 160593916, 102732282, 187946819,
 28259 136491289, 149714953, 143255272, 135916592, 198072479, 198580612,
 28260 169007332, 118844526, 179433504, 155801952, 149256630, 162048766,
 28261 116134365, 133992028, 175452085, 155344144, 109905129, 182727454,
 28262 165911813, 122232840, 151166615, 165070983, 175574337, 129548631,
 28263 120411217, 116380915, 160616116, 157320000, 183306114, 160618128,
 28264 103262586, 195951602, 146321661, 138576614, 180471993, 127077713,
 28265 116441201, 159496011, 106328305, 120759583, 148503050, 179095584,

28266 198298218, 167402898, 138551383, 123957020, 180763975, 150429225,
 28267 198476470, 171016426, 197438450, 143091658, 164528360, 132493360,
 28268 143546572, 137557916, 113663241, 120457809, 196971566, 134022158,
 28269 180545794, 131328278, 100552461, 132088901, 187421210, 192448910,
 28270 141005215, 149680971, 113720754, 100571096, 134066431, 135745439,
 28271 191597694, 135788920, 179342561, 177830222, 137011486, 142492523,
 28272 192487287, 113132021, 176673607, 156645598, 127260957, 141566023,
 28273 143787436, 129132109, 174858971, 150713073, 191040726, 143541417,
 28274 197057222, 165479803, 181512759, 157912400, 125344680, 148220261,
 28275 173422990, 101020483, 106246303, 137964746, 178190501, 181183037,
 28276 151538028, 179523433, 141955021, 135689770, 191290561, 143178787,
 28277 192086205, 174499925, 178975690, 118492103, 124206471, 138519113,
 28278 188147564, 102097605, 154895793, 178514140, 141453051, 151583964,
 28279 128232654, 106020603, 131189158, 165702720, 186250269, 191639375,
 28280 115278873, 160608114, 155694842, 110322407, 177272742, 116513642,
 28281 134366992, 171634030, 194053074, 180652685, 109301658, 192136921,
 28282 141431293, 171341061, 157153714, 106203978, 147618426, 150297807,
 28283 186062669, 169960809, 118422347, 163350477, 146719017, 145045144,
 28284 161663828, 146208240, 186735951, 102371302, 190444377, 194085350,
 28285 134454426, 133413062, 163074595, 113830310, 122931469, 134466832,
 28286 185176632, 182415152, 110179422, 164439571, 181217170, 121756492,
 28287 119644493, 196532222, 118765848, 182445119, 109401340, 150443213,
 28288 198586286, 121083179, 139396084, 143898019, 114787389, 177233102,
 28289 186310131, 148695521, 126205182, 178063494, 157118662, 177825659,
 28290 188310053, 151552316, 165984394, 109022180, 163144545, 121212978,
 28291 197344714, 188741258, 126822386, 102360271, 109981191, 152056882,
 28292 134723983, 158013366, 106837863, 128867928, 161973236, 172536066,
 28293 185216856, 132011948, 197807339, 158419190, 166595838, 167852941,
 28294 124187182, 117279875, 106103946, 106481958, 157456200, 160892122,
 28295 184163943, 173846549, 158993202, 184812364, 133466119, 170732430,
 28296 195458590, 173361878, 162906318, 150165106, 126757685, 112163575,
 28297 188696307, 145199922, 100107766, 176830946, 198149756, 122682434,
 28298 179367131, 108412102, 119520899, 148191244, 140487511, 171059184,
 28299 141399078, 189455775, 118462161, 190415309, 134543802, 180893862,
 28300 180732375, 178615267, 179711433, 123241969, 185780563, 176301808,
 28301 184386640, 160717536, 183213626, 129671224, 126094285, 140110963,
 28302 121826276, 151201170, 122552929, 128965559, 146082049, 138409069,
 28303 107606920, 103954646, 119164002, 115673360, 117909631, 187289199,
 28304 186343410, 186903200, 157966371, 103128612, 135698881, 176403642,
 28305 152540837, 109810814, 183519031, 121318624, 172281810, 150845123,
 28306 169019064, 166322359, 138872454, 163073727, 128087898, 130041018,
 28307 194859136, 173742589, 141812405, 167291912, 138003306, 134499821,
 28308 196315803, 186381054, 124578934, 150084553, 128031351, 118843410,
 28309 107373060, 159565443, 173624887, 171292628, 198074235, 139074061,
 28310 178690578, 144431052, 174262641, 176783005, 182214864, 162289361,
 28311 192966929, 192033046, 169332843, 181580535, 164864073, 118444059,
 28312 195496893, 153773183, 167266131, 130108623, 158802128, 180432893,
 28313 144562140, 147978945, 142337360, 158506327, 104399819, 132635916,
 28314 168734194, 136567839, 101281912, 120281622, 195003330, 112236091,
 28315 185875592, 101959081, 122415367, 194990954, 148881099, 175891989,
 28316 108115811, 163538891, 163394029, 123722049, 184837522, 142362091,
 28317 100834097, 156679171, 100841679, 157022331, 178971071, 102928884,
 28318 189701309, 195339954, 124415335, 106062584, 139214524, 133864640,
 28319 134324406, 157317477, 155340540, 144810061, 177612569, 108474646,

28320 114329765, 143900008, 138265211, 145210162, 136643111, 197987319,
28321 102751191, 144121361, 169620456, 193602633, 161023559, 162140467,
28322 102901215, 167964187, 135746835, 187317233, 110047459, 163339773,
28323 124770449, 118885134, 141536376, 100915375, 164267438, 145016622,
28324 113937193, 106748706, 128815954, 164819775, 119220771, 102367432,
28325 189062690, 170911791, 194127762, 112245117, 123546771, 115640433,
28326 135772061, 166615646, 174474627, 130562291, 133320309, 153340551,
28327 138417181, 194605321, 150142632, 180008795, 151813296, 175497284,
28328 167018836, 157425342, 150169942, 131069156, 134310662, 160434122,
28329 105213831, 158797111, 150754540, 163290657, 102484886, 148697402,
28330 187203725, 198692811, 149360627, 140384233, 128749423, 132178578,
28331 177507355, 171857043, 178737969, 134023369, 102911446, 196144864,
28332 197697194, 134527467, 144296030, 189437192, 154052665, 188907106,
28333 162062575, 150993037, 199766583, 167936112, 181374511, 104971506,
28334 115378374, 135795558, 167972129, 135876446, 130937572, 103221320,
28335 124605656, 161129971, 131027586, 191128460, 143251843, 143269155,
28336 129284585, 173495971, 150425653, 199302112, 118494723, 121323805,
28337 116549802, 190991967, 168151180, 122483192, 151273721, 199792134,
28338 133106764, 121874844, 126215985, 112167639, 167793529, 182985195,
28339 185453921, 106957880, 158685312, 132775454, 133229161, 198905318,
28340 190537253, 191582222, 192325972, 178133427, 181825606, 148823337,
28341 160719681, 101448145, 131983362, 137910767, 112550175, 128826351,
28342 183649210, 135725874, 110356573, 189469487, 154446940, 118175923,
28343 106093708, 128146501, 185742532, 149692127, 164624247, 183221076,
28344 154737505, 168198834, 156410354, 158027261, 125228550, 131543250,
28345 139591848, 191898263, 104987591, 115406321, 103542638, 190012837,
28346 142615518, 178773183, 175862355, 117537850, 169565995, 170028011,
28347 158412588, 170150030, 117025916, 174630208, 142412449, 112839238,
28348 105257725, 114737141, 123102301, 172563968, 130555358, 132628403,
28349 183638157, 168682846, 143304568, 105994018, 170010719, 152092970,
28350 117799058, 132164175, 179868116, 158654714, 177489647, 116547948,
28351 183121404, 131836079, 184431405, 157311793, 149677763, 173989893,
28352 102277656, 107058530, 140837477, 152640947, 143507039, 152145247,
28353 101683884, 107090870, 161471944, 137225650, 128231458, 172995869,
28354 173831689, 171268519, 139042297, 111072135, 107569780, 137262545,
28355 181410950, 138270388, 198736451, 162848201, 180468288, 120582913,
28356 153390138, 135649144, 130040157, 106509887, 192671541, 174507066,
28357 186888783, 143805558, 135011967, 145862340, 180595327, 124727843,
28358 182925939, 157715840, 136885940, 198993925, 152416883, 178793572,
28359 179679516, 154076673, 192703125, 164187609, 162190243, 104699348,
28360 159891990, 160012977, 174692145, 132970421, 167781726, 115178506,
28361 153008552, 155999794, 102099694, 155431545, 127458567, 104403686,
28362 168042864, 184045128, 181182309, 179349696, 127218364, 192935516,
28363 120298724, 169583299, 148193297, 183358034, 159023227, 105261254,
28364 121144370, 184359584, 194433836, 138388317, 175184116, 108817112,
28365 151279233, 137457721, 193398208, 119005406, 132929377, 175306906,
28366 160741530, 149976826, 147124407, 176881724, 186734216, 185881509,
28367 191334220, 175930947, 117385515, 193408089, 157124410, 163472089,
28368 131949128, 180783576, 131158294, 100549708, 191802336, 165960770,
28369 170927599, 101052702, 181508688, 197828549, 143403726, 142729262,
28370 110348701, 139928688, 153550062, 106151434, 130786653, 196085995,
28371 100587149, 139141652, 106530207, 100852656, 124074703, 166073660,
28372 153338052, 163766757, 120188394, 197277047, 122215363, 138511354,
28373 183463624, 161985542, 159938719, 133367482, 104220974, 149956672,

```

28374      170250544, 164232439, 157506869, 159133019, 137469191, 142980999,
28375      134242305, 150172665, 121209241, 145596259, 160554427, 159095199,
28376      168243130, 184279693, 171132070, 121049823, 123819574, 171759855,
28377      119501864, 163094029, 175943631, 194450091, 191506160, 149228764,
28378      132319212, 197034460, 193584259, 126727638, 168143633, 109856853,
28379      127860243, 132141052, 133076065, 188414958, 158718197, 107124299,
28380      159592267, 181172796, 144388537, 196763139, 127431422, 179531145,
28381      100064922, 112650013, 132686230, 121550837,
28382      }

```

(End of definition for `\c__fp_trig_intarray`.)

```

\__fp_trig_large:ww The exponent #1 is between 1 and 10000. We wish to look up decimals  $10^{\#1-16}/(2\pi)$ 
\__fp_trig_large_auxi:w starting from the digit #1 + 1. Since they are stored in batches of 8, compute  $\lceil \#1/8 \rceil$ 
\__fp_trig_large_auxii:w and fetch blocks of 8 digits starting there. The numbering of items in \c__fp_trig_
\__fp_trig_large_auxiii:w intarray starts at 1, so the block  $\lceil \#1/8 \rceil + 1$  contains the digit we want, at one of the
eight positions. Each call to \int_value:w \__kernel_intarray_item:Nn expands the
next, until being stopped by \__fp_trig_large_auxiii:w using \exp_stop_f:. Once
all these blocks are unpacked, the \exp_stop_f:. and 0 to 7 digits are removed by \use_
none:n...n. Finally, \__fp_trig_large_auxii:w packs 64 digits (there are between 65
and 72 at this point) into groups of 4 and the auxv auxiliary is called.

```

```

28383 \cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
28384   {
28385     \exp_after:wN \__fp_trig_large_auxi:w
28386     \int_value:w \__fp_int_eval:w (#1 - 4) / 8 \exp_after:wN ,
28387     \int_value:w #1 , ;
28388     {#2}{#3}{#4}{#5} ;
28389   }
28390 \cs_new:Npn \__fp_trig_large_auxi:w #1, #2,
28391   {
28392     \exp_after:wN \exp_after:wN
28393     \exp_after:wN \__fp_trig_large_auxii:w
28394     \cs:w
28395       use_none:n \prg_replicate:mn { #2 - #1 * 8 } { n }
28396     \exp_after:wN
28397     \cs_end:
28398     \int_value:w
28399     \__kernel_intarray_item:Nn \c__fp_trig_intarray
28400     { \__fp_int_eval:w #1 + 1 \scan_stop: }
28401     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
28402     \__kernel_intarray_item:Nn \c__fp_trig_intarray
28403     { \__fp_int_eval:w #1 + 2 \scan_stop: }
28404     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
28405     \__kernel_intarray_item:Nn \c__fp_trig_intarray
28406     { \__fp_int_eval:w #1 + 3 \scan_stop: }
28407     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
28408     \__kernel_intarray_item:Nn \c__fp_trig_intarray
28409     { \__fp_int_eval:w #1 + 4 \scan_stop: }
28410     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
28411     \__kernel_intarray_item:Nn \c__fp_trig_intarray
28412     { \__fp_int_eval:w #1 + 5 \scan_stop: }
28413     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
28414     \__kernel_intarray_item:Nn \c__fp_trig_intarray
28415     { \__fp_int_eval:w #1 + 6 \scan_stop: }

```

```

28416 \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
28417 \_kernel_intarray_item:Nn \c\_fp_trig_intarray
28418 { \_fp_int_eval:w #1 + 7 \scan_stop: }
28419 \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
28420 \_kernel_intarray_item:Nn \c\_fp_trig_intarray
28421 { \_fp_int_eval:w #1 + 8 \scan_stop: }
28422 \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
28423 \_kernel_intarray_item:Nn \c\_fp_trig_intarray
28424 { \_fp_int_eval:w #1 + 9 \scan_stop: }
28425 \exp_stop_f:
28426 }
28427 \cs_new:Npn \_fp_trig_large_auxii:w
28428 {
28429 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
28430 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
28431 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
28432 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
28433 \_fp_trig_large_auxv:www ;
28434 }
28435 \cs_new:Npn \_fp_trig_large_auxiii:w 1 { \exp_stop_f: }

```

(End of definition for `_fp_trig_large:ww` and others.)

```

\_fp_trig_large_auxv:www
\_fp_trig_large_auxvi:wNNNNNNNN
\_fp_trig_large_pack:NNNNNw

```

First come the first 64 digits of the fractional part of $10^{#1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a semicolon. Then a few more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the `auxvi` auxiliary receives the significand as `#2#3#4#5` and 16 digits of the fractional part as `#6#7#8#9`, and computes one step of the usual ladder of `pack` functions we use for multiplication (see *e.g.*, `_fp_fixed_mul:wwn`), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last middle shift by the appropriate trailing shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute any more step in the ladder. The last semicolon closes the ladder, and we return control to the `auxvii` auxiliary.

```

28436 \cs_new:Npn \_fp_trig_large_auxv:www #1; #2; #3;
28437 {
28438 \exp_after:wN \_fp_use_i_until_s:nw
28439 \exp_after:wN \_fp_trig_large_auxvii:w
28440 \int_value:w \_fp_int_eval:w \c\_fp_leading_shift_int
28441 \prg_replicate:nn { 13 }
28442 { \_fp_trig_large_auxvi:wNNNNNNNN }
28443 + \c\_fp_trailing_shift_int - \c\_fp_middle_shift_int
28444 \_fp_use_i_until_s:nw
28445 ; #3 #1 ; ;
28446 }
28447 \cs_new:Npn \_fp_trig_large_auxvi:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
28448 {
28449 \exp_after:wN \_fp_trig_large_pack:NNNNNw
28450 \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
28451 + #2*#9 + #3*#8 + #4*#7 + #5*#6
28452 #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
28453 }

```

```

28454 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
28455   { + #1#2#3#4#5 ; #6 }

```

(End of definition for __fp_trig_large_auxv:www, __fp_trig_large_auxvi:wnnnnnnnn, and __fp_trig_large_pack:NNNNNw.)

```

\__fp_trig_large_auxvii:w
\__fp_trig_large_auxviii:w
\__fp_trig_large_auxix:Nw
\__fp_trig_large_auxx:wNNNNN
\__fp_trig_large_auxxi:w

```

The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of $\#1\#2\#3/125$, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last middle shift is converted to a trailing shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `__fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by $2\pi/8$, but first, build an extended precision number by abusing `__fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `__fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```

28456 \cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
28457   {
28458     \exp_after:wN \__fp_trig_large_auxviii:ww
28459     \int_value:w \__fp_int_eval:w (#1#2#3 - 62) / 125 ;
28460     #1#2#3
28461   }
28462 \cs_new:Npn \__fp_trig_large_auxviii:ww #1;
28463   {
28464     + #1
28465     \if_int_odd:w #1 \exp_stop_f:
28466       \exp_after:wN \__fp_trig_large_auxix:Nw
28467       \exp_after:wN -
28468     \else:
28469       \exp_after:wN \__fp_trig_large_auxix:Nw
28470       \exp_after:wN +
28471     \fi:
28472   }
28473 \cs_new:Npn \__fp_trig_large_auxix:Nw
28474   {
28475     \exp_after:wN \__fp_use_i_until_s:nw
28476     \exp_after:wN \__fp_trig_large_auxxi:w
28477     \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
28478     \prg_replicate:nn { 13 }
28479     { \__fp_trig_large_auxx:wNNNNN }
28480     + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
28481     ;
28482   }
28483 \cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
28484   {
28485     \exp_after:wN \__fp_trig_large_pack:NNNNNw
28486     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
28487     #2 8 * #3#4#5#6
28488     #1; #2
28489   }
28490 \cs_new:Npn \__fp_trig_large_auxxi:w #1;
28491   {
28492     \exp_after:wN \__fp_ep_mul_raw:wwwN
28493     \int_value:w \__fp_int_eval:w 0 \__fp_ep_to_ep_loop:N #1 ; ; !

```

```

28494     0,{7853}{9816}{3397}{4483}{0961}{5661};
28495     \_fp_trig_small:ww
28496   }

```

(End of definition for `_fp_trig_large_auxvii:w` and others.)

77.1.6 Computing the power series

`_fp_sin_series_o:NNwww` Here we receive a conversion function `_fp_ep_to_float_o:wwN` or `_fp_ep_inv_to_float_o:wwN`, a *sign* (0 or 2), a (non-negative) *octant* delimited by a dot, a *fixed point* number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function #1;
- the final sign, which depends on the octant #3 and the sign #2;
- the octant #3, which controls the series we use;
- the square #4 * #4 of the argument as a fixed point number, computed with `_fp_fixed_mul:wwn`;
- the number itself as an extended-precision number.

If the octant is in {1,2,5,6,...}, we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2 \left(\frac{1}{2!} - x^2 \left(\frac{1}{4!} - x^2 \left(\dots \right) \right) \right).$$

Otherwise, the series

$$\sin(x) = x \left(1 - x^2 \left(\frac{1}{3!} - x^2 \left(\frac{1}{5!} - x^2 \left(\dots \right) \right) \right) \right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and `_fp_sanitize:Nw` checks for overflow and underflow.

```

28497 \cs_new:Npn \_fp_sin_series_o:NNwww #1#2#3. #4;
28498   {
28499     \_fp_fixed_mul:wwn #4; #4;
28500     {
28501       \exp_after:wN \_fp_sin_series_aux_o:NNwww
28502       \exp_after:wN #1
28503       \int_value:w
28504       \if_int_odd:w \_fp_int_eval:w (#3 + 2) / 4 \_fp_int_eval_end:
28505         #2
28506       \else:
28507         \if_meaning:w #2 0 2 \else: 0 \fi:
28508         \fi:
28509       {#3}
28510     }
28511   }
28512 \cs_new:Npn \_fp_sin_series_aux_o:NNwww #1#2#3 #4; #5,#6;
28513   {
28514     \if_int_odd:w \_fp_int_eval:w #3 / 2 \_fp_int_eval_end:
28515     \exp_after:wN \use_i:nn

```

```

28516 \else:
28517   \exp_after:wN \use_ii:nn
28518 \fi:
28519 { % 1/18!
28520   \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0001}{5619}{2070};
28521   #4;{0000}{0000}{0000}{0477}{9477}{3324};
28522   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
28523   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
28524   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
28525   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
28526   \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
28527   \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
28528   \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
28529   \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
28530   { \__fp_fixed_continue:wn 0, }
28531 }
28532 { % 1/17!
28533   \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254};
28534   #4;{0000}{0000}{0000}{7647}{1637}{3182};
28535   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
28536   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
28537   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
28538   \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
28539   \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
28540   \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
28541   \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
28542   { \__fp_ep_mul:wwwn 0, } #5,#6;
28543 }
28544 {
28545   \exp_after:wN \__fp_sanitize:Nw
28546   \exp_after:wN #2
28547   \int_value:w \__fp_int_eval:w #1
28548 }
28549 #2
28550 }

```

(End of definition for `__fp_sin_series_o:NNwww` and `__fp_sin_series_aux_o:NNwww`.)

Contrarily to `__fp_sin_series_o:NNwww` which received a conversion auxiliary as `#1`, here, `#1` is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant `#3` starts at 1, which means that it is 1 or 2 for $|x| \in [0, \pi/2]$, it is 3 or 4 for $|x| \in [\pi/2, \pi]$, and so on: the intervals on which $\tan|x| \geq 0$ coincide with those for which $\lfloor (\#3 + 1)/2 \rfloor$ is odd. We also have to take into account the original sign of x to get the sign of the final result; it is straightforward to check that the first `\int_value:w` expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for $\cot(x)$.

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2 a_5))))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2 b_5)))}$$

The ratio is computed by `__fp_ep_div:wwwn`, then converted to a floating point number. For octants `#3` (really, quadrants) next to a pole of the functions, the fixed point

numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```

28551 \cs_new:Npn \__fp_tan_series_o:NNwww #1#2#3. #4;
28552 {
28553   \__fp_fixed_mul:wwn #4; #4;
28554   {
28555     \exp_after:wN \__fp_tan_series_aux_o:Nnwww
28556     \int_value:w
28557       \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
28558       \exp_after:wN \reverse_if:N
28559       \fi:
28560       \if_meaning:w #1#2 2 \else: 0 \fi:
28561     {#3}
28562   }
28563 }
28564 \cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
28565 {
28566   \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{1527}{3493}{0856}{7059};
28567   #3; {0000}{0159}{6080}{0274}{5257}{6472};
28568   \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
28569   \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
28570   \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
28571   \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
28572   { \__fp_ep_mul:wwwn 0, } #4,#5;
28573   {
28574     \__fp_fixed_mul_sub_back:wwwn {0000}{0007}{0258}{0681}{9408}{4706};
28575     #3; {0000}{2343}{7175}{1399}{6151}{7670};
28576     \__fp_fixed_mul_sub_back:wwwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
28577     \__fp_fixed_mul_sub_back:wwwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
28578     \__fp_fixed_mul_sub_back:wwwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
28579     \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
28580     {
28581       \reverse_if:N \if_int_odd:w
28582       \__fp_int_eval:w (#2 - 1) / 2 \__fp_int_eval_end:
28583       \exp_after:wN \__fp_reverse_args:Nww
28584       \fi:
28585       \__fp_ep_div:wwwn 0,
28586     }
28587   }
28588   {
28589     \exp_after:wN \__fp_sanitize:Nw
28590     \exp_after:wN #1
28591     \int_value:w \__fp_int_eval:w \__fp_ep_to_float_o:wwN
28592   }
28593   #1
28594 }

```

(End of definition for `__fp_tan_series_o:NNwww` and `__fp_tan_series_aux_o:Nnwww`.)

77.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcsecant, and arcsecant) are based on a function often denoted `atan2`. This func-

tion is accessed directly by feeding two arguments to arctangent, and is defined by $\text{atan}(y, x) = \text{atan}(y/x)$ for generic y and x . Its advantages over the conventional arctangent is that it takes values in $[-\pi, \pi]$ rather than $[-\pi/2, \pi/2]$, and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of atan as

$$\text{acos } x = \text{atan}(\sqrt{1-x^2}, x) \quad (5)$$

$$\text{asin } x = \text{atan}(x, \sqrt{1-x^2}) \quad (6)$$

$$\text{asec } x = \text{atan}(\sqrt{x^2-1}, 1) \quad (7)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2-1}) \quad (8)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (9)$$

$$\text{acot } x = \text{atan}(1, x). \quad (10)$$

Rather than introducing a new function, `atan2`, the arctangent function `atan` is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument y and the second x , because $\text{atan}(y, x) = \text{atan}(y/x)$ is the angular coordinate of the point (x, y) .

As for direct trigonometric functions, the first step in computing $\text{atan}(y, x)$ is argument reduction. The sign of y gives that of the result. We distinguish eight regions where the point $(x, |y|)$ can lie, of angular size roughly $\pi/8$, characterized by their “octant”, between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of $\pi/4$ and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume $y > 0$; otherwise replace y by $-y$ below):

0 $0 < |y| < 0.41421x$, then $\text{atan } \frac{|y|}{x}$ is given by a nicely convergent Taylor series;

1 $0 < 0.41421x < |y| < x$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{4} - \text{atan } \frac{x-|y|}{x+|y|}$;

2 $0 < 0.41421|y| < x < |y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{4} + \text{atan } \frac{-x+|y|}{x+|y|}$;

3 $0 < x < 0.41421|y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{2} - \text{atan } \frac{x}{|y|}$;

4 $0 < -x < 0.41421|y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{2} + \text{atan } \frac{-x}{|y|}$;

5 $0 < 0.41421|y| < -x < |y|$, then $\text{atan } \frac{|y|}{x} = \frac{3\pi}{4} - \text{atan } \frac{x+|y|}{-x+|y|}$;

6 $0 < -0.41421x < |y| < -x$, then $\text{atan } \frac{|y|}{x} = \frac{3\pi}{4} + \text{atan } \frac{-x-|y|}{-x+|y|}$;

7 $0 < |y| < -0.41421x$, then $\text{atan } \frac{|y|}{x} = \pi - \text{atan } \frac{|y|}{-x}$.

In the following, we denote by z the ratio among $|\frac{y}{x}|$, $|\frac{x}{y}|$, $|\frac{x+y}{x-y}|$, $|\frac{x-y}{x+y}|$ which appears in the right-hand side above.

77.2.1 Arctangent and arccotangent

```

__fp_atan_o:Nw
__fp_acot_o:Nw
__fp_atan_default:w

```

The parsing step manipulates `atan` and `acot` like `min` and `max`, reading in an array of operands, but also leaves `\use_i:nn` or `\use_ii:nn` depending on whether the result

should be given in radians or in degrees. The helper `__fp_parse_function_one_two:nw` checks that the operand is one or two floating point numbers (not tuples) and leaves its second argument or its tail accordingly (its first argument is used for error messages). More precisely if we are given a single floating point number `__fp_atan_default:w` places `\c_one_fp` (expanded) after it; otherwise `__fp_atan_default:w` is omitted by `__fp_parse_function_one_two:nw`.

```

28595 \cs_new:Npn \__fp_atan_o:Nw #1
28596 {
28597   \__fp_parse_function_one_two:nw
28598   { #1 { atan } { atand } }
28599   { \__fp_atan_default:w \__fp_atanii_o:Nww #1 }
28600 }
28601 \cs_new:Npn \__fp_acot_o:Nw #1
28602 {
28603   \__fp_parse_function_one_two:nw
28604   { #1 { acot } { acotd } }
28605   { \__fp_atan_default:w \__fp_acotii_o:Nww #1 }
28606 }
28607 \cs_new:Npe \__fp_atan_default:w #1#2#3 @ { #1 #2 #3 \c_one_fp @ }

```

(End of definition for `__fp_atan_o:Nw`, `__fp_acot_o:Nw`, and `__fp_atan_default:w`.)

`__fp_atanii_o:Nww`
`__fp_acotii_o:Nww`

If either operand is `nan`, we return it. If both are normal, we call `__fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `__fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). Otherwise, one is much bigger than the other, and we call `__fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ±90), or 0, leading to $\{\pm0, \pm\pi\}$ (in degrees, $\{\pm0, \pm180\}$). Since $\text{acot}(x,y) = \text{atan}(y,x)$, `__fp_acotii_o:w` simply reverses its two arguments.

```

28608 \cs_new:Npn \__fp_atanii_o:Nww
28609   #1 \s__fp \__fp_chk:w #2#3#4; \s__fp \__fp_chk:w #5 #6 @
28610 {
28611   \if_meaning:w 3 #2 \__fp_case_return_i_o:ww \fi:
28612   \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
28613   \if_case:w
28614     \if_meaning:w #2 #5
28615       \if_meaning:w 1 #2 10 \else: 0 \fi:
28616     \else:
28617       \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
28618     \fi:
28619     \exp_stop_f:
28620     \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 2 }
28621   \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 4 }
28622   \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 0 }
28623   \fi:
28624   \__fp_atan_normal_o:NNnwNnw #1
28625   \s__fp \__fp_chk:w #2#3#4;
28626   \s__fp \__fp_chk:w #5 #6
28627 }
28628 \cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
28629   { \__fp_atanii_o:Nww #1#3; #2; }

```

(End of definition for `__fp_atanii_o:Nww` and `__fp_acotii_o:Nww`.)

`__fp_atan_inf_o:NNNw` This auxiliary is called whenever one number is ± 0 or $\pm\infty$ (and neither is `nan`). Then the result only depends on the signs, and its value is a multiple of $\pi/4$. We use the same auxiliary as for normal numbers, `__fp_atan_combine_o:NwwwwwN`, with arguments the final sign `#2`; the octant `#3`; $\operatorname{atan} z/z = 1$ as a fixed point number; $z = 0$ as a fixed point number; and $z = 0$ as an extended-precision number. Given the values we provide, $\operatorname{atan} z$ is computed to be 0, and the result is $[\#3/2] \cdot \pi/4$ if the sign `#5` of x is positive, and $[(7 - \#3)/2] \cdot \pi/4$ for negative x , where the divisions are rounded up.

```

28630 \cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
28631 {
28632   \exp_after:wN \__fp_atan_combine_o:NwwwwwN
28633   \exp_after:wN #2
28634   \int_value:w \__fp_int_eval:w
28635   \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
28636   \c__fp_one_fixed_t1
28637   {0000}{0000}{0000}{0000}{0000}{0000};
28638   0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
28639 }

```

(End of definition for `__fp_atan_inf_o:NNNw`.)

`__fp_atan_normal_o:NNnwNnw` Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\operatorname{atan}(x, \sqrt{1-x^2})$ without intermediate rounding errors.

```

28640 \cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
28641   #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
28642 {
28643   \__fp_atan_test_o:NwwNwwN
28644   #2 #3, #4{0000}{0000};
28645   #5 #6, #7{0000}{0000}; #1
28646 }

```

(End of definition for `__fp_atan_normal_o:NNnwNnw`.)

`__fp_atan_test_o:NwwNwwN` This receives: the sign `#1` of y , its exponent `#2`, its 24 digits `#3` in groups of 4, and similarly for x . We prepare to call `__fp_atan_combine_o:NwwwwwN` which expects the sign `#1`, the octant, the ratio $(\operatorname{atan} z)/z = 1 - \dots$, and the value of z , both as a fixed point number and as an extended-precision floating point number with a mantissa in $[0.01, 1)$. For now, we place `#1` as a first argument, and start an integer expression for the octant. The sign of x does not affect z , so we simply leave a contribution to the octant: $\langle \text{octant} \rangle \rightarrow 7 - \langle \text{octant} \rangle$ for negative x . Then we order $|y|$ and $|x|$ in a non-decreasing order: if $|y| > |x|$, insert 3- in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by `__fp_atan_div:wNwwnw` after the operands have been ordered.

```

28647 \cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
28648 {
28649   \exp_after:wN \__fp_atan_combine_o:NwwwwwN
28650   \exp_after:wN #1
28651   \int_value:w \__fp_int_eval:w
28652   \if_meaning:w 2 #4
28653     7 - \__fp_int_eval:w
28654   \fi:

```

```

28655     \if_int_compare:w
28656         \__fp_ep_compare:www #2,#3; #5,#6; > \c_zero_int
28657         3 -
28658         \exp_after:wN \__fp_reverse_args:Nww
28659     \fi:
28660     \__fp_atan_div:wnwnw #2,#3; #5,#6;
28661 }

```

(End of definition for __fp_atan_test_o:NwwNwwN.)

```

\__fp_atan_div:wnwnw
\__fp_atan_near:www
\__fp_atan_near_aux:wn

```

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are “near”, hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the 7- and 3- inserted earlier) and we wish to compute $\operatorname{atan} \frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute $\operatorname{atan} \frac{a}{b}$. In any case, call __fp_atan_auxi:ww followed by z , as a comma-delimited exponent and a fixed point number.

```

28662 \cs_new:Npn \__fp_atan_div:wnwnw #1,#2#3; #4,#5#6;
28663 {
28664     \if_int_compare:w
28665         \__fp_int_eval:w 41421 * #5 < #2 000
28666         \if_case:w \__fp_int_eval:w #4 - #1 \__fp_int_eval_end:
28667             00 \or: 0 \fi:
28668         \exp_stop_f:
28669         \exp_after:wN \__fp_atan_near:www
28670     \fi:
28671     0
28672     \__fp_ep_div:www #1,{#2}#3; #4,{#5}#6;
28673     \__fp_atan_auxi:w
28674 }
28675 \cs_new:Npn \__fp_atan_near:www
28676     0 \__fp_ep_div:www #1,#2; #3,
28677     {
28678         1
28679         \__fp_ep_to_fixed:wn #1 - #3, #2;
28680         \__fp_atan_near_aux:wn
28681     }
28682 \cs_new:Npn \__fp_atan_near_aux:wn #1; #2;
28683     {
28684         \__fp_fixed_add:wn #1; #2;
28685         { \__fp_fixed_sub:wn #2; #1; { \__fp_ep_div:www 0, } 0, }
28686     }

```

(End of definition for __fp_atan_div:wnwnw, __fp_atan_near:www, and __fp_atan_near_aux:wn.)

```

\__fp_atan_auxi:ww
\__fp_atan_auxii:w

```

Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a fixed point number only, then set up the call to __fp_atan_Taylor_loop:www, followed by the fixed point representation of z and the old representation.

```

28687 \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
28688     { \__fp_ep_to_fixed:wn #1,#2; \__fp_atan_auxii:w #1,#2; }
28689 \cs_new:Npn \__fp_atan_auxii:w #1;
28690     {
28691         \__fp_fixed_mul:wn #1; #1;

```

```

28692 {
28693   \__fp_atan_Taylor_loop:www 39 ;
28694   {0000}{0000}{0000}{0000}{0000}{0000} ;
28695 }
28696 ! #1;
28697 }

```

(End of definition for __fp_atan_auxi:ww and __fp_atan_auxii:w.)

__fp_atan_Taylor_loop:www We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has $\#1 = 2k - 1$,
 __fp_atan_Taylor_break:w $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\dots - z^2\frac{1}{39}))$, and $\#3 = z^2$. To go to the next step $k \rightarrow k - 1$,
 we compute $\frac{1}{2k-1}$, then subtract from it z^2 times $\#2$. The loop stops when $k = 0$: then
 $\#2$ is $(\operatorname{atan} z)/z$, and there is a need to clean up all the unnecessary data, end the integer
 expression computing the octant with a semicolon, and leave the result $\#2$ afterwards.

```

28698 \cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
28699 {
28700   \if_int_compare:w #1 = - \c_one_int
28701     \__fp_atan_Taylor_break:w
28702   \fi:
28703   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_t1 #1;
28704   \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
28705   {
28706     \exp_after:wN \__fp_atan_Taylor_loop:www
28707     \int_value:w \__fp_int_eval:w #1 - 2 ;
28708   }
28709   #3;
28710 }
28711 \cs_new:Npn \__fp_atan_Taylor_break:w
28712   \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
28713   { \fi: ; #2 ; }

```

(End of definition for __fp_atan_Taylor_loop:www and __fp_atan_Taylor_break:w.)

__fp_atan_combine_o:NwwwwN This receives a $\langle \text{sign} \rangle$, an $\langle \text{octant} \rangle$, a fixed point value of $(\operatorname{atan} z)/z$, a fixed point
 __fp_atan_combine_aux:ww number z , and another representation of z , as an $\langle \text{exponent} \rangle$ and the fixed point number
 $10^{-\langle \text{exponent} \rangle} z$, followed by either $\backslash\text{use_i:nn}$ (when working in radians) or $\backslash\text{use_ii:nn}$
 (when working in degrees). The function computes the floating point result

$$\langle \text{sign} \rangle \left(\left[\frac{\langle \text{octant} \rangle}{2} \right] \frac{\pi}{4} + (-1)^{\langle \text{octant} \rangle} \frac{\operatorname{atan} z}{z} \cdot z \right), \quad (11)$$

multiplied by $180/\pi$ if working in degrees, and using in any case the most appropriate representation of z . The floating point result is passed to $\backslash\text{__fp_sanitize:Nw}$, which checks for overflow or underflow. If the octant is 0, leave the exponent $\#5$ for $\backslash\text{__fp_sanitize:Nw}$, and multiply $\#3 = \frac{\operatorname{atan} z}{z}$ with $\#6$, the adjusted z . Otherwise, multiply $\#3 = \frac{\operatorname{atan} z}{z}$ with $\#4 = z$, then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product $\#3 \cdot \#4$. In both cases, convert to a floating point with $\backslash\text{__fp_fixed_to_float_o:wN}$.

```

28714 \cs_new:Npn \__fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5,#6; #7
28715 {
28716   \exp_after:wN \__fp_sanitize:Nw
28717   \exp_after:wN #1
28718   \int_value:w \__fp_int_eval:w

```

```

28719     \if_meaning:w 0 #2
28720         \exp_after:wN \use_i:nn
28721     \else:
28722         \exp_after:wN \use_ii:nn
28723     \fi:
28724     { #5 \__fp_fixed_mul:wwn #3; #6; }
28725     {
28726         \__fp_fixed_mul:wwn #3; #4;
28727         {
28728             \exp_after:wN \__fp_atan_combine_aux:ww
28729             \int_value:w \__fp_int_eval:w #2 / 2 ; #2;
28730         }
28731     }
28732     { #7 \__fp_fixed_to_float_o:wN \__fp_fixed_to_float_rad_o:wN }
28733     #1
28734 }
28735 \cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
28736 {
28737     \__fp_fixed_mul_short:wwn
28738     {7853}{9816}{3397}{4483}{0961}{5661};
28739     {#1}{0000}{0000};
28740     {
28741         \if_int_odd:w #2 \exp_stop_f:
28742             \exp_after:wN \__fp_fixed_sub:wwn
28743         \else:
28744             \exp_after:wN \__fp_fixed_add:wwn
28745         \fi:
28746     }
28747 }

```

(End of definition for __fp_atan_combine_o:NwwwwN and __fp_atan_combine_aux:ww.)

77.2.2 Arcsine and arccosine

__fp_asin_o:w Again, the first argument provided by l3fp-parse is \use_i:nn if we are to work in radians and \use_ii:nn for degrees. Then comes a floating point number. The arcsine of ± 0 or `nan` is the same floating point number. The arcsine of $\pm\infty$ raises an invalid operation exception. Otherwise, call an auxiliary common with __fp_acos_o:w, feeding it information about what function is being performed (for “invalid operation” exceptions).

```

28748 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
28749 {
28750     \if_case:w #2 \exp_stop_f:
28751         \__fp_case_return_same_o:w
28752     \or:
28753         \__fp_case_use:nw
28754         { \__fp_asin_normal_o:NfwNnnnw #1 { #1 { asin } { asind } } }
28755     \or:
28756         \__fp_case_use:nw
28757         { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
28758     \else:
28759         \__fp_case_return_same_o:w
28760     \fi:
28761     \s__fp \__fp_chk:w #2 #3;
28762 }

```

(End of definition for `__fp_asin_o:w`.)

`__fp_acos_o:w` The arccosine of ± 0 is $\pi/2$ (in degrees, 90). The arccosine of $\pm\infty$ raises an invalid operation exception. The arccosine of `nan` is itself. Otherwise, call an auxiliary common with `__fp_sin_o:w`, informing it that it was called by `acos` or `acosd`, and preparing to swap some arguments down the line.

```

28763 \cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
28764   {
28765     \if_case:w #2 \exp_stop_f:
28766       \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
28767     \or:
28768       \__fp_case_use:nw
28769         {
28770           \__fp_asin_normal_o:NfwNnnnw #1 { #1 { acos } { acosd } }
28771           \__fp_reverse_args:Nww
28772         }
28773     \or:
28774       \__fp_case_use:nw
28775         { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
28776     \else:
28777       \__fp_case_return_same_o:w
28778     \fi:
28779     \s__fp \__fp_chk:w #2 #3;
28780   }

```

(End of definition for `__fp_acos_o:w`.)

`__fp_asin_normal_o:NfwNnnnw` If the exponent `#5` is at most 0, the operand lies within $(-1, 1)$ and the operation is permitted: call `__fp_asin_auxi_o:NnNww` with the appropriate arguments. If the number is exactly ± 1 (the test works because we know that `#5` ≥ 1 , `#6#7` ≥ 10000000 , `#8#9` ≥ 0 , with equality only for ± 1), we also call `__fp_asin_auxi_o:NnNww`. Otherwise, `__fp_use_i:w` gets rid of the `asin` auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```

28781 \cs_new:Npn \__fp_asin_normal_o:NfwNnnnw
28782   #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
28783   {
28784     \if_int_compare:w #5 < \c_one_int
28785       \exp_after:wN \__fp_use_none_until_s:w
28786     \fi:
28787     \if_int_compare:w \__fp_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
28788       \exp_after:wN \__fp_use_none_until_s:w
28789     \fi:
28790     \__fp_use_i:ww
28791     \__fp_invalid_operation_o:fw {#2}
28792     \s__fp \__fp_chk:w 1#4#{#5}{#6}{#7}{#8}{#9};
28793     \__fp_asin_auxi_o:NnNww
28794     #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
28795   }

```

(End of definition for `__fp_asin_normal_o:NfwNnnnw`.)

`__fp_asin_auxi_o:NnNww` We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate $1-x^2$ as $(1+x)(1-x)$: this behaves better near $x = 1$. We do the

addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root $1/\sqrt{1-x^2}$. Finally, multiply by the (positive) extended-precision float $|x|$, and feed the (signed) result, and the number +1, as arguments to the arctangent function. When computing the arccosine, the arguments $x/\sqrt{1-x^2}$ and +1 are swapped by #2 (`__fp_reverse_args:Nww` in that case) before `__fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and continue after `ep_mul`.

```

28796 \cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
28797 {
28798   \__fp_ep_to_fixed:wwn #4,#5;
28799   \__fp_asin_isqrt:wn
28800   \__fp_ep_mul:wwwn #4,#5;
28801   \__fp_ep_to_ep:wwN
28802   \__fp_fixed_continue:wn
28803   { #2 \__fp_atan_test_o:NwwNwwN #3 }
28804   0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
28805 }
28806 \cs_new:Npn \__fp_asin_isqrt:wn #1;
28807 {
28808   \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl #1;
28809   {
28810     \__fp_fixed_add_one:wN #1;
28811     \__fp_fixed_continue:wn { \__fp_ep_mul:wwwn 0, } 0,
28812   }
28813   \__fp_ep_isqrt:wwn
28814 }

```

(End of definition for `__fp_asin_auxi_o:NnNww` and `__fp_asin_isqrt:wn`.)

77.2.3 Arccosecant and arcsecant

`__fp_acsc_o:w` Cases are mostly labelled by #2, except when #2 is 2: then we use #3#2, which is 02 = 2 when the number is $+\infty$ and 22 when the number is $-\infty$. The arccosecant of ± 0 raises an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arcosecant of `nan` is itself. Otherwise, `__fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (`acsc` or `acscd`) as an argument for invalid operation exceptions.

```

28815 \cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
28816 {
28817   \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
28818     \__fp_case_use:nw
28819     { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
28820   \or: \__fp_case_use:nw
28821     { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
28822   \or: \__fp_case_return_o:Nw \c_zero_fp
28823   \or: \__fp_case_return_same_o:w
28824   \else: \__fp_case_return_o:Nw \c_minus_zero_fp
28825   \fi:
28826   \s__fp \__fp_chk:w #2 #3 #4;
28827 }

```

(End of definition for `__fp_acsc_o:w`.)

`__fp_asec_o:w` The arcsecant of ± 0 raises an invalid operation exception. The arcsecant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arcosecant of `nan` is itself. Otherwise, do some more tests, keeping the function name `asec` (or `asecd`) as an argument for invalid operation exceptions, and a `__fp_reverse_args:Nww` following precisely that appearing in `__fp_acos_o:w`.

```

28828 \cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
28829 {
28830   \if_case:w #2 \exp_stop_f:
28831     \__fp_case_use:nw
28832     { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
28833   \or:
28834     \__fp_case_use:nw
28835     {
28836       \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
28837       \__fp_reverse_args:Nww
28838     }
28839   \or: \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
28840   \else: \__fp_case_return_same_o:w
28841   \fi:
28842   \s__fp \__fp_chk:w #2 #3;
28843 }

```

(End of definition for `__fp_asec_o:w`.)

`__fp_acsc_normal_o:NfwNnw` If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to `__fp_asin_auxi_o:NnNww` (with all the appropriate arguments). This computes what we want thanks to $\text{acsc}(x) = \text{asin}(1/x)$ and $\text{asec}(x) = \text{acos}(1/x)$.

```

28844 \cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
28845 {
28846   \int_compare:nNnTF {#5} < 1
28847   {
28848     \__fp_invalid_operation_o:fw {#2}
28849     \s__fp \__fp_chk:w 1#4{#5}#6;
28850   }
28851   {
28852     \__fp_ep_div:wwwn
28853     1,{1000}{0000}{0000}{0000}{0000}{0000};
28854     #5,#6{0000}{0000};
28855     { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
28856   }
28857 }

```

(End of definition for `__fp_acsc_normal_o:NfwNnw`.)

```

28858 </package>

```

Chapter 78

l3fp-convert implementation

```
28859 <*package>
28860 <@@=fp>
```

78.1 Dealing with tuples

The first argument is for instance `_fp_to_tl_dispatch:w`, which converts any floating point object to the appropriate representation. We loop through all items, putting ,~ between all of them and making sure to remove the leading ,~.

```
\_fp_tuple_convert:Nw
\_fp_tuple_convert_loop:nNw
\_fp_tuple_convert_end:w

28861 \cs_new:Npn \_fp_tuple_convert:Nw #1 \s\_fp_tuple \_fp_tuple_chk:w #2 ;
28862   {
28863     \int_case:nnF { \_fp_array_count:n {#2} }
28864     {
28865       { 0 } { ( ) }
28866       { 1 } { \_fp_tuple_convert_end:w @ { #1 #2 , } }
28867     }
28868     {
28869       \_fp_tuple_convert_loop:nNw { } #1
28870       #2 { ? \_fp_tuple_convert_end:w } ;
28871       @ { \use_none:nn }
28872     }
28873   }
28874 \cs_new:Npn \_fp_tuple_convert_loop:nNw #1#2#3#4; #5 @ #6
28875   {
28876     \use_none:n #3
28877     \exp_args:Nf \_fp_tuple_convert_loop:nNw { #2 #3#4 ; } #2 #5
28878     @ { #6 , ~ #1 }
28879   }
28880 \cs_new:Npn \_fp_tuple_convert_end:w #1 @ #2
28881   { \exp_after:wN ( \exp:w \exp_end_continue_f:w #2 ) }
```

(End of definition for `_fp_tuple_convert:Nw`, `_fp_tuple_convert_loop:nNw`, and `_fp_tuple_convert_end:w`.)

78.2 Trimming trailing zeros

```
\_fp_trim_zeros:w
\_fp_trim_zeros_loop:w
\_fp_trim_zeros_dot:w
\_fp_trim_zeros_end:w
```

If #1 ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing

zero is reached, the second argument is the `dot` auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```

28882 \cs_new:Npn \__fp_trim_zeros:w #1 ;
28883 {
28884   \__fp_trim_zeros_loop:w #1
28885   ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__fp_stop
28886 }
28887 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
28888 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
28889 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__fp_stop { #1 }

```

(End of definition for `__fp_trim_zeros:w` and others.)

78.3 Scientific notation

`\fp_to_scientific:N` The three public functions evaluate their argument, then pass it to `__fp_to_scientific_dispatch:w`.

```

\fp_to_scientific:c
\fp_to_scientific:n
28890 \cs_new:Npn \fp_to_scientific:N #1
28891 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
28892 \cs_generate_variant:Nn \fp_to_scientific:N { c }
28893 \cs_new:Npn \fp_to_scientific:n
28894 {
28895   \exp_after:wN \__fp_to_scientific_dispatch:w
28896   \exp:w \exp_end_continue_f:w \__fp_parse:n
28897 }

```

(End of definition for `\fp_to_scientific:N` and `\fp_to_scientific:n`. These functions are documented on page 262.)

`__fp_to_scientific_dispatch:w` We allow tuples.

```

\__fp_to_scientific_recover:w
\__fp_tuple_to_scientific:w
28898 \cs_new:Npn \__fp_to_scientific_dispatch:w #1
28899 {
29000   \__fp_change_func_type:NNN
29001   #1 \__fp_to_scientific:w \__fp_to_scientific_recover:w
29002   #1
29003 }
29004 \cs_new:Npn \__fp_to_scientific_recover:w #1 #2 ;
29005 {
29006   \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
29007   nan
29008 }
29009 \cs_new:Npn \__fp_tuple_to_scientific:w
29010 { \__fp_tuple_convert:Nw \__fp_to_scientific_dispatch:w }

```

(End of definition for `__fp_to_scientific_dispatch:w`, `__fp_to_scientific_recover:w`, and `__fp_tuple_to_scientific:w`.)

`__fp_to_scientific:w` Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (`#2 = 2`) start with `-`; we then only need to care about positive numbers and `nan`. Then filter the special cases: ± 0 are represented as `0`; infinities are converted to a number slightly larger than the largest after an “invalid_operation” exception; `nan` is represented as `0` after an “invalid_operation” exception. In the normal case, decrement the exponent

and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

```

28911 \cs_new:Npn \__fp_to_scientific:w \s__fp \__fp_chk:w #1#2
28912 {
28913   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
28914   \if_case:w #1 \exp_stop_f:
28915     \__fp_case_return:nw { 0.000000000000000e0 }
28916   \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
28917   \or:
28918     \__fp_case_use:nw
28919     {
28920       \__fp_invalid_operation:nnw
28921       { \fp_to_scientific:N \c__fp_overflowing_fp }
28922       { fp_to_scientific }
28923     }
28924   \or:
28925     \__fp_case_use:nw
28926     {
28927       \__fp_invalid_operation:nnw
28928       { \fp_to_scientific:N \c_zero_fp }
28929       { fp_to_scientific }
28930     }
28931   \fi:
28932   \s__fp \__fp_chk:w #1 #2
28933 }
28934 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
28935 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
28936 {
28937   \exp_after:wN \__fp_to_scientific_normal:wNw
28938   \exp_after:wN e
28939   \int_value:w \__fp_int_eval:w #2 - 1
28940   ; #3 #4 #5 #6 ;
28941 }
28942 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
28943 { #2.#3 #1 }

```

(End of definition for __fp_to_scientific:w, __fp_to_scientific_normal:wnnnnn, and __fp_to_scientific_normal:wNw.)

78.4 Decimal representation

`\fp_to_decimal:N` All three public variants are based on the same `__fp_to_decimal_dispatch:w` after evaluating their argument to an internal floating point.

```

\fp_to_decimal:c
\fp_to_decimal:n
28944 \cs_new:Npn \fp_to_decimal:N #1
28945 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
28946 \cs_generate_variant:Nn \fp_to_decimal:N { c }
28947 \cs_new:Npn \fp_to_decimal:n
28948 {
28949   \exp_after:wN \__fp_to_decimal_dispatch:w
28950   \exp:w \exp_end_continue_f:w \__fp_parse:n
28951 }

```

(End of definition for \fp_to_decimal:N and \fp_to_decimal:n. These functions are documented on page 261.)

```

\__fp_to_decimal_dispatch:w
\__fp_to_decimal_recover:w
\__fp_tuple_to_decimal:w

```

We allow tuples.

```

28952 \cs_new:Npn \__fp_to_decimal_dispatch:w #1
28953 {
28954   \__fp_change_func_type:NNN
28955   #1 \__fp_to_decimal:w \__fp_to_decimal_recover:w
28956   #1
28957 }
28958 \cs_new:Npn \__fp_to_decimal_recover:w #1 #2 ;
28959 {
28960   \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
28961   nan
28962 }
28963 \cs_new:Npn \__fp_tuple_to_decimal:w
28964 { \__fp_tuple_convert:Nw \__fp_to_decimal_dispatch:w }

```

(End of definition for `__fp_to_decimal_dispatch:w`, `__fp_to_decimal_recover:w`, and `__fp_tuple_to_decimal:w`.)

```

\__fp_to_decimal:w
\__fp_to_decimal_normal:wnnnnn
\__fp_to_decimal_large:Nnnw
\__fp_to_decimal_huge:wnnnn

```

The structure is similar to `__fp_to_scientific:w`. Insert `-` for negative numbers. Zero gives 0, $\pm\infty$ and `nan` yield an “invalid operation” exception; note that $\pm\infty$ produces a very large output, which we don’t expand now since it most likely won’t be needed. Normal numbers with an exponent in the range [1, 15] have that number of digits before the decimal separator: “decimate” them, and remove leading zeros with `\int_value:w`, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be `0.<zeros><digits>`, trimmed.

```

28965 \cs_new:Npn \__fp_to_decimal:w \s__fp \__fp_chk:w #1#2
28966 {
28967   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
28968   \if_case:w #1 \exp_stop_f:
28969     \__fp_case_return:nw { 0 }
28970   \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
28971   \or:
28972     \__fp_case_use:nw
28973     {
28974       \__fp_invalid_operation:nnw
28975       { \fp_to_decimal:N \c__fp_overflowing_fp }
28976       { fp_to_decimal }
28977     }
28978   \or:
28979     \__fp_case_use:nw
28980     {
28981       \__fp_invalid_operation:nnw
28982       { 0 }
28983       { fp_to_decimal }
28984     }
28985   \fi:
28986   \s__fp \__fp_chk:w #1 #2
28987 }
28988 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
28989 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
28990 {
28991   \int_compare:nNnTF {#2} > 0
28992   {

```

```

28993     \int_compare:nNnTF {#2} < \c__fp_prec_int
28994     {
28995         \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
28996         \__fp_to_decimal_large:Nnnw
28997     }
28998     {
28999         \exp_after:wN \exp_after:wN
29000         \exp_after:wN \__fp_to_decimal_huge:wnnnn
29001         \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } ;
29002     }
29003     {#3} {#4} {#5} {#6}
29004     }
29005     {
29006         \exp_after:wN \__fp_trim_zeros:w
29007         \exp_after:wN 0
29008         \exp_after:wN .
29009         \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
29010         #3#4#5#6 ;
29011     }
29012 }
29013 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
29014 {
29015     \exp_after:wN \__fp_trim_zeros:w \int_value:w
29016     \if_int_compare:w #2 > \c_zero_int
29017     #2
29018     \fi:
29019     \exp_stop_f:
29020     #3.#4 ;
29021 }
29022 \cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }

```

(End of definition for `__fp_to_decimal:w` and others.)

78.5 Token list representation

`\fp_to_tl:N` These three public functions evaluate their argument, then pass it to `__fp_to_tl_dispatch:w`.
`\fp_to_tl:c`
`\fp_to_tl:n`

```

29023 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
29024 \cs_generate_variant:Nn \fp_to_tl:N { c }
29025 \cs_new:Npn \fp_to_tl:n
29026 {
29027     \exp_after:wN \__fp_to_tl_dispatch:w
29028     \exp:w \exp_end_continue_f:w \__fp_parse:n
29029 }

```

(End of definition for `\fp_to_tl:N` and `\fp_to_tl:n`. These functions are documented on page 262.)

`__fp_to_tl_dispatch:w`
`__fp_to_tl_recover:w`
`__fp_tuple_to_tl:w`

We allow tuples.

```

29030 \cs_new:Npn \__fp_to_tl_dispatch:w #1
29031 { \__fp_change_func_type:NNN #1 \__fp_to_tl:w \__fp_to_tl_recover:w #1 }
29032 \cs_new:Npn \__fp_to_tl_recover:w #1 #2 ;
29033 {
29034     \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }

```

```

29035     nan
29036   }
29037   \cs_new:Npn \__fp_tuple_to_tl:w
29038     { \__fp_tuple_convert:Nw \__fp_to_tl_dispatch:w }

```

(End of definition for `__fp_to_tl_dispatch:w`, `__fp_to_tl_recover:w`, and `__fp_tuple_to_tl:w`.)

```

\__fp_to_tl:w
\__fp_to_tl_normal:nnnnn
\__fp_to_tl_scientific:wnnnnn
\__fp_to_tl_scientific:wNw

```

A structure similar to `__fp_to_scientific_dispatch:w` and `__fp_to_decimal_dispatch:w`, but without the “invalid operation” exception. First filter special cases. We express normal numbers in decimal notation if the exponent is in the range $[-2, 16]$, and otherwise use scientific notation.

```

29039 \cs_new:Npn \__fp_to_tl:w \s__fp \__fp_chk:w #1#2
29040   {
29041     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
29042     \if_case:w #1 \exp_stop_f:
29043       \__fp_case_return:nw { 0 }
29044     \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
29045     \or: \__fp_case_return:nw { inf }
29046     \else: \__fp_case_return:nw { nan }
29047     \fi:
29048   }
29049 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
29050   {
29051     \int_compare:nTF
29052       { -2 <= #1 <= \c__fp_prec_int }
29053       { \__fp_to_decimal_normal:wnnnnn }
29054       { \__fp_to_tl_scientific:wnnnnn }
29055     \s__fp \__fp_chk:w 1 0 {#1}
29056   }
29057 \cs_new:Npn \__fp_to_tl_scientific:wnnnnn
29058   \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
29059   {
29060     \exp_after:wN \__fp_to_tl_scientific:wNw
29061     \exp_after:wN e
29062     \int_value:w \__fp_int_eval:w #2 - 1
29063     ; #3 #4 #5 #6 ;
29064   }
29065 \cs_new:Npn \__fp_to_tl_scientific:wNw #1 ; #2#3;
29066   { \__fp_trim_zeros:w #2.#3 ; #1 }

```

(End of definition for `__fp_to_tl:w` and others.)

78.6 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.

78.7 Convert to dimension or integer

`\fp_to_dim:N` All three public variants are based on the same `__fp_to_dim_dispatch:w` after evaluating their argument to an internal floating point. We only allow floating point numbers,

```

\fp_to_dim:c
\fp_to_dim:n
\__fp_to_dim_dispatch:w
\__fp_to_dim_recover:w
\__fp_to_dim:w

```

not tuples.

```

29067 \cs_new:Npn \fp_to_dim:N #1
29068   { \exp_after:wN \__fp_to_dim_dispatch:w #1 }
29069 \cs_generate_variant:Nn \fp_to_dim:N { c }
29070 \cs_new:Npn \fp_to_dim:n
29071   {
29072     \exp_after:wN \__fp_to_dim_dispatch:w
29073     \exp:w \exp_end_continue_f:w \__fp_parse:n
29074   }
29075 \cs_new:Npn \__fp_to_dim_dispatch:w #1#2 ;
29076   {
29077     \__fp_change_func_type:NNN #1 \__fp_to_dim:w \__fp_to_dim_recover:w
29078     #1 #2 ;
29079   }
29080 \cs_new:Npn \__fp_to_dim_recover:w #1
29081   { \__fp_invalid_operation:nmw { Opt } { fp_to_dim } }
29082 \cs_new:Npn \__fp_to_dim:w #1 ; { \__fp_to_decimal:w #1 ; pt }

```

(End of definition for `\fp_to_dim:N` and others. These functions are documented on page 261.)

`\fp_to_int:N` For the most part identical to `\fp_to_dim:N` but without `pt`, and where `__fp_to_int:w` does more work. To convert to an integer, first round to 0 places (to the nearest integer), then express the result as a decimal number: the definition of `__fp_to_decimal_dispatch:w` is such that there are no trailing dot nor zero.

```

\__fp_to_int_dispatch:w
\__fp_to_int_recover:w
29083 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
29084 \cs_generate_variant:Nn \fp_to_int:N { c }
29085 \cs_new:Npn \fp_to_int:n
29086   {
29087     \exp_after:wN \__fp_to_int_dispatch:w
29088     \exp:w \exp_end_continue_f:w \__fp_parse:n
29089   }
29090 \cs_new:Npn \__fp_to_int_dispatch:w #1#2 ;
29091   {
29092     \__fp_change_func_type:NNN #1 \__fp_to_int:w \__fp_to_int_recover:w
29093     #1 #2 ;
29094   }
29095 \cs_new:Npn \__fp_to_int_recover:w #1
29096   { \__fp_invalid_operation:nmw { 0 } { fp_to_int } }
29097 \cs_new:Npn \__fp_to_int:w #1;
29098   {
29099     \exp_after:wN \__fp_to_decimal:w \exp:w \exp_end_continue_f:w
29100     \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
29101   }

```

(End of definition for `\fp_to_int:N` and others. These functions are documented on page 261.)

78.8 Convert from a dimension

`\dim_to_fp:n` The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by $2^{-16} = 0.0000152587890625$ to give a value expressed in points. The auxiliary `__fp_mul_npos_o:Nww` expects the desired *final sign* and two floating point operands (of the form `\s__fp ... ;`) as arguments. This set of functions is also used to convert dimension

registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `_fp_from_dim_test:ww`, and is combined with the exponent -4 of 2^{-16} . There is also a need to expand afterwards: this is performed by `_fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing`: here.

```

29102 \cs_new:Npn \dim_to_fp:n #1
29103 {
29104   \exp_after:wN \_fp_from_dim_test:ww
29105   \exp_after:wN 0
29106   \exp_after:wN ,
29107   \int_value:w \tex_glueexpr:D #1 ;
29108 }
29109 \cs_new:Npn \_fp_from_dim_test:ww #1, #2
29110 {
29111   \if_meaning:w 0 #2
29112   \_fp_case_return:nw { \exp_after:wN \c_zero_fp }
29113   \else:
29114     \exp_after:wN \_fp_from_dim:wNw
29115     \int_value:w \_fp_int_eval:w #1 - 4
29116     \if_meaning:w - #2
29117       \exp_after:wN , \exp_after:wN 2 \int_value:w
29118     \else:
29119       \exp_after:wN , \exp_after:wN 0 \int_value:w #2
29120     \fi:
29121   \fi:
29122 }
29123 \cs_new:Npn \_fp_from_dim:wNw #1,#2#3;
29124 {
29125   \_fp_pack_twice_four:wNNNNNNNN \_fp_from_dim:wNnnnnnn ;
29126   #3 000 0000 00 {10}987654321; #2 {#1}
29127 }
29128 \cs_new:Npn \_fp_from_dim:wNnnnnnn #1; #2#3#4#5#6#7#8#9
29129 { \_fp_from_dim:wnnnwNn #1 {#2#300} {0000} ; }
29130 \cs_new:Npn \_fp_from_dim:wnnnwNn #1; #2#3#4#5#6; #7#8
29131 {
29132   \_fp_mul_npos_o:Nww #7
29133   \s_fp \_fp_chk:w 1 #7 {#5} #1 ;
29134   \s_fp \_fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
29135   \prg_do_nothing:
29136 }

```

(End of definition for `\dim_to_fp:n` and others. This function is documented on page 231.)

78.9 Use and eval

`\fp_use:N` Those public functions are simple copies of the decimal conversions.

```

\fp_use:c 29137 \cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\fp_eval:n 29138 \cs_generate_variant:Nn \fp_use:N { c }
29139 \cs_new_eq:NN \fp_eval:n \fp_to_decimal:n

```

(End of definition for `\fp_use:N` and `\fp_eval:n`. These functions are documented on page 262.)

`\fp_sign:n` Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `_fp_parse:n`, namely, for better error reporting.

```

29140 \cs_new:Npn \fp_sign:n #1
29141   { \fp_to_decimal:n { sign \__fp_parse:n {#1} } }

```

(End of definition for `\fp_sign:n`. This function is documented on page 261.)

`\fp_abs:n` Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `__fp_parse:n`, namely, for better error reporting.

```

29142 \cs_new:Npn \fp_abs:n #1
29143   { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }

```

(End of definition for `\fp_abs:n`. This function is documented on page 280.)

`\fp_max:nn` Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, etc.

```

\fp_min:nn 29144 \cs_new:Npn \fp_max:nn #1#2
29145   { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
29146 \cs_new:Npn \fp_min:nn #1#2
29147   { \fp_to_decimal:n { min ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }

```

(End of definition for `\fp_max:nn` and `\fp_min:nn`. These functions are documented on page 280.)

78.10 Convert an array of floating points to a comma list

`__fp_array_to_clist:n` Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```

\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
{
  \use_none:n #1
  { , ~ } \fp_to_tl:n { #1 #2 ; }
  \__fp_array_to_clist_loop:Nw
}

```

The `\use_ii:nn` function is expanded after `__fp_expand:n` is done, and it removes `,~` from the start of the representation.

```

29148 \cs_new:Npn \__fp_array_to_clist:n #1
29149   {
29150     \tl_if_empty:nF {#1}
29151     {
29152       \exp_last_unbraced:Ne \use_ii:nn
29153       {
29154         \__fp_array_to_clist_loop:Nw #1 { ? \prg_break: } ;
29155         \prg_break_point:
29156       }
29157     }
29158   }
29159 \cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
29160   {
29161     \use_none:n #1
29162     , ~
29163     \exp_not:f { \__fp_to_tl_dispatch:w #1 #2 ; }
29164     \__fp_array_to_clist_loop:Nw
29165   }

```

(End of definition for _fp_array_to_clist:n and _fp_array_to_clist_loop:Nw.)

29166 </package>

Chapter 79

l3fp-random implementation

```
29167 (*package)
```

```
29168 (@@=fp)
```

```
\_fp_parse_word_rand:N  
\_fp_parse_word_randint:N
```

Those functions may receive a variable number of arguments. We won't use the argument ?.

```
29169 \cs_new:Npn \_fp_parse_word_rand:N
```

```
29170   { \_fp_parse_function:NNN \_fp_rand_o:Nw ? }
```

```
29171 \cs_new:Npn \_fp_parse_word_randint:N
```

```
29172   { \_fp_parse_function:NNN \_fp_randint_o:Nw ? }
```

(End of definition for `_fp_parse_word_rand:N` and `_fp_parse_word_randint:N`.)

79.1 Engine support

Obviously, every word “random” below means “pseudo-random”, as we have no access to entropy (except a very unreliable source of entropy: the time it takes to run some code).

The primitive random number generator (RNG) is provided as `\tex_uniformdeviate:D`. Under the hood, it maintains an array of 55 28-bit numbers, updated with a linear recursion relation (similar to Fibonacci numbers) modulo 2^{28} . When `\tex_uniformdeviate:D` (`<integer>`) is called (for brevity denote by N the `<integer>`), the next 28-bit number is read from the array, scaled by $N/2^{28}$, and rounded. To prevent 0 and N from appearing half as often as other numbers, they are both mapped to the result 0.

This process means that `\tex_uniformdeviate:D` only gives a uniform distribution from 0 to $N-1$ if N is a divisor of 2^{28} , so we will mostly call the RNG with such power of 2 arguments. If N does not divide 2^{28} , then the relative non-uniformity (difference between probabilities of getting different numbers) is about $N/2^{28}$. This implies that detecting deviation from $1/N$ of the probability of a fixed value X requires about $2^{56}/N$ random trials. But collective patterns can reduce this to about $2^{56}/N^2$. For instance with $N = 3 \times 2^k$, the modulo 3 repartition of such random numbers is biased with a non-uniformity about $2^k/2^{28}$ (which is much worse than the circa $3/2^{28}$ non-uniformity from taking directly $N = 3$). This is detectable after about $2^{56}/2^{2k} = 9 \cdot 2^{56}/N^2$ random numbers. For $k = 15$, $N = 98304$, this means roughly 2^{26} calls to the RNG (experimentally this takes at the very least 16 seconds on a 2 giga-hertz processor). While this bias is not quite problematic, it is uncomfortably close to being so, and it becomes worse as N is increased. In our code, we shall thus combine several results from the RNG.

The RNG has three types of unexpected correlations. First, everything is linear modulo 2^{28} , hence the lowest k bits of the random numbers only depend on the lowest k bits of the seed (and of course the number of times the RNG was called since setting the seed). The recommended way to get a number from 0 to $N - 1$ is thus to scale the raw 28-bit integer, as the engine's RNG does. We will go further and in fact typically we discard some of the lowest bits.

Second, suppose that we call the RNG with the same argument N to get a set of K integers in $[0, N - 1]$ (throwing away repeats), and suppose that $N > K^3$ and $K > 55$. The recursion used to construct more 28-bit numbers from previous ones is linear: $x_n = x_{n-55} - x_{n-24}$ or $x_n = x_{n-55} - x_{n-24} + 2^{28}$. After rescaling and rounding we find that the result $N_n \in [0, N - 1]$ is among $N_{n-55} - N_{n-24} + \{-1, 0, 1\}$ modulo N (a more detailed analysis shows that 0 appears with frequency close to $3/4$). The resulting set thus has more triplets (a, b, c) than expected obeying $a = b + c$ modulo N . Namely it will have of order $(K - 55) \times 3/4$ such triplets, when one would expect $K^3/(6N)$. This starts to be detectable around $N = 2^{18} > 55^3$ (earlier if one keeps track of positions too, but this is more subtle than it looks because the array of 28-bit integers is read backwards by the engine). Hopefully the correlation is subtle enough to not affect realistic documents so we do not specifically mitigate against this. Since we typically use two calls to the RNG per `\int_rand:nn` we would need to investigate linear relations between the x_{2n} on the one hand and between the x_{2n+1} on the other hand. Such relations will have more complicated coefficients than ± 1 , which alleviates the issue.

Third, consider successive batches of 165 calls to the RNG (with argument 2^{28} or with argument 2 for instance), then most batches have more odd than even numbers. Note that this does not mean that there are more odd than even numbers overall. Similar issues are discussed in Knuth's TAOCP volume 2 near exercise 3.3.2-31. We do not have any mitigation strategy for this.

Ideally, our algorithm should be:

- Uniform. The result should be as uniform as possible assuming that the RNG's underlying 28-bit integers are uniform.
- Uncorrelated. The result should not have detectable correlations between different seeds, similar to the lowest-bit ones mentioned earlier.
- Quick. The algorithm should be fast in \TeX , so no “bit twiddling”, but “digit twiddling” is ok.
- Simple. The behaviour must be documentable precisely.
- Predictable. The number of calls to the RNG should be the same for any `\int_rand:nn`, because then the algorithm can be modified later without changing the result of other uses of the RNG.
- Robust. It should work even for `\int_rand:nn { - \c_max_int } { \c_max_int }` where the range is not representable as an integer. In fact, we also provide later a floating-point `randint` whose range can go all the way up to $2 \times 10^{16} - 1$ possible values.

Some of these requirements conflict. For instance, uniformity cannot be achieved with a fixed number of calls to the RNG.

Denote by `random(N)` one call to `\tex_uniformdeviate:D` with argument N , and by `ediv(p, q)` the ε - \TeX rounding division giving $\lfloor p/q + 1/2 \rfloor$. Denote by $\langle \min \rangle$, $\langle \max \rangle$

and $R = \langle \mathit{max} \rangle - \langle \mathit{min} \rangle + 1$ the arguments of `\int_min:nn` and the number of possible outcomes. Note that $R \in [1, 2^{32} - 1]$ cannot necessarily be represented as an integer (however, $R - 2^{31}$ can). Our strategy is to get two 28-bit integers X and Y from the RNG, split each into 14-bit integers, as $X = X_1 \times 2^{14} + X_0$ and $Y = Y_1 \times 2^{14} + Y_0$ then return essentially $\langle \mathit{min} \rangle + \lfloor R(X_1 \times 2^{-14} + Y_1 \times 2^{-28} + Y_0 \times 2^{-42} + X_0 \times 2^{-56}) \rfloor$. For small R the X_0 term has a tiny effect so we ignore it and we can compute $R \times Y/2^{28}$ much more directly by `random(R)`.

- If $R \leq 2^{17} - 1$ then return `ediv(R random(214) + random(R) + 213, 214) - 1 + <min>`. The shifts by 2^{13} and -1 convert ε -TeX division to truncated division. The bound on R ensures that the number obtained after the shift is less than `\c_max_int`. The non-uniformity is at most of order $2^{17}/2^{42} = 2^{-25}$.
- Split $R = R_2 \times 2^{28} + R_1 \times 2^{14} + R_0$, where $R_2 \in [0, 15]$. Compute $\langle \mathit{min} \rangle + R_2 X_1 2^{14} + (R_2 Y_1 + R_1 X_1) + \text{ediv}(R_2 Y_0 + R_1 Y_1 + R_0 X_1 + \text{ediv}(R_2 X_0 + R_0 Y_1 + \text{ediv}((2^{14} R_1 + R_0)(2^{14} Y_0 + X_0), 2^{28}), 2^{14}), 2^{14})$ then map a result of $\langle \mathit{max} \rangle + 1$ to $\langle \mathit{min} \rangle$. Writing each `ediv` in terms of truncated division with a shift, and using $\lfloor (p + \lceil r/s \rceil)/q \rfloor = \lfloor (ps + r)/(sq) \rfloor$, what we compute is equal to $\lfloor \langle \mathit{exact} \rangle + 2^{-29} + 2^{-15} + 2^{-1} \rfloor$ with $\langle \mathit{exact} \rangle = \langle \mathit{min} \rangle + R \times 0.X_1 Y_1 Y_0 X_0$. Given we map $\langle \mathit{max} \rangle + 1$ to $\langle \mathit{min} \rangle$, the shift has no effect on uniformity. The non-uniformity is bounded by $R/2^{56} < 2^{-24}$. It may be possible to speed up the code by dropping tiny terms such as $R_0 X_0$, but the analysis of non-uniformity proves too difficult.

To avoid the overflow when the computation yields $\langle \mathit{max} \rangle + 1$ with $\langle \mathit{max} \rangle = 2^{31} - 1$ (note that R is then arbitrary), we compute the result in two pieces. Compute $\langle \mathit{first} \rangle = \langle \mathit{min} \rangle + R_2 X_1 2^{14}$ if $R_2 < 8$ or $\langle \mathit{min} \rangle + 8 X_1 2^{14} + (R_2 - 8) X_1 2^{14}$ if $R_2 \geq 8$, the expressions being chosen to avoid overflow. Compute $\langle \mathit{second} \rangle = R_2 Y_1 + R_1 X_1 + \text{ediv}(\dots)$, at most $R_2 2^{14} + R_1 2^{14} + R_0 \leq 2^{28} + 15 \times 2^{14} - 1$, not at risk of overflowing. We have $\langle \mathit{first} \rangle + \langle \mathit{second} \rangle = \langle \mathit{max} \rangle + 1 = \langle \mathit{min} \rangle + R$ if and only if $\langle \mathit{second} \rangle = R 12^{14} + R_0 + R_2 2^{14}$ and $2^{14} R_2 X_1 = 2^{28} R_2 - 2^{14} R_2$ (namely $R_2 = 0$ or $X_1 = 2^{14} - 1$). In that case, return $\langle \mathit{min} \rangle$, otherwise return $\langle \mathit{first} \rangle + \langle \mathit{second} \rangle$, which is safe because it is at most $\langle \mathit{max} \rangle$. Note that the decision of what to return does not need $\langle \mathit{first} \rangle$ explicitly so we don't actually compute it, just put it in an integer expression in which $\langle \mathit{second} \rangle$ is eventually added (or not).

- To get a floating point number in $[0, 1)$ just call the $R = 10000 \leq 2^{17} - 1$ procedure above to produce four blocks of four digits.
- To get an integer floating point number in a range (whose size can be up to $2 \times 10^{16} - 1$), work with fixed-point numbers: get six times four digits to build a fixed point number, multiply by R and add $\langle \mathit{min} \rangle$. This requires some care because `l3fp-extended` only supports non-negative numbers.

`\c__kernel_randint_max_int` Constant equal to $2^{17} - 1$, the maximal size of a range that `\int_range:nn` can do with its “simple” algorithm.

29173 `\int_const:Nn \c__kernel_randint_max_int { 131071 }`

(End of definition for `\c__kernel_randint_max_int`.)

`__kernel_randint:n` Used in an integer expression, `__kernel_randint:n {R}` gives a random number $1 + \lfloor (R \text{random}(2^{14}) + \text{random}(R))/2^{14} \rfloor$ that is in $[1, R]$. Previous code was computing $\lfloor p/2^{14} \rfloor$ as `ediv(p - 213, 214)` but that wrongly gives -1 for $p = 0$.

```

29174 \cs_new:Npn \__kernel_randint:n #1
29175 {
29176   (#1 * \tex_uniformdeviate:D 16384
29177   + \tex_uniformdeviate:D #1 + 8192 ) / 16384
29178 }

```

(End of definition for __kernel_randint:n.)

Used as `__fp_rand_myriads:n {XXX}` with one letter X (specifically) per block of four digit we want; it expands to `;` followed by the requested number of brace groups, each containing four (pseudo-random) digits. Digits are produced as a random number in [10000, 19999] for the usual reason of preserving leading zeros.

```

29179 \cs_new:Npn \__fp_rand_myriads:n #1
29180 { \__fp_rand_myriads_loop:w #1 \prg_break: X \prg_break_point: ; }
29181 \cs_new:Npn \__fp_rand_myriads_loop:w #1 X
29182 {
29183   #1
29184   \exp_after:wN \__fp_rand_myriads_get:w
29185   \int_value:w \__fp_int_eval:w 9999 +
29186   \__kernel_randint:n { 10000 }
29187   \__fp_rand_myriads_loop:w
29188 }
29189 \cs_new:Npn \__fp_rand_myriads_get:w 1 #1 ; { ; {#1} }

```

(End of definition for __fp_rand_myriads:n, __fp_rand_myriads_loop:w, and __fp_rand_myriads_get:w.)

79.2 Random floating point

First we check that `random` was called without argument. Then get four blocks of four digits and convert that fixed point number to a floating point number (this correctly sets the exponent). This has a minor bug: if all of the random numbers are zero then the result is correctly 0 but it raises the underflow flag; it should not do that.

```

29190 \cs_new:Npn \__fp_rand_o:Nw ? #1 @
29191 {
29192   \tl_if_empty:nTF {#1}
29193   {
29194     \exp_after:wN \__fp_rand_o:w
29195     \exp:w \exp_end_continue_f:w
29196     \__fp_rand_myriads:n { XXXX } { 0000 } { 0000 } ; 0
29197   }
29198   {
29199     \msg_expandable_error:nnnnn
29200     { fp } { num-args } { rand() } { 0 } { 0 }
29201     \exp_after:wN \c_nan_fp
29202   }
29203 }
29204 \cs_new:Npn \__fp_rand_o:w ;
29205 {
29206   \exp_after:wN \__fp_sanitize:Nw
29207   \exp_after:wN 0
29208   \int_value:w \__fp_int_eval:w \c_zero_int
29209   \__fp_fixed_to_float_o:wN
29210 }

```

(End of definition for `__fp_rand_o:Nw` and `__fp_rand_o:w`.)

79.3 Random integer

```

\__fp_randint_o:Nw Enforce that there is one argument (then add first argument 1) or two arguments. Call
\__fp_randint_default:w \__fp_randint_badarg:w on each; this function inserts 1 \exp_stop_f: to end the
\__fp_randint_badarg:w \if_case:w statement if either the argument is not an integer or if its absolute value is
\__fp_randint_o:w  $\geq 10^{16}$ . Also bail out if \__fp_compare_back:ww yields 1, meaning that the bounds are
\__fp_randint_auxi_o:ww not in the right order. Otherwise an auxiliary converts each argument times  $10^{-16}$  (hence
\__fp_randint_auxii:wn the shift in exponent) to a 24-digit fixed point number (see l3fp-extended). Then compute
\__fp_randint_auxiii_o:ww the number of choices,  $\langle max \rangle + 1 - \langle min \rangle$ . Create a random 24-digit fixed-point number
\__fp_randint_auxiv_o:ww with \__fp_rand_myriads:n, then use a fused multiply-add instruction to multiply the
\__fp_randint_auxv_o:w number of choices to that random number and add it to  $\langle min \rangle$ . Then truncate to 16
digits (namely select the integer part of  $10^{16}$  times the result) before converting back to
a floating point number (\__fp_sanitize:Nw takes care of zero). To avoid issues with
negative numbers, add 1 to all fixed point numbers (namely  $10^{16}$  to the integers they
represent), except of course when it is time to convert back to a float.
29211 \cs_new:Npn \__fp_randint_o:Nw ?
29212 {
29213   \__fp_parse_function_one_two:nnw
29214   { randint }
29215   { \__fp_randint_default:w \__fp_randint_o:w }
29216 }
29217 \cs_new:Npn \__fp_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }
29218 \cs_new:Npn \__fp_randint_badarg:w \s__fp \__fp_chk:w #1#2#3;
29219 {
29220   \__fp_int:wTF \s__fp \__fp_chk:w #1#2#3;
29221   {
29222     \if_meaning:w 1 #1
29223     \if_int_compare:w
29224       \__fp_use_i_until_s:nw #3 ; > \c__fp_prec_int
29225       \c_one_int
29226     \fi:
29227     \fi:
29228   }
29229   { \c_one_int }
29230 }
29231 \cs_new:Npn \__fp_randint_o:w #1; #2; @
29232 {
29233   \if_case:w
29234     \__fp_randint_badarg:w #1;
29235     \__fp_randint_badarg:w #2;
29236     \if:w 1 \__fp_compare_back:ww #2; #1; \c_one_int \fi:
29237     \c_zero_int
29238     \__fp_randint_auxi_o:ww #1; #2;
29239   \or:
29240     \__fp_invalid_operation_tl_o:ff
29241     { randint } { \__fp_array_to_clist:n { #1; #2; } }
29242     \exp:w
29243     \fi:
29244     \exp_after:wN \exp_end:
29245   }

```



```

29246 \cs_new:Npn \__fp_randint_auxi_o:ww #1 ; #2 ; #3 \exp_end:
29247 {
29248   \fi:
29249   \__fp_randint_auxii:wn #2 ;
29250   { \__fp_randint_auxii:wn #1 ; \__fp_randint_auxiii_o:ww }
29251 }
29252 \cs_new:Npn \__fp_randint_auxii:wn \s__fp \__fp_chk:w #1#2#3#4 ;
29253 {
29254   \if_meaning:w 0 #1
29255     \exp_after:wN \use_i:nn
29256   \else:
29257     \exp_after:wN \use_ii:nn
29258   \fi:
29259   { \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl }
29260   {
29261     \exp_after:wN \__fp_ep_to_fixed:wwn
29262     \int_value:w \__fp_int_eval:w
29263     #3 - \c__fp_prec_int , #4 {0000} {0000} ;
29264     {
29265       \if_meaning:w 0 #2
29266         \exp_after:wN \use_i:nmmn
29267         \exp_after:wN \__fp_fixed_add_one:wn
29268       \fi:
29269       \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
29270     }
29271     \__fp_fixed_continue:wn
29272   }
29273 }
29274 \cs_new:Npn \__fp_randint_auxiii_o:ww #1 ; #2 ;
29275 {
29276   \__fp_fixed_add:wwn #2 ;
29277   {0000} {0000} {0000} {0001} {0000} {0000} ;
29278   \__fp_fixed_sub:wwn #1 ;
29279   {
29280     \exp_after:wN \use_i:nn
29281     \exp_after:wN \__fp_fixed_mul_add:wwwn
29282     \exp:w \exp_end_continue_f:w \__fp_rand_myriads:n { XXXXXX } ;
29283   }
29284   #1 ;
29285   \__fp_randint_auxiv_o:ww
29286   #2 ;
29287   \__fp_randint_auxv_o:w #1 ; @
29288 }
29289 \cs_new:Npn \__fp_randint_auxiv_o:ww #1#2#3#4#5 ; #6#7#8#9
29290 {
29291   \if_int_compare:w
29292     \if_int_compare:w #1#2 > #6#7 \exp_stop_f: 1 \else:
29293     \if_int_compare:w #1#2 < #6#7 \exp_stop_f: - \fi: \fi:
29294     #3#4 > #8#9 \exp_stop_f:
29295     \__fp_use_i_until_s:nw
29296   \fi:
29297   \__fp_randint_auxv_o:w {#1}{#2}{#3}{#4}#5
29298 }
29299 \cs_new:Npn \__fp_randint_auxv_o:w #1#2#3#4#5 ; #6 @

```

```

29300 {
29301   \exp_after:wN \_fp_sanitize:Nw
29302   \int_value:w
29303   \if_int_compare:w #1 < 10000 \exp_stop_f:
29304     2
29305   \else:
29306     0
29307     \exp_after:wN \exp_after:wN
29308     \exp_after:wN \_fp_reverse_args:Nww
29309   \fi:
29310   \exp_after:wN \_fp_fixed_sub:wwn \c__fp_one_fixed_tl
29311   {#1} {#2} {#3} {#4} {0000} {0000} ;
29312   {
29313     \exp_after:wN \exp_stop_f:
29314     \int_value:w \_fp_int_eval:w \c__fp_prec_int
29315     \_fp_fixed_to_float_o:wN
29316   }
29317   0
29318   \exp:w \exp_after:wN \exp_end:
29319 }

```

(End of definition for _fp_randint_o:Nw and others.)

`\int_rand:nn` Evaluate the argument and filter out the case where the lower bound #1 is more than the upper bound #2. Then determine whether the range is narrower than `\c__kernel_randint_max_int`; #2-#1 may overflow for very large positive #2 and negative #1. If the range is narrow, call `_kernel_randint:n` {<choices>} where <choices> is the number of possible outcomes. If the range is wide, use somewhat slower code.

`_fp_randint:ww`

```

29320 \cs_new:Npn \int_rand:nn #1#2
29321 {
29322   \int_eval:n
29323   {
29324     \exp_after:wN \_fp_randint:ww
29325     \int_value:w \int_eval:n {#1} \exp_after:wN ;
29326     \int_value:w \int_eval:n {#2} ;
29327   }
29328 }
29329 \cs_new:Npn \_fp_randint:ww #1; #2;
29330 {
29331   \if_int_compare:w #1 > #2 \exp_stop_f:
29332   \msg_expandable_error:nmmn
29333   { kernel } { randint-backward-range } {#1} {#2}
29334   \_fp_randint:ww #2; #1;
29335   \else:
29336   \if_int_compare:w \_fp_int_eval:w #2
29337   \if_int_compare:w #1 > \c_zero_int
29338   - #1 < \_fp_int_eval:w
29339   \else:
29340     < \_fp_int_eval:w #1 +
29341   \fi:
29342   \c__kernel_randint_max_int
29343   \_fp_int_eval_end:
29344   \_kernel_randint:n
29345   { \_fp_int_eval:w #2 - #1 + 1 \_fp_int_eval_end: }

```

```

29346     - 1 + #1
29347     \else:
29348         \__kernel_randint:nn {#1} {#2}
29349     \fi:
29350 \fi:
29351 }

```

(End of definition for `\int_rand:nn` and `__fp_randint:ww`. This function is documented on page 177.)

`__kernel_randint:nn` Any $n \in [-2^{31} + 1, 2^{31} - 1]$ is uniquely written as $2^{14}n_1 + n_2$ with $n_1 \in [-2^{17}, 2^{17} - 1]$ and $n_2 \in [0, 2^{14} - 1]$. Calling `__fp_randint_split_o:Nw n` ; gives n_1 ; n_2 ; and expands the next token once. We do this for two random numbers and apply `__fp_randint_split_o:Nw` twice to fully decompose the range R . One subtlety is that we compute $R - 2^{31} = \langle \max \rangle - \langle \min \rangle - (2^{31} - 1) \in [-2^{31} + 1, 2^{31} - 1]$ rather than R to avoid overflow.

Then we have `__fp_randint_wide_aux:w` $\langle X_1 \rangle; \langle X_0 \rangle; \langle Y_1 \rangle; \langle Y_0 \rangle; \langle R_2 \rangle; \langle R_1 \rangle; \langle R_0 \rangle; .$ and we apply the algorithm described earlier.

```

29352 \cs_new:Npn \__kernel_randint:nn #1#2
29353 {
29354     #1
29355     \exp_after:wN \__fp_randint_wide_aux:w
29356     \int_value:w
29357         \exp_after:wN \__fp_randint_split_o:Nw
29358         \tex_uniformdeviate:D 268435456 ;
29359     \int_value:w
29360         \exp_after:wN \__fp_randint_split_o:Nw
29361         \tex_uniformdeviate:D 268435456 ;
29362     \int_value:w
29363         \exp_after:wN \__fp_randint_split_o:Nw
29364         \int_value:w \__fp_int_eval:w 131072 +
29365         \exp_after:wN \__fp_randint_split_o:Nw
29366         \int_value:w
29367         \__kernel_int_add:nnn {#2} { -#1 } { -\c_max_int } ;
29368     .
29369 }
29370 \cs_new:Npn \__fp_randint_split_o:Nw #1#2 ;
29371 {
29372     \if_meaning:w 0 #1
29373     0 \exp_after:wN ; \int_value:w 0
29374     \else:
29375         \exp_after:wN \__fp_randint_split_aux:w
29376         \int_value:w \__fp_int_eval:w (#1#2 - 8192) / 16384 ;
29377         + #1#2
29378     \fi:
29379     \exp_after:wN ;
29380 }
29381 \cs_new:Npn \__fp_randint_split_aux:w #1 ;
29382 {
29383     #1 \exp_after:wN ;
29384     \int_value:w \__fp_int_eval:w - #1 * 16384
29385 }
29386 \cs_new:Npn \__fp_randint_wide_aux:w #1;#2; #3;#4; #5;#6;#7; .
29387 {
29388     \exp_after:wN \__fp_randint_wide_auxii:w
29389     \int_value:w \__fp_int_eval:w #5 * #3 + #6 * #1 +

```

```

29390     (#5 * #4 + #6 * #3 + #7 * #1 +
29391     (#5 * #2 + #7 * #3 +
29392     (16384 * #6 + #7) * (16384 * #4 + #2) / 268435456) / 16384
29393     ) / 16384 \exp_after:wN ;
29394     \int_value:w \__fp_int_eval:w (#5 + #6) * 16384 + #7 ;
29395     #1 ; #5 ;
29396   }
29397 \cs_new:Npn \__fp_randint_wide_auxii:w #1; #2; #3; #4;
29398 {
29399   \if_int_odd:w 0
29400     \if_int_compare:w #1 = #2 \else: \exp_stop_f: \fi:
29401     \if_int_compare:w #4 = \c_zero_int 1 \fi:
29402     \if_int_compare:w #3 = 16383 ~ 1 \fi:
29403     \exp_stop_f:
29404     \exp_after:wN \prg_break:
29405   \fi:
29406   \if_int_compare:w #4 < 8 \exp_stop_f:
29407     + #4 * #3 * 16384
29408   \else:
29409     + 8 * #3 * 16384 + (#4 - 8) * #3 * 16384
29410   \fi:
29411   + #1
29412   \prg_break_point:
29413 }

```

(End of definition for __kernel_randint:nn and others.)

`\int_rand:n` Similar to `\int_rand:nn`, but needs fewer checks.
`__fp_randint:n`

```

29414 \cs_new:Npn \int_rand:n #1
29415 {
29416   \int_eval:n
29417   { \exp_args:Nf \__fp_randint:n { \int_eval:n {#1} } }
29418 }
29419 \cs_new:Npn \__fp_randint:n #1
29420 {
29421   \if_int_compare:w #1 < \c_one_int
29422     \msg_expandable_error:nnnn
29423     { kernel } { randint-backward-range } { 1 } {#1}
29424     \__fp_randint:ww #1; 1;
29425   \else:
29426     \if_int_compare:w #1 > \c__kernel_randint_max_int
29427     \__kernel_randint:nn { 1 } {#1}
29428   \else:
29429     \__kernel_randint:n {#1}
29430   \fi:
29431   \fi:
29432 }

```

(End of definition for `\int_rand:n` and `__fp_randint:n`. This function is documented on page 177.)

```

29433 </package>

```

Chapter 80

l3fp-types implementation

```
29434 (*package)
29435 (@@=fp)
```

80.1 Support for types

Despite lack of documentation, the l3fp internals support types. Each additional type must define

- `\s__fp_⟨type⟩` and `__fp_⟨type⟩_chk:w`;
- `__fp_exp_after_⟨type⟩_f:nw`;
- `__fp_⟨type⟩_to_⟨out⟩:w` for `⟨out⟩` among `decimal`, `scientific`, `tl`;

and may define

- `__fp_⟨type⟩_to_int:w` and `__fp_⟨type⟩_to_dim:w`;
- `__fp_⟨op⟩_⟨type⟩_o:w` for any of the `⟨op⟩` that the type implements, among `acos`, `acsc`, `asec`, `asin`, `cos`, `cot`, `csc`, `exp`, `ln`, `not`, `sec`, `set_sign`, `sin`, `tan`;
- `__fp_⟨type1⟩_⟨op⟩_⟨type2⟩_o:ww` for `⟨op⟩` among `^*/-+&|` and for every pair of types;
- `__fp_⟨type1⟩_bcmp_⟨type2⟩:ww` for every pair of types.

The latter is set up in l3fp-logic.

80.2 Dispatch according to the type

```
\__fp_types_cs_to_op:N From \__fp_⟨op⟩_o:w produce ⟨op⟩, otherwise ?.
\__fp_types_cs_to_op_auxi:www
29436 \cs_new:Npe \__fp_types_cs_to_op:N #1
29437 {
29438   \exp_not:N \exp_after:wN \exp_not:N \__fp_types_cs_to_op_auxi:wwwn
29439   \exp_not:N \token_to_str:N #1 \s__fp_mark
29440   \exp_not:N \__fp_use_i_delimit_by_s_stop:nw
29441   \tl_to_str:n { \__fp_o:w } \s__fp_mark
29442   { \exp_not:N \__fp_use_i_delimit_by_s_stop:nw ? }
```

```

29443     \s__fp_stop
29444   }
29445 \use:e
29446   {
29447     \cs_new:Npn \exp_not:N \__fp_types_cs_to_op_auxi:wwwn
29448       #1 \tl_to_str:n { __fp_ } #2
29449       \tl_to_str:n { _o:w } #3 \s__fp_mark #4 { #4 {#2} }
29450   }

```

(End of definition for __fp_types_cs_to_op:N and __fp_types_cs_to_op_auxi:wwwn.)

```

\__fp_types_unary:NNw     \__fp_types_unary:NNw \__fp_⟨function⟩_o:w
\__fp_types_unary_auxi:nNw  ⟨token⟩ ⟨operand⟩ @
\__fp_types_unary_auxii:NnNw
29451 \cs_new:Npn \__fp_types_unary:NNw #1
29452   {
29453     \exp_args:Nf \__fp_types_unary_auxi:nNw
29454       { \__fp_types_cs_to_op:N #1 }
29455   }
29456 \cs_new:Npn \__fp_types_unary_auxi:nNw #1#2#3
29457   {
29458     \exp_after:wN \__fp_types_unary_auxii:NnNw
29459     \cs:w __fp_#1 \__fp_type_from_scan:N #3 _o:w \cs_end:
29460     {#1}
29461     #2#3
29462   }
29463 \cs_new:Npn \__fp_types_unary_auxii:NnNw #1#2#3
29464   {
29465     \token_if_eq_meaning:NNTF \scan_stop: #1
29466     { \__fp_invalid_operation_o:nw {#2} }
29467     { #1 #3 }
29468   }

```

(End of definition for __fp_types_unary:NNw, __fp_types_unary_auxi:nNw, and __fp_types_unary_auxii:NnNw.)

```

\__fp_types_binary:Nww     \__fp_types_binary:Nww \__fp_⟨binop⟩_o:ww
\__fp_types_binary_auxi:Nww  ⟨operand1⟩ ⟨operand2⟩ @
\__fp_types_binary_auxii:NNww
29469 \cs_new:Npn \__fp_types_binary:Nww #1
29470   {
29471     \exp_last_unbraced:Nf \__fp_types_binary_auxi:Nww
29472       { \__fp_types_cs_to_op:N #1 }
29473   }
29474 \cs_new:Npn \__fp_types_binary_auxi:Nww #1#2#3; #4#5; @
29475   {
29476     \exp_after:wN \__fp_types_binary_auxii:NNww
29477     \cs:w
29478       __fp
29479       \__fp_type_from_scan:N #2
29480       _#1
29481       \__fp_type_from_scan:N #4
29482       _o:ww
29483     \cs_end:
29484     #1 #2#3; #4#5;
29485   }

```

```
29486 \cs_new:Npn \__fp_types_binary_auxii:NNww #1#2
29487 {
29488   \token_if_eq_meaning:NNTF \scan_stop: #1
29489   { \__fp_invalid_operation_o:Nww #2 }
29490   {#1}
29491 }
```

(End of definition for __fp_types_binary:Nww, __fp_types_binary_auxi:Nww, and __fp_types_-binary_auxii:NNww.)

```
29492 \end{package}
```

Chapter 81

I3fp-symbolic implementation

```
29493 (*package)
```

```
29494 (@@=fp)
```

81.1 Misc

`\l__fp_symbolic_fp` Scratch floating point.

```
29495 \fp_new:N \l__fp_symbolic_fp
```

(End of definition for \l__fp_symbolic_fp.)

81.2 Building blocks for expressions

Every symbolic expression has the form `\s__fp_symbolic __fp_symbolic_chk:w <operation> , {<operands>} <junk>`; where the `<operation>` is a list of N-type tokens, the `<operands>` is an array of floating point objects, and the `<junk>` is to be discarded. If the outermost operator (last to be evaluated) is unary, the expression has the form

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_types_unary:NNw \__fp_<op>_o:w <token> ,
{ <operand> } <junk> ;
```

where the `<op>` is a unary operation (`set_sign`, `cos`, ...), and the `<token>` and `<operand>` are used as arguments for `__fp_<op>_o:w` (or the type-specific analog of this function). If the outermost operator is binary, the expression has the form

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_types_binary:Nww \__fp_<op>_o:ww ,
{ <operand1> <operand2> } <junk> ;
```

where the `<op>` is an operation (`+`, `&`, ...), and `__fp_<op>_o:ww` receives the `<operands>` as arguments. If the expression consists of a single variable, it is stored as

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_variable_o:w <identifier> ,
{ } <junk> ;
```


Symbolic expressions are stored in a prefix form. When encountering a symbolic expression in a floating point computation, we attempt to evaluate the operands as much as possible, and if that yields floating point numbers rather than expressions, we apply the operator which follows (if the function is known).

For instance, the expression $a + b * \sin(c)$ is stored as

```
\s__fp_symbolic \__fp_symbolic_chk:w
  \__fp_types_binary:Nww \__fp+_o:ww ,
  {
    \s__fp_symbolic \__fp_symbolic_chk:w
      \__fp_variable_o:w a , { } ;
    \s__fp_symbolic \__fp_symbolic_chk:w
      \__fp_types_binary:Nww \__fp*_o:ww ,
      {
        \s__fp_symbolic \__fp_symbolic_chk:w
          \__fp_variable_o:w b , { } ;
        \s__fp_symbolic \__fp_symbolic_chk:w
          \__fp_types_unary:NNw \__fp_sin_o:w \use_i:nn ,
          {
            \s__fp_symbolic \__fp_symbolic_chk:w
              \__fp_variable_o:w c , { } ;
          } ;
        } ;
      } ;
  } ;
```

`\s__fp_symbolic` Scan mark indicating the start of a symbolic expression.

29496 `\scan_new:N \s__fp_symbolic`

(End of definition for `\s__fp_symbolic`.)

`__fp_symbolic_chk:w` Analog of `__fp_chk:w` for symbolic expressions.

29497 `\cs_new_protected:Npn __fp_symbolic_chk:w #1,#2#3;`

```
29498 {
29499   \msg_error:nne { fp } { misused-fp }
29500   {
29501     \__fp_to_tl_dispatch:w
29502     \s__fp_symbolic \__fp_symbolic_chk:w #1,{#2};
29503   }
29504 }
```

(End of definition for `__fp_symbolic_chk:w`.)

81.3 Expanding after a symbolic expression

`__fp_if_has_symbolic:nTF` Tests if #1 contains `\s__fp_symbolic` at top-level. This test should be precise enough
`__fp_if_has_symbolic_aux:w` to determine if a given array contains a symbolic expression or only consists of floating points. Used in `__fp_exp_after_symbolic_f:nw`.

```
29505 \cs_new:Npn \__fp_if_has_symbolic:nTF #1
29506 {
29507   \__fp_if_has_symbolic_aux:w
29508   #1 \s__fp_mark \use_i:nn
```

```

29509     \s__fp_symbolic \s__fp_mark \use_ii:nn
29510     \s__fp_stop
29511   }
29512 \cs_new:Npn \__fp_if_has_symbolic_aux:w
29513   #1 \s__fp_symbolic #2 \s__fp_mark #3#4 \s__fp_stop { #3 }

```

(End of definition for `__fp_if_has_symbolic:nTF` and `__fp_if_has_symbolic_aux:w`.)

```

\__fp_exp_after_symbolic_f:nw
\__fp_exp_after_symbolic_aux:w
\__fp_exp_after_symbolic_loop:N

```

This function does two things: trigger an f-expansion of the argument #1 after the following symbolic expression, and evaluate all pieces of the expression which can be evaluated.

```

29514 \cs_new:Npn \__fp_exp_after_symbolic_f:nw
29515   #1 \s__fp_symbolic \__fp_symbolic_chk:w #2, #3#4;
29516   {
29517     \exp_after:wN \__fp_exp_after_symbolic_aux:w
29518     \exp:w
29519     \__fp_exp_after_symbolic_loop:N #2
29520     { , \exp:w \use_none:nn }
29521     \exp_after:wN \exp_end: \exp_after:wN
29522     {
29523       \exp:w \exp_end_continue_f:w
29524       \__fp_exp_after_array_f:w #3 \s__fp_expr_stop
29525       \exp_after:wN
29526     }
29527     \exp_after:wN ;
29528     \exp:w \exp_end_continue_f:w #1
29529   }
29530 \cs_new:Npn \__fp_exp_after_symbolic_aux:w #1, #2;
29531   {
29532     \__fp_if_has_symbolic:nTF {#2}
29533     { \s__fp_symbolic \__fp_symbolic_chk:w #1, {#2} ; }
29534     { #1 #2 @ \prg_do_nothing: }
29535   }
29536 \cs_new:Npn \__fp_exp_after_symbolic_loop:N #1
29537   {
29538     \exp_after:wN \exp_end:
29539     \exp_after:wN #1
29540     \exp:w
29541     \__fp_exp_after_symbolic_loop:N
29542   }

```

(End of definition for `__fp_exp_after_symbolic_f:nw`, `__fp_exp_after_symbolic_aux:w`, and `__fp_exp_after_symbolic_loop:N`.)

81.4 Applying infix operators to expressions

```

\__fp_symbolic_binary_o:Nww

```

Used when applying infix operators to expressions.

```

29543 \cs_new:Npn \__fp_symbolic_binary_o:Nww #1 #2; #3;
29544   {
29545     \__fp_exp_after_symbolic_f:nw { \exp_after:wN \exp_stop_f: }
29546     \s__fp_symbolic \__fp_symbolic_chk:w
29547     \__fp_types_binary:Nww #1 , { #2; #3; } ;
29548   }

```

(End of definition for `__fp_symbolic_binary_o:Nww`.)

^^A Hack! ^^A Hack! ^^A Hack!

```
\__fp_symbolic+_symbolic_o:ww
\__fp_symbolic+_o:ww      29549 \cs_set_protected:Npn \__fp_tmp:w #1#2
\__fp+_symbolic_o:ww     29550 {
\__fp_symbolic-_symbolic_o:ww 29551   \cs_new:cpn
\__fp_symbolic-_o:ww     29552     { __fp_symbolic_#2_symbolic_o:ww }
\__fp-_symbolic_o:ww    29553     { \__fp_symbolic_binary_o:Nww #1 }
\__fp_symbolic*_symbolic_o:ww 29554   \cs_new_eq:cc
\__fp_symbolic*_o:ww    29555     { __fp_symbolic_#2_o:ww }
\__fp*_symbolic_o:ww    29556     { __fp_symbolic_#2_symbolic_o:ww }
\__fp_symbolic/_symbolic_o:ww 29557   \cs_new_eq:cc
\__fp_symbolic/_o:ww    29558     { __fp_o:ww }
\__fp/_symbolic_o:ww    29559     { __fp_symbolic_#2_symbolic_o:ww }
\__fp/_o:ww             29560 }
\__fp_symbolic^symbolic_o:ww 29561 \tl_map_inline:nn { + - * / ^ & | }
\__fp_symbolic^o:ww     29562   { \exp_args:Nc \__fp_tmp:w { __fp_#1_o:ww } {#1} }
\__fp^symbolic_o:ww
```

(End of definition for `__fp_symbolic+_symbolic_o:ww` and others.)

81.5 Applying prefix functions to expressions

Used when applying infix operators to expressions.

```
\__fp_symbolic_unary_o:NNw
\__fp_&_symbolic_o:ww    29563 \cs_new:Npn \__fp_symbolic_unary_o:NNw #1#2#3; @
                          29564 {
                          29565   \__fp_exp_after_symbolic_f:nw { \exp_after:wN \exp_stop_f: }
                          29566   \s__fp_symbolic \__fp_symbolic_chk:w
                          29567   \__fp_types_unary:NNw #1#2 , { #3; } ;
                          29568 }
```

(End of definition for `__fp_symbolic_unary_o:NNw`.)

```
\__fp_symbolic_acos_o:w
\__fp_symbolic_acsc_o:w  29569 \tl_map_inline:nn
\__fp_symbolic_asec_o:w  29570 {
\__fp_symbolic_asin_o:w  29571   {acos} {acsc} {asec} {asin} {cos} {cot} {csc} {exp} {ln}
\__fp_symbolic_cos_o:w   29572   {not} {sec} {set_sign} {sin} {sqrt} {tan}
\__fp_symbolic_cot_o:w   29573 }
\__fp_symbolic_csc_o:w   29574 {
\__fp_symbolic_exp_o:w   29575   \cs_new:cpe { __fp_symbolic_#1_o:w }
\__fp_symbolic_ln_o:w    29576   {
\__fp_symbolic_not_o:w   29577     \exp_not:N \__fp_symbolic_unary_o:NNw
\__fp_symbolic_sec_o:w   29578     \exp_not:c { __fp_#1_o:w }
\__fp_symbolic_set_sign_o:w 29579   }
\__fp_symbolic_sin_o:w   29580 }
\__fp_symbolic_tan_o:w
```

(End of definition for `__fp_symbolic_acos_o:w` and others.)

81.6 Conversions

`_fp_symbolic_to_decimal:w` Symbolic expressions cannot be converted to decimal, integer, or scientific notation unless they can be reduced to

```

\__fp_symbolic_to_int:w
  \__fp_symbolic_to_scientific:w
\__fp_symbolic_convert:wnnN
29581 \cs_set_protected:Npn \__fp_tmp:w #1#2#3
29582 {
29583   \cs_new:cpn { \__fp_symbolic_to_#1:w }
29584   {
29585     \exp_after:wN \__fp_symbolic_convert:wnnN
29586     \exp:w \exp_end_continue_f:w
29587     \__fp_exp_after_symbolic_f:nw { { #2 } { fp_to_#1 } #3 }
29588   }
29589 }
29590 \__fp_tmp:w { decimal } { 0 } \__fp_to_decimal_dispatch:w
29591 \__fp_tmp:w { int } { 0 } \__fp_to_int_dispatch:w
29592 \__fp_tmp:w { scientific } { nan } \__fp_to_scientific_dispatch:w
29593 \cs_new:Npn \__fp_symbolic_convert:wnnN #1#2; #3#4#5
29594 {
29595   \str_if_eq:nnTF {#1} { \s__fp_symbolic }
29596   { \__fp_invalid_operation:nw {#3} {#4} #1#2; }
29597   { #5 #1#2; }
29598 }

```

(End of definition for `_fp_symbolic_to_decimal:w` and others.)

```

\__fp_symbolic_cs_arg_to_fn:NN
\__fp_symbolic_op_arg_to_fn:nN
29599 \cs_new:Npn \__fp_symbolic_cs_arg_to_fn:NN #1
29600 {
29601   \exp_args:Nf \__fp_symbolic_op_arg_to_fn:nN
29602   { \__fp_types_cs_to_op:N #1 }
29603 }
29604 \cs_new:Npn \__fp_symbolic_op_arg_to_fn:nN #1#2
29605 {
29606   \str_case:nnF { #1 #2 }
29607   {
29608     { not ? } { ! }
29609     { set_sign 0 } { abs }
29610     { set_sign 2 } { - }
29611   }
29612   {
29613     \token_if_eq_meaning:NNTF #2 \use_ii:nn
29614     { #1 d } {#1}
29615   }
29616 }

```

(End of definition for `_fp_symbolic_cs_arg_to_fn:NN` and `_fp_symbolic_op_arg_to_fn:nN`.)

`_fp_symbolic_to_tl:w` Converting a symbolic expression to a token list is possible.

```

\__fp_symbolic_unary_to_tl:NNw
\__fp_symbolic_binary_to_tl:Nww
\__fp_symbolic_function_to_tl:Nw
29617 \cs_new:Npn \__fp_symbolic_to_tl:w
29618   \s__fp_symbolic \__fp_symbolic_chk:w #1#2, #3#4;
29619   {
29620     \str_case:nnTF {#1}
29621     {
29622       { \__fp_types_unary:NNw } { \__fp_symbolic_unary_to_tl:NNw }

```

```

29623     { \_fp_types_binary:Nww } { \_fp_symbolic_binary_to_tl:Nww }
29624     { \_fp_function_o:w } { \_fp_symbolic_function_to_tl:Nw }
29625   }
29626   { #2, #3 @ }
29627   { \tl_to_str:n {#2} }
29628 }
29629 \cs_new:Npn \_fp_symbolic_unary_to_tl:NNw #1#2 , #3 @
29630 {
29631   \use:e
29632   {
29633     \_fp_symbolic_cs_arg_to_fn:NN #1#2
29634     ( \_fp_to_tl_dispatch:w #3 )
29635   }
29636 }
29637 \cs_new:Npn \_fp_symbolic_binary_to_tl:Nww #1, #2; #3; @
29638 {
29639   \use:e
29640   {
29641     ( \_fp_to_tl_dispatch:w #2; )
29642     \_fp_types_cs_to_op:N #1
29643     ( \_fp_to_tl_dispatch:w #3; )
29644   }
29645 }
29646 \cs_new:Npn \_fp_symbolic_function_to_tl:Nw #1, #2@
29647 {
29648   \use:e
29649   {
29650     \_fp_types_cs_to_op:N #1
29651     ( \_fp_array_to_clist:n {#2} )
29652   }
29653 }

```

(End of definition for _fp_symbolic_to_tl:w and others.)

81.7 Identifiers

Functions defined here are not necessarily tied to symbolic expressions.

_fp_id_if_invalid:nTF
_fp_id_if_invalid_aux:N

If #1 contains a space, it is not a valid identifier. Otherwise, loop through letters in #1: if it is not a letter, break the loop and return true. If the end of the loop is reached without finding any non-letter, return false. Note #1 must be a str (i.e., resulted from \tl_to_str:n).

```

29654 \prg_new_protected_conditional:Npnn
29655   \_fp_id_if_invalid:n #1 { T , F , TF }
29656 {
29657   \tl_if_empty:nTF {#1}
29658   { \prg_return_true: }
29659   {
29660     \tl_if_in:nnTF { #1 } { ~ }
29661     { \prg_return_true: }
29662     {
29663       \_fp_id_if_invalid_aux:N #1
29664       { ? \prg_break:n \prg_return_false: }

```

```

29665         \prg_break_point:
29666     }
29667 }
29668 }
29669 \cs_new:Npn \__fp_id_if_invalid_aux:N #1
29670 {
29671     \use_none:n #1
29672     \int_compare:nF { 'a <= '#1 <= 'z }
29673     {
29674         \int_compare:nF { 'A <= '#1 <= 'Z }
29675         { \prg_break:n \prg_return_true: }
29676     }
29677     \__fp_id_if_invalid_aux:N
29678 }

```

(End of definition for __fp_id_if_invalid:nTF and __fp_id_if_invalid_aux:N.)

81.8 Declaring variables and assigning values

__fp_variable_o:w We do not use \exp_last_unbraced:Nv to extract the value of \l__fp_variable_#1_fp because in \fp_set_variable:nn we define this fp variable to be something which f-expands to an actual floating point, rather than a genuine floating point.

```

29679 \cs_new:Npn \__fp_variable_o:w #1 @ #2
29680 {
29681     \fp_if_exist:cTF { l__fp_variable_#1_fp }
29682     {
29683         \exp_last_unbraced:Nf \__fp_exp_after_array_f:w
29684         { \use:c { l__fp_variable_#1_fp } } \s__fp_expr_stop
29685         \exp_after:wN \exp_stop_f: #2
29686     }
29687     {
29688         \token_if_eq_meaning:NNTF #2 \prg_do_nothing:
29689         {
29690             \s__fp_symbolic \__fp_symbolic_chk:w
29691             \__fp_variable_o:w #1 , { } ;
29692         }
29693         {
29694             \exp_after:wN \s__fp_symbolic
29695             \exp_after:wN \__fp_symbolic_chk:w
29696             \exp_after:wN \__fp_variable_o:w
29697             \exp:w
29698             \__fp_exp_after_symbolic_loop:N #1
29699             { , \exp:w \use_none:nn }
29700             \exp_after:wN \exp_end:
29701             \exp_after:wN { \exp_after:wN } \exp_after:wN ;
29702             #2
29703         }
29704     }
29705 }

```

(End of definition for __fp_variable_o:w.)

```

\__fp_variable_set_parsing:Nn
\__fp_variable_set_parsing_aux:NNn

```

```

29706 \cs_new_protected:Npn \__fp_variable_set_parsing:Nn #1#2
29707 {
29708   \cs_set:Npn \__fp_tmp:w
29709   {
29710     \__fp_exp_after_symbolic_f:nw { \__fp_parse_infix:NN }
29711     \s__fp_symbolic \__fp_symbolic_chk:w
29712     \__fp_variable_o:w #2 , { } ;
29713   }
29714   \exp_args:NNc \__fp_variable_set_parsing_aux:NNn #1
29715   { \__fp_parse_word_#2:N } {#2}
29716 }
29717 \cs_new_protected:Npn \__fp_variable_set_parsing_aux:NNn #1#2#3
29718 {
29719   \cs_if_eq:NNF #2 \__fp_tmp:w
29720   {
29721     \cs_if_exist:NTF #2
29722     {
29723       \msg_warning:nnnn
29724       { fp } { id-used-elsewhere } {#3} { variable }
29725       #1 #2 \__fp_tmp:w
29726     }
29727     {
29728       \cs_new_eq:NN #2 \scan_stop: % to declare the function
29729       #1 #2 \__fp_tmp:w
29730     }
29731   }
29732 }

```

(End of definition for `__fp_variable_set_parsing:Nn` and `__fp_variable_set_parsing_aux:NNn`.)

`\fp_clear_variable:n` We need local undefining, so have to do it low-level. `__fp_clear_variable_aux:n` is needed by `__fp_set_function:Nnnn` to skip `__fp_id_if_invalid:nTF`.

```

\__fp_clear_variable:n
\__fp_clear_variable_aux:n
29733 \cs_new_protected:Npn \fp_clear_variable:n #1
29734 {
29735   \exp_args:No \__fp_clear_variable:n { \tl_to_str:n {#1} }
29736 }
29737 \cs_new_protected:Npn \__fp_clear_variable:n #1
29738 {
29739   \__fp_id_if_invalid:nTF {#1}
29740   { \msg_error:nnn { fp } { id-invalid } {#1} }
29741   { \__fp_clear_variable_aux:n {#1} }
29742 }
29743 \cs_new_protected:Npn \__fp_clear_variable_aux:n #1
29744 {
29745   \cs_set_eq:cN { l__fp_variable_#1_fp } \tex_undefined:D
29746   \__fp_variable_set_parsing:Nn \cs_set_eq:NN {#1}
29747 }

```

(End of definition for `\fp_clear_variable:n`, `__fp_clear_variable:n`, and `__fp_clear_variable_aux:n`. This function is documented on page 268.)

`\fp_new_variable:n` Check that #1 is a valid identifier. If the identifier is already in use, complain. Then set `__fp_parse_word_#1:N` to use `__fp_variable_o:w`.

```

29748 \cs_new_protected:Npn \fp_new_variable:n #1

```

```

29749 {
29750   \exp_args:No \__fp_new_variable:n { \tl_to_str:n {#1} }
29751 }
29752 \cs_new_protected:Npn \__fp_new_variable:n #1
29753 {
29754   \__fp_id_if_invalid:nTF {#1}
29755   { \msg_error:nnn { fp } { id-invalid } {#1} }
29756   {
29757     \cs_if_exist:cT { __fp_parse_word_#1:N }
29758     {
29759       \msg_error:nnn
29760       { fp } { id-already-defined } {#1}
29761       \cs_undefine:c { __fp_parse_word_#1:N }
29762       \cs_set_eq:cN { l__fp_variable_#1_fp } \tex_undefined:D
29763     }
29764     \__fp_variable_set_parsing:Nn \cs_gset_eq:NN {#1}
29765   }
29766 }

```

(End of definition for `\fp_new_variable:n` and `__fp_new_variable:n`. This function is documented on page 267.)

`\l__fp_symbolic_flag` Refuse invalid identifiers. If the variable does not exist yet, define it just as in `\fp_new_variable:n` (but without unnecessary checks). Then evaluate #2. If the result contains the identifier #1, we would later get a loop in cases such as

```

\fp_set_variable:nn {A} {A}
\fp_show:n {A}

```

To detect this, define `\l__fp_variable_#1_fp` to raise an internal flag and evaluate to `nan`. Then re-evaluate `\l__fp_symbolic_fp`, and store the result in #1. If the flag is raised, #1 was present in `\l__fp_symbolic_fp`. In all cases, the #1-free result ends up in `\l__fp_variable_#1_fp`.

```

29767 \flag_new:N \l__fp_symbolic_flag
29768 \cs_new_protected:Npn \fp_set_variable:nn #1
29769 {
29770   \exp_args:No \__fp_set_variable:nn { \tl_to_str:n {#1} }
29771 }
29772 \cs_new_protected:Npn \__fp_set_variable:nn #1#2
29773 {
29774   \__fp_id_if_invalid:nTF {#1}
29775   { \msg_error:nnn { fp } { id-invalid } {#1} }
29776   {
29777     \__fp_variable_set_parsing:Nn \cs_set_eq:NN {#1}
29778     \fp_set:Nn \l__fp_symbolic_fp {#2}
29779     \cs_set_nopar:cpn { l__fp_variable_#1_fp }
29780     { \flag_ensure_raised:N \l__fp_symbolic_flag \c_nan_fp }
29781     \flag_clear:N \l__fp_symbolic_flag
29782     \fp_set:cn { l__fp_variable_#1_fp } { \l__fp_symbolic_fp }
29783     \flag_if_raised:NT \l__fp_symbolic_flag
29784     {
29785       \msg_error:nneee { fp } { id-loop }
29786       { #1 }
29787       { \tl_to_str:n {#2} }

```



```

29788         { \fp_to_tl:N \l__fp_symbolic_fp }
29789     }
29790 }
29791 }

```

(End of definition for `\l__fp_symbolic_flag`, `\fp_set_variable:nn`, and `__fp_set_variable:nn`. This variable is documented on page 267.)

81.9 Messages

```

29792 \msg_new:nnnn { fp } { id-invalid }
29793 { Floating-point-identifier~'#1'~invalid. }
29794 {
29795     LaTeX-has-been-asked-to-create-a-new-floating-point-identifier~'#1'~
29796     but-this-may-only-contain-ASCII-letters.
29797 }
29798 \msg_new:nnnn { fp } { id-already-defined }
29799 { Floating-point-identifier~'#1'~already-defined. }
29800 {
29801     LaTeX-has-been-asked-to-create-a-new-floating-point-identifier~'#1'~
29802     but-this-name-has-already-been-used-elsewhere.
29803 }
29804 \msg_new:nnnn { fp } { id-used-elsewhere }
29805 { Floating-point-identifier~'#1'~already-used-for-something-else. }
29806 {
29807     LaTeX-has-been-asked-to-create-a-new-floating-point-identifier~'#1'~
29808     but-this-name-is-used,-and-is-not-a-user-defined~#2.
29809 }
29810 \msg_new:nnnn { fp } { id-loop }
29811 { Variable~'#1'~used-in-the-definition-of~'#1'. }
29812 {
29813     LaTeX-has-been-asked-to-set-the-floating-point-identifier~'#1'~
29814     to-the-expression~'#2'.~Evaluating-this-expression-yields~'#3',~
29815     which-contains~'#1'~itself.
29816 }

```

81.10 Road-map

The following functions are not implemented: `min`, `max`, `?:`, comparisons, `round`, `atan`, `acot`.

```

29817 </package>

```

Chapter 82

13fp-functions implementation

```
29818 (*package)
29819 <@@=fp>
```

82.1 Declaring functions

```
\fp_new_function:n
\__fp_new_function:n
29820 \cs_new_protected:Npn \fp_new_function:n #1
29821   { \exp_args:No \__fp_new_function:n { \tl_to_str:n {#1} } }
29822 \cs_new_protected:Npn \__fp_new_function:n #1
29823   {
29824     \__fp_id_if_invalid:nTF {#1}
29825     { \msg_error:nnn { fp } { id-invalid } {#1} }
29826     {
29827       \cs_if_exist:cT { __fp_parse_word_#1:N }
29828       {
29829         \msg_error:nnn
29830         { fp } { id-already-defined } {#1}
29831         \cs_undefine:c { __fp_parse_word_#1:N }
29832         \cs_undefine:c { __fp_#1_o:w }
29833       }
29834       \__fp_function_set_parsing:Nn \cs_gset_eq:NN {#1}
29835     }
29836   }
```

(End of definition for \fp_new_function:n and __fp_new_function:n. This function is documented on page 268.)

```
\__fp_function_set_parsing:Nn
\__fp_function_set_parsing_aux:NNn
29837 \cs_new_protected:Npn \__fp_function_set_parsing:Nn #1#2
29838   {
29839     \exp_args:NNc \__fp_function_set_parsing_aux:NNn #1
29840     { __fp_parse_word_#2:N } {#2}
29841   }
29842 \cs_new_protected:Npn \__fp_function_set_parsing_aux:NNn #1#2#3
29843   {
29844     \cs_set:Npe \__fp_tmp:w
29845     {
29846       \exp_not:N \__fp_parse_function:NNN
```

```

29847     \exp_not:N \__fp_function_o:w
29848     \exp_not:c { __fp_#3_o:w }
29849   }
29850 \cs_if_eq:NNF #2 \__fp_tmp:w
29851 {
29852   \cs_if_exist:NTF #2
29853   {
29854     \msg_warning:nnnn
29855       { fp } { id-used-elsewhere } {#3} { function }
29856       #1 #2 \__fp_tmp:w
29857   }
29858   {
29859     \cs_new_eq:NN #2 \scan_stop: % to declare the function
29860     #1 #2 \__fp_tmp:w
29861   }
29862 }
29863 }

```

(End of definition for `__fp_function_set_parsing:Nn` and `__fp_function_set_parsing_aux:NNn`.)

`__fp_function_o:w`

```

29864 \cs_new:Npn \__fp_function_o:w #1#2 @
29865 {
29866   \cs_if_exist:NTF #1
29867     { #1 #2 @ }
29868     {
29869       \exp_after:wN \s__fp_symbolic
29870       \exp_after:wN \__fp_symbolic_chk:w
29871       \exp_after:wN \__fp_function_o:w
29872       \exp_after:wN #1
29873       \exp_after:wN ,
29874       \exp_after:wN {
29875         \exp:w \exp_end_continue_f:w
29876         \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
29877         \exp_after:wN
29878       }
29879       \exp_after:wN ;
29880     }
29881 }

```

(End of definition for `__fp_function_o:w`.)

82.2 Defining functions by their expression

`\l__fp_function_arg_int` Labels the arguments of a function being defined.

```
29882 \int_new:N \l__fp_function_arg_int
```

(End of definition for `\l__fp_function_arg_int`.)

`\fp_set_function:nnn`
`__fp_set_function:Nnnn`

```

\fp_set_function:nnn {<identifier>}
{<comma-list of variables>} {<expression>}

```

Defines the `<identifier>` to stand for a function which expects some arguments defined by the `<comma-list of variables>`, and evaluates to the `<expression>`.

```

29883 \cs_new_protected:Npn \fp_set_function:nnn #1
29884 {
29885   \exp_args:NNo \__fp_set_function:Nnnn \cs_set_eq:cN
29886   { \tl_to_str:n {#1} }
29887 }
29888 \cs_new_protected:Npn \__fp_set_function:Nnnn #1#2#3#4
29889 {
29890   \__fp_id_if_invalid:nTF {#2}
29891   { \msg_error:nnn { fp } { id-invalid } {#2} }
29892   {
29893     \cs_if_exist:cF { __fp_parse_word_#2:N }
29894     { \__fp_function_set_parsing:Nn \cs_set_eq:NN {#2} }
29895     \group_begin:
29896     \int_zero:N \l__fp_function_arg_int
29897     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#3} }
29898     {
29899       \int_incr:N \l__fp_function_arg_int
29900       \exp_args:Ne \__fp_clear_variable_aux:n
29901       {
29902         \c_underscore_str \tex_romannumeral:D \l__fp_function_arg_int
29903       }
29904       \fp_clear_variable:n {##1}
29905       \cs_set_nopar:cpe { l__fp_variable_##1_fp }
29906       {
29907         \exp_not:N \s__fp_symbolic
29908         \exp_not:N \__fp_symbolic_chk:w
29909         \exp_not:N \__fp_function_arg_o:w
29910         \int_use:N \l__fp_function_arg_int
29911         #####1 , { } ;
29912       }
29913     }
29914     \cs_set:Npn \__fp_function_arg_o:w ##1 @
29915     {
29916       \exp_after:wN \s__fp_symbolic
29917       \exp_after:wN \__fp_symbolic_chk:w
29918       \exp_after:wN \__fp_function_arg_o:w
29919       \tex_romannumeral:D
29920       \__fp_exp_after_symbolic_loop:N ##1
29921       { , \tex_romannumeral:D \use_none:nn }
29922       \exp_after:wN \c_zero_int
29923       \exp_after:wN { \exp_after:wN } \exp_after:wN ;
29924     }
29925     \fp_set:Nn \l__fp_symbolic_fp {#4}
29926     \use:e
29927     {
29928       \exp_not:n { \cs_gset:Npn \__fp_tmp:w ##1 }
29929       { \exp_not:o { \l__fp_symbolic_fp } }
29930     }
29931     \use:e
29932     {
29933       \exp_not:n { \cs_gset:Npn \__fp_tmp:w ##1 @ }
29934       {
29935         \exp_not:N \__fp_exp_after_symbolic_f:nw
29936         \exp_not:n { { \exp_after:wN \exp_stop_f: } }

```

```

29937         \exp_not:o { \__fp_tmp:w { . , {##1} } }
29938     }
29939 }
29940 \group_end:
29941 #1 { \__fp_#2_o:w } \__fp_tmp:w
29942 }
29943 }

```

```

\__fp_function_arg_o:w 29944 \cs_new:Npn \__fp_function_arg_o:w #1. #2
\__fp_function_arg_few:w 29945 {
\__fp_function_arg_get:w 29946   \if_meaning:w @ #2
29947     \exp_after:wN \__fp_function_arg_few:w
29948   \fi:
29949   \if_int_compare:w #1 = \c_one_int
29950     \exp_after:wN \__fp_function_arg_get:w
29951   \fi:
29952   \__fp_use_i_until_s:nw
29953   {
29954     \exp_after:wN \__fp_function_arg_o:w
29955     \int_value:w \int_eval:n { #1 - 1 } .
29956   }
29957   #2
29958 }
29959 \cs_new:Npn \__fp_function_arg_few:w #1 @ { \exp_after:wN \c_nan_fp }
29960 \cs_new:Npn \__fp_function_arg_get:w #1#2#3; #4 @
29961 {
29962   \__fp_exp_after_array_f:w #3; \s__fp_expr_stop
29963   \exp_after:wN \exp_stop_f:
29964 }

```

(End of definition for `\fp_set_function:nnn` and others. This function is documented on page 268.)

`\fp_clear_function:n`

```

\__fp_clear_function:n 29965 \cs_new_protected:Npn \fp_clear_function:n #1
29966   { \exp_args:No \__fp_clear_function:n { \tl_to_str:n {#1} } }
29967 \cs_new_protected:Npn \__fp_clear_function:n #1
29968   {
29969     \__fp_id_if_invalid:nTF {#1}
29970     { \msg_error:nnn { fp } { id-invalid } {#1} }
29971     {
29972       \cs_set_eq:cN { \__fp_#1_o:w } \tex_undefine:D
29973       \__fp_function_set_parsing:Nn \cs_set_eq:NN {#1}
29974     }
29975   }

```

(End of definition for `\fp_clear_function:n` and `__fp_clear_function:n`. This function is documented on page 268.)

```
29976 </package>
```

Chapter 83

l3fparray implementation

```
29977 (*package)
```

```
29978 (@@=fp)
```

In analogy to l3intarray it would make sense to have <@@=fparray>, but we need direct access to __fp_parse:n from l3fp-parse, and a few other (less crucial) internals of the l3fp family.

83.1 Allocating arrays

There are somewhat more than $(2^{31} - 1)^2$ floating point numbers so we store each floating point number as three entries in integer arrays. To avoid having to multiply indices by three or to add 1 etc, a floating point array is just a token list consisting of three tokens: integer arrays of the same size.

`\g__fp_array_int` Used to generate unique names for the three integer arrays.

```
29979 \int_new:N \g__fp_array_int
```

(End of definition for \g__fp_array_int.)

`\l__fp_array_loop_int` Used to loop in __fp_array_gzero:N.

```
29980 \int_new:N \l__fp_array_loop_int
```

(End of definition for \l__fp_array_loop_int.)

`\fparray_new:Nn` Build a three-token token list, then define all three tokens to be integer arrays of the same size. No need to initialize the data: the integer arrays start with zeros, and three zeros denote precisely `\c_zero_fp`, as we want.

`\fparray_new:cn`
`__fp_array_new:nNNN`

```
29981 \cs_new_protected:Npn \fparray_new:Nn #1#2
```

```
29982 {
```

```
29983   \tl_new:N #1
```

```
29984   \prg_replicate:nn { 3 }
```

```
29985   {
```

```
29986     \int_gincr:N \g__fp_array_int
```

```
29987     \exp_args:NNc \tl_gput_right:Nn #1
```

```
29988     { g__fp_array_ \__fp_int_to_roman:w \g__fp_array_int _intarray }
```

```
29989   }
```

```
29990 \exp_last_unbraced:Nfo \__fp_array_new:nNNNN
```

```
29991 { \int_eval:n {#2} } #1 #1
```

```

29992 }
29993 \cs_generate_variant:Nn \fpararray_new:Nn { c }
29994 \cs_new_protected:Npn \__fp_array_new:nNNNN #1#2#3#4#5
29995 {
29996   \int_compare:nNnTF {#1} < 0
29997   {
29998     \msg_error:nnn { kernel } { negative-array-size } {#1}
29999     \cs_undefine:N #1
30000     \int_gsub:Nn \g__fp_array_int { 3 }
30001   }
30002   {
30003     \intarray_new:Nn #2 {#1}
30004     \intarray_new:Nn #3 {#1}
30005     \intarray_new:Nn #4 {#1}
30006   }
30007 }

```

(End of definition for `\fpararray_new:Nn` and `__fp_array_new:nNNNN`. This function is documented on page 283.)

`\fpararray_count:N` Size of any of the intarrays, here we pick the third.

```

\fparray_count:c 30008 \cs_new:Npn \fpararray_count:N #1
30009 {
30010   \exp_after:wN \use_i:nnn
30011   \exp_after:wN \intarray_count:N #1
30012 }
30013 \cs_generate_variant:Nn \fpararray_count:N { c }

```

(End of definition for `\fpararray_count:N`. This function is documented on page 284.)

83.2 Array items

`__fp_array_bounds:NNnTF` See the `\intarray` analogue: only names change. The functions `\fpararray_gset:Nnn` and `__fp_array_bounds_error:NNn` `\fpararray_item:Nn` share bounds checking. The T branch is used if #3 is within bounds of the array #2.

```

30014 \cs_new:Npn \__fp_array_bounds:NNnTF #1#2#3#4#5
30015 {
30016   \if_int_compare:w 1 > #3 \exp_stop_f:
30017     \__fp_array_bounds_error:NNn #1 #2 {#3}
30018     #5
30019   \else:
30020     \if_int_compare:w #3 > \fpararray_count:N #2 \exp_stop_f:
30021     \__fp_array_bounds_error:NNn #1 #2 {#3}
30022     #5
30023   \else:
30024     #4
30025   \fi:
30026 \fi:
30027 }
30028 \cs_new:Npn \__fp_array_bounds_error:NNn #1#2#3
30029 {
30030   #1 { kernel } { out-of-bounds }
30031   { \token_to_str:N #2 } {#3} { \fpararray_count:N #2 }
30032 }

```

(End of definition for `__fp_array_bounds:NNnTF` and `__fp_array_bounds_error:NNn`.)

`\fparray_gset:Nnn` Evaluate, then store exponent in one intarray, sign and 8 digits of mantissa in the next, and 8 trailing digits in the last.

`\fparray_gset:cnm`

```

__fp_array_gset:NNNNnw 30033 \cs_new_protected:Npn \fparray_gset:Nnn #1#2#3
  __fp_array_gset:w 30034 {
__fp_array_gset_recover:Nw 30035 \exp_after:wN \exp_after:wN
  __fp_array_gset_special:nnNNN 30036 \exp_after:wN __fp_array_gset:NNNNnw
__fp_array_gset_normal:w 30037 \exp_after:wN #1
  30038 \exp_after:wN #1
  30039 \int_value:w \int_eval:n {#2} \exp_after:wN ;
  30040 \exp:w \exp_end_continue_f:w __fp_parse:n {#3}
  30041 }
  30042 \cs_generate_variant:Nn \fparray_gset:Nnn { c }
  30043 \cs_new_protected:Npn __fp_array_gset:NNNNnw #1#2#3#4#5 ; #6 ;
  30044 {
  30045 __fp_array_bounds:NNnTF \msg_error:nneee #4 {#5}
  30046 {
  30047 \exp_after:wN __fp_change_func_type:NNN
  30048 __fp_use_i_until_s:nw #6 ;
  30049 __fp_array_gset:w
  30050 __fp_array_gset_recover:Nw
  30051 #6 ; {#5} #1 #2 #3
  30052 }
  30053 { }
  30054 }
  30055 \cs_new_protected:Npn __fp_array_gset_recover:Nw #1#2 ;
  30056 {
  30057 __fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
  30058 \exp_after:wN #1 \c_nan_fp
  30059 }
  30060 \cs_new_protected:Npn __fp_array_gset:w \s__fp __fp_chk:w #1#2
  30061 {
  30062 \if_case:w #1 \exp_stop_f:
  30063 __fp_case_return:nw { __fp_array_gset_special:nnNNN {#2} }
  30064 \or: \exp_after:wN __fp_array_gset_normal:w
  30065 \or: __fp_case_return:nw { __fp_array_gset_special:nnNNN { #2 3 } }
  30066 \or: __fp_case_return:nw { __fp_array_gset_special:nnNNN { 1 } }
  30067 \fi:
  30068 \s__fp __fp_chk:w #1 #2
  30069 }
  30070 \cs_new_protected:Npn __fp_array_gset_normal:w
  30071 \s__fp __fp_chk:w 1 #1 #2 #3#4#5 ; #6#7#8#9
  30072 {
  30073 __kernel_intarray_gset:Nnn #7 {#6} {#2}
  30074 __kernel_intarray_gset:Nnn #8 {#6}
  30075 { \if_meaning:w 2 #1 3 \else: 1 \fi: #3#4 }
  30076 __kernel_intarray_gset:Nnn #9 {#6} { 1 \use:nn #5 }
  30077 }
  30078 \cs_new_protected:Npn __fp_array_gset_special:nnNNN #1#2#3#4#5
  30079 {
  30080 __kernel_intarray_gset:Nnn #3 {#2} {#1}
  30081 __kernel_intarray_gset:Nnn #4 {#2} {0}
  30082 __kernel_intarray_gset:Nnn #5 {#2} {0}

```



```
30083 }
```

(End of definition for `\fpararray_gset:Nnn` and others. This function is documented on page 283.)

`\fpararray_gzero:N`

`\fpararray_gzero:c`

```
30084 \cs_new_protected:Npn \fpararray_gzero:N #1
30085 {
30086   \int_zero:N \l__fp_array_loop_int
30087   \prg_replicate:nn { \fpararray_count:N #1 }
30088   {
30089     \int_incr:N \l__fp_array_loop_int
30090     \exp_after:wN \__fp_array_gset_special:nnNNN
30091     \exp_after:wN 0
30092     \exp_after:wN \l__fp_array_loop_int
30093     #1
30094   }
30095 }
30096 \cs_generate_variant:Nn \fpararray_gzero:N { c }
```

(End of definition for `\fpararray_gzero:N`. This function is documented on page 283.)

`\fpararray_item:Nn`

`\fpararray_item:cn`

`\fpararray_item_to_tl:Nn`

`\fpararray_item_to_tl:cn`

`__fp_array_item:NwN`

`__fp_array_item:NNNnN`

`__fp_array_item:N`

`__fp_array_item:w`

`__fp_array_item_special:w`

`__fp_array_item_normal:w`

```
30097 \cs_new:Npn \fpararray_item:Nn #1#2
30098 {
30099   \exp_after:wN \__fp_array_item:NwN
30100   \exp_after:wN #1
30101   \int_value:w \int_eval:n {#2} ;
30102   \__fp_to_decimal:w
30103 }
30104 \cs_generate_variant:Nn \fpararray_item:Nn { c }
30105 \cs_new:Npn \fpararray_item_to_tl:Nn #1#2
30106 {
30107   \exp_after:wN \__fp_array_item:NwN
30108   \exp_after:wN #1
30109   \int_value:w \int_eval:n {#2} ;
30110   \__fp_to_tl:w
30111 }
30112 \cs_generate_variant:Nn \fpararray_item_to_tl:Nn { c }
30113 \cs_new:Npn \__fp_array_item:NwN #1#2 ; #3
30114 {
30115   \__fp_array_bounds:NNnTF \msg_expandable_error:nnfff #1 {#2}
30116   { \exp_after:wN \__fp_array_item:NNNnN #1 {#2} #3 }
30117   { \exp_after:wN #3 \c_nan_fp }
30118 }
30119 \cs_new:Npn \__fp_array_item:NNNnN #1#2#3#4
30120 {
30121   \exp_after:wN \__fp_array_item:N
30122   \int_value:w \__kernel_intarray_item:Nn #2 {#4} \exp_after:wN ;
30123   \int_value:w \__kernel_intarray_item:Nn #3 {#4} \exp_after:wN ;
30124   \int_value:w \__kernel_intarray_item:Nn #1 {#4} ;
30125 }
30126 \cs_new:Npn \__fp_array_item:N #1
30127 {
30128   \if_meaning:w 0 #1 \exp_after:wN \__fp_array_item_special:w \fi:
30129   \__fp_array_item:w #1
```

```

30130 }
30131 \cs_new:Npn \__fp_array_item:w #1 #2#3#4#5 #6 ; 1 #7 ;
30132 {
30133   \exp_after:wN \__fp_array_item_normal:w
30134   \int_value:w \if_meaning:w #1 1 0 \else: 2 \fi: \exp_stop_f:
30135   #7 ; {#2#3#4#5} {#6} ;
30136 }
30137 \cs_new:Npn \__fp_array_item_special:w #1 ; #2 ; #3 ; #4
30138 {
30139   \exp_after:wN #4
30140   \exp:w \exp_end_continue_f:w
30141   \if_case:w #3 \exp_stop_f:
30142     \exp_after:wN \c_zero_fp
30143   \or: \exp_after:wN \c_nan_fp
30144   \or: \exp_after:wN \c_minus_zero_fp
30145   \or: \exp_after:wN \c_inf_fp
30146   \else: \exp_after:wN \c_minus_inf_fp
30147   \fi:
30148 }
30149 \cs_new:Npn \__fp_array_item_normal:w #1 #2#3#4#5 #6 ; #7 ; #8 ; #9
30150 { #9 \s__fp \__fp_chk:w 1 #1 {#8} #7 {#2#3#4#5} {#6} ; }

```

(End of definition for `\farray_item:Nn` and others. These functions are documented on page 284.)

`\farray_if_exist_p:N`
`\farray_if_exist_p:c`
`\farray_if_exist:NTF`
`\farray_if_exist:cTF`

Copies of the `cs` functions defined in `l3basics`.

```

30151 \prg_new_eq_conditional:NNn \farray_if_exist:N \cs_if_exist:N
30152   { TF , T , F , p }
30153 \prg_new_eq_conditional:NNn \farray_if_exist:c \cs_if_exist:c
30154   { TF , T , F , p }

```

(End of definition for `\farray_if_exist:NTF`. This function is documented on page 284.)

```

30155 </package>

```

Chapter 84

l3bitset implementation

```
30156 (*package)
30157 (@@=bitset)

Transitional support.
30158 \cs_if_exist:NT \@expl@finalise@setup@@
30159 {
30160   \tl_gput_right:Nn \@expl@finalise@setup@@
30161   { \declare@file@substitution { l3bitset.sty } { null.tex } }
30162 }
```

A bitset is a string variable.

```
\bitset_new:N
\bitset_new:c
\bitset_new:Nn
\bitset_new:cn
30163 \cs_new_protected:Npn \bitset_new:N #1
30164 {
30165   \__kernel_chk_if_free_cs:N #1
30166   \cs_gset_eq:NN #1 \c_zero_str
30167   \prop_new:c { g__bitset_ \cs_to_str:N #1 _name_prop }
30168 }
30169 \cs_new_protected:Npn \bitset_new:Nn #1 #2
30170 {
30171   \__kernel_chk_if_free_cs:N #1
30172   \cs_gset_eq:NN #1 \c_zero_str
30173   \prop_new:c { g__bitset_ \cs_to_str:N #1 _name_prop }
30174   \prop_gset_from_keyval:cn
30175   { g__bitset_ \cs_to_str:N #1 _name_prop }
30176   {#2}
30177 }
30178 \cs_generate_variant:Nn \bitset_new:N { c }
30179 \cs_generate_variant:Nn \bitset_new:Nn { c }
```

(End of definition for \bitset_new:N and \bitset_new:Nn. These functions are documented on page 286.)

```
\bitset_addto_named_index:Nn
```

```
30180 \cs_new_protected:Npn \bitset_addto_named_index:Nn #1#2
30181 {
30182   \prop_gput_from_keyval:cn
30183   { g__bitset_ \cs_to_str:N #1 _name_prop } { #2 }
30184 }
```

(End of definition for `\bitset_addto_named_index:Nn`. This function is documented on page 286.)

`\bitset_if_exist_p:N` Existence tests.

```

\bitset_if_exist_p:c 30185 \prg_new_eq_conditional:NNn
\bitset_if_exist:NTF 30186 \bitset_if_exist:N \str_if_exist:N { p , T , F , TF }
\bitset_if_exist:cTF 30187 \prg_new_eq_conditional:NNn
30188 \bitset_if_exist:c \str_if_exist:c { p , T , F , TF }

```

(End of definition for `\bitset_if_exist:NTF`. This function is documented on page 287.)

`__bitset_set_true:Nn` The internal command uses only numbers (integer expressions) for the position. A bit is set by either extending the string or by splitting it and then inserting an 1. It is not checked if the value was already 1.

```

\__bitset_gset_true:Nn 30189 \cs_new_protected:Npn \__bitset_set_true:Nn #1#2
\__bitset_set_false:Nn 30190 { \__bitset_set:NNnN \str_set:Ne #1 {#2} 1 }
\__bitset_gset_false:Nn 30191 \cs_new_protected:Npn \__bitset_gset_true:Nn #1#2
\__bitset_set:NNnN 30192 { \__bitset_set:NNnN \str_gset:Ne #1 {#2} 1 }
30193 \cs_new_protected:Npn \__bitset_set_false:Nn #1#2
30194 { \__bitset_set:NNnN \str_set:Ne #1 {#2} 0 }
30195 \cs_new_protected:Npn \__bitset_gset_false:Nn #1#2
30196 { \__bitset_set:NNnN \str_gset:Ne #1 {#2} 0 }
30197 \cs_new_protected:Npn \__bitset_set:NNnN #1#2#3#4
30198 {
30199 \int_compare:nNnT {#3} > { 0 }
30200 {
30201 \int_compare:nNnTF { \str_count:N #2 } < {#3}
30202 {
30203 #1 #2
30204 {
30205 #4
30206 \prg_replicate:nn { #3 - \str_count:N #2 - 1 } { 0 }
30207 #2
30208 }
30209 }
30210 {
30211 #1 #2
30212 {
30213 \str_range:Nnn #2 { 1 } { -1 - (#3) }
30214 #4
30215 \str_range:Nnn #2 { 1 - (#3) } { -1 }
30216 }
30217 }
30218 }
30219 }

```

(End of definition for `__bitset_set_true:Nn` and others.)

`\l__bitset_internal_int`

```
30220 \int_new:N \l__bitset_internal_int
```

(End of definition for `\l__bitset_internal_int`.)

<https://chat.stackexchange.com/transcript/message/56878159#56878159>

```

\__bitset_test_digits:nTF
\__bitset_test_digits_end:n
\__bitset_test_digits:w
30221 \prg_new_protected_conditional:Npnn \__bitset_test_digits:n #1 { TF }
30222 {
30223   \tex_afterassignment:D \__bitset_test_digits:w
30224   \l__bitset_internal_int = 0 \tl_trim_spaces_apply:nN {#1} \tl_to_str:n
30225   \__bitset_test_digits_end:
30226   \use_i:nnn \if_false:
30227   \__bitset_test_digits_end:
30228   \if_int_compare:w \c_zero_int < \l__bitset_internal_int
30229   \prg_return_true:
30230   \else:
30231   \prg_return_false:
30232   \fi:
30233 }
30234 \cs_new_eq:NN \__bitset_test_digits_end: \exp_stop_f:
30235 \cs_new_protected:Npn \__bitset_test_digits:w #1 \__bitset_test_digits_end: { }

(End of definition for \__bitset_test_digits:nTF, \__bitset_test_digits_end:n, and \__bitset_
test_digits:w.)
```

```

\bitset_set_true:Nn
\bitset_set_true:cn
\bitset_gset_true:Nn
\bitset_gset_true:cn
\bitset_set_false:Nn
\bitset_set_false:cn
\bitset_gset_false:Nn
\bitset_gset_false:cn
\__bitset_set_aux:NNn
```

The user commands must first translate the argument to an index number.

```

30236 \cs_new_protected:Npn \bitset_set_true:Nn #1#2
30237 { \__bitset_set:NNn \__bitset_set_true:Nn #1 {#2} }
30238 \cs_new_protected:Npn \bitset_gset_true:Nn #1#2
30239 { \__bitset_set:NNn \__bitset_gset_true:Nn #1 {#2} }
30240 \cs_new_protected:Npn \bitset_set_false:Nn #1#2
30241 { \__bitset_set:NNn \__bitset_set_false:Nn #1 {#2} }
30242 \cs_new_protected:Npn \bitset_gset_false:Nn #1#2
30243 { \__bitset_set:NNn \__bitset_gset_false:Nn #1 {#2} }
30244 \cs_new_protected:Npn \__bitset_set:NNn #1#2#3
30245 {
30246   \prop_if_in:cnTF { g__bitset_ \cs_to_str:N #2 _name_prop } {#3}
30247   {
30248     #1 #2
30249     {
30250       \prop_item:cn { g__bitset_ \cs_to_str:N #2 _name_prop } {#3}
30251     }
30252   }
30253   {
30254     \__bitset_test_digits:nTF {#3}
30255     {
30256       #1 #2 {#3}
30257       \prop_gput:cn { g__bitset_ \cs_to_str:N #2 _name_prop } {#3} {#3}
30258     }
30259     {
30260       \msg_warning:nnee { bitset } { unknown-name }
30261       { \token_to_str:N #2 }
30262       { \tl_to_str:n {#3} }
30263     }
30264   }
30265 }
30266 \cs_generate_variant:Nn \bitset_set_true:Nn { c }
30267 \cs_generate_variant:Nn \bitset_gset_true:Nn { c }
30268 \cs_generate_variant:Nn \bitset_set_false:Nn { c }
30269 \cs_generate_variant:Nn \bitset_gset_false:Nn { c }
```

(End of definition for `\bitset_set_true:Nn` and others. These functions are documented on page 287.)

```

\bitset_clear:N
\bitset_clear:c 30270 \cs_new_protected:Npn \bitset_clear:N #1
\bitset_gclear:N 30271 {
\bitset_gclear:c 30272   \str_set_eq:NN #1 \c_zero_str
30273 }
30274 \cs_new_protected:Npn \bitset_gclear:N #1
30275 {
30276   \str_gset_eq:NN #1 \c_zero_str
30277 }
30278 \cs_generate_variant:Nn \bitset_clear:N { c }
30279 \cs_generate_variant:Nn \bitset_gclear:N { c }

```

(End of definition for `\bitset_clear:N` and `\bitset_gclear:N`. These functions are documented on page 287.)

```

\bitset_to_arabic:N The naming of the commands follow the names in the int module. \bitset_to_-
\bitset_to_arabic:c arabic:N uses \int_from_bin:n if the string is shorter than 32 and the slower \fp_-
\bitset_to_bin:N eval:n for larger bitsets.
\bitset_to_bin:c 30280 \cs_new:Npn \bitset_to_arabic:N #1
\__bitset_to_int:nN 30281 {
30282   \int_compare:nNnTF { \str_count:N #1 } < { 32 }
30283     { \exp_args:No \int_from_bin:n {#1} }
30284     {
30285       \exp_after:wN \__bitset_to_int:nN \exp_after:wN 0
30286       #1 \q_recursion_tail \q_recursion_stop
30287     }
30288   }
30289 \cs_new:Npn \__bitset_to_int:nN #1#2
30290 {
30291   \quark_if_recursion_tail_stop_do:Nn #2 {#1}
30292   \exp_args:Nf \__bitset_to_int:nN { \fp_eval:n { #1 * 2 + #2 } }
30293 }
30294 \cs_new:Npn \bitset_to_bin:N #1
30295 {
30296   #1
30297 }
30298 \cs_generate_variant:Nn \bitset_to_arabic:N { c }
30299 \cs_generate_variant:Nn \bitset_to_bin:N { c }

```

(End of definition for `\bitset_to_arabic:N`, `\bitset_to_bin:N`, and `__bitset_to_int:nN`. These functions are documented on page 288.)

`\bitset_item:Nn` All bits that have been set at anytime have an entry in the prop, so we can take everything else as 0.

```

\bitset_item:cN 30300 \cs_new:Npn \bitset_item:Nn #1#2
30301 {
30302   \prop_if_in:cnTF { g__bitset_ \cs_to_str:N #1 _name_prop } {#2}
30303   {
30304     \int_eval:n
30305     {
30306       \str_item:Nn #1
30307       { 0 - ( \prop_item:cn { g__bitset_ \cs_to_str:N #1 _name_prop } {#2} ) }

```

```

30308         +0
30309     }
30310 }
30311 {
30312     0
30313 }
30314 }
30315 \cs_generate_variant:Nn \bitset_item:Nn { c }

```

(End of definition for \bitset_item:Nn. This function is documented on page 287.)

```

\bitset_show:N
\bitset_show:c 30316 \cs_new_protected:Npn \bitset_show:N { \__bitset_show:NN \msg_show:nneeee }
\bitset_log:N  30317 \cs_generate_variant:Nn \bitset_show:N { c }
\bitset_log:c  30318 \cs_new_protected:Npn \bitset_log:N { \__bitset_show:NN \msg_log:nneeee }
30319 \cs_generate_variant:Nn \bitset_log:N { c }
30320 \cs_new_protected:Npn \__bitset_show:NN #1#2
30321 {
30322     \__kernel_chk_defined:NT #2
30323     {
30324         #1 { bitset } { show }
30325         { \token_to_str:N #2 }
30326         { \bitset_to_bin:N #2 }
30327         { \bitset_to_arabic:N #2 }
30328         { }
30329     }
30330 }

```

(End of definition for \bitset_show:N and \bitset_log:N. These functions are documented on page 288.)

```

\bitset_show_named_index:N
\bitset_show_named_index:c 30331 \cs_new_protected:Npn \bitset_show_named_index:N
\bitset_log_named_index:N  30332 { \__bitset_show_named_index:NN \msg_show:nneeee }
\bitset_log_named_index:c  30333 \cs_generate_variant:Nn \bitset_show_named_index:N { c }
30334 \cs_new_protected:Npn \bitset_log_named_index:N
30335 { \__bitset_show_named_index:NN \msg_log:nneeee }
30336 \cs_generate_variant:Nn \bitset_log_named_index:N { c }
30337 \cs_new_protected:Npn \__bitset_show_named_index:NN #1#2
30338 {
30339     \__kernel_chk_defined:NT #2
30340     {
30341         #1 { bitset } { show-names }
30342         { \token_to_str:N #2 }
30343         { \prop_map_function:cN { g__bitset_ \cs_to_str:N #2 _name_prop } \msg_show_item:
30344         { } { } }
30345     }
30346 }

```

(End of definition for \bitset_show_named_index:N and \bitset_log_named_index:N. These functions are documented on page 288.)

84.1 Messages

```
30347 \msg_new:nnn { bitset } { show }
30348   {
30349     The-bitset~#1~has~the~representation: \\
30350     >~binary:~#2 \\
30351     >~arabic:~#3 .
30352   }
30353 \msg_new:nnn { bitset } { show-names }
30354   {
30355     The-bitset~#1~
30356     \tl_if_empty:nTF {#2}
30357       { knows~no~names~yet \\>~ . }
30358       { knows~the~name/index~pairs~(without~outer~braces): #2 . }
30359   }
30360 \msg_new:nnn { bitset } { unknown-name }
30361   { The~name~'~#2'~is~unknown~for~bitset~\tl_to_str:n {#1} }
30362 \prop_gput:Nnn \g_msg_module_name_prop { bitset } { LaTeX }
30363 \prop_gput:Nnn \g_msg_module_type_prop { bitset } { }
30364 </package>
```


Chapter 85

l3cctab implementation

30365 `*package)`

30366 `\@@=cctab)`

As LuaTeX offers engine support for category code tables, and this is entirely lacking from the other engines, we need two complementary approaches. (Some future XeTeX may add support, at which point the conditionals below would be different.)

85.1 Variables

`\g__cctab_stack_seq` List of catcode tables saved by nested `\cctab_begin:N`, to restore catcodes at the matching `\cctab_end:.` When popped from the `\g__cctab_stack_seq` the table numbers are stored in `\g__cctab_unused_seq` for later reuse.

30367 `\seq_new:N \g__cctab_stack_seq`

30368 `\seq_new:N \g__cctab_unused_seq`

(End of definition for `\g__cctab_stack_seq` and `\g__cctab_unused_seq`.)

`\g__cctab_group_seq` A stack to store the group level when a catcode table started.

30369 `\seq_new:N \g__cctab_group_seq`

(End of definition for `\g__cctab_group_seq`.)

`\g__cctab_allocate_int` Integer to keep track of what category code table to allocate. In LuaTeX it is only used in format mode to implement `\cctab_new:N`. In other engines it is used to make csnames for dynamic tables.

30370 `\int_new:N \g__cctab_allocate_int`

(End of definition for `\g__cctab_allocate_int`.)

`\l__cctab_internal_a_tl` Scratch space. For instance, when popping `\g__cctab_stack_seq`/`\g__cctab_unused_seq`, consists of the catcodetable number (integer denotation) in LuaTeX, or of an intarray variable (as a single token) in other engines.

30371 `\tl_new:N \l__cctab_internal_a_tl`

30372 `\tl_new:N \l__cctab_internal_b_tl`

(End of definition for `\l__cctab_internal_a_tl` and `\l__cctab_internal_b_tl`.)

`\g__cctab_endlinechar_prop` In LuaTeX we store the `\endlinechar` associated to each `\catcodetable` in a property list, unless it is the default value 13.

```
30373 \prop_new:N \g__cctab_endlinechar_prop
```

(End of definition for `\g__cctab_endlinechar_prop`.)

85.2 Allocating category code tables

`\cctab_new:N` The `__cctab_new:N` auxiliary allocates a new catcode table but does not attempt to set its value consistently across engines. It is used both in `\cctab_new:N`, which sets catcodes to iniTeX values, and in `\cctab_begin:N/\cctab_end:` for dynamically allocated tables. First, the LuaTeX case. Creating a new category code table is done like other registers. In ConTeXt, `\newcatcodetable` does not include the initialisation, so that is added explicitly.

```
30374 \sys_if_engine luatex:TF
30375 {
30376   \cs_new_protected:Npn \cctab_new:N #1
30377   {
30378     \__kernel_chk_if_free_cs:N #1
30379     \__cctab_new:N #1
30380   }
30381   \cs_new_protected:Npn \__cctab_new:N #1
30382   {
30383     \newcatcodetable #1
30384     \tex_initcatcodetable:D #1
30385   }
30386 }
```

Now the case for other engines. Here, each table is an integer array. Following the LuaTeX pattern, a new table starts with iniTeX codes. The `\debug_suspend:` and `\debug_resume:` functions prevent errors and logging from debug commands which are either duplicate or false when `__cctab_new:N` is used by `\cctab_new:N` or `\cctab_const:Nn`. The index base is out-by-one, so we have an internal function to handle that. The iniTeX `\endlinechar` is 13.

```
30387 {
30388   \cs_new_protected:Npn \__cctab_new:N #1
30389   {
30390     \debug_suspend:
30391     \intarray_new:Nn #1 { 257 }
30392     \debug_resume:
30393   }
30394   \cs_new_protected:Npn \__cctab_gstore:Nnn #1#2#3
30395   { \intarray_gset:Nnn #1 { #2 + 1 } {#3} }
30396   \cs_new_protected:Npn \cctab_new:N #1
30397   {
30398     \__kernel_chk_if_free_cs:N #1
30399     \__cctab_new:N #1
30400     \int_step_inline:nn { 256 }
30401     { \__kernel_intarray_gset:Nnn #1 {##1} { 12 } }
30402     \__kernel_intarray_gset:Nnn #1 { 257 } { 13 }
30403     \__cctab_gstore:Nnn #1 { 0 } { 9 }
30404     \__cctab_gstore:Nnn #1 { 13 } { 5 }

```

```

30405     \__cctab_gstore:Nnn #1 { 32 } { 10 }
30406     \__cctab_gstore:Nnn #1 { 37 } { 14 }
30407     \int_step_inline:nnn { 65 } { 90 }
30408         { \__cctab_gstore:Nnn #1 {##1} { 11 } }
30409     \__cctab_gstore:Nnn #1 { 92 } { 0 }
30410     \int_step_inline:nnn { 97 } { 122 }
30411         { \__cctab_gstore:Nnn #1 {##1} { 11 } }
30412     \__cctab_gstore:Nnn #1 { 127 } { 15 }
30413 }
30414 }
30415 \cs_generate_variant:Nn \cctab_new:N { c }

```

(End of definition for `\cctab_new:N`, `__cctab_new:N`, and `__cctab_gstore:Nnn`. This function is documented on page 289.)

85.3 Saving category code tables

`__cctab_gset:n` In various functions we need to save the current catcodes (globally) in a table. In LuaTeX, saving the catcodes is a primitives, but the `\endlinechar` needs more work: to avoid filling `\g__cctab_endlinechar_prop` with many entries we special-case the default value 13. In other engines we store 256 current catcodes and the `\endlinechar` in an intarray variable.

```

30416 \sys_if_engine luatex:TF
30417 {
30418     \cs_new_protected:Npn \__cctab_gset:n #1
30419         { \exp_args:Nf \__cctab_gset_aux:n { \int_eval:n {#1} } }
30420     \cs_new_protected:Npn \__cctab_gset_aux:n #1
30421         {
30422             \tex_savecatcodetable:D #1 \scan_stop:
30423             \int_compare:nNnTF { \tex_endlinechar:D } = { 13 }
30424                 { \prop_gremove:Nn \g__cctab_endlinechar_prop {#1} }
30425                 {
30426                     \prop_gput:NnV \g__cctab_endlinechar_prop {#1}
30427                     \tex_endlinechar:D
30428                 }
30429         }
30430 }
30431 {
30432     \cs_new_protected:Npn \__cctab_gset:n #1
30433         {
30434             \int_step_inline:nn { 256 }
30435                 {
30436                     \__kernel_intarray_gset:Nnn #1 {##1}
30437                     { \char_value_catcode:n { ##1 - 1 } }
30438                 }
30439             \__kernel_intarray_gset:Nnn #1 { 257 }
30440             { \tex_endlinechar:D }
30441         }
30442 }

```

(End of definition for `__cctab_gset:n` and `__cctab_gset_aux:n`.)

`\cctab_gset:Nn` Category code tables are always global, so only one version of assignments is needed.
`\cctab_gset:cn` Simply run the setup in a group and save the result in a category code table #1, provided it is valid. The internal function is defined above depending on the engine.

```

30443 \cs_new_protected:Npn \cctab_gset:Nn #1#2
30444 {
30445   \__cctab_chk_if_valid:NT #1
30446   {
30447     \group_begin:
30448     \cctab_select:N \c_initex_cctab
30449     #2 \scan_stop:
30450     \__cctab_gset:n {#1}
30451     \group_end:
30452   }
30453 }
30454 \cs_generate_variant:Nn \cctab_gset:Nn { c }

```

(End of definition for `\cctab_gset:Nn`. This function is documented on page 289.)

`\cctab_gsave_current:N` Very simple.

```

\cctab_gsave_current:c
30455 \cs_new_protected:Npn \cctab_gsave_current:N #1
30456 {
30457   \__cctab_chk_if_valid:NT #1
30458   { \__cctab_gset:n {#1} }
30459 }
30460 \cs_generate_variant:Nn \cctab_gsave_current:N { c }

```

(End of definition for `\cctab_gsave_current:N`. This function is documented on page 289.)

85.4 Using category code tables

`\g__cctab_internal_cctab` In Lua_{TeX}, we must ensure that the saved tables are read-only. This is done by applying the saved table, then switching immediately to a scratch table. Any later catcode assignment will affect that scratch table rather than the saved one. If we simply switched to the saved tables, then `\char_set_catcode_other:N` in the example below would change `\c_document_cctab` and a later use of that table would give the wrong category code to

```

\use:n
{
  \cctab_begin:N \c_document_cctab
  \char_set_catcode_other:N \_
  \cctab_end:
  \cctab_begin:N \c_document_cctab
  \int_compare:nTF { \char_value_catcode:n { '_' } = 8 }
  { \TRUE } { \ERROR }
  \cctab_end:
}

```

We must also make sure that a scratch table is never reused in a nested group: in the following example, the scratch table used by the first `\cctab_begin:N` would be changed globally by the second one issuing `\savecatcodetable`, and after `\group_end:` the wrong

category codes (those of `\c_str_cctab`) would be imposed. Note that the inner `\cctab_end` restores the correct catcodes only locally, so the problem really comes up because of the different grouping level. The simplest is to use a scratch table labeled by the `\currentgrouplevel`. We initialize one of them as an example.

```

\use:n
{
  \cctab_begin:N \c_document_cctab
  \group_begin:
  \cctab_begin:N \c_str_cctab
  \cctab_end:
  \group_end:
  \cctab_end:
}

30461 \sys_if_engine_luatex:T
30462 {
30463   \__cctab_new:N \g__cctab_internal_cctab
30464   \cs_new:Npn \__cctab_internal_cctab_name:
30465     {
30466       g__cctab_internal
30467       \tex_romannumeral:D \tex_currentgrouplevel:D
30468       _cctab
30469     }
30470 }

```

(End of definition for `\g__cctab_internal_cctab` and `__cctab_internal_cctab_name:`.)

`\cctab_select:N` The public function simply checks the `\cctab var` exists before using the engine-dependent `__cctab_select:N`. Skipping these checks would result in low-level engine-dependent errors. First, the LuaTeX case. In other engines, selecting a catcode table is a matter of doing 256 catcode assignments and setting the `\endlinechar`.

```

30471 \cs_new_protected:Npn \cctab_select:N #1
30472   { \__cctab_chk_if_valid:NT #1 { \__cctab_select:N #1 } }
30473 \cs_generate_variant:Nn \cctab_select:N { c }
30474 \sys_if_engine_luatex:TF
30475 {
30476   \cs_new_protected:Npn \__cctab_select:N #1
30477     {
30478       \tex_catcodetable:D #1
30479       \prop_get:NVNTF \g__cctab_endlinechar_prop #1 \l__cctab_internal_a_tl
30480         { \int_set:Nn \tex_endlinechar:D { \l__cctab_internal_a_tl } }
30481         { \int_set:Nn \tex_endlinechar:D { 13 } }
30482       \cs_if_exist:cF { \__cctab_internal_cctab_name: }
30483         { \exp_args:Nc \__cctab_new:N { \__cctab_internal_cctab_name: } }
30484       \exp_args:Nc \tex_savecatcodetable:D { \__cctab_internal_cctab_name: }
30485       \exp_args:Nc \tex_catcodetable:D { \__cctab_internal_cctab_name: }
30486     }
30487 }
30488 {
30489   \cs_new_protected:Npn \__cctab_select:N #1
30490     {
30491       \int_step_inline:nm { 256 }
30492       {

```

```

30493         \char_set_catcode:nn { ##1 - 1 }
30494         { \__kernel_intarray_item:Nn #1 {##1} }
30495     }
30496     \int_set:Nn \tex_endlinechar:D
30497     { \__kernel_intarray_item:Nn #1 { 257 } }
30498 }
30499 }

```

(End of definition for `\cctab_select:N` and `__cctab_select:N`. This function is documented on page 290.)

`\g__cctab_next_cctab` For `\cctab_begin:N/\cctab_end:` we will need to allocate dynamic tables. This is done here by `__cctab_begin_aux:`, which puts a table number (in LuaTeX) or name (in other engines) into `\l__cctab_internal_a_tl`. In LuaTeX this simply calls `__cctab_new:N` and uses the resulting catcodetable number; in other engines we need to give a name to the intarray variable and use that. In LuaTeX, to restore catcodes at `\cctab_end:` we cannot just set `\catcodetable` to its value before `\cctab_begin:N`, because that table may have been altered by other code in the mean time. So we must make sure to save the catcodes in a table we control and restore them at `\cctab_end:`.

```

30500 \sys_if_engine luatex:TF
30501 {
30502     \cs_new_protected:Npn \__cctab_begin_aux:
30503     {
30504         \__cctab_new:N \g__cctab_next_cctab
30505         \tl_set:NW \l__cctab_internal_a_tl \g__cctab_next_cctab
30506         \cs_undefine:N \g__cctab_next_cctab
30507     }
30508 }
30509 {
30510     \cs_new_protected:Npn \__cctab_begin_aux:
30511     {
30512         \int_gincr:N \g__cctab_allocate_int
30513         \exp_args:Nc \__cctab_new:N
30514         { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
30515         \exp_args:NNc \tl_set:Nn \l__cctab_internal_a_tl
30516         { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
30517     }
30518 }

```

(End of definition for `\g__cctab_next_cctab` and `__cctab_begin_aux:`.)

`\cctab_begin:N` Check the `<cctab var>` exists, to avoid low-level errors. Get in `\l__cctab_internal_a_tl` the number/name of a dynamic table, either from `\g__cctab_unused_seq` where we save tables that are not currently in use, or from `__cctab_begin_aux:` if none are available. Then save the current catcodes into the table (pointed to by) `\l__cctab_internal_a_tl` and save that table number in a stack before selecting the desired catcodes.

```

30519 \cs_new_protected:Npn \cctab_begin:N #1
30520 {
30521     \__cctab_chk_if_valid:NT #1
30522     {
30523         \seq_gpop:NNF \g__cctab_unused_seq \l__cctab_internal_a_tl
30524         { \__cctab_begin_aux: }
30525         \__cctab_chk_group_begin:e

```

```

30526         { \__cctab_nesting_number:N \l__cctab_internal_a_tl }
30527         \seq_gpush:NV \g__cctab_stack_seq \l__cctab_internal_a_tl
30528         \exp_args:NV \__cctab_gset:n \l__cctab_internal_a_tl
30529         \__cctab_select:N #1
30530     }
30531 }
30532 \cs_generate_variant:Nn \cctab_begin:N { c }

```

(End of definition for `\cctab_begin:N`. This function is documented on page 290.)

\cctab_end: Make sure a `\cctab_begin:N` was used some time earlier, get in `\l__cctab_internal_a_tl` the catcode table number/name in which the prevailing catcodes were stored, then restore these catcodes. The dynamic table is now unused hence stored in `\g__cctab_unused_seq` for recycling by later `\cctab_begin:N`.

```

30533 \cs_new_protected:Npn \cctab_end:
30534 {
30535     \seq_gpop:NNTF \g__cctab_stack_seq \l__cctab_internal_a_tl
30536     {
30537         \seq_gpush:NV \g__cctab_unused_seq \l__cctab_internal_a_tl
30538         \exp_args:Ne \__cctab_chk_group_end:n
30539         { \__cctab_nesting_number:N \l__cctab_internal_a_tl }
30540         \__cctab_select:N \l__cctab_internal_a_tl
30541     }
30542     { \msg_error:nn { cctab } { extra-end } }
30543 }

```

(End of definition for `\cctab_end:`. This function is documented on page 290.)

`__cctab_chk_group_begin:n` `__cctab_chk_group_begin:e` `__cctab_chk_group_end:n` Catcode tables are not allowed to be intermixed with groups, so here we check that they are properly nested regarding TeX groups. `__cctab_chk_group_begin:n` stores the current group level in a stack, and locally defines a dummy control sequence `__cctab_group_⟨cctab-level⟩_chk:`.

`__cctab_chk_group_end:n` pops the stack, and compares the returned value with `\tex_currentgrouplevel:D`. If they differ, `\cctab_end:` is in a different grouping level than the matching `\cctab_begin:N`. If they are the same, both happened at the same level, however a group might have ended and another started between `\cctab_begin:N` and `\cctab_end:`:

```

\group_begin:
  \cctab_begin:N \c_document_cctab
\group_end:
\group_begin:
  \cctab_end:
\group_end:

```

In this case checking `\tex_currentgrouplevel:D` is not enough, so we locally define `__cctab_group_⟨cctab-level⟩_chk:`, and then check if it exist in `\cctab_end:`. If it doesn't, we know there was a group end where it shouldn't.

The `⟨cctab-level⟩` in the sentinel macro above cannot be replaced by the more convenient `\tex_currentgrouplevel:D` because with the latter we might be tricked. Suppose:

```

\group_begin:
  \cctab_begin:N \c_code_cctab % A
\group_end:
\group_begin:
  \cctab_begin:N \c_code_cctab % B
  \cctab_end: % C
  \cctab_end: % D
\group_end:

```

The line marked with A would start a `cctab` with a sentinel token named `__cctab_group_1_chk:`, which would disappear at the `\group_end:` that follows. But B would create the same sentinel token, since both are at the same group level. Line C would end the `cctab` from line B correctly, but so would line D because line B created the same sentinel token. Using `\cctab-level` works correctly because it signals that certain `cctab` level was activated somewhere, but if it doesn't exist when the `\cctab_end:` is reached, we had a problem.

Unfortunately these tests only flag the wrong usage at the `\cctab_end:`, which might be far from the `\cctab_begin:N`. However it isn't possible to signal the wrong usage at the `\group_end:` without using `\tex_aftergroup:D`, which is unsafe in certain types of groups.

The three cases checked here just raise an error, and no recovery is attempted: usually interleaving groups and catcode tables will work predictably.

```

30544 \cs_new_protected:Npn \__cctab_chk_group_begin:n #1
30545   {
30546     \seq_gpush:Ne \g__cctab_group_seq
30547     { \int_use:N \tex_currentgrouplevel:D }
30548     \cs_set_eq:cN { __cctab_group_ #1 _chk: } \prg_do_nothing:
30549   }
30550 \cs_generate_variant:Nn \__cctab_chk_group_begin:n { e }
30551 \cs_new_protected:Npn \__cctab_chk_group_end:n #1
30552   {
30553     \seq_gpop:NN \g__cctab_group_seq \l__cctab_internal_b_tl
30554     \bool_lazy_and:nnF
30555       {
30556         \int_compare_p:nNn
30557           { \tex_currentgrouplevel:D } = { \l__cctab_internal_b_tl }
30558       }
30559     { \cs_if_exist_p:c { __cctab_group_ #1 _chk: } }
30560     {
30561       \msg_error:nne { cctab } { group-mismatch }
30562     }
30563     \int_sign:n
30564       { \tex_currentgrouplevel:D - \l__cctab_internal_b_tl }
30565   }
30566 }
30567 \cs_undefine:c { __cctab_group_ #1 _chk: }
30568 }

```

(End of definition for `__cctab_chk_group_begin:n` and `__cctab_chk_group_end:n`.)

`__cctab_nesting_number:N` This macro returns the numeric index of the current catcode table. In LuaTeX this is just the argument, which is a count reference to a `\catcodetable` register. In other engines, the number is extracted from the `cctab` variable.


```

30569 \sys_if_engine luatex:TF
30570 { \cs_new:Npn \__cctab_nesting_number:N #1 {#1} }
30571 {
30572   \cs_new:Npn \__cctab_nesting_number:N #1
30573   {
30574     \exp_after:wN \exp_after:wN \exp_after:wN \__cctab_nesting_number:w
30575     \exp_after:wN \token_to_str:N #1
30576   }
30577   \use:e
30578   {
30579     \cs_new:Npn \exp_not:N \__cctab_nesting_number:w
30580     #1 \tl_to_str:n { g__cctab_ } #2 \tl_to_str:n { _cctab } {#2}
30581   }
30582 }

```

(End of definition for `__cctab_nesting_number:N` and `__cctab_nesting_number:w`.)

Finally, install some code at the end of the \TeX run to check that all `\cctab_begin:N` were ended by some `\cctab_end:`.

```

30583 \cs_if_exist:NT \hook_gput_code:nnn
30584 {
30585   \hook_gput_code:nnn { enddocument/end } { cctab }
30586   {
30587     \seq_if_empty:NF \g__cctab_stack_seq
30588     { \msg_error:nn { cctab } { missing-end } }
30589   }
30590 }

```

`\cctab_item:Nn` Evaluate the integer argument only once. In most engines the `cctab` variable only has 256 entries so we only look up the catcode for these entries, otherwise we use the current catcode. In particular, for out-of-range values we use whatever fall-back `\char_value_catcode:n`. In $\text{Lua}\TeX$, we use the `tex.getcatcode` function.

`\cctab_item:cn`

```

30591 \cs_new:Npn \cctab_item:Nn #1#2
30592 { \exp_args:Nf \__cctab_item:nN { \int_eval:n {#2} } #1 }
30593 \sys_if_engine luatex:TF
30594 {
30595   \cs_new:Npn \__cctab_item:nN #1#2
30596   { \lua_now:e { tex.print(-2, tex.getcatcode(\int_use:N #2, #1)) } }
30597 }
30598 {
30599   \cs_new:Npn \__cctab_item:nN #1#2
30600   {
30601     \int_compare:nNnTF {#1} < { 256 }
30602     { \intarray_item:Nn #2 { #1 + 1 } }
30603     { \char_value_catcode:n {#1} }
30604   }
30605 }
30606 \cs_generate_variant:Nn \cctab_item:Nn { c }

```

(End of definition for `\cctab_item:Nn`. This function is documented on page 290.)

85.5 Category code table conditionals

`\cctab_if_exist_p:N` Checks whether a `\cctab var` is defined.

`\cctab_if_exist_p:c`

`\cctab_if_exist:NTF`

`\cctab_if_exist:cTF`

```

30607 \prg_new_eq_conditional:NNn \cctab_if_exist:N \cs_if_exist:N
30608 { TF , T , F , p }
30609 \prg_new_eq_conditional:NNn \cctab_if_exist:c \cs_if_exist:c
30610 { TF , T , F , p }

```

(End of definition for \cctab_if_exist:NTF. This function is documented on page 290.)

`__cctab_chk_if_valid:NTF` Checks whether the argument is defined and whether it is a valid `<cctab var>`. In LuaTeX the validity of the `<cctab var>` is checked by the engine, which complains if the argument is not a `\chardef`'ed constant. In other engines, check if the given command is an intarray variable (the underlying definition is a copy of the cmr10 font).

```

30611 \prg_new_protected_conditional:Npnn \__cctab_chk_if_valid:N #1
30612 { TF , T , F }
30613 {
30614   \cctab_if_exist:NTF #1
30615   {
30616     \__cctab_chk_if_valid_aux:NTF #1
30617     { \prg_return_true: }
30618     {
30619       \msg_error:nne { cctab } { invalid-cctab }
30620       { \token_to_str:N #1 }
30621       \prg_return_false:
30622     }
30623   }
30624   {
30625     \msg_error:nne { kernel } { command-not-defined }
30626     { \token_to_str:N #1 }
30627     \prg_return_false:
30628   }
30629 }
30630 \sys_if_engine luatex:TF
30631 {
30632   \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
30633   {
30634     \int_compare:nNnTF {#1-1} < { \e@alloc@ccodetable@count }
30635   }
30636   \cs_if_exist:NT \c_syst_catcodes_n
30637   {
30638     \cs_gset_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
30639     {
30640       \int_compare:nTF { #1 <= \c_syst_catcodes_n }
30641     }
30642   }
30643 }
30644 {
30645   \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
30646   {
30647     \exp_args:Nf \str_if_in:nnTF
30648     { \cs_meaning:N #1 }
30649     { select~font~cmr10~at~ }
30650   }
30651 }

```

(End of definition for __cctab_chk_if_valid:NTF and __cctab_chk_if_valid_aux:NTF.)

85.6 Constant category code tables

`\cctab_const:Nn` Creates a new `\cctab var` then sets it with the `iniTeX` and user-supplied codes. To avoid false debug errors, we write out implementation of `\cctab_new:N` and `\cctab_gset:Nn` instead of directly using them here. The initialization part in `\cctab_new:N` in non-LuaTeX is omitted as it's covered by the `iniTeX` settings.

```

30652 \cs_new_protected:Npn \cctab_const:Nn #1#2
30653   {
30654     \__kernel_chk_if_free_cs:N #1
30655     \__cctab_new:N #1
30656     \group_begin:
30657       \cctab_select:N \c_initex_cctab
30658       #2 \scan_stop:
30659       \__cctab_gset:n {#1}
30660     \group_end:
30661   }
30662 \cs_generate_variant:Nn \cctab_const:Nn { c }

```

(End of definition for `\cctab_const:Nn`. This function is documented on page 289.)

`\c_initex_cctab` Creating category code tables means thinking starting from `iniTeX`. For all-other and `\c_other_cctab` the standard “string” tables that’s easy.

`\c_str_cctab`

```

30663 \cctab_new:N \c_initex_cctab
30664 \cctab_const:Nn \c_other_cctab
30665   {
30666     \cctab_select:N \c_initex_cctab
30667     \int_set:Nn \tex_endlinechar:D { -1 }
30668     \int_step_inline:nnn { 0 } { 127 }
30669       { \char_set_catcode_other:n {#1} }
30670   }
30671 \cctab_const:Nn \c_str_cctab
30672   {
30673     \cctab_select:N \c_other_cctab
30674     \char_set_catcode_space:n { 32 }
30675   }

```

(End of definition for `\c_initex_cctab`, `\c_other_cctab`, and `\c_str_cctab`. These variables are documented on page 291.)

`\c_code_cctab`
`\c_document_cctab`

To pick up document-level category codes, we need to delay set up to the end of the format, where that’s possible. Also, as there are a *lot* of category codes to set, we avoid using the official interface and store the document codes using internal code. Depending on whether we are in the hook or not, the catcodes may be code or document, so we explicitly set up both correctly.

```

30676 \cs_if_exist:NTF \@expl@finalise@setup@@
30677   { \tl_gput_right:Nn \@expl@finalise@setup@@ }
30678   { \use:n }
30679   {
30680     \__cctab_new:N \c_code_cctab
30681     \group_begin:
30682       \int_set:Nn \tex_endlinechar:D { 32 }
30683       \char_set_catcode_invalid:n { 0 }
30684       \bool_lazy_or:nnTF
30685         { \sys_if_engine_xetex_p: } { \sys_if_engine luatex_p: }

```

```

30686     { \int_step_function:nnN { 31 } \char_set_catcode_invalid:n }
30687     { \int_step_function:nnN { 31 } \char_set_catcode_active:n }
30688 \int_step_function:nnN { 33 } { 64 } \char_set_catcode_other:n
30689 \int_step_function:nnN { 65 } { 90 } \char_set_catcode_letter:n
30690 \int_step_function:nnN { 91 } { 96 } \char_set_catcode_other:n
30691 \int_step_function:nnN { 97 } { 122 } \char_set_catcode_letter:n
30692 \char_set_catcode_ignore:n      { 9 } % tab
30693 \char_set_catcode_other:n       { 10 } % lf
30694 \char_set_catcode_active:n      { 12 } % ff
30695 \char_set_catcode_end_line:n    { 13 } % cr
30696 \char_set_catcode_ignore:n      { 32 } % space
30697 \char_set_catcode_parameter:n   { 35 } % hash
30698 \char_set_catcode_math_toggle:n { 36 } % dollar
30699 \char_set_catcode_comment:n     { 37 } % percent
30700 \char_set_catcode_alignment:n   { 38 } % ampersand
30701 \char_set_catcode_letter:n      { 58 } % colon
30702 \char_set_catcode_escape:n      { 92 } % backslash
30703 \char_set_catcode_math_superscript:n { 94 } % circumflex
30704 \char_set_catcode_letter:n      { 95 } % underscore
30705 \char_set_catcode_group_begin:n { 123 } % left brace
30706 \char_set_catcode_other:n       { 124 } % pipe
30707 \char_set_catcode_group_end:n    { 125 } % right brace
30708 \char_set_catcode_space:n       { 126 } % tilde
30709 \bool_set_catcode_invalid:n     { 127 } % ^^?
30710 \bool_lazy_or:nnF
30711   { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
30712   { \int_step_function:nnN { 128 } { 255 } \char_set_catcode_active:n }
30713 \__cctab_gset:n { \c_code_cctab }
30714 \group_end:
30715 \cctab_const:Nn \c_document_cctab
30716   {
30717     \cctab_select:N \c_code_cctab
30718     \int_set:Nn \tex_endlinechar:D { 13 }
30719     \char_set_catcode_space:n      { 9 }
30720     \char_set_catcode_space:n      { 32 }
30721     \char_set_catcode_other:n      { 58 }
30722     \char_set_catcode_math_subscript:n { 95 }
30723     \char_set_catcode_active:n     { 126 }
30724   }
30725 }

```

(End of definition for \c_code_cctab and \c_document_cctab. These variables are documented on page 290.)

\g_tmpa_cctab
\g_tmpb_cctab

```

30726 \cctab_new:N \g_tmpa_cctab
30727 \cctab_new:N \g_tmpb_cctab

```

(End of definition for \g_tmpa_cctab and \g_tmpb_cctab. These variables are documented on page 291.)

85.7 Messages

```

30728 \msg_new:nnnn { cctab } { stack-full }
30729   { The-category-code-table-stack-is-exhausted. }

```

```

30730 {
30731   LaTeX-has-been-asked-to-switch-to-a-new-category-code-table,~
30732   but-there-is-no-more-space-to-do-this!
30733 }
30734 \msg_new:nnnn { cctab } { extra-end }
30735 { Extra~\iow_char:N\cctab_end:~ignored~\msg_line_context:. }
30736 {
30737   LaTeX-came-across-a~\iow_char:N\cctab_end:~without-a-matching~
30738   \iow_char:N\cctab_begin:N.~This-command-will-be-ignored.
30739 }
30740 \msg_new:nnnn { cctab } { missing-end }
30741 { Missing~\iow_char:N\cctab_end:~before-end-of-TeX-run. }
30742 {
30743   LaTeX-came-across-more~\iow_char:N\cctab_begin:N~than~
30744   \iow_char:N\cctab_end:.
30745 }
30746 \msg_new:nnnn { cctab } { invalid-cctab }
30747 { Invalid~\iow_char:N\catcode-table. }
30748 {
30749   You-can-only-switch-to-a~\iow_char:N\catcode-table-that-is~
30750   initialized-using~\iow_char:N\cctab_new:N-or~
30751   \iow_char:N\cctab_const:Nn.
30752 }
30753 \msg_new:nnnn { cctab } { group-mismatch }
30754 {
30755   \iow_char:N\cctab_end:~occurred-in-a~
30756   \int_case:nn {#1}
30757   {
30758     { 0 } { different-group }
30759     { 1 } { higher-group-level }
30760     { -1 } { lower-group-level }
30761   } ~than~
30762   the-matching~\iow_char:N\cctab_begin:N.
30763 }
30764 {
30765   Catcode-tables-and-groups-must-be-properly-nested,~but~
30766   you-tried-to-interleave-them.~LaTeX-will-try-to-proceed,~
30767   but-results-may-be-unexpected.
30768 }
30769 \prop_gput:Nnn \g_msg_module_name_prop { cctab } { LaTeX }
30770 \prop_gput:Nnn \g_msg_module_type_prop { cctab } { }
30771 </package>

```

Chapter 86

Unicode implementation

```
30772 ⟨*package⟩
30773 ⟨@@=codepoint⟩
```

86.1 User functions

`\codepoint_str_generate:n` Conversion of a codepoint to a character (Unicode engines) or to one or more bytes (8-bit engines) is required. For loading the data, all that is needed is the form which creates strings: these are outside the group as they will also be used when looking up data in the hash table storage at point-of-use. Later, we will also need functions that can generate character tokens for document use: those are defined below, in the data recovery setup.

```
30774 \bool_lazy_or:nmTF
30775 { \sys_if_engine luatex_p: }
30776 { \sys_if_engine xetex_p: }
30777 {
30778   \cs_new:Npn \codepoint_str_generate:n #1
30779     {
30780       \int_compare:nNnTF {#1} = { '\ }
30781         { ~ }
30782         { \char_generate:nn {#1} { 12 } }
30783     }
30784   \cs_new:Npn \codepoint_generate:nn #1#2
30785     {
30786       \int_compare:nNnTF {#1} = { '\ }
30787         { ~ }
30788         {
30789           \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
30790             { \char_generate:nn {#1} {#2} }
30791         }
30792     }
30793 }
30794 {
30795   \cs_new:Npn \codepoint_str_generate:n #1
30796     {
30797       \int_compare:nNnTF {#1} = { '\ }
30798         { ~ }
30799         {
30800           \use:e
```

```

30801         {
30802             \exp_not:N \__codepoint_str_generate:nmmm
30803             \__kernel_codepoint_to_bytes:n {#1}
30804         }
30805     }
30806 }
30807 \cs_new:Npn \__codepoint_str_generate:nmmm #1#2#3#4
30808 {
30809     \char_generate:nn {#1} { 12 }
30810     \tl_if_blank:nF {#2}
30811     {
30812         \char_generate:nn {#2} { 12 }
30813         \tl_if_blank:nF {#3}
30814         {
30815             \char_generate:nn {#3} { 12 }
30816             \tl_if_blank:nF {#4}
30817             { \char_generate:nn {#4} { 12 } }
30818         }
30819     }
30820 }
30821 \cs_new:Npn \codepoint_generate:nn #1#2
30822 {
30823     \int_compare:nNnTF {#1} = { '\ }
30824     { ~ }
30825     {
30826         \int_compare:nNnTF {#1} < { "80 }
30827         {
30828             \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
30829             { \char_generate:nn {#1} {#2} }
30830         }
30831         {
30832             \use:e
30833             {
30834                 \exp_not:N \__codepoint_generate:nmmm
30835                 \__kernel_codepoint_to_bytes:n {#1}
30836             }
30837         }
30838     }
30839 }
30840 \cs_new:Npn \__codepoint_generate:nmmm #1#2#3#4
30841 {
30842     \__kernel_exp_not:w \exp_after:wN
30843     {
30844         \tex_expanded:D
30845         {
30846             \__codepoint_generate:n {#1}
30847             \__codepoint_generate:n {#2}
30848             \tl_if_blank:nF {#3}
30849             {
30850                 \__codepoint_generate:n {#3}
30851                 \tl_if_blank:nF {#4}
30852                 { \__codepoint_generate:n {#4} }
30853             }
30854         }

```

```

30855     }
30856   }
30857   \cs_new:Npn \__codepoint_generate:n #1
30858   {
30859     \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
30860     { \char_generate:nn {#1} { 13 } }
30861   }
30862 }

```

(End of definition for `\codepoint_str_generate:n` and others. These functions are documented on page 294.)

This code converts a codepoint into the correct UTF-8 representation. In terms of the algorithm itself, see <https://en.wikipedia.org/wiki/UTF-8> for the octet pattern.

```

\__kernel_codepoint_to_bytes:n
\__codepoint_to_bytes_auxi:n
\__codepoint_to_bytes_auxii:Nnn
\__codepoint_to_bytes_auxiii:n
\__codepoint_to_bytes_outputi:nw
\__codepoint_to_bytes_outputii:nw
\__codepoint_to_bytes_outputiii:nw
\__codepoint_to_bytes_outputiv:nw
\__codepoint_to_bytes_output:nmn
\__codepoint_to_bytes_output:fnm
\__codepoint_to_bytes_end:

```

```

30863 \cs_new:Npn \__kernel_codepoint_to_bytes:n #1
30864 {
30865   \exp_args:Nf \__codepoint_to_bytes_auxi:n
30866   { \int_eval:n {#1} }
30867 }
30868 \cs_new:Npn \__codepoint_to_bytes_auxi:n #1
30869 {
30870   \if_int_compare:w #1 > "80 \exp_stop_f:
30871   \if_int_compare:w #1 < "800 \exp_stop_f:
30872     \__codepoint_to_bytes_outputi:nw
30873     { \__codepoint_to_bytes_auxii:Nnn C {#1} { 64 } }
30874     \__codepoint_to_bytes_outputii:nw
30875     { \__codepoint_to_bytes_auxiii:n {#1} }
30876   \else:
30877     \if_int_compare:w #1 < "10000 \exp_stop_f:
30878     \__codepoint_to_bytes_outputi:nw
30879     { \__codepoint_to_bytes_auxii:Nnn E {#1} { 64 * 64 } }
30880     \__codepoint_to_bytes_outputii:nw
30881     {
30882       \__codepoint_to_bytes_auxiii:n
30883       { \int_div_truncate:nn {#1} { 64 } }
30884     }
30885     \__codepoint_to_bytes_outputiii:nw
30886     { \__codepoint_to_bytes_auxiii:n {#1} }
30887   \else:
30888     \__codepoint_to_bytes_outputi:nw
30889     {
30890       \__codepoint_to_bytes_auxii:Nnn F
30891       {#1} { 64 * 64 * 64 }
30892     }
30893     \__codepoint_to_bytes_outputii:nw
30894     {
30895       \__codepoint_to_bytes_auxiii:n
30896       { \int_div_truncate:nn {#1} { 64 * 64 } }
30897     }
30898     \__codepoint_to_bytes_outputiii:nw
30899     {
30900       \__codepoint_to_bytes_auxiii:n
30901       { \int_div_truncate:nn {#1} { 64 } }
30902     }

```



```

30903         \__codepoint_to_bytes_outputiv:nw
30904         { \__codepoint_to_bytes_auxiii:n {#1} }
30905     \fi:
30906     \fi:
30907     \else:
30908         \__codepoint_to_bytes_outputi:nw {#1}
30909     \fi:
30910     \__codepoint_to_bytes_end: { } { } { } { }
30911 }
30912 \cs_new:Npn \__codepoint_to_bytes_auxii:Nnn #1#2#3
30913 { "#10 + \int_div_truncate:nn {#2} {#3} }
30914 \cs_new:Npn \__codepoint_to_bytes_auxiii:n #1
30915 { \int_mod:nn {#1} { 64 } + 128 }
30916 \cs_new:Npn \__codepoint_to_bytes_outputi:nw
30917 #1 #2 \__codepoint_to_bytes_end: #3
30918 { \__codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { } {#2} }
30919 \cs_new:Npn \__codepoint_to_bytes_outputii:nw
30920 #1 #2 \__codepoint_to_bytes_end: #3#4
30921 { \__codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { {#3} } {#2} }
30922 \cs_new:Npn \__codepoint_to_bytes_outputiii:nw
30923 #1 #2 \__codepoint_to_bytes_end: #3#4#5
30924 {
30925     \__codepoint_to_bytes_output:fnn
30926     { \int_eval:n {#1} } { {#3} {#4} } {#2}
30927 }
30928 \cs_new:Npn \__codepoint_to_bytes_outputiv:nw
30929 #1 #2 \__codepoint_to_bytes_end: #3#4#5#6
30930 {
30931     \__codepoint_to_bytes_output:fnn
30932     { \int_eval:n {#1} } { {#3} {#4} {#5} } {#2}
30933 }
30934 \cs_new:Npn \__codepoint_to_bytes_output:nnn #1#2#3
30935 {
30936     #3
30937     \__codepoint_to_bytes_end: #2 {#1}
30938 }
30939 \cs_generate_variant:Nn \__codepoint_to_bytes_output:nnn { f }
30940 \cs_new:Npn \__codepoint_to_bytes_end: { }

```

(End of definition for `__kernel_codepoint_to_bytes:n` and others.)

`\codepoint_to_category:n` Get the value and convert back to the string.

```

30941 \cs_new:Npn \codepoint_to_category:n #1
30942 {
30943     \cs:w
30944     c__codepoint_category_
30945     \tex_romannumeral:D
30946     \__kernel_codepoint_data:nn { category } {#1}
30947     _str
30948     \cs_end:
30949 }

```

(End of definition for `\codepoint_to_category:n`. This function is documented on page 295.)

```

\codepoint_to_nfd:n
  \__codepoint_to_nfd:n
    \__codepoint_to_nfd:nn
      \__codepoint_to_nfd:nnn
        \__codepoint_to_nfd:nnnn
          30950 \cs_new:Npn \codepoint_to_nfd:n #1
          30951   { \exp_args:Ne \__codepoint_to_nfd:n { \int_eval:n {#1} } }
          30952 \cs_new:Npn \__codepoint_to_nfd:n #1
          30953   { \__codepoint_to_nfd:nn {#1} { \char_value_catcode:n {#1} } }
          30954 \bool_lazy_or:nnF
          30955   { \sys_if_engine luatex_p: }
          30956   { \sys_if_engine xetex_p: }
          30957   {
          30958     \cs_gset:Npn \__codepoint_to_nfd:n #1
          30959     {
          30960       \int_compare:nNnTF {#1} > { "80 }
          30961       { \__codepoint_to_nfd:nn {#1} { 12 } }
          30962       { \__codepoint_to_nfd:nn {#1} { \char_value_catcode:n {#1} } }
          30963     }
          30964   }
          30965 \cs_new:Npn \__codepoint_to_nfd:nn #1#2
          30966   {
          30967     \exp_args:Ne \__codepoint_to_nfd:nnn
          30968     { \__codepoint_nfd:n {#1} } {#1} {#2}
          30969   }
          30970 \cs_new:Npn \__codepoint_to_nfd:nnn #1#2#3 { \__codepoint_to_nfd:nnnn #1 {#2} {#3} }
          30971 \cs_new:Npn \__codepoint_to_nfd:nnnn #1#2#3#4
          30972   {
          30973     \int_compare:nNnTF {#1} = {#3}
          30974     { \codepoint_generate:nn {#1} {#4} }
          30975     {
          30976       \__codepoint_to_nfd:nn {#1} {#4}
          30977       \tl_if_blank:nF {#2}
          30978       { \__codepoint_to_nfd:nn {#2} {#4} }
          30979     }
          30980   }

```

(End of definition for `\codepoint_to_nfd:n` and others. This function is documented on page 295.)

86.2 Data loader

Text operations requires data from the Unicode Consortium. Data read into Unicode engine formats is at best a small part of what we need, so there is a loader here to set up the appropriate data structures.

Where we need data for most or all of the Unicode range, we use the two-stage table approach recommended by the Unicode Consortium and demonstrated in a model implementation in Python in https://www.strchr.com/multi-stage_tables. This approach uses the `intarray` (`fontdimen`-based) data type as it is fast for random access and avoids significant hash table usage. In contrast, where only a small subset of codepoints are required, storage as macros is preferable. There is also some consideration of the effort needed to load data: see for example the grapheme breaking information, which would be problematic to convert into a two-stage table but which can be used with reasonable performance in a small number of comma lists (at the cost that breaking at higher codepoint Hangul characters will be slightly slow).

`\c__codepoint_block_size_int` Choosing the block size for the blocks in the two-stage approach is non-trivial: depending on the data stored, the optimal size for memory usage will vary. At the same time, for us there is also the question of load-time: larger blocks require longer comma lists as intermediates, so are slower. As this is going to be needed to use the data, we set it up outside of the group for clarity.

```
30981 \int_const:Nn \c__codepoint_block_size_int { 64 }
```

(End of definition for `\c__codepoint_block_size_int`.)

Parsing the data files can be the same way for all engines, but where they are stored as character tokens, the construction method depends on whether they are Unicode or 8-bit internally. Parsing is therefore done by common functions, with some data storage using engine-specific auxiliaries.

As only the data needs to remain at the end of this process, everything is set up inside a group. The only thing that is outside is creating a stream: they are global anyway and it is best to force a stream for all engines.

`\g__codepoint_data_ior`

```
30982 \ior_new:N \g__codepoint_data_ior
```

(End of definition for `\g__codepoint_data_ior`.)

We need some setup for the two-part table approach. The number of blocks we need will be variable, but the resulting size of the stage one table is predictable. For performance reasons, we therefore create the stage one tables now so they can be used immediately, and will later rename them as a constant tables. For each two-stage table construction, we need a comma list to hold the partial block and a couple of integers to track where we are up to. To avoid burning registers, the latter are stored in macros and are “fake” integers. We also avoid any `new` functions, keeping as much as possible local.

As we need both positive and negative values, case data requires one two-stage table for each transformation. In contrasts, general Unicode properties could be stored in one table with appropriate combination rules: that is not done at present but is likely to be added over time. Here, all that is needed is additional entries into the comma-list to create the structures.

Notice that in the standard `expl3` way we are indexes position not offset: that does mean a little work later.

```
30983 \group_begin:
30984   \clist_map_inline:nn
30985     { category , uppercase , lowercase }
30986     {
30987       \cs_set_nopar:cpn { l__codepoint_ #1_block_clist } { }
30988       \cs_set_nopar:cpn { l__codepoint_ #1_block_t1 } { 1 }
30989       \cs_set_nopar:cpn { l__codepoint_ #1_pos_t1 } { 0 }
30990       \intarray_new:cn { g__codepoint_ #1_index_intarray }
30991         { \int_div_truncate:nm { "110000 } \c__codepoint_block_size_int }
30992     }
```

We need an integer value when matching the current block to those we have already seen, and a way to track codepoints for handling ranges. Again, we avoid using up registers or creating global names.

```
30993   \cs_set_nopar:Npn \l__codepoint_next_codepoint_fint_t1 { 0 }
30994   \cs_set_nopar:Npn \l__codepoint_matched_block_t1 { 0 }
```

For Unicode general category, there needs to be numerical representation of each possible value. As we need to go from string to number here, but the other way elsewhere, we set up fast mappings both ways, but one set local and the other as constants.

```

30995 \cs_set_protected:Npn \__codepoint_data_auxi:w #1#2
30996 {
30997   \quark_if_recursion_tail_stop:n {#2}
30998   \cs_set_nopar:cpn { l__codepoint_category_ #2 _tl } {#1}
30999   \str_const:cn { c__codepoint_category_ \tex_romannumeral:D #1 _str } {#2}
31000   \exp_args:Ne \__codepoint_data_auxi:w { \int_eval:n { #1 + 1 } }
31001 }
31002 \__codepoint_data_auxi:w { 1 }
31003 { Lu } { Ll } { Lt } { Lm } { Lo }
31004 { Mn } { Me } { Mc }
31005 { Nd } { Nl } { No }
31006 { Zs } { Zl } { Zp }
31007 { Cc } { Cf } { Co } { Cs } { Cn }
31008 { Pd } { Ps } { Pe } { Pc } { Po } { Pi } { Pf }
31009 { Sm } { Sc } { Sk } { So }
31010 \q_recursion_tail
31011 \q_recursion_stop

```

Parse the main Unicode data file and pull out the NFD and case changing data. The NFD data is stored on using the hash table approach and can yield a predictable number of codepoints: one or two. We also need the case data, which will be modified further below. To allow for finding ranges, the description of the codepoint needs to be carried forward.

```

31012 \cs_set_protected:Npn \__codepoint_data_auxi:w
31013 #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
31014 {
31015   \tl_if_blank:nF {#6}
31016   {
31017     \tl_if_head_eq_charcode:nNF {#6} < % >
31018     { \__codepoint_data_auxii:w #1 ; #6 ~ \q_stop }
31019   }
31020   \__codepoint_data_auxiii:w #1 ; #2 ; #3 ;
31021 }
31022 \cs_set_protected:Npn \__codepoint_data_auxii:w #1 ; #2 ~ #3 \q_stop
31023 {
31024   \tl_const:ce
31025   { c__codepoint_nfd_ \codepoint_str_generate:n {"#1} _tl }
31026   {
31027     {"#2}
31028     { \tl_if_blank:nF {#3} {"#3} }
31029   }
31030 }

```

The category data needs to be converted from a string to the numerical equivalent: a simple operation. The case data is going to be stored as an offset from the parent character, rather than an absolute value. We therefore deal with that plus the situation where a codepoint has no mapping data in one shot.

```

31031 \cs_set_protected:Npn \__codepoint_data_auxiii:w
31032 #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ~ \q_stop
31033 {
31034   \use:e

```

```

31035     {
31036         \__codepoint_data_auxiv:w
31037         #1 ; #2 ;
31038         \__codepoint_data_category:n {#3} ;
31039         \__codepoint_data_offset:nn {#1} {#7} ;
31040         \__codepoint_data_offset:nn {#1} {#8} ;
31041         #9;
31042     }
31043 }
31044 \cs_set:Npn \__codepoint_data_category:n #1
31045 { \use:c { l__codepoint_category_ #1 _tl } }
31046 \cs_set:Npn \__codepoint_data_offset:nn #1#2
31047 {
31048     \tl_if_blank:nTF {#2}
31049     { 0 }
31050     { \int_eval:n { "#2 - "#1 } }
31051 }

```

To deal with ranges, we track the position of the next codepoint expected. If there is a gap, we deal with that separately: it could be a range or an unused part of the Unicode space. As such, we deal with the current codepoint here whether or not there is range to fill in. Upper- and lowercase data go into the two-stage table, any titlecase exception is just stored in a macro. The data for the codepoint is added to the current block, and if that is now complete we move on to save the block. The case exceptions are all stored as codepoints, with a fixed number of balanced text as we know that there are never more than three.

```

31052 \cs_set_protected:Npn \__codepoint_data_auxiv:w #1 ; #2 ; #3 ; #4 ; #5 ; #6 ;
31053 {
31054     \int_compare:nNnT {"#1} > \l__codepoint_next_codepoint_fint_tl
31055     {
31056         \__codepoint_data_auxv:nnnw {#1} {#3} {#4} {#5}
31057         #2 Last> \q_stop
31058     }
31059     \__codepoint_add:nn { category } {#3}
31060     \__codepoint_add:nn { uppercase } {#4}
31061     \__codepoint_add:nn { lowercase } {#5}
31062     \int_compare:nNnF {#4} = { \__codepoint_data_offset:nn {#1} {#6} }
31063     {
31064         \tl_const:ce
31065         { c__codepoint_titlecase_ \codepoint_str_generate:n {"#1} _tl }
31066         { {"#6} { } { } }
31067     }
31068     \tl_set:Ne \l__codepoint_next_codepoint_fint_tl
31069     { \int_eval:n { "#1 + 1 } }
31070 }
31071 \cs_set_protected:Npn \__codepoint_add:nn #1#2
31072 {
31073     \clist_put_right:cn { l__codepoint_ #1 _block_clist } {#2}
31074     \int_compare:nNnT { \clist_count:c { l__codepoint_ #1 _block_clist } }
31075     = \c__codepoint_block_size_int
31076     { \__codepoint_save_blocks:nn {#1} { 1 } }
31077 }

```

Distinguish between a range and a gap, and pass on the appropriate value(s). The general

category for unassigned characters is Cn, so we find the correct value once and then use that.

```

31078 \cs_set_protected:Npe \__codepoint_data_auxv:nnnw #1#2#3#4#5 Last> #6 \q_stop
31079 {
31080   \exp_not:N \tl_if_blank:nTF {#6}
31081   {
31082     \exp_not:N \__codepoint_range:nnn {#1} { category }
31083     { \exp_not:N \l__codepoint_category_Cn_tl }
31084     \exp_not:N \__codepoint_range:nnn {#1} { uppercase } { 0 }
31085     \exp_not:N \__codepoint_range:nnn {#1} { lowercase } { 0 }
31086   }
31087   {
31088     \exp_not:N \__codepoint_range:nnn {#1} { category } {#2}
31089     \exp_not:N \__codepoint_range:nnn {#1} { uppercase } {#3}
31090     \exp_not:N \__codepoint_range:nnn {#1} { lowercase } {#4}
31091   }
31092 }

```

Calculated the length of the range and the space remaining in the current block.

```

31093 \cs_set_protected:Npn \__codepoint_range:nnn #1
31094 {
31095   \exp_args:Nf \__codepoint_range_aux:nnn
31096   { \int_eval:n { "#1 - \l__codepoint_next_codepoint_fint_tl } }
31097 }
31098 \cs_set_protected:Npn \__codepoint_range_aux:nnn #1#2
31099 {
31100   \exp_args:Nf \__codepoint_range:nnnn
31101   {
31102     \int_min:nn
31103     {#1}
31104     {
31105       \c__codepoint_block_size_int
31106       - \clist_count:c { l__codepoint_ #2_block_clist }
31107     }
31108   }
31109   {#1} {#2}
31110 }

```

Here we want to do three things: add to and possibly complete the current block, add complete blocks quickly, then finish up the range in a final open block. We need to avoid as far as possible avoid dealing with every single codepoint, so the middle step is optimised.

```

31111 \cs_set_protected:Npn \__codepoint_range:nnnn #1#2#3#4
31112 {
31113   \prg_replicate:nn {#1}
31114   { \clist_put_right:cn { l__codepoint_ #3_block_clist } {#4} }
31115   \int_compare:nNnT { \clist_count:c { l__codepoint_ #3_block_clist } }
31116   = \c__codepoint_block_size_int
31117   { \__codepoint_save_blocks:nn {#3} { 1 } }
31118   \int_compare:nNnF
31119   { \int_div_truncate:nn { #2 - #1 } \c__codepoint_block_size_int } = 0
31120   {
31121     \tl_set:ce { l__codepoint_ #3_block_clist }
31122     {

```

```

31123         \exp_args:NNe \use:nn \use_none:n
31124         { \prg_replicate:nn { \c__codepoint_block_size_int } { , #4 } }
31125     }
31126     \__codepoint_save_blocks:nn {#3}
31127     { \int_div_truncate:nn { (#2 - #1) } \c__codepoint_block_size_int }
31128 }
31129 \prg_replicate:nn
31130 { \int_mod:nn { #2 - #1 } \c__codepoint_block_size_int }
31131 { \clist_put_right:ce { l__codepoint_ #3_block_clist } {#4} }
31132 }

```

To allow rapid comparison, each completed block is stored locally as a comma list: once all of the blocks have been created, they are converted into an `intarray` in one step. The aim here is to check the current block against those we've already used, and either match to an existing block or save a new block.

```

31133 \cs_set_protected:Npn \__codepoint_save_blocks:nn #1#2
31134 {
31135     \tl_set_eq:Nc \l__codepoint_matched_block_tl { l__codepoint_ #1_block_tl }
31136     \int_step_inline:nn { \tl_use:c { l__codepoint_ #1_block_tl } - 1 }
31137     {
31138         \tl_if_eq:ccT { l__codepoint_ #1_block_clist }
31139         { l__codepoint_ #1_block_ ##1_clist }
31140         { \tl_set:Nn \l__codepoint_matched_block_tl {##1} }
31141     }
31142     \int_compare:nNnT
31143     { \tl_use:c { l__codepoint_ #1_block_tl } } = \l__codepoint_matched_block_tl
31144     {
31145         \clist_set_eq:cc
31146         {
31147             l__codepoint_ #1_block_
31148             \tl_use:c { l__codepoint_ #1_block_tl }_clist
31149         }
31150         { l__codepoint_ #1_block_clist }
31151         \tl_set:ce { l__codepoint_ #1_block_tl }
31152         { \int_eval:n { \tl_use:c { l__codepoint_ #1_block_tl } + 1 } }
31153     }
31154     \prg_replicate:nn {#2}
31155     {
31156         \tl_set:ce { l__codepoint_ #1_pos_tl }
31157         { \int_eval:n { \tl_use:c { l__codepoint_ #1_pos_tl } + 1 } }
31158         \exp_args:Nc \__kernel_intarray_gset:Nnn
31159         { g__codepoint_ #1_index_intarray }
31160         { \tl_use:c { l__codepoint_ #1_pos_tl } }
31161         \l__codepoint_matched_block_tl
31162     }
31163     \clist_clear:c { l__codepoint_ #1_block_clist }
31164 }

```

Close out the final block, rename the first stage table, then combine all of the block comma-lists into one large second-stage table with offsets. As we use an index not an offset, there is a little back-and-forward to do.

```

31165 \cs_set_protected:Npn \__codepoint_finalise_blocks:
31166 {
31167     \clist_map_inline:nn { category , uppercase , lowercase }

```

```

31168     {
31169     \__codepoint_range:nnn { 110000 } {##1} { 0 }
31170     \__codepoint_finalise_blocks:n {##1}
31171     }
31172   }
31173   \cs_set_protected:Npn \__codepoint_finalise_blocks:n #1
31174   {
31175     \cs_gset_eq:cc { c__codepoint_ #1 _index_intarray } { g__codepoint_ #1 _index_intarra
31176     \cs_undefine:c { g__codepoint_ #1 _index_intarray }
31177     \intarray_new:cn { g__codepoint_ #1 _blocks_intarray }
31178     { ( \tl_use:c { l__codepoint_ #1 _block_tl } - 1 ) * \c__codepoint_block_size_int }
31179     \int_step_inline:nn { \tl_use:c { l__codepoint_ #1 _block_tl } - 1 }
31180     {
31181       \exp_args:Nv \__codepoint_finalise_blocks:nnn
31182       { l__codepoint_ #1 _block_ ##1 _clist }
31183       {##1} {#1}
31184     }
31185     \cs_gset_eq:cc { c__codepoint_ #1 _blocks_intarray }
31186     { g__codepoint_ #1 _blocks_intarray }
31187     \cs_undefine:c { g__codepoint_ #1 _blocks_intarray }
31188   }
31189   \cs_set_protected:Npn \__codepoint_finalise_blocks:nnn #1#2#3
31190   {
31191     \exp_args:Nnf \__codepoint_finalise_blocks:nnnw { 1 }
31192     { \int_eval:n { ( #2 - 1 ) * \c__codepoint_block_size_int } }
31193     {#3}
31194     #1 , \q_recursion_tail , \q_recursion_stop
31195   }
31196   \cs_set_protected:Npn \__codepoint_finalise_blocks:nnnw #1#2#3#4 ,
31197   {
31198     \quark_if_recursion_tail_stop:n {#4}
31199     \intarray_gset:cnn { g__codepoint_ #3 _blocks_intarray }
31200     { #1 + #2 }
31201     {#4}
31202     \exp_args:Nf \__codepoint_finalise_blocks:nnnw
31203     { \int_eval:n { #1 + 1 } } {#2} {#3}
31204   }

```

With the setup done, read the main data file: it's easiest to do that as a token list with spaces retained.

```

31205   \ior_open:Nn \g__codepoint_data_ior { UnicodeData.txt }
31206   \group_begin:
31207     \char_set_catcode_space:n { '\ }%
31208     \ior_map_variable:NNn \g__codepoint_data_ior \l__codepoint_tmpa_tl
31209     {%
31210       \if_meaning:w \l__codepoint_tmpa_tl \c_space_tl
31211       \exp_after:wN \ior_map_break:
31212       \fi:
31213       \exp_after:wN \__codepoint_data_auxi:w \l__codepoint_tmpa_tl \q_stop
31214     }%
31215     \__codepoint_finalise_blocks:
31216   \group_end:
31217 \group_end:

```

__kernel_codepoint_data:nn Recover data from a two-stage table: entirely generic as this applies to all tables (as we
 __codepoint_data:nnn

use the same block size for all of them). Notice that as we use indices not offsets we have to shuffle out-by-one issues. This function is needed *before* loading the special casing data, as there we need to be able to check the standard case mappings.

```

31218 \cs_new:Npn \__kernel_codepoint_data:nn #1#2
31219 {
31220   \exp_args:Nf \__codepoint_data:nnn
31221   {
31222     \int_eval:n
31223     {
31224       \c__codepoint_block_size_int *
31225       (
31226         \intarray_item:cn { c__codepoint_ #1 _index_intarray }
31227         {
31228           \int_div_truncate:nn {#2}
31229           \c__codepoint_block_size_int
31230           + 1
31231         }
31232         - 1
31233       )
31234     }
31235   }
31236   {#2} {#1}
31237 }
31238 \cs_new:Npn \__codepoint_data:nnn #1#2#3
31239 {
31240   \intarray_item:cn { c__codepoint_ #3 _blocks_intarray }
31241   { #1 + \int_mod:nn {#2} \c__codepoint_block_size_int + 1 }
31242 }

```

(End of definition for __kernel_codepoint_data:nn and __codepoint_data:nnn.)

The other data files all use C-style comments so we have to worry about # tokens (and reading as strings). The set up for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For the more complex foldings, always store the result, splitting up the two or three code points in the input as required.

```

31243 \group_begin:
31244   \ior_open:Nn \g__codepoint_data_ior { CaseFolding.txt }
31245   \cs_set_protected:Npn \__codepoint_data_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
31246   {
31247     \if:w \tl_head:n { #2 ? } C
31248     \reverse_if:N \if_int_compare:w
31249     \int_eval:n { \__kernel_codepoint_data:nn { lowercase } {"#1} + "#1 }
31250     = "#3 ~
31251     \tl_const:ce
31252     { c__codepoint_casefold_ \codepoint_str_generate:n {"#1} _tl }
31253     { {"#3} { } { } }
31254     \fi:
31255     \else:
31256     \if:w \tl_head:n { #2 ? } F
31257     \__codepoint_data_auxii:w #1 ~ #3 ~ \q_stop
31258     \fi:
31259     \fi:
31260   }

```

Here, #4 can have a trailing space, so we tidy up a bit at the cost of speed for these small number of cases it applies to.

```

31261 \cs_set_protected:Npn \__codepoint_data_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
31262 {
31263   \tl_const:ce { c__codepoint_casefold_ \codepoint_str_generate:n {"#1} _tl }
31264   {
31265     {"#2}
31266     {"#3}
31267     { \tl_if_blank:nF {#4} { " \int_to_Hex:n {"#4} } }
31268   }
31269 }
31270 \ior_str_map_inline:Nn \g__codepoint_data_ior
31271 {
31272   \reverse_if:N \if:w \c_hash_str \tl_head:w #1 \c_hash_str \q_stop
31273   \__codepoint_data_auxi:w #1 \q_stop
31274   \fi:
31275 }
31276 \ior_close:N \g__codepoint_data_ior

```

For upper- and lowercasing special situations, there is a bit more to do as we also have titl casing to consider, plus we need to stop part-way through the file.

```

31277 \ior_open:Nn \g__codepoint_data_ior { SpecialCasing.txt }
31278 \cs_set_protected:Npn \__codepoint_data_auxi:w
31279 #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
31280 {
31281   \use:n { \__codepoint_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
31282   \use:n { \__codepoint_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
31283   \str_if_eq:nnF {#3} {#4}
31284   { \use:n { \__codepoint_data_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop }
31285 }
31286 \cs_set_protected:Npn \__codepoint_data_auxii:w
31287 #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
31288 {
31289   \tl_if_empty:nF {#4}
31290   {
31291     \tl_const:ce { c__codepoint_ #2 case_ \codepoint_str_generate:n {"#1} _tl }
31292     {
31293       {"#3}
31294       {"#4}
31295       { \tl_if_blank:nF {#5} {"#5} }
31296     }
31297   }
31298 }
31299 \ior_str_map_inline:Nn \g__codepoint_data_ior
31300 {
31301   \str_if_eq:eeTF { \tl_head:w #1 \c_hash_str \q_stop } { \c_hash_str }
31302   {
31303     \str_if_eq:eeT
31304     {#1}
31305     { \c_hash_str \c_space_tl Conditional~Mappings }
31306     { \ior_map_break: }
31307   }
31308   { \__codepoint_data_auxi:w #1 \q_stop }
31309 }

```

```

31310 \ior_close:N \g__codepoint_data_ior
31311 \group_end:

```

`__kernel_codepoint_case:nn` With the core data files loaded, there is now a need to provide access to this information for other modules. That is done here such that case folding can also be covered. At this level, all that needs to be returned is the

```

\__codepoint_case:nnn
\__codepoint_uppercase:n
\__codepoint_lowercase:n
\__codepoint_titlecase:n
\__codepoint_casefold:n
\__codepoint_case:nn
31312 \cs_new:Npn \__kernel_codepoint_case:nn #1#2
31313 {
31314   \exp_args:Ne \__codepoint_case:nnn
31315     { \codepoint_str_generate:n {#2} } {#1} {#2}
31316 }
31317 \cs_new:Npn \__codepoint_case:nnn #1#2#3
31318 {
31319   \cs_if_exist:cTF { c__codepoint_ #2 _ #1 _tl }
31320     {
31321       \tl_use:c
31322         { c__codepoint_ #2 _ #1 _tl }
31323     }
31324     { \use:c { __codepoint_ #2 :n } {#3} }
31325 }
31326 \cs_new:Npn \__codepoint_uppercase:n { \__codepoint_case:nn { uppercase } }
31327 \cs_new:Npn \__codepoint_lowercase:n { \__codepoint_case:nn { lowercase } }
31328 \cs_new:Npn \__codepoint_titlecase:n { \__codepoint_case:nn { uppercase } }
31329 \cs_new:Npn \__codepoint_casefold:n { \__codepoint_case:nn { lowercase } }
31330 \cs_new:Npn \__codepoint_case:nn #1#2
31331 {
31332   { \int_eval:n { \__kernel_codepoint_data:nn {#1} {#2} + #2 } }
31333   { }
31334   { }
31335 }

```

(End of definition for `__kernel_codepoint_case:nn` and others.)

`__codepoint_nfd:n` A simple interface.
`__codepoint_nfd:nn`

```

31336 \cs_new:Npn \__codepoint_nfd:n #1
31337 { \exp_args:Ne \__codepoint_nfd:nn { \codepoint_str_generate:n {#1} } {#1} }
31338 \cs_new:Npn \__codepoint_nfd:nn #1#2
31339 {
31340   \tl_if_exist:cTF { c__codepoint_nfd_ #1 _tl }
31341     { \tl_use:c { c__codepoint_nfd_ #1 _tl } }
31342     { {#2} { } }
31343 }

```

(End of definition for `__codepoint_nfd:n` and `__codepoint_nfd:nn`.)

```

31344 (@@=text)

```

Read the Unicode grapheme data. This is quite easy to handle and we only need codepoints, not characters, so there is no need to worry about the engine in use. As reading as a string is most convenient, we have to do some work to remove spaces: the hardest part of the entire process!

```

31345 \ior_new:N \g__text_data_ior
31346 \group_begin:
31347 \ior_open:Nn \g__text_data_ior { GraphemeBreakProperty.txt }
31348 \cs_set_nopar:Npn \l__text_tmpa_str { }

```

```

31349 \cs_set_nopar:Npn \l__text_tmpb_str { }
31350 \cs_set_protected:Npn \__text_data_auxi:w #1 ;~ #2 ~ #3 \q_stop
31351 {
31352   \str_if_eq:VnF \l__text_tmpb_str {#2}
31353   {
31354     \str_if_empty:NF \l__text_tmpb_str
31355     {
31356       \clist_const:ce { c__text_grapheme_ \l__text_tmpb_str _clist }
31357       { \exp_after:wN \use_none:n \l__text_tmpa_str }
31358       \cs_set_nopar:Npn \l__text_tmpa_str { }
31359     }
31360     \cs_set_nopar:Npn \l__text_tmpb_str {#2}
31361   }
31362   \__text_data_auxii:w #1 .. #1 .. #1 \q_stop
31363 }
31364 \cs_set_protected:Npn \__text_data_auxii:w #1 .. #2 .. #3 \q_stop
31365 {
31366   \cs_set_nopar:Npe \l__text_tmpa_str
31367   {
31368     \l__text_tmpa_str ,
31369     \tl_trim_spaces:n {#1} .. \tl_trim_spaces:n {#2}
31370   }
31371 }
31372 \ior_str_map_inline:Nn \g__text_data_ior
31373 {
31374   \str_if_eq:eeF { \tl_head:w #1 \c_hash_str \q_stop } { \c_hash_str }
31375   {
31376     \tl_if_blank:nF {#1}
31377     { \__text_data_auxi:w #1 \q_stop }
31378   }
31379 }
31380 \ior_close:N \g__text_data_ior
31381 \group_end:
31382 </package>

```

Chapter 87

l3text implementation

```
31383 (*package)
31384 (@@=text)
31385 \cs_generate_variant:Nn \tl_if_head_eq_meaning_p:nN { o }
```

87.1 Internal auxiliaries

`\s__text_stop` Internal scan marks.
31386 `\scan_new:N \s__text_stop`
(End of definition for `\s__text_stop`.)

`\q__text_nil` Internal quarks.
31387 `\quark_new:N \q__text_nil`
(End of definition for `\q__text_nil`.)

`__text_quark_if_nil_p:n` Branching quark conditional.
`__text_quark_if_nil:nTF` 31388 `__kernel_quark_new_conditional:Nn __text_quark_if_nil:n { TF }`
(End of definition for `__text_quark_if_nil:nTF`.)

`\q__text_recursion_tail` Internal recursion quarks.
`\q__text_recursion_stop` 31389 `\quark_new:N \q__text_recursion_tail`
31390 `\quark_new:N \q__text_recursion_stop`
(End of definition for `\q__text_recursion_tail` and `\q__text_recursion_stop`.)

`__text_use_i_delimit_by_q_recursion_stop:nw` Functions to gobble up to a quark.
31391 `\cs_new:Npn __text_use_i_delimit_by_q_recursion_stop:nw`
31392 `#1 #2 \q__text_recursion_stop {#1}`
(End of definition for `__text_use_i_delimit_by_q_recursion_stop:nw`.)

`__text_if_q_recursion_tail_stop_do:Nn` Functions to query recursion quarks.
`__text_if_q_recursion_tail_stop_do:mn` 31393 `__kernel_quark_new_test:N __text_if_q_recursion_tail_stop_do:Nn`
31394 `__kernel_quark_new_test:N __text_if_q_recursion_tail_stop_do:mn`

(End of definition for `__text_if_q_recursion_tail_stop_do:Nn` and `__text_if_q_recursion_tail_stop_do:nn`.)

```
\s__text_recursion_tail Internal scan marks quarks.
\s__text_recursion_stop 31395 \scan_new:N \s__text_recursion_tail
                        31396 \scan_new:N \s__text_recursion_stop
```

(End of definition for `\s__text_recursion_tail` and `\s__text_recursion_stop`.)

```
\__text_use_i_delimit_by_s_recursion_stop:nw Functions to gobble up to a scan mark.
31397 \cs_new:Npn \__text_use_i_delimit_by_s_recursion_stop:nw
31398 #1 #2 \s__text_recursion_stop {#1}
```

(End of definition for `__text_use_i_delimit_by_s_recursion_stop:nw`.)

`__text_if_s_recursion_tail_stop_do:Nn` Functions to query recursion scan marks. Slower than a quark test but needed to avoid issues in the outer expansion loop with unterminated `\romannumeral` primitives.

```
31399 \cs_new:Npn \__text_if_s_recursion_tail_stop_do:Nn #1
31400 {
31401   \bool_lazy_and:nnTF
31402     { \cs_if_eq_p:NN \s__text_recursion_tail #1 }
31403     { \str_if_eq_p:nn { \s__text_recursion_tail } {#1} }
31404     { \__text_use_i_delimit_by_s_recursion_stop:nw }
31405     { \use_none:n }
31406 }
```

(End of definition for `__text_if_s_recursion_tail_stop_do:Nn`.)

87.2 Utilities

`__text_token_to_explicit:N` The idea here is to take a token and ensure that if it's an implicit char, we output the explicit version. Otherwise, the token needs to be unchanged. First, we have to split between control sequences and everything else.

```
\__text_token_to_explicit:N
  \__text_token_to_explicit_char:N
  \__text_token_to_explicit_cs:N
  \__text_token_to_explicit_cs_aux:N
\__text_token_to_explicit:n 31407 \group_begin:
  \__text_token_to_explicit_auxi:w 31408   \char_set_catcode_active:n { 0 }
  \__text_token_to_explicit_auxii:w 31409   \cs_new:Npn \__text_token_to_explicit:N #1
  \__text_token_to_explicit_auxiii:w 31410   {
31411     \if_catcode:w \exp_not:N #1
31412       \if_catcode:w \scan_stop: \exp_not:N #1
31413         \scan_stop:
31414       \else:
31415         \exp_not:N ^^@
31416       \fi:
31417       \exp_after:wN \__text_token_to_explicit_cs:N
31418     \else:
31419       \exp_after:wN \__text_token_to_explicit_char:N
31420     \fi:
31421     #1
31422   }
31423 \group_end:
```

For control sequences, we can check for macros versus other cases using `\if_meaning:w`, then explicitly check for `\chardef` and `\mathchardef`.

```

31424 \cs_new:Npn \__text_token_to_explicit_cs:N #1
31425 {
31426   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
31427   \exp_after:wN \use:nn \exp_after:wN
31428     \__text_token_to_explicit_cs_aux:N
31429   \else:
31430     \exp_after:wN \exp_not:n
31431   \fi:
31432   {#1}
31433 }
31434 \cs_new:Npn \__text_token_to_explicit_cs_aux:N #1
31435 {
31436   \bool_lazy_or:nnTF
31437     { \token_if_chardef_p:N #1 }
31438     { \token_if_mathchardef_p:N #1 }
31439   {
31440     \char_generate:nn {#1}
31441     {
31442       \if_int_compare:w \char_value_catcode:n {#1} = 10 \exp_stop_f:
31443         10
31444       \else:
31445         12
31446       \fi:
31447     }
31448   }
31449   {#1}
31450 }

```

For character tokens, we need to filter out the implicit characters from those that are explicit. That's done here, then if necessary we work out the category code and generate the char. To avoid issues with alignment tabs, that one is done by elimination rather than looking up the code explicitly. The trick with finding the charcode is that the \TeX messages are either the *something* character *char* or the *type* *char*.

```

31451 \cs_new:Npn \__text_token_to_explicit_char:N #1
31452 {
31453   \if:w
31454     \if_catcode:w ^ \exp_args:No \str_tail:n { \token_to_str:N #1 } ^
31455     \token_to_str:N #1 #1
31456   \else:
31457     AB
31458   \fi:
31459   \exp_after:wN \exp_not:n
31460   \else:
31461     \exp_after:wN \__text_token_to_explicit:n
31462   \fi:
31463   {#1}
31464 }
31465 \cs_new:Npn \__text_token_to_explicit:n #1
31466 {
31467   \exp_after:wN \__text_token_to_explicit_auxi:w
31468   \int_value:w
31469   \if_catcode:w \c_group_begin_token #1 1 \else:

```

```

31470     \if_catcode:w \c_group_end_token #1 2 \else:
31471     \if_catcode:w \c_math_toggle_token #1 3 \else:
31472     \if_catcode:w ## #1 6 \else:
31473     \if_catcode:w ^ #1 7 \else:
31474     \if_catcode:w \c_math_subscript_token #1 8 \else:
31475     \if_catcode:w \c_space_token #1 10 \else:
31476     \if_catcode:w A #1 11 \else:
31477     \if_catcode:w + #1 12 \else:
31478     4 \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi:
31479     \exp_after:wN ;
31480     \token_to_meaning:N #1 \s__text_stop
31481   }
31482 \cs_new:Npn \__text_token_to_explicit_auxi:w #1 ; #2 \s__text_stop
31483   {
31484     \char_generate:nn
31485     {
31486       \if_int_compare:w #1 < 9 \exp_stop_f:
31487         \exp_after:wN \__text_token_to_explicit_auxii:w
31488       \else:
31489         \exp_after:wN \__text_token_to_explicit_auxiii:w
31490       \fi:
31491       #2
31492     }
31493     {#1}
31494   }
31495 \exp_last_unbraced:NNNN \cs_new:Npn \__text_token_to_explicit_auxii:w
31496   #1 { \tl_to_str:n { character ~ } } { ' }
31497 \cs_new:Npn \__text_token_to_explicit_auxiii:w #1 ~ #2 ~ { ' }

```

(End of definition for `__text_token_to_explicit:N` and others.)

`__text_char_catcode:N` An idea from `l3char`: we need to get the category code of a specific token, not the general case.

```

31498 \cs_new:Npn \__text_char_catcode:N #1
31499   {
31500     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
31501     3
31502   \else:
31503     \if_catcode:w \exp_not:N #1 \c_alignment_token
31504     4
31505   \else:
31506     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
31507     7
31508   \else:
31509     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
31510     8
31511   \else:
31512     \if_catcode:w \exp_not:N #1 \c_space_token
31513     10
31514   \else:
31515     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
31516     11
31517   \else:
31518     \if_catcode:w \exp_not:N #1 \c_catcode_other_token

```



```

31519             12
31520             \else:
31521             13
31522             \fi:
31523         \fi:
31524     \fi:
31525 \fi:
31526 \fi:
31527 \fi:
31528 \fi:
31529 }

```

(End of definition for `_text_char_catcode:N`.)

`_text_if_expandable:NTF` Test for tokens that make sense to expand here: that is more restrictive than the engine view.

```

31530 \prg_new_conditional:Npnn \_text_if_expandable:N #1 { T , F , TF }
31531 {
31532     \token_if_expandable:NTF #1
31533     {
31534         \bool_lazy_any:nTF
31535         {
31536             { \token_if_protected_macro_p:N #1 }
31537             { \token_if_protected_long_macro_p:N #1 }
31538             { \token_if_eq_meaning_p:NN \q_text_recursion_tail #1 }
31539         }
31540         { \prg_return_false: }
31541         { \prg_return_true: }
31542     }
31543     { \prg_return_false: }
31544 }

```

(End of definition for `_text_if_expandable:NTF`.)

87.3 Codepoint utilities

For working with codepoints in an engine-neutral way.

`_text_codepoint_process:nN` Grab a codepoint and apply some code to it: here #1 should expect one following *balanced text*.

```

\_text_codepoint_process_aux:nN
\_text_codepoint_process:nNN
\_text_codepoint_process:nNNN
\_text_codepoint_process:nNNNN
31545 \bool_lazy_or:nnTF
31546 { \sys_if_engine luatex_p: }
31547 { \sys_if_engine xetex_p: }
31548 {
31549     \cs_new:Npn \_text_codepoint_process:nN #1#2 { #1 {#2} }
31550 }
31551 {
31552     \cs_new:Npe \_text_codepoint_process:nN #1#2
31553     {
31554         \exp_not:N \int_compare:nNnTF {'#2} > { "80 }
31555         {
31556             \sys_if_engine pdftex:TF
31557             { \exp_not:N \_text_codepoint_process_aux:nN }

```

```

31558         {
31559             \exp_not:N \int_compare:nNnTF { '#2 } > { "FF }
31560             { \exp_not:N \use:n }
31561             { \exp_not:N \__text_codepoint_process_aux:nN }
31562         }
31563     }
31564     { \exp_not:N \use:n }
31565     { #1 } #2
31566 }
31567 \cs_new:Npn \__text_codepoint_process_aux:nN #1#2
31568 {
31569     \int_compare:nNnTF { '#2 } < { "EO }
31570     { \__text_codepoint_process:nNN }
31571     {
31572         \int_compare:nNnTF { '#2 } < { "FO }
31573         { \__text_codepoint_process:nNNN }
31574         { \__text_codepoint_process:nNNNN }
31575     }
31576     { #1 } #2
31577 }
31578 \cs_new:Npn \__text_codepoint_process:nNN #1#2#3
31579 { #1 { #2#3 } }
31580 \cs_new:Npn \__text_codepoint_process:nNNN #1#2#3#4
31581 { #1 { #2#3#4 } }
31582 \cs_new:Npn \__text_codepoint_process:nNNNN #1#2#3#4#5
31583 { #1 { #2#3#4#5 } }
31584 }

```

(End of definition for __text_codepoint_process:nN and others.)

__text_codepoint_compare_p:nNn Allows comparison for all engines using a first “character” followed by a codepoint.

```

\__text_codepoint_compare:nNnTF
\__text_codepoint_from_chars:Nw
\__text_codepoint_from_chars_aux:Nw
\__text_codepoint_from_chars:N
\__text_codepoint_from_chars:NN
\__text_codepoint_from_chars:NNN
\__text_codepoint_from_chars:NNNN
31585 \bool_lazy_or:nmTF
31586 { \sys_if_engine luatex_p: }
31587 { \sys_if_engine xetex_p: }
31588 {
31589     \prg_new_conditional:Npnn
31590     \__text_codepoint_compare:nNn #1#2#3 { TF , p }
31591     {
31592         \int_compare:nNnTF { '#1 } #2 { #3 }
31593         \prg_return_true: \prg_return_false:
31594     }
31595     \cs_new:Npn \__text_codepoint_from_chars:Nw #1 { '#1 }
31596 }
31597 {
31598     \prg_new_conditional:Npnn
31599     \__text_codepoint_compare:nNn #1#2#3 { TF , p }
31600     {
31601         \int_compare:nNnTF { \__text_codepoint_from_chars:Nw #1 }
31602         #2 { #3 }
31603         \prg_return_true: \prg_return_false:
31604     }
31605     \cs_new:Npe \__text_codepoint_from_chars:Nw #1
31606     {
31607         \exp_not:N \if_int_compare:w '#1 > "80 \exp_not:N \exp_stop_f:

```

```

31608     \sys_if_engine_pdftex:TF
31609     {
31610         \exp_not:N \exp_after:wN
31611         \exp_not:N \__text_codepoint_from_chars_aux:Nw
31612     }
31613     {
31614         \exp_not:N \if_int_compare:w '#1 > "FF \exp_not:N \exp_stop_f:
31615         \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
31616         \exp_not:N \exp_after:wN
31617         \exp_not:N \__text_codepoint_from_chars:N
31618         \exp_not:N \else:
31619         \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
31620         \exp_not:N \exp_after:wN
31621         \exp_not:N \__text_codepoint_from_chars_aux:Nw
31622         \exp_not:N \fi:
31623     }
31624     \exp_not:N \else:
31625     \exp_not:N \exp_after:wN \exp_not:N \__text_codepoint_from_chars:N
31626     \exp_not:N \fi:
31627     #1
31628 }
31629 \cs_new:Npn \__text_codepoint_from_chars_aux:Nw #1
31630 {
31631     \if_int_compare:w '#1 < "E0 \exp_stop_f:
31632     \exp_after:wN \__text_codepoint_from_chars:NN
31633     \else:
31634     \if_int_compare:w '#1 < "F0 \exp_stop_f:
31635     \exp_after:wN \exp_after:wN \exp_after:wN
31636     \__text_codepoint_from_chars:NNN
31637     \else:
31638     \exp_after:wN \exp_after:wN \exp_after:wN
31639     \__text_codepoint_from_chars:NNNN
31640     \fi:
31641     \fi:
31642     #1
31643 }
31644 \cs_new:Npn \__text_codepoint_from_chars:N #1 {'#1}
31645 \cs_new:Npn \__text_codepoint_from_chars:NN #1#2
31646 { ('#1 - "C0) * "40 + '#2 - "80 }
31647 \cs_new:Npn \__text_codepoint_from_chars:NNN #1#2#3
31648 { ('#1 - "E0) * "1000 + ('#2 - "80) * "40 + '#3 - "80 }
31649 \cs_new:Npn \__text_codepoint_from_chars:NNNN #1#2#3#4
31650 {
31651     ('#1 - "F0) * "40000
31652     + ('#2 - "80) * "1000
31653     + ('#3 - "80) * "40
31654     + '#4 - "80
31655 }
31656 }

```

(End of definition for __text_codepoint_compare:nNnTF and others.)

87.4 Configuration variables

`\l_text_accents_tl` Used to be used for excluding these ideas from expansion: now deprecated.
`\l_text_letterlike_tl`

```

31657 \tl_new:N \l_text_accents_tl
31658 \tl_new:N \l_text_letterlike_tl

```

(End of definition for `\l_text_accents_tl` and `\l_text_letterlike_tl`.)

`\l_text_case_exclude_arg_tl` Non-text arguments, including covering the case of `\protected@edef` applied to `\cite`.

```

31659 \tl_new:N \l_text_case_exclude_arg_tl
31660 \tl_set:Nc \l_text_case_exclude_arg_tl
31661   {
31662     \exp_not:n { \begin \cite \end \label \ref }
31663     \exp_not:c { cite ~ }
31664     \exp_not:n { \babelshorthand }
31665   }

```

(End of definition for `\l_text_case_exclude_arg_tl`. This variable is documented on page 299.)

`\l_text_math_arg_tl` Math mode as arguments.

```

31666 \tl_new:N \l_text_math_arg_tl
31667 \tl_set:Nn \l_text_math_arg_tl { \ensuremath }

```

(End of definition for `\l_text_math_arg_tl`. This variable is documented on page 299.)

`\l_text_math_delims_tl` Paired math mode delimiters.

```

31668 \tl_new:N \l_text_math_delims_tl
31669 \tl_set:Nn \l_text_math_delims_tl { $ $ \ ( \ ) }

```

(End of definition for `\l_text_math_delims_tl`. This variable is documented on page 299.)

`\l_text_expand_exclude_tl` Commands which need not to expand. We start with a somewhat historical list, and tidy up if possible.

```

31670 \tl_new:N \l_text_expand_exclude_tl
31671 \tl_set:Nn \l_text_expand_exclude_tl
31672   { \begin \cite \end \label \ref }
31673 \bool_lazy_and:nnT
31674   { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
31675   { \tl_if_exist_p:N \@expl@finalise@setup@@ }
31676   {
31677     \tl_gput_right:Nn \@expl@finalise@setup@@
31678     {
31679       \tl_gput_right:Nn \@kernel@after@begindocument
31680       {
31681         \group_begin:
31682         \cs_set_protected:Npn \__text_tmp:w #1
31683         {
31684           \tl_clear:N \l_text_expand_exclude_tl
31685           \tl_map_inline:nn {#1}
31686           {
31687             \bool_lazy_any:nF
31688             {
31689               { \token_if_protected_macro_p:N ##1 }
31690               { \token_if_protected_long_macro_p:N ##1 }
31691             }

```

```

31692         \str_if_eq_p:ee
31693         { \cs_replacement_spec:N ##1 }
31694         { \exp_not:n { \protect ##1 } \c_space_tl }
31695     }
31696 }
31697 { \tl_put_right:Nn \l_text_expand_exclude_tl {##1} }
31698 }
31699 }
31700 \exp_args:NV \__text_tmp:w \l_text_expand_exclude_tl
31701 \exp_args:NNNV \group_end:
31702 \tl_set:Nn \l_text_expand_exclude_tl \l_text_expand_exclude_tl
31703 }
31704 }
31705 }

```

(End of definition for `\l_text_expand_exclude_tl`. This variable is documented on page 299.)

`\l__text_math_mode_tl` Used to control math mode output: internal as there is a dedicated setter.

```
31706 \tl_new:N \l__text_math_mode_tl
```

(End of definition for `\l__text_math_mode_tl`.)

87.5 Expansion to formatted text

```

\c__text_chardef_space_token
\c__text_mathchardef_space_token
\c__text_chardef_group_begin_token
\c__text_mathchardef_group_begin_token
\c__text_chardef_group_end_token
\c__text_mathchardef_group_end_token

```

Markers for implicit char handling.

```

31707 \tex_global:D \tex_chardef:D \c__text_chardef_space_token = '\ %
31708 \tex_global:D \tex_mathchardef:D \c__text_mathchardef_space_token = '\ %
31709 \tex_global:D \tex_chardef:D \c__text_chardef_group_begin_token = '\{ % '\} '\{
31710 \tex_global:D \tex_mathchardef:D \c__text_mathchardef_group_begin_token = '\{ % '\} '\{
31711 \tex_global:D \tex_chardef:D \c__text_chardef_group_end_token = '\} % '\}
31712 \tex_global:D \tex_mathchardef:D \c__text_mathchardef_group_end_token = '\} %

```

(End of definition for `\c__text_chardef_space_token` and others.)

`\text_expand:n`

After precautions against `&` tokens, start a simple loop: that of course means that “text” cannot contain the two recursion quarks. The loop here must be f-type expandable; we have arbitrary user commands which might be protected *and* take arguments, and if the expansion code is used in a typesetting context, that will otherwise explode. (The same issue applies more clearly to case changing: see the example there.) The outer loop has to use scan marks as delimiters to protect against unterminated `\romannumeral` usage in the input.

```

\__text_expand:n
\__text_expand_result:n
\__text_expand_store:n
\__text_expand_store:o
\__text_expand_store:nw
\__text_expand_end:w
\__text_expand_loop:w
\__text_expand_group:n
\__text_expand_space:w
\__text_expand_N_type:N
\__text_expand_math_search:NNN
\__text_expand_math_loop:Nw
\__text_expand_math_N_type:NN
\__text_expand_math_group:Nn
\__text_expand_math_space:Nw
\__text_expand_explicit:N
\__text_expand_exclude:N
\__text_expand_exclude_switch:Nmnmn
\__text_expand_exclude:nN
\__text_expand_exclude:NN
\__text_expand_exclude:Nw
\__text_expand_exclude:Nnn
\__text_expand_accent:N
\__text_expand_accent:NN
\__text_expand_letterlike:N
\__text_expand_letterlike:NN
\__text_expand_cs:N

```

```

31713 \cs_new:Npn \text_expand:n #1
31714 {
31715     \__kernel_exp_not:w \exp_after:wN
31716     {
31717         \exp:w
31718         \__text_expand:n {#1}
31719     }
31720 }
31721 \cs_new:Npn \__text_expand:n #1
31722 {
31723     \group_align_safe_begin:
31724     \__text_expand_loop:w #1

```

```

31725     \s__text_recursion_tail \s__text_recursion_stop
31726     \__text_expand_result:n { }
31727 }

```

The approach to making the code f-type expandable is to use a marker result token and to shuffle the collected tokens

```

31728 \cs_new:Npn \__text_expand_store:n #1
31729 { \__text_expand_store:nw {#1} }
31730 \cs_generate_variant:Nn \__text_expand_store:n { o }
31731 \cs_new:Npn \__text_expand_store:nw #1#2 \__text_expand_result:n #3
31732 { #2 \__text_expand_result:n { #3 #1 } }
31733 \cs_new:Npn \__text_expand_end:w #1 \__text_expand_result:n #2
31734 {
31735     \group_align_safe_end:
31736     \exp_end:
31737     #2
31738 }

```

The main loop is a standard “tl action”; groups are handled recursively, while spaces are just passed through. Thus all of the action is in handling N-type tokens.

```

31739 \cs_new:Npn \__text_expand_loop:w #1 \s__text_recursion_stop
31740 {
31741     \tl_if_head_is_N_type:nTF {#1}
31742     { \__text_expand_N_type:N }
31743     {
31744         \tl_if_head_is_group:nTF {#1}
31745         { \__text_expand_group:n }
31746         { \__text_expand_space:w }
31747     }
31748     #1 \s__text_recursion_stop
31749 }
31750 \cs_new:Npn \__text_expand_group:n #1
31751 {
31752     \__text_expand_store:o
31753     {
31754         \exp_after:wN
31755         {
31756             \exp:w
31757             \__text_expand:n {#1}
31758         }
31759     }
31760     \__text_expand_loop:w
31761 }
31762 \exp_last_unbraced:NNo \cs_new:Npn \__text_expand_space:w \c_space_tl
31763 {
31764     \__text_expand_store:n { ~ }
31765     \__text_expand_loop:w
31766 }

```

The first step in dealing with N-type tokens is to look for math mode material: that needs to be left alone. The starting function has to be split into two as we need `\quark_if_recursion_tail_stop:N` first before we can trigger the search. We then look for matching pairs of delimiters, allowing for the case where math mode starts but does not end. Within math mode, we simply pass all the tokens through unchanged, just checking the N-type ones against the end marker.

```

31767 \cs_new:Npn \__text_expand_N_type:N #1
31768 {
31769   \__text_if_s_recursion_tail_stop_do:Nn #1
31770   { \__text_expand_end:w }
31771   \exp_after:wN \__text_expand_math_search:NNN
31772   \exp_after:wN #1 \l_text_math_delims_tl
31773   \q__text_recursion_tail \q__text_recursion_tail
31774   \q__text_recursion_stop
31775 }
31776 \cs_new:Npn \__text_expand_math_search:NNN #1#2#3
31777 {
31778   \__text_if_q_recursion_tail_stop_do:Nn #2
31779   { \__text_expand_explicit:N #1 }
31780   \token_if_eq_meaning:NNTF #1 #2
31781   {
31782     \__text_use_i_delimit_by_q_recursion_stop:nw
31783     {
31784       \__text_expand_store:n {#1}
31785       \__text_expand_math_loop:Nw #3
31786     }
31787   }
31788   { \__text_expand_math_search:NNN #1 }
31789 }
31790 \cs_new:Npn \__text_expand_math_loop:Nw #1#2 \s__text_recursion_stop
31791 {
31792   \tl_if_head_is_N_type:nTF {#2}
31793   { \__text_expand_math_N_type:NN }
31794   {
31795     \tl_if_head_is_group:nTF {#2}
31796     { \__text_expand_math_group:Nn }
31797     { \__text_expand_math_space:Nw }
31798   }
31799   #1#2 \s__text_recursion_stop
31800 }
31801 \cs_new:Npn \__text_expand_math_N_type:NN #1#2
31802 {
31803   \__text_if_s_recursion_tail_stop_do:Nn #2
31804   { \__text_expand_end:w }
31805   \token_if_eq_meaning:NNF #2 \exp_not:N
31806   { \__text_expand_store:n {#2} }
31807   \token_if_eq_meaning:NNTF #2 #1
31808   { \__text_expand_loop:w }
31809   { \__text_expand_math_loop:Nw #1 }
31810 }
31811 \cs_new:Npn \__text_expand_math_group:Nn #1#2
31812 {
31813   \__text_expand_store:n { {#2} }
31814   \__text_expand_math_loop:Nw #1
31815 }
31816 \exp_after:wN \cs_new:Npn \exp_after:wN \__text_expand_math_space:Nw
31817 \exp_after:wN # \exp_after:wN 1 \c_space_tl
31818 {
31819   \__text_expand_store:n { ~ }
31820   \__text_expand_math_loop:Nw #1

```

```
31821 }
```

At this stage, either we have a control sequence or a simple character: split and handle. The need to check for non-protected actives arises from handling of legacy input encodings: they need to end up in a representation we can deal with in further processing. The tests for explicit parts of the L^AT_EX 2_ε UTF-8 mechanism cover the case of bookmarks, where definitions change and are no longer protected. The same is true for babel shorthands.

```
31822 \cs_new:Npn \__text_expand_explicit:N #1
31823 {
31824   \token_if_cs:NTF #1
31825   { \__text_expand_exclude:N #1 }
31826   {
31827     \bool_lazy_and:nnTF
31828     { \token_if_active_p:N #1 }
31829     {
31830       ! \bool_lazy_any_p:n
31831       {
31832         { \token_if_protected_macro_p:N #1 }
31833         { \token_if_protected_long_macro_p:N #1 }
31834         { \tl_if_head_eq_meaning_p:oN {#1} \UTFviii@two@octets }
31835         { \tl_if_head_eq_meaning_p:oN {#1} \UTFviii@three@octets }
31836         { \tl_if_head_eq_meaning_p:oN {#1} \UTFviii@four@octets }
31837         { \tl_if_head_eq_meaning_p:oN {#1} \active@prefix }
31838       }
31839     }
31840     { \exp_after:wN \__text_expand_loop:w #1 }
31841     {
31842       \__text_expand_store:n {#1}
31843       \__text_expand_loop:w
31844     }
31845   }
31846 }
```

Next we exclude math commands: this is mainly as there *might* be an `\ensuremath`. The switching command for case needs special handling as it has to work by meaning.

```
31847 \cs_new:Npn \__text_expand_exclude:N #1
31848 {
31849   \cs_if_eq:NNTF #1 \text_case_switch:nnnn
31850   { \__text_expand_exclude_switch:Nnnnn #1 }
31851   {
31852     \exp_args:Ne \__text_expand_exclude:nN
31853     {
31854       \exp_not:V \l_text_math_arg_tl
31855       \exp_not:V \l_text_expand_exclude_tl
31856       \exp_not:V \l_text_case_exclude_arg_tl
31857     }
31858     #1
31859   }
31860 }
31861 \cs_new:Npn \__text_expand_exclude_switch:Nnnnn #1#2#3#4#5
31862 {
31863   \__text_expand_store:n { #1 {#2} {#3} {#4} {#5} }
31864   \__text_expand_loop:w
```



```

31865 }
31866 \cs_new:Npn \__text_expand_exclude:nN #1#2
31867 {
31868   \__text_expand_exclude:NN #2 #1
31869   \q__text_recursion_tail \q__text_recursion_stop
31870 }
31871 \cs_new:Npn \__text_expand_exclude:NN #1#2
31872 {
31873   \__text_if_q_recursion_tail_stop_do:Nn #2
31874   { \__text_expand_accent:N #1 }
31875   \str_if_eq:nnTF {#1} {#2}
31876   {
31877     \__text_use_i_delimit_by_q_recursion_stop:nw
31878     { \__text_expand_exclude:Nw #1 }
31879   }
31880   { \__text_expand_exclude:NN #1 }
31881 }
31882 \cs_new:Npn \__text_expand_exclude:Nw #1#2#
31883 { \__text_expand_exclude:Nnn #1 {#2} }
31884 \cs_new:Npn \__text_expand_exclude:Nnn #1#2#3
31885 {
31886   \__text_expand_store:n { #1#2 {#3} }
31887   \__text_expand_loop:w
31888 }

```

Accents.

```

31889 \cs_new:Npn \__text_expand_accent:N #1
31890 {
31891   \exp_after:wN \__text_expand_accent:NN \exp_after:wN
31892   #1 \l_text_accents_tl
31893   \q__text_recursion_tail \q__text_recursion_stop
31894 }
31895 \cs_new:Npn \__text_expand_accent:NN #1#2
31896 {
31897   \__text_if_q_recursion_tail_stop_do:Nn #2
31898   { \__text_expand_letterlike:N #1 }
31899   \cs_if_eq:NNTF #2 #1
31900   {
31901     \__text_use_i_delimit_by_q_recursion_stop:nw
31902     {
31903       \__text_expand_store:n {#1}
31904       \__text_expand_loop:w
31905     }
31906   }
31907   { \__text_expand_accent:NN #1 }
31908 }

```

Another list of exceptions: these ones take no arguments so are easier to handle.

```

31909 \cs_new:Npn \__text_expand_letterlike:N #1
31910 {
31911   \exp_after:wN \__text_expand_letterlike:NN \exp_after:wN
31912   #1 \l_text_letterlike_tl
31913   \q__text_recursion_tail \q__text_recursion_stop
31914 }
31915 \cs_new:Npn \__text_expand_letterlike:NN #1#2

```

```

31916 {
31917   \_text_if_q_recursion_tail_stop_do:Nn #2
31918   { \_text_expand_cs:N #1 }
31919   \cs_if_eq:NNTF #2 #1
31920   {
31921     \_text_use_i_delimit_by_q_recursion_stop:nw
31922     {
31923       \_text_expand_store:n {#1}
31924       \_text_expand_loop:w
31925     }
31926   }
31927   { \_text_expand_letterlike:NN #1 }
31928 }

```

LaTeX 2_ε's `\protect` makes life interesting. Where possible, we simply remove it and replace with the “parent” command; of course, the `\protect` might be explicit, in which case we need to leave it alone. That includes the case where it's not even followed by an N-type token. There is also the case of a straight `\@protected@testopt` to cover.

```

31929 \cs_new:Npe \_text_expand_cs:N #1
31930 {
31931   \exp_not:N \str_if_eq:nnTF {#1} { \exp_not:N \protect }
31932   { \exp_not:N \_text_expand_protect:w }
31933   {
31934     \bool_lazy_and:nnTF
31935     { \cs_if_exist_p:N \fmtname }
31936     { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
31937     { \exp_not:N \_text_expand_testopt:N #1 }
31938     { \exp_not:N \_text_expand_replace:N #1 }
31939   }
31940 }
31941 \cs_new:Npn \_text_expand_protect:w #1 \s__text_recursion_stop
31942 {
31943   \tl_if_head_is_N_type:nTF {#1}
31944   { \_text_expand_protect:N }
31945   {
31946     \_text_expand_store:n { \protect }
31947     \_text_expand_loop:w
31948   }
31949   #1 \s__text_recursion_stop
31950 }
31951 \cs_new:Npn \_text_expand_protect:N #1
31952 {
31953   \_text_if_s_recursion_tail_stop_do:Nn #1
31954   {
31955     \_text_expand_store:n { \protect }
31956     \_text_expand_end:w
31957   }
31958   \exp_args:Ne \_text_expand_protect:nN
31959   { \cs_to_str:N #1 } #1
31960 }
31961 \cs_new:Npn \_text_expand_protect:nN #1#2
31962 { \_text_expand_protect:Nw #2 #1 \q__text_nil #1 ~ \q__text_nil \q__text_nil \s__text_stop
31963 \cs_new:Npn \_text_expand_protect:Nw #1 #2 ~ \q__text_nil #3 \q__text_nil #4 \s__text_stop
31964 {

```

```

31965 \__text_quark_if_nil:nTF {#4}
31966 {
31967   \cs_if_exist:cTF {#2}
31968     { \exp_args:Ne \__text_expand_store:n { \exp_not:c {#2} } }
31969     { \__text_expand_store:n { \protect #1 } }
31970 }
31971 { \__text_expand_store:n { \protect #1 } }
31972 \__text_expand_loop:w
31973 }
31974 \cs_new:Npn \__text_expand_testopt:N #1
31975 {
31976   \token_if_eq_meaning:NNTF #1 \@protected@testopt
31977     { \__text_expand_testopt:NNn }
31978     { \__text_expand_encoding:N #1 }
31979 }
31980 \cs_new:Npn \__text_expand_testopt:NNn #1#2#3
31981 {
31982   \__text_expand_store:n {#1}
31983   \__text_expand_loop:w
31984 }

```

Deal with encoding-specific commands

```

31985 \cs_new:Npn \__text_expand_encoding:N #1
31986 {
31987   \bool_lazy_or:nnTF
31988     { \cs_if_eq_p:NN #1 \@current@cmd }
31989     { \cs_if_eq_p:NN #1 \@changed@cmd }
31990     { \exp_after:wN \__text_expand_loop:w \__text_expand_encoding_escape:NN }
31991     { \__text_expand_replace:N #1 }
31992 }
31993 \cs_new:Npn \__text_expand_encoding_escape:NN #1#2 { \exp_not:n {#1} }

```

See if there is a dedicated replacement, and if there is, insert it.

```

31994 \cs_new:Npn \__text_expand_replace:N #1
31995 {
31996   \bool_lazy_and:nnTF
31997     { \cs_if_exist_p:c { l__text_expand_ \token_to_str:N #1 _t1 } }
31998     {
31999       \bool_lazy_or_p:nn
32000         { \token_if_cs_p:N #1 }
32001         { \token_if_active_p:N #1 }
32002     }
32003   {
32004     \exp_args:Nv \__text_expand_replace:n
32005       { l__text_expand_ \token_to_str:N #1 _t1 }
32006   }
32007   { \__text_expand_cs_expand:N #1 }
32008 }
32009 \cs_new:Npn \__text_expand_replace:n #1 { \__text_expand_loop:w #1 }

```

Finally, expand any macros which can be: this then loops back around to deal with what they produce. The only issue is if the token is `\exp_not:n`, as that must apply to the following balanced text.

```

32010 \cs_new:Npn \__text_expand_cs_expand:N #1
32011 {

```

```

32012 \text_if_expandable:NTF #1
32013 {
32014 \token_if_eq_meaning:NNTF #1 \exp_not:n
32015 { \text_expand_unexpanded:w }
32016 { \exp_after:wN \text_expand_loop:w #1 }
32017 }
32018 {
32019 \text_expand_store:n {#1}
32020 \text_expand_loop:w
32021 }
32022 }

```

Since `\exp_not:n` is actually a primitive, it allows a strange syntax and its particular primitive expands what follows and discards spaces and `\scan_stop:` until finding a braced argument (the opening brace can be implicit but we will not support this here). Here, we repeatedly f-expand after such an `\exp_not:n`, and test what follows. If it is a brace group, then we found the intended argument of `\exp_not:n`. If it is a space, then the next f-expansion will eliminate it. If it is an N-type token then `\text_expand_unexpanded:N` leaves the token to be expanded if it is expandable, and otherwise removes it, assuming that it is `\scan_stop:`. This silently hides errors when `\exp_not:n` is incorrectly followed by some non-expandable token other than `\scan_stop:`, but this should be pretty rare, and there is no good error recovery anyways.

```

32023 \cs_new:Npn \text_expand_unexpanded:w
32024 {
32025 \exp_after:wN \text_expand_unexpanded_test:w
32026 \exp:w \exp_end_continue_f:w
32027 }
32028 \cs_new:Npn \text_expand_unexpanded_test:w #1 \s_text_recursion_stop
32029 {
32030 \tl_if_head_is_group:NTF {#1}
32031 { \text_expand_unexpanded:n }
32032 {
32033 \text_expand_unexpanded:w
32034 \tl_if_head_is_N_type:N {#1} { \text_expand_unexpanded:N }
32035 }
32036 #1 \s_text_recursion_stop
32037 }
32038 \cs_new:Npn \text_expand_unexpanded:N #1
32039 {
32040 \exp_after:wN \if_meaning:w \exp_not:N #1 #1
32041 \else:
32042 \exp_after:wN #1
32043 \fi:
32044 }
32045 \cs_new:Npn \text_expand_unexpanded:n #1
32046 {
32047 \text_expand_store:n {#1}
32048 \text_expand_loop:w
32049 }

```

(End of definition for `\text_expand:n` and others. This function is documented on page 296.)

`\text_declare_expand_equivalent:Nn`
`\text_declare_expand_equivalent:cn`

Create equivalents to allow replacement.

```

32050 \cs_new_protected:Npn \text_declare_expand_equivalent:Nn #1#2

```

```

32051 {
32052   \tl_clear_new:c { l__text_expand_ \token_to_str:N #1 _tl }
32053   \tl_set:cn { l__text_expand_ \token_to_str:N #1 _tl } {#2}
32054 }
32055 \cs_generate_variant:Nn \text_declare_expand_equivalent:Nn { c }

```

(End of definition for \text_declare_expand_equivalent:Nn. This function is documented on page 296.)

Prevent expansion of various standard values.

```

32056 \tl_map_inline:nn
32057 { \‘ \’ \^ \~ \= \u \. \” \r \H \v \d \c \k \b \t }
32058 { \text_declare_expand_equivalent:Nn #1 { \exp_not:n {#1} } }
32059 \tl_map_inline:nn
32060 {
32061   \AA \aa
32062   \AE \ae
32063   \DH \dh
32064   \DJ \dj
32065   \IJ \ij
32066   \L \l
32067   \NG \ng
32068   \O \o
32069   \OE \oe
32070   \SS \ss
32071   \TH \th
32072 }
32073 { \text_declare_expand_equivalent:Nn #1 { \exp_not:n {#1} } }
32074 </package>

```

Chapter 88

l3text-case implementation

```
32075 (*package)
```

```
32076 (@@=text)
```

88.1 Case changing

`\l_text_titlecase_check_letter_bool` Needed to determine the route used in titlecasing.

```
32077 \bool_new:N \l_text_titlecase_check_letter_bool
```

```
32078 \bool_set_true:N \l_text_titlecase_check_letter_bool
```

(End of definition for `\l_text_titlecase_check_letter_bool`. This variable is documented on page 299.)

`\text_lowercase:n`
`\text_uppercase:n` The user level functions here are all wrappers around the internal functions for case changing.

```
\text_titlecase_all:n 32079 \cs_new:Npn \text_lowercase:n #1
\text_titlecase_first:n 32080 { \__text_change_case:nnn { lower } { } {#1} }
\text_lowercase:nn 32081 \cs_new:Npn \text_uppercase:n #1
\text_uppercase:nn 32082 { \__text_change_case:nnn { upper } { } {#1} }
\text_titlecase_all:nn 32083 \cs_new:Npn \text_titlecase_all:n #1
\text_titlecase_first:nn 32084 { \__text_change_case:nnn { title } { } {#1} }
\__text_change_case:nnnn 32085 \cs_new:Npn \text_titlecase_first:n #1
32086 { \__text_change_case:nnnn { title } { break } { } {#1} }
32087 \cs_new:Npn \text_lowercase:nn #1#2
32088 { \__text_change_case:nnn { lower } {#1} {#2} }
32089 \cs_new:Npn \text_uppercase:nn #1#2
32090 { \__text_change_case:nnn { upper } {#1} {#2} }
32091 \cs_new:Npn \text_titlecase_all:nn #1#2
32092 { \__text_change_case:nnn { title } {#1} {#2} }
32093 \cs_new:Npn \text_titlecase_first:nn #1#2
32094 { \__text_change_case:nnnn { title } { break } {#1} {#2} }
32095 \cs_new:Npn \__text_change_case:nnn #1#2#3
32096 { \__text_change_case:nnnn {#1} {#1} {#2} {#3} }
```

(End of definition for `\text_lowercase:n` and others. These functions are documented on page 297.)

`__text_change_case:nnnn`
`__text_change_case_auxi:nnnn`
`__text_change_case_auxii:nnnn` As for the expansion code, the business end of case changing is the handling of N-type tokens. First, we expand the input fully (so the loops here don't need to worry about awkward look-aheads and the like). Then we split into the different paths.

```
\__text_change_case_BCP:nnnn
```

```
\__text_change_case_BCP:nnnw
```

```
\__text_change_case_BCP:nnnnnw
```

```
\__text_change_case_store:n
```

```
\__text_change_case_store:o
```

```
\__text_change_case_store:V
```

```
\__text_change_case_store:v
```

```
\__text_change_case_store:e
```

```
\__text_change_case_store:nw
```

```
\__text_change_case_result:n
```

The code here needs to be f-type expandable to deal with the situation where case changing is applied in running text. There, we might have case changing as a document command and the text containing other non-expandable document commands.

```
\cs_set_eq:NN \MakeLowercase \text_lowercase
...
\MakeLowercase{\enquote*{A} text}
```

If we use an e-type expansion and wrap each token in `\exp_not:n`, that would explode: the document command grabs `\exp_not:n` as an argument, and things go badly wrong. So we have to wrap the entire result in exactly one `\exp_not:n`, or rather in the kernel version.

```
32097 \cs_new:Npn \__text_change_case:nnnn #1#2#3#4
32098 {
32099   \__kernel_exp_not:w \exp_after:wN
32100   {
32101     \exp:w
32102     \exp_args:Ne \__text_change_case_auxi:nnnn
32103     { \text_expand:n {#4} }
32104     {#1} {#2} {#3}
32105   }
32106 }
32107 \cs_new:Npn \__text_change_case_auxi:nnnn #1#2#3#4
32108 {
32109   \exp_args:No \__text_change_case_BCP:nnnn
32110   { \tl_to_str:n {#4} } {#1} {#2} {#3}
32111 }
32112 \cs_new:Npe \__text_change_case_BCP:nnnn #1#2#3#4
32113 {
32114   \exp_not:N \__text_change_case_BCP:nnnw
32115   {#2} {#3} {#4} #1 \tl_to_str:n { -x- -x- } \exp_not:N \q__text_stop
32116 }
32117 \use:e
32118 {
32119   \cs_new:Npn \exp_not:N \__text_change_case_BCP:nnnw
32120   #1#2#3#4 \tl_to_str:n { -x- } #5 \tl_to_str:n { -x- } #6
32121   \exp_not:N \q__text_stop
32122 }
32123 { \__text_change_case_BCP:nnnnnw {#1} {#2} {#3} {#5} {#4} #4 - \q__text_stop }
32124 \cs_new:Npn \__text_change_case_BCP:nnnnnw #1#2#3#4#5#6 - #7 \q__text_stop
32125 {
32126   \bool_lazy_or:nnTF
32127   { \cs_if_exist_p:c { __text_change_case_ #2 _ #6 -x- #4 :nnnn } }
32128   { \tl_if_exist_p:c { l__text_ #2 case_special_ #6 -x- #4 _tl } }
32129   { \__text_change_case_auxii:nnnn {#1} {#2} {#3} { #6 -x- #4 } }
32130   {
32131     \cs_if_exist:cTF { __text_change_case_ #2 _ #6 :nnnn }
32132     { \__text_change_case_auxii:nnnn {#1} {#2} {#3} {#6} }
32133     { \__text_change_case_auxii:nnnn {#1} {#2} {#3} {#5} }
32134   }
32135 }
32136 \cs_new:Npn \__text_change_case_auxii:nnnn #1#2#3#4
32137 {
32138   \group_align_safe_begin:
```

```

32139 \cs_if_exist_use:c { __text_change_case_boundary_ #2 _ #4 :Nnnnw }
32140 \__text_change_case_loop:nnnw {#2} {#3} {#4} #1
32141 \q__text_recursion_tail \q__text_recursion_stop
32142 \__text_change_case_result:n { }
32143 }

```

As for expansion, collect up the tokens for future use.

```

32144 \cs_new:Npn \__text_change_case_store:n #1
32145 { \__text_change_case_store:nw {#1} }
32146 \cs_generate_variant:Nn \__text_change_case_store:n { o , e , V , v }
32147 \cs_new:Npn \__text_change_case_store:nw #1#2 \__text_change_case_result:n #3
32148 { #2 \__text_change_case_result:n { #3 #1 } }
32149 \cs_new:Npn \__text_change_case_end:w #1 \__text_change_case_result:n #2
32150 {
32151 \group_align_safe_end:
32152 \exp_end:
32153 #2
32154 }

```

The main loop is the standard tl action type.

```

32155 \cs_new:Npn \__text_change_case_loop:nnnw #1#2#3#4 \q__text_recursion_stop
32156 {
32157 \tl_if_head_is_N_type:nTF {#4}
32158 { \__text_change_case_N_type:nnnN }
32159 {
32160 \tl_if_head_is_group:nTF {#4}
32161 { \use:c { __text_change_case_group_ #1 :nnnn } }
32162 { \__text_change_case_space:nnnw }
32163 }
32164 {#1} {#2} {#3} #4 \q__text_recursion_stop
32165 }
32166 \cs_new:Npn \__text_change_case_break:w
32167 { \__text_change_case_break_aux:w \prg_do_nothing: }
32168 \cs_new:Npn \__text_change_case_break_aux:w
32169 #1 \q__text_recursion_tail \q__text_recursion_stop
32170 {
32171 \__text_change_case_store:o {#1}
32172 \__text_change_case_end:w
32173 }

```

For a group, we *could* worry about whether this contains a character or not. However, that would make life very complex for little gain: exactly what a first character is is rather weakly-defined anyway. So if there is a group, we simply assume that a character has been seen, and for title case we switch to the “rest of the tokens” situation. To avoid having too much testing, we use a two-step process here to allow the titlecase functions to be separate.

```

32174 \cs_new:Npn \__text_change_case_group_lower:nnnn #1#2#3#4
32175 {
32176 \__text_change_case_store:o
32177 {
32178 \exp_after:wN
32179 {
32180 \exp:w
32181 \__text_change_case_auxii:nnnn {#4} {#1} {#2} {#3}
32182 }

```



```

32183     }
32184     \__text_change_case_loop:nnnw {#1} {#2} {#3}
32185   }
32186 \cs_new_eq:NN \__text_change_case_group_upper:nnnn
32187   \__text_change_case_group_lower:nnnn
32188 \cs_new:Npn \__text_change_case_group_title:nnnn #1#2#3#4
32189   {
32190     \__text_change_case_store:o
32191     {
32192       \exp_after:wN
32193       {
32194         \exp:w
32195         \__text_change_case_auxii:nnnn {#4} {#1} {#2} {#3}
32196       }
32197     }
32198   \__text_change_case_skip:nnw {#2} {#3}
32199 }
32200 \use:e
32201 {
32202   \cs_new:Npn \exp_not:N \__text_change_case_space:nnnw #1#2#3 \c_space_tl
32203 }
32204 {
32205   \__text_change_case_store:n { ~ }
32206   \cs_if_exist_use:cF { __text_change_case_space_ #2 :nnn }
32207   {
32208     \cs_if_exist_use:c { __text_change_case_boundary_ #1 _ #3 :Nnnnw }
32209     \__text_change_case_loop:nnnw
32210   }
32211   {#2} {#2} {#3}
32212 }
32213 \cs_new:Npn \__text_change_case_space_break:nnn #1#2#3
32214   { \__text_change_case_break:w }

```

The first step of handling N-type tokens is to filter out the end-of-loop. That has to be done separately from the first real step as otherwise we pick up the wrong delimiter. The loop here is the same as the `expand` one, just passing the additional data long. If no close-math token is found then the final clean-up is forced (i.e. there is no assumption of “well-behaved” input in terms of math mode).

```

32215 \cs_new:Npn \__text_change_case_N_type:nnnN #1#2#3#4
32216   {
32217     \__text_if_q_recursion_tail_stop_do:Nn #4
32218     { \__text_change_case_end:w }
32219     \__text_change_case_N_type_aux:nnnN {#1} {#2} {#3} #4
32220   }
32221 \cs_new:Npn \__text_change_case_N_type_aux:nnnN #1#2#3#4
32222   {
32223     \exp_args:NV \__text_change_case_N_type:nnnnN
32224     \l_text_math_delims_tl {#1} {#2} {#3} #4
32225   }
32226 \cs_new:Npn \__text_change_case_N_type:nnnnN #1#2#3#4#5
32227   {
32228     \__text_change_case_math_search:nnnNNN {#2} {#3} {#4} #5 #1
32229     \q__text_recursion_tail \q__text_recursion_tail
32230     \q__text_recursion_stop

```

```

32231 }
32232 \cs_new:Npn \__text_change_case_math_search:nnnNNN #1#2#3#4#5#6
32233 {
32234   \__text_if_q_recursion_tail_stop_do:Nn #5
32235   { \__text_change_case_cs_check:nnnN {#1} {#2} {#3} #4 }
32236   \token_if_eq_meaning:NNTF #4 #5
32237   {
32238     \__text_use_i_delimit_by_q_recursion_stop:nw
32239     {
32240       \__text_change_case_store:n {#4}
32241       \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #6
32242     }
32243   }
32244   { \__text_change_case_math_search:nnnNNN {#1} {#2} {#3} #4 }
32245 }
32246 \cs_new:Npn \__text_change_case_math_loop:nnnNw #1#2#3#4#5 \q__text_recursion_stop
32247 {
32248   \tl_if_head_is_N_type:nTF {#5}
32249   { \__text_change_case_math_N_type:nnnNN }
32250   {
32251     \tl_if_head_is_group:nTF {#5}
32252     { \__text_change_case_math_group:nnnNn }
32253     { \__text_change_case_math_space:nnnNw }
32254   }
32255   {#1} {#2} {#3} #4 #5 \q__text_recursion_stop
32256 }
32257 \cs_new:Npn \__text_change_case_math_N_type:nnnNN #1#2#3#4#5
32258 {
32259   \__text_if_q_recursion_tail_stop_do:Nn #5
32260   { \__text_change_case_end:w }
32261   \__text_change_case_store:n {#5}
32262   \token_if_eq_meaning:NNTF #5 #4
32263   { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
32264   { \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #4 }
32265 }
32266 \cs_new:Npn \__text_change_case_math_group:nnnNn #1#2#3#4#5
32267 {
32268   \__text_change_case_store:n { {#5} }
32269   \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #4
32270 }
32271 \use:e
32272 {
32273   \cs_new:Npn \exp_not:N \__text_change_case_math_space:nnnNw #1#2#3#4
32274   \c_space_tl
32275 }
32276 {
32277   \__text_change_case_store:n { ~ }
32278   \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #4
32279 }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: the two routes the code may take are then very different.

```

32280 \cs_new:Npn \__text_change_case_cs_check:nnnN #1#2#3#4
32281 {

```

```

32282 \token_if_cs:NTF #4
32283   { \__text_change_case_exclude:nnnN {#1} {#2} {#3} }
32284   {
32285     \__text_codepoint_process:nN
32286     { \use:c { \__text_change_case_custom_ #1 :nnnn } {#1} {#2} {#3} }
32287   }
32288   #4
32289 }

```

To deal with a control sequence there is first a need to test if it is on the list which indicate that case changing should be skipped. That's done using a loop as for the other special cases. If a hit is found then the argument is grabbed and passed through as-is.

```

32290 \cs_new:Npn \__text_change_case_exclude:nnnN #1#2#3#4
32291   {
32292     \exp_args:Ne \__text_change_case_exclude:nnnnN
32293     {
32294       \exp_not:V \l_text_math_arg_tl
32295       \exp_not:V \l_text_case_exclude_arg_tl
32296     }
32297     {#1} {#2} {#3} #4
32298   }
32299 \cs_new:Npn \__text_change_case_exclude:nnnnN #1#2#3#4#5
32300   {
32301     \__text_change_case_exclude:nnnNN {#2} {#3} {#4} #5 #1
32302     \q__text_recursion_tail \q__text_recursion_stop
32303   }
32304 \cs_new:Npn \__text_change_case_exclude:nnnNN #1#2#3#4#5
32305   {
32306     \__text_if_q_recursion_tail_stop_do:Nn #5
32307     { \__text_change_case_replace:nnnN {#1} {#2} {#3} #4 }
32308     \str_if_eq:nnTF {#4} {#5}
32309     {
32310       \__text_use_i_delimit_by_q_recursion_stop:nw
32311       { \__text_change_case_exclude:nnnNw {#1} {#2} {#3} #4 }
32312     }
32313     { \__text_change_case_exclude:nnnNN {#1} {#2} {#3} #4 }
32314   }
32315 \cs_new:Npn \__text_change_case_exclude:nnnNw #1#2#3#4#5#
32316   { \__text_change_case_exclude:nnnNnn {#1} {#2} {#3} {#4} {#5} }
32317 \cs_new:Npn \__text_change_case_exclude:nnnNnn #1#2#3#4#5#6
32318   {
32319     \tl_if_blank:nTF {#5}
32320     { \__text_change_case_store:n { #4 {#6} } }
32321     {
32322       \__text_change_case_store:o
32323       {
32324         \exp_after:wN #4
32325         \exp:w \__text_change_case_auxii:nnnn {#5} {#1} {#2} {#3}
32326         {#6}
32327       }
32328     }
32329     \__text_change_case_loop:nnnw {#1} {#2} {#3}
32330   }

```

Deal with any specialist replacement for case changing.

```

32331 \cs_new:Npn \__text_change_case_replace:nnnN #1#2#3#4
32332 {
32333   \cs_if_exist:cTF { l__text_case_ \token_to_str:N #4 _tl }
32334   {
32335     \__text_change_case_replace:vnnn
32336     { l__text_case_ \token_to_str:N #4 _tl } {#1} {#2} {#3}
32337   }
32338   { \__text_change_case_switch:nnnN {#1} {#2} {#3} #4 }
32339 }
32340 \cs_new:Npn \__text_change_case_replace:nnnn #1#2#3#4
32341 { \__text_change_case_loop:nnnw {#2} {#3} {#4} #1 }
32342 \cs_generate_variant:Nn \__text_change_case_replace:nnnn { v }

```

Allow for manually-controlled case switching.

```

32343 \cs_new:Npn \__text_change_case_switch:nnnN #1#2#3#4
32344 {
32345   \cs_if_eq:NNTF #4 \text_case_switch:nnnn
32346   { \use:c { __text_change_case_switch_ #1 :nnnNnnnn } }
32347   { \use:c { __text_change_case_letterlike_ #1 :nnnN } }
32348   {#1} {#2} {#3} #4
32349 }
32350 \cs_new:Npn \__text_change_case_switch_lower:nnnNnnnn #1#2#3#4#5#6#7#8
32351 {
32352   \__text_change_case_store:n {#7}
32353   \__text_change_case_loop:nnnw {#1} {#2} {#3}
32354 }
32355 \cs_new:Npn \__text_change_case_switch_upper:nnnNnnnn #1#2#3#4#5#6#7#8
32356 {
32357   \__text_change_case_store:n {#6}
32358   \__text_change_case_loop:nnnw {#1} {#2} {#3}
32359 }
32360 \cs_new:Npn \__text_change_case_switch_title:nnnNnnnn #1#2#3#4#5#6#7#8
32361 {
32362   \__text_change_case_store:n {#8}
32363   \__text_change_case_skip:nnw {#2} {#3}
32364 }

```

Skip over material quickly after titlecase-first-only initials

```

32365 \cs_new:Npn \__text_change_case_skip:nnw #1#2#3 \q__text_recursion_stop
32366 {
32367   \tl_if_head_is_N_type:nTF {#3}
32368   { \__text_change_case_skip_N_type:nnN }
32369   {
32370     \tl_if_head_is_group:nTF {#3}
32371     { \__text_change_case_skip_group:nnn }
32372     { \__text_change_case_skip_space:nnw }
32373   }
32374   {#1} {#2} #3 \q__text_recursion_stop
32375 }
32376 \cs_new:Npn \__text_change_case_skip_N_type:nnN #1#2#3
32377 {
32378   \__text_if_q_recursion_tail_stop_do:Nn #3
32379   { \__text_change_case_end:w }
32380   \__text_change_case_store:n {#3}
32381   \__text_change_case_skip:nnw {#1} {#2}

```

```

32382 }
32383 \cs_new:Npn \__text_change_case_skip_group:nnn #1#2#3
32384 {
32385   \__text_change_case_store:n { {#3} }
32386   \__text_change_case_skip:nnw {#1} {#2}
32387 }
32388 \cs_new:Npn \__text_change_case_skip_space:nnw #1#2
32389 { \__text_change_case_space:nnnw {#1} {#1} {#2} }

```

Letter-like commands may still be present: they are set up using a simple lookup approach, so can easily be handled with no loop. If there is no hit, we are at the end of the process: we loop around. Letter-like chars are all available only in upper- and lowercase, so titlecasing maps to the uppercase version.

```

32390 \cs_new:Npn \__text_change_case_letterlike_lower:nnnN #1#2#3#4
32391 { \__text_change_case_letterlike:nnnnN {#1} {#1} {#1} {#2} {#3} #4 }
32392 \cs_new_eq:NN \__text_change_case_letterlike_upper:nnnN
32393   \__text_change_case_letterlike_lower:nnnN
32394 \cs_new:Npn \__text_change_case_letterlike_title:nnnN #1#2#3#4
32395 { \__text_change_case_letterlike:nnnnN { upper } { end } {#1} {#2} {#3} #4 }
32396 \cs_new:Npn \__text_change_case_letterlike:nnnnN #1#2#3#4#5#6
32397 {
32398   \cs_if_exist:cTF { c__text_ #1 case_ \token_to_str:N #6 _tl }
32399   {
32400     \__text_change_case_store:v
32401     { c__text_ #1 case_ \token_to_str:N #6 _tl }
32402     \use:c { __text_change_case_next_ #2 :nnn } {#2} {#4} {#5}
32403   }
32404   {
32405     \__text_change_case_store:n {#6}
32406     \cs_if_exist:cTF
32407     {
32408       c__text_
32409       \str_if_eq:nnTF {#1} { lower } { upper } { lower }
32410       case_ \token_to_str:N #6 _tl
32411     }
32412     { \use:c { __text_change_case_next_ #2 :nnn } {#2} {#4} {#5} }
32413     { \__text_change_case_loop:nnnw {#3} {#4} {#5} }
32414   }
32415 }

```

Check for a customised codepoint result.

```

32416 \cs_new:Npn \__text_change_case_custom_lower:nnnn #1#2#3#4
32417 {
32418   \__text_change_case_custom:nnnnn {#1} {#1} {#2} {#3} {#4}
32419   { \use:c { __text_change_case_codepoint_ #1 :nnnn } {#1} {#2} {#3} {#4} }
32420 }
32421 \cs_new_eq:NN \__text_change_case_custom_upper:nnnn
32422   \__text_change_case_custom_lower:nnnn
32423 \cs_new:Npn \__text_change_case_custom_title:nnnn #1#2#3#4
32424 {
32425   \__text_change_case_custom:nnnnn { title } {#1} {#2} {#3} {#4}
32426   {
32427     \__text_change_case_custom:nnnnn { upper } {#1} {#2} {#3} {#4}
32428     { \use:c { __text_change_case_codepoint_ #1 :nnnn } {#1} {#2} {#3} {#4} }
32429   }

```

```

32430 }
32431 \cs_new:Npn \__text_change_case_custom:nnnnn #1#2#3#4#5#6
32432 {
32433   \tl_if_exist:cTF { l__text_ #1 case _ \tl_to_str:n {#5} _ #4 _tl }
32434   {
32435     \__text_change_case_replace:vnnc
32436     { l__text_ #1 case _ \tl_to_str:n {#5} _ #4 _tl } {#2} {#3} {#4}
32437   }
32438   {
32439     \tl_if_exist:cTF { l__text_ #1 case _ \tl_to_str:n {#5} _tl }
32440     {
32441       \__text_change_case_replace:vnnc
32442       { l__text_ #1 case _ \tl_to_str:n {#5} _tl } {#2} {#3} {#4}
32443     }
32444     {#6}
32445   }
32446 }

```

For upper- and lowercase changes, once we get to this stage there are only a couple of questions remaining: is there a language-specific mapping and is there the special case of a terminal sigma. If not, then we pass to a simple codepoint mapping.

```

32447 \cs_new:Npn \__text_change_case_codepoint_lower:nnnn #1#2#3#4
32448 {
32449   \cs_if_exist_use:cF { __text_change_case_lower_ #3 :nnnnn }
32450   { \__text_change_case_lower_sigma:nnnnn }
32451   {#1} {#1} {#2} {#3} {#4}
32452 }
32453 \cs_new:Npn \__text_change_case_codepoint_upper:nnnn #1#2#3#4
32454 {
32455   \cs_if_exist_use:cF { __text_change_case_upper_ #3 :nnnnn }
32456   { \__text_change_case_codepoint:nnnnn }
32457   {#1} {#1} {#2} {#3} {#4}
32458 }

```

If the current character is an uppercase sigma, the a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase: the logic here is simply based on letters or actives (to cover 8-bit engines).

```

32459 \cs_new:Npn \__text_change_case_lower_sigma:nnnnn #1#2#3#4#5
32460 {
32461   \__text_codepoint_compare:nNnTF {#5} = { "03A3 }
32462   { \__text_change_case_lower_sigma:nnnnw {#2} }
32463   { \__text_change_case_codepoint:nnnnn {#1} {#2} }
32464   {#3} {#4} {#5}
32465 }
32466 \cs_new:Npn \__text_change_case_lower_sigma:nnnnw #1#2#3#4#5 \q__text_recursion_stop
32467 {
32468   \tl_if_head_is_N_type:nTF {#5}
32469   { \__text_change_case_lower_sigma:nnnnN {#4} }
32470   {
32471     \__text_change_case_store:e
32472     { \codepoint_generate:nn { "03C2 } { \__text_char_catcode:N #4 } }
32473     \__text_change_case_loop:nnnw
32474   }
32475   {#1} {#2} {#3} #5 \q__text_recursion_stop
32476 }

```

```

32477 \cs_new:Npn \__text_change_case_lower_sigma:nnnnN #1#2#3#4#5
32478 {
32479   \__text_change_case_store:e
32480   {
32481     \bool_lazy_or:nnTF
32482     { \token_if_letter_p:N #5 }
32483     {
32484       \bool_lazy_and_p:nn
32485       { \token_if_active_p:N #5 }
32486       { \int_compare_p:nNn {'#5} > { "80 } }
32487     }
32488     { \codepoint_generate:nn { "03C3 } { \__text_char_catcode:N #1 } }
32489     { \codepoint_generate:nn { "03C2 } { \__text_char_catcode:N #1 } }
32490   }
32491   \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
32492 }

```

For titlecasing, we need to obtain the general category of the current codepoint.

```

32493 \cs_new:Npn \__text_change_case_codepoint_title:nnnn #1#2#3#4
32494 {
32495   \bool_if:NTF \l_text_titlecase_check_letter_bool
32496   {
32497     \exp_args:Ne \__text_change_case_codepoint_title_auxi:nnnn
32498     {
32499       \codepoint_to_category:n
32500       { \__text_codepoint_from_chars:Nw #4 }
32501     }
32502   }
32503   { \__text_change_case_codepoint_title:nnn }
32504   {#2} {#3} {#4}
32505 }
32506 \cs_new:Npn \__text_change_case_codepoint_title_auxi:nnnn #1#2#3#4
32507 {
32508   \tl_if_head_eq_charcode:nNTF {#1} { L }
32509   { \__text_change_case_codepoint_title:nnn }
32510   { \__text_change_case_codepoint_title_auxii:nnnn { title } }
32511   {#2} {#3} {#4}
32512 }
32513 \cs_new:Npn \__text_change_case_codepoint_title:nnn #1#2#3
32514 { \__text_change_case_codepoint_title_auxii:nnnn { end } {#1} {#2} {#3} }
32515 \cs_new:Npn \__text_change_case_codepoint_title_auxii:nnnn #1#2#3#4
32516 {
32517   \cs_if_exist_use:cF { __text_change_case_title_ #3 :nnnn }
32518   {
32519     \cs_if_exist_use:cF { __text_change_case_upper_ #3 :nnnn }
32520     { \__text_change_case_codepoint:nnnn }
32521   }
32522   { title } {#1} {#2} {#3} {#4}
32523 }
32524 \cs_new:Npn \__text_change_case_codepoint:nnnnn #1#2#3#4#5
32525 {
32526   \bool_lazy_and:nnTF
32527   { \tl_if_single_p:n {#5} }
32528   { \token_if_active_p:N #5 }
32529   { \__text_change_case_store:n {#5} }

```

```

32530     {
32531         \__text_change_case_store:e
32532         { \__text_change_case_codepoint:nn {#1} {#5} }
32533     }
32534     \use:c { \__text_change_case_next_ #2 :nnn } {#2} {#3} {#4}
32535 }
32536 \cs_new:Npn \__text_change_case_codepoint:nn #1#2
32537 {
32538     \__text_change_case_codepoint:fnn
32539     { \int_eval:n { \__text_codepoint_from_chars:Nw #2 } } {#1} {#2}
32540 }
32541 \cs_new:Npn \__text_change_case_codepoint:nnn #1#2#3
32542 {
32543     \exp_args:Ne \__text_change_case_codepoint_aux:nn
32544     { \__kernel_codepoint_case:nn { #2 case } {#1} } {#3}
32545 }
32546 \cs_generate_variant:Nn \__text_change_case_codepoint:nnn { f }

```

Avoid high chars with pTeX.

```

32547 \sys_if_engine_ptex:T
32548 {
32549     \cs_new_eq:NN \__text_change_case_codepoint_aux:nnn
32550     \__text_change_case_codepoint:nnn
32551     \cs_gset:Npn \__text_change_case_codepoint:nnn #1#2#3
32552     {
32553         \int_compare:nNnTF {#1} = { -1 }
32554         { \exp_not:n {#3} }
32555         { \__text_change_case_codepoint_aux:nnn {#1} {#2} {#3} }
32556     }
32557 }
32558 \cs_new:Npn \__text_change_case_codepoint_aux:nn #1#2
32559 {
32560     \use:e { \__text_change_case_codepoint_aux:nnnn #1 {#2} }
32561 }
32562 \cs_new:Npn \__text_change_case_codepoint_aux:nnnn #1#2#3#4
32563 {
32564     \__text_codepoint_compare:nNnTF {#4} = {#1}
32565     { \exp_not:n {#4} }
32566     {
32567         \codepoint_generate:nn {#1}
32568         { \__text_change_case_catcode:nn {#4} {#1} }
32569         \tl_if_blank:nF {#2}
32570         {
32571             \codepoint_generate:nn {#2}
32572             { \char_value_catcode:n {#2} }
32573             \tl_if_blank:nF {#3}
32574             {
32575                 \codepoint_generate:nn {#3}
32576                 { \char_value_catcode:n {#3} }
32577             }
32578         }
32579     }
32580 }

```

We need to ensure that only valid catcode-extraction is attempted. That's fine with

Unicode engines but needs a bit of work with 8-bit ones. The logic is that if the original codepoint was in the ASCII range, we keep the catcode. Otherwise, if the target is in the ASCII range, we use the standard catcode. If neither are true, we set as 13 on the grounds that this will be what is used anyway!

```

32581 \bool_lazy_or:nNTF
32582 { \sys_if_engine_luatex_p: }
32583 { \sys_if_engine_xetex_p: }
32584 {
32585   \cs_new:Npn \__text_change_case_catcode:nm #1#2
32586     { \__text_char_catcode:N #1 }
32587 }
32588 {
32589   \cs_new:Npn \__text_change_case_catcode:nm #1#2
32590     {
32591       \__text_codepoint_compare:nNnTF {#1} < { "80 }
32592       { \__text_char_catcode:N #1 }
32593       {
32594         \int_compare:nNnTF {#2} < { "80 }
32595         { \char_value_catcode:n {#2} }
32596         { 13 }
32597       }
32598     }
32599 }
32600 \cs_new:Npn \__text_change_case_next_lower:nnn #1#2#3
32601   { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
32602 \cs_new_eq:NN \__text_change_case_next_upper:nnn
32603   \__text_change_case_next_lower:nnn
32604 \cs_new_eq:NN \__text_change_case_next_title:nnn
32605   \__text_change_case_next_lower:nnn
32606 \cs_new:Npn \__text_change_case_next_end:nnn #1#2#3
32607   { \__text_change_case_skip:nnw {#2} {#3} }

```

(End of definition for `__text_change_case:nnnn` and others.)

`\text_declare_case_equivalent:Nn` Create equivalents to allow replacement.

```

32608 \cs_new_protected:Npn \text_declare_case_equivalent:Nn #1#2
32609   {
32610     \tl_clear_new:c { l__text_case_ \token_to_str:N #1 _tl }
32611     \tl_set:cn { l__text_case_ \token_to_str:N #1 _tl } {#2}
32612   }

```

(End of definition for `\text_declare_case_equivalent:Nn`. This function is documented on page 298.)

`\text_declare_lowercase_mapping:nn` Codepoint customisation.

```

\text_declare_titlecase_mapping:nn
\text_declare_uppercase_mapping:nn
\__text_declare_case_mapping:nnn
\__text_declare_case_mapping_aux:nnn
\text_declare_lowercase_mapping:nnn
\text_declare_titlecase_mapping:nnn
\text_declare_uppercase_mapping:nnn
\__text_declare_case_mapping:nnnn
\__text_declare_case_mapping_aux:nnnn
32613 \cs_new_protected:Npn \text_declare_lowercase_mapping:nn #1#2
32614   { \__text_declare_case_mapping:nnn { lower } {#1} {#2} }
32615 \cs_new_protected:Npn \text_declare_titlecase_mapping:nn #1#2
32616   { \__text_declare_case_mapping:nnn { title } {#1} {#2} }
32617 \cs_new_protected:Npn \text_declare_uppercase_mapping:nn #1#2
32618   { \__text_declare_case_mapping:nnn { upper } {#1} {#2} }
32619 \cs_new_protected:Npn \__text_declare_case_mapping:nnn #1#2#3
32620   {
32621     \exp_args:Ne \__text_declare_case_mapping_aux:nnn
32622     { \codepoint_str_generate:n {#2} } {#1} {#3}

```

```

32623 }
32624 \cs_new_protected:Npn \__text_declare_case_mapping_aux:nnn #1#2#3
32625 {
32626   \tl_clear_new:c { l__text_ #2 case _ #1 _tl }
32627   \tl_set:cn { l__text_ #2 case _ #1 _tl } {#3}
32628 }
32629 \cs_new_protected:Npn \text_declare_lowercase_mapping:nnn #1#2#3
32630 { \__text_declare_case_mapping:nnnn { lower } {#1} {#2} {#3} }
32631 \cs_new_protected:Npn \text_declare_titlecase_mapping:nnn #1#2#3
32632 { \__text_declare_case_mapping:nnnn { title } {#1} {#2} {#3} }
32633 \cs_new_protected:Npn \text_declare_uppercase_mapping:nnn #1#2#3
32634 { \__text_declare_case_mapping:nnnn { upper } {#1} {#2} {#3} }
32635 \cs_new_protected:Npn \__text_declare_case_mapping:nnnn #1#2#3#4
32636 {
32637   \exp_args:Ne \__text_declare_case_mapping_aux:nnnn
32638   { \codepoint_str_generate:n {#3} } {#1} {#2} {#4}
32639 }
32640 \cs_new_protected:Npn \__text_declare_case_mapping_aux:nnnn #1#2#3#4
32641 {
32642   \tl_clear_new:c { l__text_ #2 case _ #1 _ #3 _tl }
32643   \tl_set:cn { l__text_ #2 case _ #1 _ #3 _tl } {#4}
32644   \tl_clear_new:c { l__text_ #2 case_special_ #3 _tl }
32645 }

```

(End of definition for `\text_declare_lowercase_mapping:nn` and others. These functions are documented on page 298.)

`\text_case_switch:nnnn` Set up the mechanism for manual case switching.

```

\__text_case_switch_marker: 32646 \cs_new:Npn \text_case_switch:nnnn #1#2#3#4
32647 {
32648   \__text_case_switch_marker:
32649   #1
32650 }
32651 \cs_new:Npn \__text_case_switch_marker: { }

```

(End of definition for `\text_case_switch:nnnn` and `__text_case_switch_marker:.` This function is documented on page 298.)

`__text_change_case_generate:n` A utility.

```

32652 \cs_new:Npn \__text_change_case_generate:n #1
32653 { \codepoint_generate:nn {#1} { \char_value_catcode:n {#1} } }

```

(End of definition for `__text_change_case_generate:n`.)

`__text_change_case_upper_de-x-eszett:nnnn` A simple alternative version for German.

```

\__text_change_case_upper_de-alt:nnnn 32654 \cs_new:cpn { __text_change_case_upper_de-x-eszett:nnnn } #1#2#3#4#5
32655 {
32656   \__text_codepoint_compare:nNnTF {#5} = { "00DF }
32657   {
32658     \__text_change_case_store:e
32659     {
32660       \codepoint_generate:nn { "1E9E }
32661       { \__text_change_case_catcode:nn {#5} { "1E9E } }
32662     }
32663     \use:c { __text_change_case_next_ #2 :nnn }

```

```

32664         {#2} {#3} {#4}
32665     }
32666     { \_text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
32667 }
32668 \cs_new_eq:cc { \_text_change_case_upper_de-alt:nnnnn }
32669 { \_text_change_case_upper_de-x-eszett:nnnnn }

```

(End of definition for _text_change_case_upper_de-x-eszett:nnnnn and _text_change_case_upper_de-alt:nnnnn.)

```

\_text_change_case_upper_el:nnnnn
\_text_change_case_upper_el-x-iota:nnnnn
\_text_change_case_upper_el_aux:nnnnn
\_text_change_case_upper_el:nnnn
\_text_change_case_upper_el:nnnnw
\_text_change_case_upper_el:nnnnN
\_text_change_case_upper_el_aux:nnnnN
change_case_upper_el ypogegrammeni:nnnnnnw
change_case_upper_el ypogegrammeni:nnnnnnN
change_case_upper_el ypogegrammeni:nnnnnnn
\_text_change_case_upper_el dialytika:nnnn
\_text_change_case_upper_el dialytika:n
\_text_change_case_upper_el hiatus:nnnnw
\_text_change_case_upper_el hiatus:nnnnN
\_text_change_case_upper_el hiatus:nnnnn
\_text_change_case_upper_el ypogegrammeni:n
change_case_upper_el-x-iota ypogegrammeni:n
\_text_change_case_upper_el_stress:nn
\_text_change_case_upper_el_gobble:nnnw
\_text_change_case_upper_el_gobble:nnnN
\_text_change_case_upper_el_gobble:nnnn
\_text_change_case_if_greek:n
\_text_change_case_if_greek:nTF
xt_change_case_if_greek_spacing_diacritic:n
change_case_if_greek_spacing_diacritic:nTF
\_text_change_case_if_greek_accent:n
\_text_change_case_if_greek_accent:nTF
\_text_change_case_if_greek_breathing:n
\_text_change_case_if_greek_breathing:nTF
\_text_change_case_if_greek_stress:n
\_text_change_case_if_greek_stress:nTF
\_text_change_case_if_takes_dialytika:n
\_text_change_case_if_takes_dialytika:nTF
\_text_change_case_if_takes ypogegrammeni:n
text_change_case_if_takes ypogegrammeni:nTF

```

For Greek uppercasing, we need to know if characters *in the Greek range* have accents. That means doing a NFD conversion first, then starting a search. As described by the Unicode CLDR, Greek accents need to be found *after* any U+0308 (diaeresis) and are done in two groups to allow for the canonical ordering. The implementation here follows the data and examples from ICU (<https://icu.unicode.org/design/case/greek-upper>), although necessarily the implementation is somewhat different. The *ypogegrammeni* is filtered out here as it is not actually in the Greek range, so gets lost if we leave until later. The one Greek codepoint we skip is the numeral sign and question mark: the first has an awkward NFD for pdfTeX so is best left unchanged, and the latter has issues concerning how LGR outputs the input and output (differently!).

```

32670 \cs_new:Npn \_text_change_case_upper_el:nnnnn #1#2#3#4#5
32671 {
32672     \bool_lazy_and:nnTF
32673     { \_text_change_case_if_greek_p:n {#5} }
32674     {
32675         ! \bool_lazy_or_p:nn
32676         { \_text_codepoint_compare_p:nNn {#5} = { "0374 } }
32677         { \_text_codepoint_compare_p:nNn {#5} = { "037E } }
32678     }
32679     {
32680         \_text_change_case_if_greek_spacing_diacritic:nTF {#5}
32681         {
32682             \_text_change_case_store:n {#5}
32683             \_text_change_case_loop:nnnw
32684         }
32685         {
32686             \exp_args:Ne \_text_change_case_upper_el:nnnn
32687             {
32688                 \codepoint_to_nfd:n
32689                 { \_text_codepoint_from_chars:Nw #5 }
32690             }
32691         }
32692         {#2} {#3} {#4}
32693     }
32694     {
32695         \_text_codepoint_compare:nNnTF {#5} = { "0345 }
32696         {
32697             \_text_change_case_store:e
32698             {
32699                 \codepoint_generate:nn { "0399 }
32700                 { \char_value_catcode:n { "0399 } }
32701             }
32702             \_text_change_case_loop:nnnw {#2} {#3} {#4}
32703         }

```

```

32704         { \_text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
32705     }
32706 }
32707 \cs_new_eq:cN { \_text_change_case_upper_el-x-iota:nnnnn }
32708   \_text_change_case_upper_el:nnnnn
32709 \cs_new:Npn \_text_change_case_upper_el:nnnn #1#2#3#4
32710 {
32711   \_text_codepoint_process:nN
32712     { \_text_change_case_upper_el:nnnnw {#2} {#3} {#4} } #1
32713 }

```

At this stage we have the first NFD codepoint as #3. What we need to know is whether after that we have another character, either from the NFD or directly in the input. If not, we store the changed character at this stage.

```

32714 \cs_new:Npn \_text_change_case_upper_el:nnnnw #1#2#3#4#5 \q_text_recursion_stop
32715 {
32716   \tl_if_head_is_N_type:nTF {#5}
32717     { \_text_change_case_upper_el:nnnnN {#4} }
32718     {
32719       \_text_change_case_store:e
32720         { \_text_change_case_codepoint:nn { upper } {#4} }
32721       \_text_change_case_loop:nnnw
32722     }
32723     {#1} {#2} {#3} #5 \q_text_recursion_stop
32724 }

```

Now, we check the detail of the next codepoint: again we filter out the not-a-char cases, before checking if it's an dialytika, accent or diacritic. (The latter do not have the same hiatus behavior as accents.) There is additional work if the codepoint can take a ypogegrammeni: there, we need to move any ypogegrammeni to after accents (in case the input is not normalised). The ypogegrammeni itself is handled separately.

```

32725 \cs_new:Npn \_text_change_case_upper_el:nnnnN #1#2#3#4#5
32726 {
32727   \token_if_cs:NTF #5
32728     {
32729       \_text_change_case_store:e
32730         { \_text_change_case_codepoint:nn { upper } {#1} }
32731       \_text_change_case_loop:nnnw {#2} {#3} {#4} #5
32732     }
32733     {
32734       \_text_change_case_if_takes_ypogegrammeni:nTF {#1}
32735       {
32736         \_text_change_case_upper_el_ypogegrammeni:nnnnnnw
32737           {#1} {#2} {#3} {#4} { } { } #5
32738       }
32739       { \_text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5 }
32740     }
32741 }
32742 \cs_new:Npn \_text_change_case_upper_el_ypogegrammeni:nnnnnnw
32743   #1#2#3#4#5#6#7 \q_text_recursion_stop
32744 {
32745   \tl_if_head_is_N_type:nTF {#7}
32746     {
32747       \_text_change_case_upper_el_ypogegrammeni:nnnnnnN

```

```

32748         {#1} {#2} {#3} {#4} {#5} {#6}
32749     }
32750     { \_text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5#6 }
32751     #7 \q_text_recursion_stop
32752 }
32753 \cs_new:Npn \_text_change_case_upper_el_ypogegrammeni:nnnnnnN #1#2#3#4#5#6#7
32754 {
32755     \token_if_cs:NTF #7
32756     { \_text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5#6 }
32757     {
32758         \_text_codepoint_process:nN
32759         {
32760             \_text_change_case_upper_el_ypogegrammeni:nnnnnnN
32761             {#1} {#2} {#3} {#4} {#5} {#6}
32762         }
32763     }
32764     #7
32765 }
32766 \cs_new:Npn \_text_change_case_upper_el_ypogegrammeni:nnnnnnn #1#2#3#4#5#6#7
32767 {
32768     \_text_codepoint_compare:nNnTF {#7} = { "0345 }
32769     {
32770         \_text_change_case_upper_el_ypogegrammeni:nnnnnnw
32771         {#1} {#2} {#3} {#4} {#5} {#7}
32772     }
32773     {
32774         \bool_lazy_or:nnTF
32775         { \_text_change_case_if_greek_accent_p:n {#7} }
32776         { \_text_change_case_if_greek_breathing_p:n {#7} }
32777         {
32778             \_text_change_case_upper_el_ypogegrammeni:nnnnnnw
32779             {#1} {#2} {#3} {#4} {#5#7} {#6}
32780         }
32781         { \_text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5#6 #7 }
32782     }
32783 }
32784 \cs_new:Npn \_text_change_case_upper_el_aux:nnnnN #1#2#3#4#5
32785 {
32786     \_text_codepoint_process:nN
32787     { \_text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} } #5
32788 }
32789 \cs_new:Npn \_text_change_case_upper_el_aux:nnnnn #1#2#3#4#5
32790 {
32791     \_text_codepoint_compare:nNnTF {#5} = { "0308 }
32792     { \_text_change_case_upper_el_dialytika:nnnn {#2} {#3} {#4} {#1} }
32793     {
32794         \_text_change_case_if_greek_accent:nTF {#5}
32795         { \_text_change_case_upper_el_hiatus:nnnw {#2} {#3} {#4} {#1} }
32796         {
32797             \_text_change_case_if_greek_breathing:nTF {#5}
32798             { \_text_change_case_upper_el:nnnn {#1} {#2} {#3} {#4} }
32799             {
32800                 \_text_codepoint_compare:nNnTF {#5} = { "0345 }
32801                 {

```

```

32802         \__text_change_case_store:e
32803         { \use:c { __text_change_case_upper_ #4 _ypogegrammeni:n } {#1} }
32804         \__text_change_case_loop:nnw {#2} {#3} {#4}
32805     }
32806     {
32807         \__text_change_case_if_greek_stress:nTF {#5}
32808         {
32809             \__text_change_case_store:e
32810             { \__text_change_case_upper_el_stress:nn {#1} {#5} }
32811             \__text_change_case_loop:nnw {#2} {#3} {#4}
32812         }
32813         {
32814             \__text_change_case_store:e
32815             { \__text_change_case_codepoint:nn { upper } {#1} }
32816             \__text_change_case_loop:nnw {#2} {#3} {#4} #5
32817         }
32818     }
32819 }
32820 }
32821 }
32822 }
32823 }

```

We handle *dialytika* in parts as it's also needed for the hiatus. We know only two letters take it, so we can shortcut here on the second part of the tests.

```

32824 \cs_new:Npn \__text_change_case_upper_el_dialytika:nnnn #1#2#3#4
32825 {
32826     \__text_change_case_if_takes_dialytika:nTF {#4}
32827     { \__text_change_case_upper_el_dialytika:n {#4} }
32828     {
32829         \__text_change_case_store:e
32830         { \__text_change_case_codepoint:nn { upper } {#4} }
32831     }
32832     \__text_change_case_upper_el_gobble:nnw {#1} {#2} {#3}
32833 }
32834 \cs_new:Npn \__text_change_case_upper_el_dialytika:n #1
32835 {
32836     \__text_change_case_store:e
32837     {
32838         \bool_lazy_or:nnTF
32839         { \__text_codepoint_compare_p:nNn {#1} = { "0399 } }
32840         { \__text_codepoint_compare_p:nNn {#1} = { "03B9 } }
32841         {
32842             \codepoint_generate:nn { "03AA }
32843             { \__text_change_case_catcode:nn {#1} { "03AA } }
32844         }
32845         {
32846             \codepoint_generate:nn { "03AB }
32847             { \__text_change_case_catcode:nn {#1} { "03AB } }
32848         }
32849     }
32850 }

```

Adding a hiatus needs some of the same ideas, but if there is not one we skip this code point, hence needing a separate function.

```

32851 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnnnw
32852   #1#2#3#4#5 \q__text_recursion_stop
32853   {
32854     \tl_if_head_is_N_type:nTF {#5}
32855     { \__text_change_case_upper_el_hiatus:nnnnN {#4} }
32856     {
32857       \__text_change_case_store:e
32858       { \__text_change_case_codepoint:nn { upper } {#4} }
32859       \__text_change_case_loop:nnw
32860     }
32861     {#1} {#2} {#3} #5 \q__text_recursion_stop
32862   }
32863 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnnnN #1#2#3#4#5
32864   {
32865     \token_if_cs:NTF #5
32866     {
32867       \__text_change_case_store:e
32868       { \__text_change_case_codepoint:nn { upper } {#1} }
32869       \__text_change_case_loop:nnw {#2} {#3} {#4} #5
32870     }
32871     {
32872       \__text_codepoint_process:nN
32873       { \__text_change_case_upper_el_hiatus:nnnnN {#1} {#2} {#3} {#4} } #5
32874     }
32875   }
32876 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnnnn #1#2#3#4#5
32877   {
32878     \__text_change_case_if_takes_dialytika:nTF {#5}
32879     {
32880       \__text_change_case_store:e
32881       { \__text_change_case_codepoint:nn { upper } {#1} }
32882       \__text_change_case_upper_el_dialytika:n {#5}
32883       \__text_change_case_upper_el_gobble:nnw {#2} {#3} {#4}
32884     }
32885     { \__text_change_case_upper_el:nnnn {#1} {#2} {#3} {#4} #5 }
32886   }

```

Handling the *ypogegrammeni* output depends on the selected approach

```

32887 \cs_new:Npn \__text_change_case_upper_el_ypogegrammeni:n #1
32888   {
32889     \exp_args:Ne \__text_change_case_generate:n
32890     {
32891       \int_case:nn
32892       { \__text_codepoint_from_chars:Nw #1 }
32893       {
32894         { "0391 } { "1FBC }
32895         { "03B1 } { "1FBC }
32896         { "0397 } { "1FCC }
32897         { "03B7 } { "1FCC }
32898         { "03A9 } { "1FFC }
32899         { "03C9 } { "1FFC }
32900       }
32901     }
32902   }
32903 \cs_new:cpn { __text_change_case_upper_el-x-iota_ypogegrammeni:n } #1

```

```

32904 {
32905   \_text_change_case_codepoint:nn { upper } {#1}
32906   \codepoint_generate:nn { "0399 }
32907   { \char_value_catcode:n { "0399 } }
32908 }

```

We choose to retain stress diacritics, but we also need to recombine them for pdf \TeX . That is handled here.

```

32909 \cs_new:Npn \_text_change_case_upper_el_stress:nn #1#2
32910 {
32911   \exp_args:Ne \_text_change_case_generate:n
32912   {
32913     \int_case:nn
32914     { \_text_codepoint_from_chars:Nw #2 }
32915     {
32916       { "0304 }
32917       {
32918         \int_case:nn { \_text_codepoint_from_chars:Nw #1 }
32919         {
32920           { "0391 } { "1FB9 }
32921           { "03B1 } { "1FB9 }
32922           { "0399 } { "1FD9 }
32923           { "03B9 } { "1FD9 }
32924           { "03A5 } { "1FE9 }
32925           { "03C5 } { "1FE9 }
32926         }
32927       }
32928       { "0306 }
32929       {
32930         \int_case:nn { \_text_codepoint_from_chars:Nw #1 }
32931         {
32932           { "0391 } { "1FB8 }
32933           { "03B1 } { "1FB8 }
32934           { "0399 } { "1FD8 }
32935           { "03B9 } { "1FD8 }
32936           { "03A5 } { "1FE8 }
32937           { "03C5 } { "1FE8 }
32938         }
32939       }
32940     }
32941   }
32942 }

```

For clearing out trailing combining marks after we have dealt with the first one.

```

32943 \cs_new:Npn \_text_change_case_upper_el_gobble:nnnw
32944 #1#2#3#4 \q_text_recursion_stop
32945 {
32946   \tl_if_head_is_N_type:nTF {#4}
32947   { \_text_change_case_upper_el_gobble:nnnN }
32948   { \_text_change_case_loop:nnnw }
32949   {#1} {#2} {#3} #4 \q_text_recursion_stop
32950 }
32951 \cs_new:Npn \_text_change_case_upper_el_gobble:nnnN #1#2#3#4
32952 {
32953   \token_if_cs:NTF #4

```



```

32954     { \_text_change_case_loop:nnnw {#1} {#2} {#3} }
32955     {
32956         \_text_codepoint_process:nN
32957         { \_text_change_case_upper_el_gobble:nnnn {#1} {#2} {#3} }
32958     }
32959     #4
32960 }
32961 \cs_new:Npn \_text_change_case_upper_el_gobble:nnnn #1#2#3#4
32962 {
32963     \bool_lazy_or:nnTF
32964     { \_text_change_case_if_greek_accent_p:n {#4} }
32965     { \_text_change_case_if_greek_breathing_p:n {#4} }
32966     { \_text_change_case_upper_el_gobble:nnnw {#1} {#2} {#3} }
32967     { \_text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
32968 }

```

Luckily the Greek range is limited and clear.

```

32969 \prg_new_conditional:Npnn \_text_change_case_if_greek:n #1 { p , TF }
32970 {
32971     \exp_args:Nf \_text_change_case_if_greek:n
32972     { \int_eval:n { \_text_codepoint_from_chars:Nw #1 } }
32973 }
32974 \cs_new:Npn \_text_change_case_if_greek:n #1
32975 {
32976     \if_int_compare:w #1 < "0370 \exp_stop_f:
32977     \prg_return_false:
32978     \else:
32979     \if_int_compare:w #1 > "03FF \exp_stop_f:
32980     \if_int_compare:w #1 < "1F00 \exp_stop_f:
32981     \prg_return_false:
32982     \else:
32983     \if_int_compare:w #1 > "1FFF \exp_stop_f:
32984     \if_int_compare:w #1 = "2126 \exp_stop_f:
32985     \prg_return_true:
32986     \else:
32987     \prg_return_false:
32988     \fi:
32989     \else:
32990     \prg_return_true:
32991     \fi:
32992     \fi:
32993     \else:
32994     \prg_return_true:
32995     \fi:
32996     \fi:
32997 }

```

We follow ICU in adding a few extras to the accent list here.

```

32998 \prg_new_conditional:Npnn \_text_change_case_if_greek_accent:n #1 { TF , p }
32999 {
33000     \exp_args:Nf \_text_change_case_if_greek_accent:n
33001     { \int_eval:n { \_text_codepoint_from_chars:Nw #1 } }
33002 }
33003 \cs_new:Npn \_text_change_case_if_greek_accent:n #1
33004 {

```

```

33005 \if_int_compare:w #1 = "0300 \exp_stop_f:
33006 \prg_return_true:
33007 \else:
33008 \if_int_compare:w #1 = "0301 \exp_stop_f:
33009 \prg_return_true:
33010 \else:
33011 \if_int_compare:w #1 = "0342 \exp_stop_f:
33012 \prg_return_true:
33013 \else:
33014 \if_int_compare:w #1 = "0302 \exp_stop_f:
33015 \prg_return_true:
33016 \else:
33017 \if_int_compare:w #1 = "0303 \exp_stop_f:
33018 \prg_return_true:
33019 \else:
33020 \if_int_compare:w #1 = "0311 \exp_stop_f:
33021 \prg_return_true:
33022 \else:
33023 \prg_return_false:
33024 \fi:
33025 \fi:
33026 \fi:
33027 \fi:
33028 \fi:
33029 \fi:
33030 }
33031 \prg_new_conditional:Npnn \_text_change_case_if_greek_spacing_diacritic:n
33032 #1 { TF }
33033 {
33034 \exp_args:Nf \_text_change_case_if_greek_spacing_diacritic:n
33035 { \int_eval:n { \_text_codepoint_from_chars:Nw #1 } }
33036 }
33037 \cs_new:Npn \_text_change_case_if_greek_spacing_diacritic:n #1
33038 {
33039 \if_int_compare:w #1 < "1FBD \exp_stop_f:
33040 \if_int_compare:w #1 = "037A \exp_stop_f:
33041 \prg_return_true:
33042 \else:
33043 \prg_return_false:
33044 \fi:
33045 \else:
33046 \if_int_compare:w #1 = "1FBD \exp_stop_f:
33047 \prg_return_true:
33048 \else:
33049 \if_int_compare:w #1 = "1FBF \exp_stop_f:
33050 \prg_return_true:
33051 \else:
33052 \if_int_compare:w #1 = "1FC0 \exp_stop_f:
33053 \prg_return_true:
33054 \else:
33055 \if_int_compare:w #1 = "1FC1 \exp_stop_f:
33056 \prg_return_true:
33057 \else:
33058 \if_int_compare:w #1 = "1FCD \exp_stop_f:

```

```

33059         \prg_return_true:
33060     \else:
33061         \if_int_compare:w #1 = "1FCE \exp_stop_f:
33062             \prg_return_true:
33063     \else:
33064         \if_int_compare:w #1 = "1FCF \exp_stop_f:
33065             \prg_return_true:
33066     \else:
33067         \if_int_compare:w #1 = "1FDD \exp_stop_f:
33068             \prg_return_true:
33069     \else:
33070         \if_int_compare:w #1 = "1FDE \exp_stop_f:
33071             \prg_return_true:
33072     \else:
33073         \if_int_compare:w #1 = "1FDF \exp_stop_f:
33074             \prg_return_true:
33075     \else:
33076         \if_int_compare:w #1 = "1FED \exp_stop_f:
33077             \prg_return_true:
33078     \else:
33079         \if_int_compare:w #1 = "1FEE \exp_stop_f:
33080             \prg_return_true:
33081     \else:
33082         \if_int_compare:w #1 = "1FEF \exp_stop_f:
33083             \prg_return_true:
33084     \else:
33085         \if_int_compare:w #1 = "1FFD \exp_stop_f:
33086             \prg_return_true:
33087     \else:
33088         \if_int_compare:w #1 = "1FFE \exp_stop_f:
33089             \prg_return_true:
33090     \else:
33091         \prg_return_false:
33092     \fi:
33093 \fi:
33094 \fi:
33095 \fi:
33096 \fi:
33097 \fi:
33098 \fi:
33099 \fi:
33100 \fi:
33101 \fi:
33102 \fi:
33103 \fi:
33104 \fi:
33105 \fi:
33106 \fi:
33107 \fi:
33108 }
33109 \prg_new_conditional:Npnn \__text_change_case_if_greek_breathing:n
33110 #1 { TF , p }
33111 {
33112     \exp_args:Nf \__text_change_case_if_greek_breathing:n

```

```

33113     { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
33114   }
33115 \cs_new:Npn \__text_change_case_if_greek_breathing:n #1
33116 {
33117   \if_int_compare:w #1 = "0313 \exp_stop_f:
33118     \prg_return_true:
33119   \else:
33120     \if_int_compare:w #1 = "0314 \exp_stop_f:
33121       \prg_return_true:
33122     \else:
33123       \prg_return_false:
33124     \fi:
33125   \fi:
33126 }
33127 \prg_new_conditional:Npnn \__text_change_case_if_greek_stress:n
33128 #1 { TF , p }
33129 {
33130   \exp_args:Nf \__text_change_case_if_greek_stress:n
33131     { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
33132 }
33133 \cs_new:Npn \__text_change_case_if_greek_stress:n #1
33134 {
33135   \if_int_compare:w #1 = "0304 \exp_stop_f:
33136     \prg_return_true:
33137   \else:
33138     \if_int_compare:w #1 = "0306 \exp_stop_f:
33139       \prg_return_true:
33140     \else:
33141       \prg_return_false:
33142     \fi:
33143   \fi:
33144 }
33145 \prg_new_conditional:Npnn \__text_change_case_if_takes_dialytika:n #1 { TF }
33146 {
33147   \exp_args:Nf \__text_change_case_if_takes_dialytika:n
33148     { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
33149 }
33150 \cs_new:Npn \__text_change_case_if_takes_dialytika:n #1
33151 {
33152   \if_int_compare:w #1 = "0399 \exp_stop_f:
33153     \prg_return_true:
33154   \else:
33155     \if_int_compare:w #1 = "03B9 \exp_stop_f:
33156       \prg_return_true:
33157     \else:
33158       \if_int_compare:w #1 = "03A5 \exp_stop_f:
33159         \prg_return_true:
33160       \else:
33161         \if_int_compare:w #1 = "03C5 \exp_stop_f:
33162           \prg_return_true:
33163         \else:
33164           \prg_return_false:
33165         \fi:
33166       \fi:

```

```

33167     \fi:
33168     \fi:
33169   }
33170 \prg_new_conditional:Npnn \__text_change_case_if_takes_ypogegrammeni:n #1 { TF }
33171   {
33172     \exp_args:Nf \__text_change_case_if_takes_ypogegrammeni:n
33173     { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
33174   }
33175 \cs_new:Npn \__text_change_case_if_takes_ypogegrammeni:n #1
33176   {
33177     \if_int_compare:w #1 = "03B1 \exp_stop_f:
33178     \prg_return_true:
33179   \else:
33180     \if_int_compare:w #1 = "03B7 \exp_stop_f:
33181     \prg_return_true:
33182   \else:
33183     \if_int_compare:w #1 = "03C9 \exp_stop_f:
33184     \prg_return_true:
33185   \else:
33186     \prg_return_false:
33187   \fi:
33188   \fi:
33189   \fi:
33190   }

```

(End of definition for __text_change_case_upper_el:nnnnn and others.)

```

\__text_change_case_boundary_upper_el:Nnnnw
\__text_change_case_boundary_upper_el-x-iota:Nnnnw
\__text_change_case_boundary_upper_el:nnnN
\__text_change_case_boundary_upper_el:nnnn
\__text_change_case_boundary_upper_el:nnnnw

```

There is one things that need special treatment at the start of words in Greek. For an isolated accent *eta*, which is handled by seeing if we have exactly one of the affected codepoints followed by a space or brace group.

```

33191 \cs_new:Npn \__text_change_case_boundary_upper_el:Nnnnw
33192   #1#2#3#4#5 \q__text_recursion_stop
33193   {
33194     \tl_if_head_is_N_type:nTF {#5}
33195     { \__text_change_case_boundary_upper_el:nnnN }
33196     { \__text_change_case_loop:nnnw }
33197     {#2} {#3} {#4} #5 \q__text_recursion_stop
33198   }
33199 \cs_new_eq:cN { \__text_change_case_boundary_upper_el-x-iota:Nnnnw }
33200   \__text_change_case_boundary_upper_el:Nnnnw
33201 \cs_new:Npn \__text_change_case_boundary_upper_el:nnnN #1#2#3#4
33202   {
33203     \token_if_cs:NTF #4
33204     { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
33205     {
33206       \__text_codepoint_process:nN
33207       { \__text_change_case_boundary_upper_el:nnnn {#1} {#2} {#3} }
33208     }
33209     #4
33210   }
33211 \cs_new:Npn \__text_change_case_boundary_upper_el:nnnn #1#2#3#4
33212   {
33213     \bool_lazy_any:nTF
33214     {

```

```

33215     { \_text_codepoint_compare_p:nNn {#4} = { "0389 } }
33216     { \_text_codepoint_compare_p:nNn {#4} = { "03AE } }
33217     { \_text_codepoint_compare_p:nNn {#4} = { "1F22 } }
33218     { \_text_codepoint_compare_p:nNn {#4} = { "1F2A } }
33219     }
33220     { \_text_change_case_boundary_upper_el:nnnw {#1} {#2} {#3} {#4} }
33221     { \_text_change_case_breathing:nnnn {#1} {#2} {#3} {#4} }
33222   }
33223 \cs_new:Npn \_text_change_case_boundary_upper_el:nnnw
33224   #1#2#3#4#5 \q_text_recursion_stop
33225   {
33226     \tl_if_head_is_N_type:nTF {#5}
33227     { \_text_change_case_loop:nnw {#1} {#2} {#3} #4 }
33228     {
33229       \_text_change_case_store:e
33230       {
33231         \codepoint_generate:nn { "0389 }
33232         { \_text_change_case_catcode:nn {#4} { "0389 } }
33233       }
33234       \_text_change_case_loop:nnw {#1} {#2} {#3}
33235     }
33236     #5 \q_text_recursion_stop
33237   }

```

(End of definition for _text_change_case_boundary_upper_el:Nnnnw and others.)

```

\_text_change_case_breathing:nnnn
\_text_change_case_breathing:nnnnn
\_text_change_case_breathing:nnnnnw
\_text_change_case_breathing:nnnnnw
\_text_change_case_breathing_aux:nnnnnn
\_text_change_case_breathing_aux:nnnnnw
\_text_change_case_breathing_aux:nnnN
\_text_change_case_breathing_dialytika:nnnn

```

In Greek, breathing diacritics are normally dropped when uppercasing: see the code for the general case. However, for the first character of a word, if there is a breather *and* the next character takes a *dialytika*, it needs to be added. We start by checking if the current codepoint is in the Greek range, then decomposing.

```

33238 \cs_new:Npn \_text_change_case_breathing:nnnn #1#2#3#4
33239   {
33240     \_text_change_case_if_greek:nTF {#4}
33241     {
33242       \exp_args:Ne \_text_change_case_breathing:nnnnn
33243       {
33244         \codepoint_to_nfd:n
33245         { \_text_codepoint_from_chars:Nw #4 }
33246       }
33247       {#1} {#2} {#3} {#4}
33248     }
33249     { \_text_change_case_loop:nnw {#1} {#2} {#3} #4 }
33250   }
33251 \cs_new:Npn \_text_change_case_breathing:nnnnn #1#2#3#4#5
33252   {
33253     \_text_codepoint_process:nN
33254     { \_text_change_case_breathing:nnnnnw {#2} {#3} {#4} {#5} }
33255     #1 \q_mark
33256   }

```

Normal form decomposition will always give between one and three codepoints. Luckily, the two breathing marks (*psili* and *dasia*) will be in a predictable position: last. So we can quickly establish first that there was a change on decomposition, and second if the final resulting codepoint is one of the two we care about.

```

33257 \cs_new:Npn \__text_change_case_breathing:nnnnnw #1#2#3#4#5#6 \q_mark
33258 {
33259   \tl_if_blank:nTF {#6}
33260   { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
33261   {
33262     \__text_codepoint_process:nN
33263     { \__text_change_case_breathing:nnnnnw {#1} {#2} {#3} {#4} {#5} }
33264     #6 \q_mark
33265   }
33266 }
33267 \cs_new:Npn \__text_change_case_breathing:nnnnnw #1#2#3#4#5#6#7 \q_mark
33268 {
33269   \tl_if_blank:nTF {#7}
33270   {
33271     \__text_change_case_breathing_aux:nnnnn
33272     {#1} {#2} {#3} {#4} {#5} {#6}
33273   }
33274   {
33275     \__text_codepoint_process:nN
33276     { \__text_change_case_breathing:nnnnnw {#1} {#2} {#3} {#4} {#5} }
33277     #7 \q_mark
33278   }
33279 }
33280 \cs_new:Npn \__text_change_case_breathing_aux:nnnnn #1#2#3#4#5#6
33281 {
33282   \bool_lazy_or:nnTF
33283   { \__text_codepoint_compare_p:nNn {#6} = { "0313 } }
33284   { \__text_codepoint_compare_p:nNn {#6} = { "0314 } }
33285   { \__text_change_case_breathing_aux:nnnw {#1} {#2} {#3} {#5} }
33286   { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
33287 }

```

Now the lookahead can be fired: check the next codepoint and assess whether it takes a *dialytika*. Drop the breathing mark or generate the *dialytika*: the latter is code shared with the general mechanism.

```

33288 \cs_new:Npn \__text_change_case_breathing_aux:nnnw #1#2#3#4#5
33289 \q_text_recursion_stop
33290 {
33291   \__text_change_case_store:e
33292   { \__text_change_case_codepoint:nn { upper } {#4} }
33293   \tl_if_head_is_N_type:nTF {#5}
33294   { \__text_change_case_breathing_aux:nnnN }
33295   { \__text_change_case_loop:nnnw }
33296   {#1} {#2} {#3} #5 \q_text_recursion_stop
33297 }
33298 \cs_new:Npn \__text_change_case_breathing_aux:nnnN #1#2#3#4
33299 {
33300   \__text_codepoint_process:nN
33301   { \__text_change_case_breathing_dialytika:nnnn {#1} {#2} {#3} } #4
33302 }
33303 \cs_new:Npn \__text_change_case_breathing_dialytika:nnnn #1#2#3#4
33304 {
33305   \__text_change_case_if_takes_dialytika:nTF {#4}
33306   {

```

```

33307     \_text_change_case_upper_el_dialytika:n {#4}
33308     \_text_change_case_loop:nnnw {#1} {#2} {#3}
33309   }
33310   { \_text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
33311 }

```

(End of definition for _text_change_case_breathing:nnnn and others.)

_text_change_case_title_el:nnnn Titlecasing retains accents, but to prevent the uppercasing code from kicking in, there has to be an explicit function here.

```

33312 \cs_new:Npn \_text_change_case_title_el:nnnnn #1#2#3#4#5
33313 { \_text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }

```

(End of definition for _text_change_case_title_el:nnnnn.)

_text_change_case_upper_hy:nnnnn See <https://www.unicode.org/L2/L2020/20143-armenian-ech-yiwn.pdf>.

```

\text_change_case_title_hy:nnnnn
\_text_change_case_upper_hy-x-yiwn:nnnnn
\_text_change_case_title_hy-x-yiwn:nnnnn
33314 \cs_new:Npn \_text_change_case_upper_hy:nnnnn #1#2#3#4#5
33315 {
33316   \_text_codepoint_compare:nNnTF {#5} = { "0587 }
33317   {
33318     \_text_change_case_store:e
33319     {
33320       \codepoint_generate:nn { "0535 }
33321       { \_text_change_case_catcode:nn {#5} { "0535 } }
33322       \codepoint_generate:nn { "054E }
33323       { \_text_change_case_catcode:nn {#5} { "054E } }
33324     }
33325     \use:c { __text_change_case_next_ #2 :nnn }
33326     {#2} {#3} {#4}
33327   }
33328   { \_text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33329 }
33330 \cs_new:Npn \_text_change_case_title_hy:nnnnn #1#2#3#4#5
33331 {
33332   \_text_codepoint_compare:nNnTF {#5} = { "0587 }
33333   {
33334     \_text_change_case_store:e
33335     {
33336       \codepoint_generate:nn { "0535 }
33337       { \_text_change_case_catcode:nn {#5} { "0535 } }
33338       \codepoint_generate:nn { "057E }
33339       { \_text_change_case_catcode:nn {#5} { "057E } }
33340     }
33341     \use:c { __text_change_case_next_ #2 :nnn }
33342     {#2} {#3} {#4}
33343   }
33344   { \_text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33345 }
33346 \cs_new:cpn { __text_change_case_upper_hy-x-yiwn:nnnnn } #1#2#3#4#5
33347 { \_text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33348 \cs_new_eq:cc { __text_change_case_title_hy-x-yiwn:nnnnn }
33349 { __text_change_case_upper_hy-x-yiwn:nnnnn }

```

(End of definition for _text_change_case_upper_hy:nnnnn and others.)

_text_change_case_lower_la-x-medieval:nnnnn
_text_change_case_upper_la-x-medieval:nnnnn

Simply swaps of characters.

```
33350 \cs_new:cpn { __text_change_case_lower_la-x-medieval:nnnnn } #1#2#3#4#5
33351 {
33352   \__text_codepoint_compare:nNnTF {#5} = { "0056 }
33353   {
33354     \__text_change_case_store:e
33355     { \char_generate:nn { "0075 } { \__text_char_catcode:N #5 } }
33356     \use:c { __text_change_case_next_ #2 :nnn }
33357     {#2} {#3} {#4}
33358   }
33359   { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33360 }
33361 \cs_new:cpn { __text_change_case_upper_la-x-medieval:nnnnn } #1#2#3#4#5
33362 {
33363   \__text_codepoint_compare:nNnTF {#5} = { "0075 }
33364   {
33365     \__text_change_case_store:e
33366     { \char_generate:nn { "0056 } { \__text_char_catcode:N #5 } }
33367     \use:c { __text_change_case_next_ #2 :nnn }
33368     {#2} {#3} {#4}
33369   }
33370   { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33371 }
```

(End of definition for `__text_change_case_lower_la-x-medieval:nnnnn` and `__text_change_case_upper_la-x-medieval:nnnnn`.)

__text_change_cases_lower_lt:nnnnn
__text_change_cases_lower_lt_auxi:nnnnn
__text_change_cases_lower_lt_auxii:nnnnn
__text_change_case_lower_lt:nnnw
__text_change_case_lower_lt:nnnN
__text_change_case_lower_lt:nnnn

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. The first step is a simple match attempt: look for the three uppercase accented letters which should gain a dot-above char in their lowercase form.

```
33372 \cs_new:Npn \__text_change_case_lower_lt:nnnnn #1#2#3#4#5
33373 {
33374   \exp_args:Ne \__text_change_case_lower_lt_auxi:nnnnn
33375   {
33376     \int_case:nn { \__text_codepoint_from_chars:Nw #5 }
33377     {
33378       { "00CC } { "0300 }
33379       { "00CD } { "0301 }
33380       { "0128 } { "0303 }
33381     }
33382   }
33383   {#2} {#3} {#4} {#5}
33384 }
```

If there was a hit, output the result with the dot-above and move on. Otherwise, look for one of the three letters that can take a combining accent: I, J, and I-ogonek.

```
33385 \cs_new:Npn \__text_change_case_lower_lt_auxi:nnnnn #1#2#3#4#5
33386 {
33387   \tl_if_blank:nTF {#1}
33388   {
33389     \exp_args:Ne \__text_change_case_lower_lt_auxii:nnnnn
33390     {
33391       \int_case:nn { \__text_codepoint_from_chars:Nw #5 }
33392       {
33393         { "0041 } { "0304 }
33394         { "0042 } { "0305 }
33395         { "0134 } { "0308 }
33396       }
33397     }
33398   }
33399 }
```

```

33392         {
33393             { "0049 } { "0069 }
33394             { "004A } { "006A }
33395             { "012E } { "012F }
33396         }
33397     }
33398     {#2} {#3} {#4} {#5}
33399 }
33400 {
33401     \__text_change_case_store:e
33402     {
33403         \codepoint_generate:nn { "0069 }
33404         { \__text_change_case_catcode:nn {#5} { "0069 } }
33405         \codepoint_generate:nn { "0307 }
33406         { \__text_change_case_catcode:nn {#5} { "0307 } }
33407         \codepoint_generate:nn {#1}
33408         { \__text_change_case_catcode:nn {#5} {#1} }
33409     }
33410     \__text_change_case_loop:nnnw {#2} {#3} {#4}
33411 }
33412 }

```

Again, branch depending on a hit. If there is one, we output the character then need to look for a combining accent: as usual, we need to be aware of the loop situation.

```

33413 \cs_new:Npn \__text_change_case_lower_lt_auxii:nnnnn #1#2#3#4#5
33414 {
33415     \tl_if_blank:nTF {#1}
33416     { \__text_change_case_codepoint:nnnnn {#2} {#2} {#3} {#4} {#5} }
33417     {
33418         \__text_change_case_store:e
33419         {
33420             \codepoint_generate:nn {#1}
33421             { \__text_change_case_catcode:nn {#5} {#1} }
33422         }
33423         \__text_change_case_lower_lt:nnnw {#2} {#3} {#4}
33424     }
33425 }
33426 \cs_new:Npn \__text_change_case_lower_lt:nnnw #1#2#3#4 \q__text_recursion_stop
33427 {
33428     \tl_if_head_is_N_type:nTF {#4}
33429     { \__text_change_case_lower_lt:nnnN }
33430     { \__text_change_case_loop:nnnw }
33431     {#1} {#2} {#3} #4 \q__text_recursion_stop
33432 }
33433 \cs_new:Npn \__text_change_case_lower_lt:nnnN #1#2#3#4
33434 {
33435     \__text_codepoint_process:nN
33436     { \__text_change_case_lower_lt:nnnn {#1} {#2} {#3} } #4
33437 }
33438 \cs_new:Npn \__text_change_case_lower_lt:nnnn #1#2#3#4
33439 {
33440     \bool_lazy_and:nnT
33441     {
33442         \bool_lazy_or_p:nn

```

```

33443     { ! \tl_if_single_p:n {#4} }
33444     { ! \token_if_cs_p:N #4 }
33445   }
33446   {
33447     \bool_lazy_any_p:n
33448     {
33449       { \__text_codepoint_compare_p:nNn {#4} = { "0300 } }
33450       { \__text_codepoint_compare_p:nNn {#4} = { "0301 } }
33451       { \__text_codepoint_compare_p:nNn {#4} = { "0303 } }
33452     }
33453   }
33454   {
33455     \__text_change_case_store:e
33456     {
33457       \codepoint_generate:nn { "0307 }
33458       { \__text_change_case_catcode:nn {#4} { "0307 } }
33459     }
33460   }
33461   \__text_change_case_loop:nnnw {#1} {#2} {#3} #4
33462 }

```

(End of definition for __text_change_cases_lower_lt:nnnnn and others.)

```

\__text_change_cases_upper_lt:nnnnn
\__text_change_cases_upper_lt_aux:nnnnn
\__text_change_case_upper_lt:nnnw
\__text_change_case_upper_lt:nnnN
\__text_change_case_upper_lt:nnnn

```

The uppercasing version: first find i/j/i-ogonek, then look for the combining char: drop it if present.

```

33463 \cs_new:Npn \__text_change_case_upper_lt:nnnnn #1#2#3#4#5
33464 {
33465   \exp_args:Ne \__text_change_case_upper_lt_aux:nnnnn
33466   {
33467     \int_case:nn { \__text_codepoint_from_chars:Nw #5 }
33468     {
33469       { "0069 } { "0049 }
33470       { "006A } { "004A }
33471       { "012F } { "012E }
33472     }
33473   }
33474   {#2} {#3} {#4} {#5}
33475 }
33476 \cs_new:Npn \__text_change_case_upper_lt_aux:nnnnn #1#2#3#4#5
33477 {
33478   \tl_if_blank:nTF {#1}
33479   { \__text_change_case_codepoint:nnnnn { upper } {#2} {#3} {#4} {#5} }
33480   {
33481     \__text_change_case_store:e
33482     {
33483       \codepoint_generate:nn {#1}
33484       { \__text_change_case_catcode:nn {#5} {#1} }
33485     }
33486     \__text_change_case_upper_lt:nnnw {#2} {#3} {#4}
33487   }
33488 }
33489 \cs_new:Npn \__text_change_case_upper_lt:nnnw #1#2#3#4 \q__text_recursion_stop
33490 {
33491   \tl_if_head_is_N_type:nTF {#4}

```

```

33492     { \_text_change_case_upper_lt:nnnN }
33493     { \use:c { \_text_change_case_next_ #1 :nnn } }
33494     {#1} {#2} {#3} #4 \q_text_recursion_stop
33495   }
33496 \cs_new:Npn \_text_change_case_upper_lt:nnnN #1#2#3#4
33497   {
33498     \_text_codepoint_process:nN
33499     { \_text_change_case_upper_lt:nnnn {#1} {#2} {#3} } #4
33500   }
33501 \cs_new:Npn \_text_change_case_upper_lt:nnnn #1#2#3#4
33502   {
33503     \bool_lazy_and:nnTF
33504     {
33505       \bool_lazy_or_p:nn
33506       { ! \tl_if_single_p:n {#4} }
33507       { ! \token_if_cs_p:N #4 }
33508     }
33509     { \_text_codepoint_compare_p:nNn {#4} = { "0307 } }
33510     { \use:c { \_text_change_case_next_ #1 :nnn } {#1} {#2} {#3} }
33511     { \use:c { \_text_change_case_next_ #1 :nnn } {#1} {#2} {#3} #4 }
33512   }

```

(End of definition for _text_change_cases_upper_lt:nnnnn and others.)

```

\_text_change_case_title_nl:nnnnn
\_text_change_case_title_nl_aux:nnnnn
\_text_change_case_title_nl:nnnw
\_text_change_case_title_nl:nnnN

```

For Dutch, there is a single look-ahead test for ij when title casing. If the appropriate letters are found, produce IJ and gobble the j/J.

```

33513 \cs_new:Npn \_text_change_case_title_nl:nnnnn #1#2#3#4#5
33514   {
33515     \tl_if_single:nTF {#5}
33516     { \_text_change_case_title_nl_aux:nnnnn }
33517     { \_text_change_case_codepoint:nnnnn }
33518     {#1} {#2} {#3} {#4} {#5}
33519   }
33520 \cs_new:Npn \_text_change_case_title_nl_aux:nnnnn #1#2#3#4#5
33521   {
33522     \bool_lazy_or:nnTF
33523     { \int_compare_p:nNn {'#5} = { "0049 } }
33524     { \int_compare_p:nNn {'#5} = { "0069 } }
33525     {
33526       \_text_change_case_store:e
33527       { \char_generate:nn { "0049 } { \_text_char_catcode:N #5 } }
33528       \_text_change_case_title_nl:nnnw {#2} {#3} {#4}
33529     }
33530     { \_text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33531   }
33532 \cs_new:Npn \_text_change_case_title_nl:nnnw #1#2#3#4 \q_text_recursion_stop
33533   {
33534     \tl_if_head_is_N_type:nTF {#4}
33535     { \_text_change_case_title_nl:nnnN }
33536     { \use:c { \_text_change_case_next_ #1 :nnn } }
33537     {#1} {#2} {#3} #4 \q_text_recursion_stop
33538   }
33539 \cs_new:Npn \_text_change_case_title_nl:nnnN #1#2#3#4
33540   {

```

```

33541 \bool_lazy_and:nnTF
33542   { ! \token_if_cs_p:N #4 }
33543   {
33544     \bool_lazy_or_p:nn
33545     { \int_compare_p:nNn {'#4} = { "004A } }
33546     { \int_compare_p:nNn {'#4} = { "006A } }
33547   }
33548   {
33549     \__text_change_case_store:e
33550     { \char_generate:nn { "004A } { \__text_char_catcode:N #4 } }
33551     \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3}
33552   }
33553   { \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3} #4 }
33554 }

```

(End of definition for `__text_change_case_title_nl:nnnnn` and others.)

```

\__text_change_case_lower_tr:nnnnn
\__text_change_case_lower_tr:nnnNw
\__text_change_case_lower_tr:NnnnN
\__text_change_case_lower_tr:Nnnnn

```

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```

33555 \cs_new:Npn \__text_change_case_lower_tr:nnnnn #1#2#3#4#5
33556   {
33557     \__text_codepoint_compare:nNnTF {#5} = { "0049 }
33558     { \__text_change_case_lower_tr:nnnNw {#1} {#3} {#4} #5 }
33559     {
33560       \__text_codepoint_compare:nNnTF {#5} = { "0130 }
33561       {
33562         \__text_change_case_store:e
33563         {
33564           \codepoint_generate:nn { "0069 }
33565           { \__text_change_case_catcode:nn {#5} { "0069 } }
33566         }
33567         \__text_change_case_loop:nnnw {#1} {#3} {#4}
33568       }
33569       { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33570     }
33571   }

```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input.

```

33572 \cs_new:Npn \__text_change_case_lower_tr:nnnNw #1#2#3#4#5 \q__text_recursion_stop
33573   {
33574     \tl_if_head_is_N_type:nTF {#5}
33575     { \__text_change_case_lower_tr:NnnnN #4 {#1} {#2} {#3} }
33576     {
33577       \__text_change_case_store:e
33578       {
33579         \codepoint_generate:nn { "0131 }
33580         { \__text_change_case_catcode:nn {#4} { "0131 } }
33581       }
33582       \__text_change_case_loop:nnnw {#1} {#2} {#3}
33583     }
33584     #5 \q__text_recursion_stop

```

```

33585 }
33586 \cs_new:Npn \__text_change_case_lower_tr:NnnnN #1#2#3#4#5
33587 {
33588   \__text_codepoint_process:nN
33589   { \__text_change_case_lower_tr:Nnnnn #1 {#2} {#3} {#4} } #5
33590 }
33591 \cs_new:Npn \__text_change_case_lower_tr:Nnnnn #1#2#3#4#5
33592 {
33593   \bool_lazy_or:nnTF
33594   {
33595     \bool_lazy_and_p:nn
33596     { \tl_if_single_p:n {#5} }
33597     { \token_if_cs_p:N #5 }
33598   }
33599   { ! \__text_codepoint_compare_p:nNn {#5} = { "0307 } }
33600   {
33601     \__text_change_case_store:e
33602     {
33603       \codepoint_generate:nn { "0131 }
33604       { \__text_change_case_catcode:nn {#1} { "0131 } }
33605     }
33606     \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
33607   }
33608   {
33609     \__text_change_case_store:e
33610     {
33611       \codepoint_generate:nn { "0069 }
33612       { \__text_change_case_catcode:nn {#1} { "0069 } }
33613     }
33614     \__text_change_case_loop:nnnw {#2} {#3} {#4}
33615   }
33616 }

```

(End of definition for __text_change_case_lower_tr:nnnn and others.)

__text_change_case_upper_tr:nnnnn Uppercasing is easier: just one exception with no context.

```

33617 \cs_new:Npn \__text_change_case_upper_tr:nnnnn #1#2#3#4#5
33618 {
33619   \__text_codepoint_compare:nNnTF {#5} = { "0069 }
33620   {
33621     \__text_change_case_store:e
33622     {
33623       \codepoint_generate:nn { "0130 }
33624       { \__text_change_case_catcode:nn {#5} { "0130 } }
33625     }
33626     \use:c { __text_change_case_next_ #2 :nnn } {#2} {#3} {#4}
33627   }
33628   { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33629 }

```

(End of definition for __text_change_case_upper_tr:nnnnn.)

__text_change_case_lower_az:nnnnn Straight copies.

```

\__text_change_case_upper_az:nnnnn
33630 \cs_new_eq:NN \__text_change_case_lower_az:nnnnn
33631 \__text_change_case_lower_tr:nnnnn

```

```

33632 \cs_new_eq:NN \__text_change_case_upper_az:nnnnn
33633 \__text_change_case_upper_tr:nnnnn

```

(End of definition for __text_change_case_lower_az:nnnnn and __text_change_case_upper_az:nnnnn.)
The (fixed) look-up mappings for letter-like control sequences.

```

33634 \group_begin:
33635 \cs_set_protected:Npn \__text_change_case_setup:NN #1#2
33636 {
33637 \quark_if_recursion_tail_stop:N #1
33638 \tl_const:cn { c__text_lowercase_ \token_to_str:N #1 _tl }
33639 { #2 }
33640 \tl_const:cn { c__text_uppercase_ \token_to_str:N #2 _tl }
33641 { #1 }
33642 \__text_change_case_setup:NN
33643 }
33644 \__text_change_case_setup:NN
33645 \AA \aa
33646 \AE \ae
33647 \DH \dh
33648 \DJ \dj
33649 \IJ \ij
33650 \L \l
33651 \NG \ng
33652 \O \o
33653 \OE \oe
33654 \SS \ss
33655 \TH \th
33656 \q_recursion_tail ?
33657 \q_recursion_stop
33658 \tl_const:cn { c__text_uppercase_ \token_to_str:N \i _tl } { I }
33659 \tl_const:cn { c__text_uppercase_ \token_to_str:N \j _tl } { J }
33660 \group_end:

```

To deal with possible encoding-specific extensions to \@uclclist, we check at the end of the preamble. This will therefore only apply to L^AT_EX 2_ε package mode.

```

33661 \tl_if_exist:NT \@expl@finalise@setup@@
33662 {
33663 \tl_gput_right:Nn \@expl@finalise@setup@@
33664 {
33665 \tl_gput_right:Nn \@kernel@after@begindocument
33666 {
33667 \group_begin:
33668 \cs_set_protected:Npn \__text_change_case_setup:Nn #1#2
33669 {
33670 \quark_if_recursion_tail_stop:N #1
33671 \tl_if_single_token:nT {#2}
33672 {
33673 \cs_if_exist:cF
33674 { c__text_uppercase_ \token_to_str:N #1 _tl }
33675 {
33676 \tl_const:cn
33677 { c__text_uppercase_ \token_to_str:N #1 _tl }
33678 { #2 }
33679 }

```

```

33680         \cs_if_exist:cF
33681         { c__text_lowercase_ \token_to_str:N #2 _t1 }
33682         {
33683           \tl_const:cn
33684           { c__text_lowercase_ \token_to_str:N #2 _t1 }
33685           { #1 }
33686         }
33687       }
33688     \__text_change_case_setup:Nn
33689   }
33690   \exp_after:wN \__text_change_case_setup:Nn \@uclclist
33691   \q_recursion_tail ?
33692   \q_recursion_stop
33693 \group_end:
33694 }
33695 }
33696 }

```

A few adjustments to case mapping for combining chars: these are not needed for the Unicode engines

```

33697 \bool_lazy_or:nnF
33698 { \sys_if_engine_luatex_p: }
33699 { \sys_if_engine_xetex_p: }
33700 {
33701   \text_declare_uppercase_mapping:nn { "01F0 } { \v { J } }
33702 }
33703 </package>

```


Chapter 89

13text-map implementation

```
33704 (*package)
```

```
33705 (@@=text)
```

89.1 Mapping to text

```
\text_map_function:nN
```

The standard lead-off for an action loop.

```
\__text_map_function:nN
```

```
33706 \cs_new:Npn \text_map_function:nN #1#2
```

```
\__text_map_loop:Nnw
```

```
33707 { \exp_args:Ne \__text_map_function:nN { \text_expand:n {#1} } #2 }
```

```
\__text_map_group:Nnn
```

```
33708 \cs_new:Npn \__text_map_group:nN #1#2
```

```
\__text_map_space:Nnw
```

```
33709 {
```

```
\__text_map_N_type:NnN
```

```
33710 \__text_map_loop:Nnw #2 { } #1
```

```
\__text_map_codepoint:Nnn
```

```
33711 \q__text_recursion_tail \q__text_recursion_stop
```

```
\__text_map_CR:Nnw
```

```
33712 \prg_break_point:Nn \text_map_break: { }
```

```
\__text_map_CR:NnN
```

```
33713 }
```

```
\__text_map_class:Nnnn
```

The standard set up for an “action” loop. Groups are handled by recursion, spaces are treated similarly: both count as grapheme boundaries. For N-type tokens, we filter out control sequences (again a boundary), then move on to further analysis.

```
\__text_map_class:nNnnn
```

```
33714 \cs_new:Npn \__text_map_loop:Nnw #1#2#3 \q__text_recursion_stop
```

```
\__text_map_class_loop:Nnnnw
```

```
33715 {
```

```
\__text_map_class_end:nw
```

```
33716 \tl_if_head_is_N_type:nTF {#3}
```

```
\__text_map_Control:Nnn
```

```
33717 { \__text_map_N_type:NnN }
```

```
\__text_map_Extend:Nnn
```

```
33718 {
```

```
\__text_map_SpacingMark:Nnn
```

```
33719 \tl_if_head_is_group:nTF {#3}
```

```
\__text_map_Prepend:Nnn
```

```
33720 { \__text_map_group:Nnn }
```

```
\__text_map_Prepend_aux:Nnn
```

```
33721 { \__text_map_space:Nnw }
```

```
\__text_map_Prepend_nNnn
```

```
33722 }
```

```
\__text_map_Prepend_loop:Nnnw
```

```
33723 #1 {#2} #3 \q__text_recursion_stop
```

```
\__text_map_not_Control:Nnn
```

```
33724 }
```

```
\__text_map_not_Extend:Nnn
```

```
33725 \cs_new:Npn \__text_map_group:Nnn #1#2#3
```

```
\__text_map_not_SpacingMark:Nnn
```

```
33726 {
```

```
\__text_map_not_Prepend:Nnn
```

```
33727 \__text_map_output:Nn #1 {#2}
```

```
\__text_map_not_L:Nnn
```

```
33728 {
```

```
\__text_map_not_LV:Nnn
```

```
33729 \__text_map_loop:Nnw #1 { } #2
```

```
\__text_map_not_V:Nnn
```

```
33730 \q__text_recursion_tail \q__text_recursion_stop
```

```
\__text_map_not_LVT:Nnn
```

```
33731 \prg_break_point:Nn \text_map_break: { }
```

```
\__text_map_not_T:Nnn
```

```
33732 }
```

```
\__text_map_L:Nnn
```

```
\__text_map_LV:Nnn
```

```
\__text_map_V:Nnn
```

```
\__text_map_LVT:Nnn
```

```
\__text_map_T:Nnn
```

```
\__text_map_hangul:Nnnw
```

```
\__text_map_hangul:NnnN
```

```
\__text_map_hangul:Nnnn
```

```
\__text_map_hangul_aux:Nnnnw
```

```
\__text_map_hangul_nNnnnw
```

```
\__text_map_hangul_loop:Nnnnw
```

```

33733     \__text_map_loop:Nnw #1 { }
33734   }
33735 \use:e
33736 { \cs_new:Npn \exp_not:N \__text_map_space:Nnw #1#2 \c_space_tl }
33737 {
33738   \__text_map_output:Nn #1 {#2}
33739   #1 { ~ }
33740   \__text_map_loop:Nnw #1 { }
33741 }
33742 \cs_new:Npn \__text_map_N_type:NnN #1#2#3
33743 {
33744   \__text_if_q_recursion_tail_stop_do:Nn #3
33745   {
33746     \__text_map_output:Nn #1 {#2}
33747     \text_map_break:
33748   }
33749   \token_if_cs:NTF #3
33750   {
33751     \__text_map_output:Nn #1 {#2}
33752     #1 {#3}
33753     \__text_map_loop:Nnw #1 { }
33754   }
33755   {
33756     \__text_codepoint_process:nN
33757     { \__text_map_codepoint:Nnn #1 {#2} } #3
33758   }
33759 }

```

We pull out a few special cases here. Carriage returns case needs a bit of context handling so has an auxiliary. Codepoint U+200D is the zero-width joiner, which has no context to concern us: just don't break.

```

33760 \cs_new:Npn \__text_map_codepoint:Nnn #1#2#3
33761 {
33762   \__text_codepoint_compare:nNnTF {#3} = { "0D }
33763   {
33764     \__text_map_output:Nn #1 {#2}
33765     \__text_map_CR:Nnw #1 {#3}
33766   }
33767   {
33768     \__text_codepoint_compare:nNnTF {#3} = { "200D }
33769     { \__text_map_loop:Nnw #1 {#2#3} }
33770     { \__text_map_class:Nnnn #1 {#2} {#3} { Control } }
33771   }
33772 }

```

A carriage return is a boundary unless it is immediately followed by a line feed, in which case that pair is a boundary.

```

33773 \cs_new:Npn \__text_map_CR:Nnw #1#2#3 \q__text_recursion_stop
33774 {
33775   \tl_if_head_is_N_type:nTF {#3}
33776   { \__text_map_CR:NnN #1 {#2} }
33777   {
33778     #1 {#2}
33779     \__text_map_loop:Nnw #1 { }
33780   }

```

```

33781         #3 \q__text_recursion_stop
33782     }
33783 \cs_new:Npn \__text_map_CR:NnN #1#2#3
33784 {
33785     \__text_if_q_recursion_tail_stop_do:Nn #3
33786     {
33787         #1 {#2}
33788         \text_map_break:
33789     }
33790     \bool_lazy_and:nnTF
33791     { ! \token_if_cs_p:N #3 }
33792     { \int_compare_p:nNn { '#3 } = { "0A } }
33793     {
33794         \__text_map_output:Nn #1 {#2#3}
33795         \__text_map_loop:Nnw #1 { }
33796     }
33797     { \__text_map_loop:Nnw #1 { } #3 }
33798 }

```

There are various classes of character, and we deal with them all in the same general way. We need to example the relevant list of codepoints: if we get a hit, then we do whatever the relevant action is. Otherwise we loop, but only if the current codepoint could still match: the loop stops early otherwise and we move forward.

```

33799 \cs_new:Npn \__text_map_class:Nnnn #1#2#3#4
33800 {
33801     \exp_args:Nv \__text_map_class:nNnnn { c__text_grapheme_ #4 _clist }
33802     #1 {#2} {#3} {#4}
33803 }
33804 \cs_new:Npn \__text_map_class:nNnnn #1#2#3#4#5
33805 {
33806     \__text_map_class_loop:Nnnnw #2 {#3} {#4} {#5}
33807     #1 , \q__text_recursion_tail .. , \q__text_recursion_stop
33808 }
33809 \cs_new:Npn \__text_map_class_loop:Nnnnw #1#2#3#4 #5 .. #6 ,
33810 {
33811     \__text_if_q_recursion_tail_stop_do:nn {#5}
33812     { \use:c { __text_map_not_ #4 :Nnn } #1 {#2} {#3} }
33813     \__text_codepoint_compare:nNnTF {#3} < { "#5 }
33814     {
33815         \__text_map_class_end:nw
33816         { \use:c { __text_map_not_ #4 :Nnn } #1 {#2} {#3} }
33817     }
33818     {
33819         \__text_codepoint_compare:nNnTF {#3} > { "#6 }
33820         { \__text_map_class_loop:Nnnnw #1 {#2} {#3} {#4} }
33821         {
33822             \__text_map_class_end:nw
33823             { \use:c { __text_map_ #4 :Nnn } #1 {#2} {#3} }
33824         }
33825     }
33826 }
33827 \cs_new:Npn \__text_map_class_end:nw #1#2 \q__text_recursion_stop {#1}

```

Break before *and* after.

```

33828 \cs_new:Npn \__text_map_Control:Nnn #1#2#3

```

```

33829 {
33830   \_text_map_output:Nn #1 {#2}
33831   \_text_map_output:Nn #1 {#3}
33832   \_text_map_loop:Nnw #1 { }
33833 }

```

Keep collecting.

```

33834 \cs_new:Npn \_text_map_Extend:Nnn #1#2#3
33835 { \_text_map_loop:Nnw #1 {#2#3} }
33836 \cs_new_eq:NN \_text_map_SpacingMark:Nnn \_text_map_Extend:Nnn

```

Outputting anything earlier, the combine with what follows. The only exclusions are control characters.

```

33837 \cs_new:Npn \_text_map_Prepend:Nnn #1#2#3
33838 {
33839   \_text_map_output:Nn #1 {#2}
33840   \_text_map_lookahead:Nnw #1 {#3} \_text_map_Prepend_aux:Nnn
33841 }
33842 \cs_new:Npn \_text_map_Prepend_aux:Nnn #1#2#3
33843 {
33844   \bool_lazy_or:nnTF
33845     { \_text_codepoint_compare_p:nNn {#3} = { "0A } }
33846     { \_text_codepoint_compare_p:nNn {#3} = { "0D } }
33847     {
33848       #1 {#2}
33849       \_text_map_loop:Nnw #1 {#3}
33850     }
33851     {
33852       \exp_args:NV \_text_map_Prepend:nNnn
33853         \c_text_grapheme_Control_clist
33854         #1 {#2} {#3}
33855     }
33856 }
33857 \cs_new:Npn \_text_map_Prepend:nNnn #1#2#3#4
33858 {
33859   \_text_map_Prepend_loop:Nnnw #2 {#3} {#4}
33860   #1 , \q_text_recursion_tail .. , \q_text_recursion_stop
33861 }
33862 \cs_new:Npn \_text_map_Prepend_loop:Nnnw #1#2#3 #4 .. #5 ,
33863 {
33864   \_text_if_q_recursion_tail_stop_do:nn {#4}
33865   { \_text_map_loop:Nnw #1 {#2#3} }
33866   \_text_codepoint_compare:nNnTF {#3} < { "#4 }
33867   {
33868     \_text_map_class_end:nw
33869     { \_text_map_loop:Nnw #1 {#2#3} }
33870   }
33871   {
33872     \_text_codepoint_compare:nNnTF {#3} > { "#5 }
33873     { \_text_map_Prepend_loop:Nnnw #1 {#2} {#3} }
33874     {
33875       \_text_map_class_end:nw
33876       { \_text_map_loop:Nnw #1 {#2} #3 }
33877     }
33878   }

```

```
33879 }
```

Dealing with end-of-class is done such that we can be flexible.

```
33880 \cs_new:Npn \__text_map_not_Control:Nnn #1#2#3
33881 { \__text_map_class:Nnnn #1 {#2} {#3} { Extend } }
33882 \cs_new:Npn \__text_map_not_Extend:Nnn #1#2#3
33883 { \__text_map_class:Nnnn #1 {#2} {#3} { SpacingMark } }
33884 \cs_new:Npn \__text_map_not_SpacingMark:Nnn #1#2#3
33885 { \__text_map_class:Nnnn #1 {#2} {#3} { Prepend } }
33886 \cs_new:Npn \__text_map_not_Prepend:Nnn #1#2#3
33887 { \__text_map_class:Nnnn #1 {#2} {#3} { L } }
33888 \cs_new:Npn \__text_map_not_L:Nnn #1#2#3
33889 { \__text_map_class:Nnnn #1 {#2} {#3} { LV } }
33890 \cs_new:Npn \__text_map_not_LV:Nnn #1#2#3
33891 { \__text_map_class:Nnnn #1 {#2} {#3} { V } }
33892 \cs_new:Npn \__text_map_not_V:Nnn #1#2#3
33893 { \__text_map_class:Nnnn #1 {#2} {#3} { LVT } }
33894 \cs_new:Npn \__text_map_not_LVT:Nnn #1#2#3
33895 { \__text_map_class:Nnnn #1 {#2} {#3} { T } }
33896 \cs_new:Npn \__text_map_not_T:Nnn #1#2#3
33897 { \__text_map_class:Nnnn #1 {#2} {#3} { Regional_Indicator } }
33898 \cs_new:Npn \__text_map_not_Regional_Indicator:Nnn #1#2#3
33899 {
33900   \__text_map_output:Nn #1 {#2}
33901   \__text_map_loop:Nnw #1 {#3}
33902 }
```

Hangul needs additional treatment. First we have to deal with the start-of-Hangul position: output what we had up to now, then move the specialist handler. The idea here is to pick off the different codepoint types one at a time, tracking what else can be considered at each stage until we hit the end of the viable types. Other than that, we just keep building up the Hangul codepoints using a dedicated version of the loop from above.

```
33903 \cs_new:Npn \__text_map_L:Nnn #1#2#3
33904 {
33905   \__text_map_output:Nn #1 {#2}
33906   \__text_map_hangul:Nnnw
33907   #1 {#3} { L ; V ; LV ; LVT }
33908 }
33909 \cs_new:Npn \__text_map_LV:Nnn #1#2#3
33910 {
33911   \__text_map_output:Nn #1 {#2}
33912   \__text_map_hangul:Nnnw
33913   #1 {#3} { V ; T }
33914 }
33915 \cs_new_eq:NN \__text_map_V:Nnn \__text_map_LV:Nnn
33916 \cs_new:Npn \__text_map_LVT:Nnn #1#2#3
33917 {
33918   \__text_map_output:Nn #1 {#2}
33919   \__text_map_hangul:Nnnw
33920   #1 {#3} { T }
33921 }
33922 \cs_new_eq:NN \__text_map_T:Nnn \__text_map_LVT:Nnn
33923 \cs_new:Npn \__text_map_hangul:Nnnw #1#2#3#4 \q__text_recursion_stop
33924 {
33925   \tl_if_head_is_N_type:nTF {#4}
```

```

33926     { \_text_map_hangul:NnnN #1 {#2} {#3} }
33927     {
33928         #1 {#2}
33929         \_text_map_loop:Nnw #1 { }
33930     }
33931     #4 \q_text_recursion_stop
33932 }
33933 \cs_new:Npn \_text_map_hangul:NnnN #1#2#3#4
33934 {
33935     \_text_if_q_recursion_tail_stop_do:Nn #4
33936     {
33937         #1 {#2}
33938         \text_map_break:
33939     }
33940     \token_if_cs:NTF #4
33941     {
33942         #1 {#2}
33943         \_text_map_loop:Nnw #1 { }
33944     }
33945     {
33946         \_text_codepoint_process:nN
33947         { \_text_map_hangul:Nnnn #1 {#2} {#3} } #4
33948     }
33949 }
33950 \cs_new:Npn \_text_map_hangul:Nnnn #1#2#3#4
33951 {
33952     \_text_map_hangul_aux:Nnnw #1 {#2} {#4}
33953     #3 ; \q_recursion_tail ; \q_recursion_stop
33954 }
33955 \cs_new:Npn \_text_map_hangul_aux:Nnnw #1#2#3#4 ;
33956 {
33957     \quark_if_recursion_tail_stop_do:nn {#4}
33958     { \_text_map_loop:Nnw #1 {#2} #3 }
33959     \exp_args:Nv \_text_map_hangul:nNnnnw { c__text_grapheme_ #4 _clist }
33960     #1 {#2} {#3} {#4}
33961 }
33962 \cs_new:Npn \_text_map_hangul:nNnnnw #1#2#3#4#5#6 \q_recursion_stop
33963 {
33964     \_text_map_hangul_loop:Nnnnnw #2 {#3} {#4} {#5} {#6}
33965     #1 , \q_text_recursion_tail .. , \q_text_recursion_stop
33966 }
33967 \cs_new:Npn \_text_map_hangul_loop:Nnnnnw #1#2#3#4#5 #6 .. #7 ,
33968 {
33969     \_text_if_q_recursion_tail_stop_do:nn {#6}
33970     { \_text_map_hangul_next:Nnnn #1 {#2} {#3} {#5} }
33971     \_text_codepoint_compare:nNnTF {#3} < { "#6 }
33972     {
33973         \_text_map_hangul_end:nw
33974         { \_text_map_hangul_next:Nnnn #1 {#2} {#3} {#5} }
33975     }
33976     {
33977         \_text_codepoint_compare:nNnTF {#3} > { "#7 }
33978         { \_text_map_hangul_loop:Nnnnnw #1 {#2} {#3} {#4} {#5} }
33979     }

```

```

33980     \__text_map_hangul_end:nw
33981     { \use:c { __text_map_hangul_ #4 :Nnn } #1 {#2} {#3} }
33982   }
33983 }
33984 }
33985 \cs_new:Npn \__text_map_hangul_next:Nnnn #1#2#3#4
33986 { \__text_map_hangul_aux:Nnnw #1 {#2} {#3} #4 \q_recursion_stop }
33987 \cs_new:Npn \__text_map_hangul_end:nw #1#2 \q__text_recursion_stop {#1}
33988 \cs_new:Npn \__text_map_hangul_L:Nnn #1#2#3
33989 {
33990   \__text_map_hangul:Nnnw
33991   #1 {#2#3} { L V { LV } { LVT } }
33992 }
33993 \cs_new:Npn \__text_map_hangul_LV:Nnn #1#2#3
33994 {
33995   \__text_map_hangul:Nnnw
33996   #1 {#2#3} { VT }
33997 }
33998 \cs_new_eq:NN \__text_map_hangul_V:Nnn \__text_map_hangul_LV:Nnn
33999 \cs_new:Npn \__text_map_hangul_LVT:Nnn #1#2#3
34000 {
34001   \__text_map_hangul:Nnnw
34002   #1 {#2#3} { T }
34003 }
34004 \cs_new_eq:NN \__text_map_hangul_T:Nnn \__text_map_hangul_LVT:Nnn

```

The Regional Indicator rule means looking ahead and dealing with the case where there are two in a row. So we use a look ahead to pick them off. As there is only one range the values are hard-coded.

```

34005 \cs_new:Npn \__text_map_Regional_Indicator:Nnn #1#2#3
34006 {
34007   \__text_map_output:Nn #1 {#2}
34008   \__text_map_lookahead:NnNw #1 {#3} \__text_map_Regional_Indicator_aux:Nnn
34009 }
34010 \cs_new:Npn \__text_map_Regional_Indicator_aux:Nnn #1#2#3
34011 {
34012   \bool_lazy_or:nnTF
34013   { \__text_codepoint_compare_p:nNn {#3} < { "1F1E6 } }
34014   { \__text_codepoint_compare_p:nNn {#3} > { "1F1FF } }
34015   {
34016     \__text_map_loop:Nnw #1 {#2} #3
34017   }
34018   { \__text_map_loop:Nnw #1 {#2#3} }
34019 }

```

A generic loop-ahead setup.

```

34020 \cs_new:Npn \__text_map_lookahead:NnNw #1#2#3#4 \q__text_recursion_stop
34021 {
34022   \tl_if_head_is_N_type:nTF {#4}
34023   { \__text_map_lookahead:NnNN #1 {#2} #3 }
34024   { \__text_map_loop:Nnw #1 {#2} }
34025   #4 \q__text_recursion_stop
34026 }
34027 \cs_new:Npn \__text_map_lookahead:NnNN #1#2#3#4
34028 {

```

```

34029  \_text_if_q_recursion_tail_stop_do:Nn #4 { #1 {#2} }
34030  \token_if_cs:NTF #4
34031  {
34032    #1 {#2}
34033    \_text_map_loop:Nnw #1 { }
34034  }
34035  { \_text_codepoint_process:nN { #3 #1 {#2} } }
34036    #4
34037  }

```

For the end of the process.

```

34038 \cs_new:Npn \_text_map_output:Nn #1#2
34039   { \tl_if_blank:nF {#2} { #1 {#2} } }
34040 \cs_new:Npn \text_map_break:
34041   { \prg_map_break:Nn \text_map_break: { } }
34042 \cs_new:Npn \text_map_break:n
34043   { \prg_map_break:Nn \text_map_break: }

```

(End of definition for \text_map_function:nN and others. These functions are documented on page 300.)

\text_map_inline:nn The standard non-expandable inline version.

```

34044 \cs_new_protected:Npn \text_map_inline:nn #1#2
34045   {
34046     \int_gincr:N \g__kernel_prg_map_int
34047     \cs_gset_protected:cpn
34048       { __text_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
34049     \exp_args:Nnc \text_map_function:nN {#1}
34050       { __text_map_ \int_use:N \g__kernel_prg_map_int :w }
34051     \prg_break_point:Nn \text_map_break:
34052       { \int_gdecr:N \g__kernel_prg_map_int }
34053   }

```

(End of definition for \text_map_inline:nn. This function is documented on page 300.)

```

34054 </package>

```


Chapter 90

l3text-purify implementation

```
34055 (*package)
```

```
34056 (@@=text)
```

90.1 Purifying text

```
\_text_if_recursion_tail_stop:N Functions to query recursion quarks.
```

```
34057 \\_kernel_quark_new_test:N \_text_if_recursion_tail_stop:N
```

(End of definition for _text_if_recursion_tail_stop:N.)

```
\text_purify:n
```

As in the other parts of the module, we start off with a standard “action” loop, with expansion applied up-front.

```
\_text_purify:n
```

```
\_text_purify_store:n
```

```
34058 \cs_new:Npn \text_purify:n #1
```

```
\_text_purify_store:nw
```

```
34059 {
```

```
\_text_purify_end:w
```

```
34060 \_kernel_exp_not:w \exp_after:wN
```

```
\_text_purify_loop:w
```

```
34061 {
```

```
\_text_purify_group:n
```

```
\exp:w
```

```
\_text_purify_space:w
```

```
34062 \exp_args:Ne \_text_purify:n
```

```
\_text_purify_N_type:N
```

```
34063 { \text_expand:n {#1} }
```

```
\_text_purify_N_type_aux:N
```

```
34064 }
```

```
\_text_purify_math_search:NNN
```

```
34065 }
```

```
\_text_purify_math_start:NNw
```

```
34066 }
```

```
\_text_purify_math_store:n
```

```
34067 \cs_new:Npn \_text_purify:n #1
```

```
\_text_purify_math_store:nw
```

```
34068 {
```

```
\_text_purify_math_end:w
```

```
34069 \group_align_safe_begin:
```

```
\_text_purify_math_loop:NNw
```

```
34070 \_text_purify_loop:w #1
```

```
\_text_purify_math_N_type:NNN
```

```
34071 \q\_text_recursion_tail \q\_text_recursion_stop
```

```
\_text_purify_math_group:NNn
```

```
34072 \_text_purify_result:n { }
```

```
\_text_purify_math_space:NNw
```

```
34073 }
```

As for expansion, collect up the tokens for future use.

```
34074 \cs_new:Npn \_text_purify_store:n #1
```

```
34075 { \_text_purify_store:nw {#1} }
```

```
34076 \cs_new:Npn \_text_purify_store:nw #1#2 \_text_purify_result:n #3
```

```
34077 { #2 \_text_purify_result:n { #3 #1 } }
```

```
34078 \cs_new:Npn \_text_purify_end:w #1 \_text_purify_result:n #2
```

```
34079 {
```

```
\_text_purify_replace_auxi:n
```

```
34080 \group_align_safe_end:
```

```
\_text_purify_replace_auxii:n
```

```
34081 \exp_end:
```

```
\_text_purify_expand:N
```

```
\_text_purify_protect:N
```

```
\_text_purify_encoding:N
```

```
\_text_purify_encoding_escape:NN
```

```

34082     #2
34083   }

```

The main loop is a standard “tl action”. Unlike the expansion or case changing, here any groups have to be run inline. Most of the business end is as before in the N-type token processing.

```

34084 \cs_new:Npn \__text_purify_loop:w #1 \q__text_recursion_stop
34085   {
34086     \tl_if_head_is_N_type:nTF {#1}
34087       { \__text_purify_N_type:N }
34088       {
34089         \tl_if_head_is_group:nTF {#1}
34090           { \__text_purify_group:n }
34091           { \__text_purify_space:w }
34092       }
34093     #1 \q__text_recursion_stop
34094   }
34095 \cs_new:Npn \__text_purify_group:n #1 { \__text_purify_loop:w #1 }
34096 \exp_last_unbraced:NNo \cs_new:Npn \__text_purify_space:w \c_space_tl
34097   {
34098     \__text_purify_store:n { ~ }
34099     \__text_purify_loop:w
34100   }

```

The first part of handling math mode is exactly the same as in the other functions: look for a start-of-math mode token and if found start a new loop tracking the closing token.

```

34101 \cs_new:Npn \__text_purify_N_type:N #1
34102   {
34103     \__text_if_q_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
34104     \__text_purify_N_type_aux:N #1
34105   }
34106 \cs_new:Npn \__text_purify_N_type_aux:N #1
34107   {
34108     \exp_after:wN \__text_purify_math_search:NNN
34109     \exp_after:wN #1 \l_text_math_delims_tl
34110     \q__text_recursion_tail ?
34111     \q__text_recursion_stop
34112   }
34113 \cs_new:Npn \__text_purify_math_search:NNN #1#2#3
34114   {
34115     \__text_if_q_recursion_tail_stop_do:Nn #2
34116       { \__text_purify_math_cmd:N #1 }
34117     \token_if_eq_meaning:NNTF #1 #2
34118       {
34119         \__text_use_i_delimit_by_q_recursion_stop:nw
34120         { \__text_purify_math_start:NNw #2 #3 }
34121       }
34122     { \__text_purify_math_search:NNN #1 }
34123   }
34124 \cs_new:Npn \__text_purify_math_start:NNw #1#2#3 \q__text_recursion_stop
34125   {
34126     \__text_purify_math_loop:NNw #1#2#3 \q__text_recursion_stop
34127     \__text_purify_math_result:n { }
34128   }
34129 \cs_new:Npn \__text_purify_math_store:n #1

```

```

34130 { \_text_purify_math_store:nw {#1} }
34131 \cs_new:Npn \_text_purify_math_store:nw #1#2 \_text_purify_math_result:n #3
34132 { #2 \_text_purify_math_result:n { #3 #1 } }
34133 \cs_new:Npn \_text_purify_math_end:w #1 \_text_purify_math_result:n #2
34134 {
34135   \_text_purify_store:n { $ #2 $ }
34136   \_text_purify_loop:w #1
34137 }
34138 \cs_new:Npn \_text_purify_math_stop:Nw #1 \_text_purify_math_result:n #2
34139 {
34140   \_text_purify_store:n {#1#2}
34141   \_text_purify_end:w
34142 }
34143 \cs_new:Npn \_text_purify_math_loop:NNw #1#2#3 \q__text_recursion_stop
34144 {
34145   \tl_if_head_is_N_type:nTF {#3}
34146     { \_text_purify_math_N_type:NNN }
34147     {
34148       \tl_if_head_is_group:nTF {#3}
34149         { \_text_purify_math_group:NNn }
34150         { \_text_purify_math_space:NNw }
34151     }
34152     #1#2#3 \q__text_recursion_stop
34153 }
34154 \cs_new:Npn \_text_purify_math_N_type:NNN #1#2#3
34155 {
34156   \_text_if_q_recursion_tail_stop_do:Nn #3
34157   { \_text_purify_math_stop:Nw #1 }
34158   \token_if_eq_meaning:NNTF #3 #2
34159   { \_text_purify_math_end:w }
34160   {
34161     \_text_purify_math_store:n {#3}
34162     \_text_purify_math_loop:NNw #1#2
34163   }
34164 }
34165 \cs_new:Npn \_text_purify_math_group:NNn #1#2#3
34166 {
34167   \_text_purify_math_store:n { {#3} }
34168   \_text_purify_math_loop:NNw #1#2
34169 }
34170 \exp_after:wN \cs_new:Npn \exp_after:wN \_text_purify_math_space:NNw
34171 \exp_after:wN # \exp_after:wN 1
34172 \exp_after:wN # \exp_after:wN 2 \c_space_tl
34173 {
34174   \_text_purify_math_store:n { ~ }
34175   \_text_purify_math_loop:NNw #1#2
34176 }

```

Then handle math mode as an argument: same outcomes, different input syntax.

```

34177 \cs_new:Npn \_text_purify_math_cmd:N #1
34178 {
34179   \exp_after:wN \_text_purify_math_cmd:NN \exp_after:wN #1
34180   \l_text_math_arg_tl \q__text_recursion_tail \q__text_recursion_stop
34181 }
34182 \cs_new:Npn \_text_purify_math_cmd:NN #1#2

```

```

34183 {
34184   \_text_if_q_recursion_tail_stop_do:Nn #2
34185   { \_text_purify_replace:N #1 }
34186   \cs_if_eq:NNTF #2 #1
34187   {
34188     \_text_use_i_delimit_by_q_recursion_stop:nw
34189     { \_text_purify_math_cmd:n }
34190   }
34191   { \_text_purify_math_cmd:NN #1 }
34192 }
34193 \cs_new:Npn \_text_purify_math_cmd:n #1
34194 { \_text_purify_math_end:w \_text_purify_math_result:n {#1} }

```

For N-type tokens, we first look for a string-context replacement before anything else: this can therefore cover anything. Assuming we don't find one, check to see if we can expand control sequences: if not, they have to be dropped. We also allow for L^AT_EX 2_ε \protect: there's an assumption that we don't have \protect { \oops } or similar, but that's also in the expansion code and seems like a reasonable balance.

```

34195 \cs_new:Npn \_text_purify_replace:N #1
34196 {
34197   \bool_lazy_and:nnTF
34198   { \cs_if_exist_p:c { l__text_purify_ \token_to_str:N #1 _t1 } }
34199   {
34200     \bool_lazy_or_p:nn
34201     { \token_if_cs_p:N #1 }
34202     { \token_if_active_p:N #1 }
34203   }
34204   {
34205     \exp_args:Nv \_text_purify_replace_auxi:n
34206     { l__text_purify_ \token_to_str:N #1 _t1 }
34207   }
34208   {
34209     \exp_args:Ne \_text_purify_replace_auxii:n
34210     { \_text_token_to_explicit:N #1 }
34211   }
34212 }
34213 \cs_new:Npn \_text_purify_replace_auxi:n #1 { \_text_purify_loop:w #1 }
34214 \cs_new:Npn \_text_purify_replace_auxii:n #1
34215 {
34216   \token_if_cs:NNTF #1
34217   { \_text_purify_expand:N #1 }
34218   {
34219     \_text_purify_store:n {#1}
34220     \_text_purify_loop:w
34221   }
34222 }
34223 \cs_new:Npn \_text_purify_expand:N #1
34224 {
34225   \str_if_eq:nnTF {#1} { \protect }
34226   { \_text_purify_protect:N }
34227   { \_text_purify_encoding:N #1 }
34228 }
34229 \cs_new:Npn \_text_purify_protect:N #1
34230 {

```

```

34231     \_text_if_q_recursion_tail_stop_do:Nn #1 { \_text_purify_end:w }
34232     \_text_purify_loop:w
34233 }

```

Handle encoding commands, as detailed for expansion.

```

34234 \cs_new:Npn \_text_purify_encoding:N #1
34235 {
34236     \bool_lazy_or:nnTF
34237     { \cs_if_eq_p:NN #1 \@current@cmd }
34238     { \cs_if_eq_p:NN #1 \@changed@cmd }
34239     { \_text_purify_encoding_escape:NN }
34240     {
34241         \_text_if_expandable:NTF #1
34242         { \exp_after:wN \_text_purify_loop:w #1 }
34243         { \_text_purify_loop:w }
34244     }
34245 }
34246 \cs_new:Npn \_text_purify_encoding_escape:NN #1#2
34247 {
34248     \_text_purify_store:n {#1}
34249     \_text_purify_loop:w
34250 }

```

(End of definition for `\text_purify:n` and others. This function is documented on page 299.)

```

\text_declare_purify_equivalent:Nn
\text_declare_purify_equivalent:Ne

```

```

34251 \cs_new_protected:Npn \text_declare_purify_equivalent:Nn #1#2
34252 {
34253     \tl_clear_new:c { l__text_purify_ \token_to_str:N #1 _tl }
34254     \tl_set:cn { l__text_purify_ \token_to_str:N #1 _tl } {#2}
34255 }
34256 \cs_generate_variant:Nn \text_declare_purify_equivalent:Nn { Ne }

```

(End of definition for `\text_declare_purify_equivalent:Nn`. This function is documented on page 299.)

Now pre-define a range of standard commands that need dedicated definitions in purified text. First handle font-related stuff: all of this needs to be disabled.

```

34257 \tl_map_inline:nn
34258 {
34259     \fontencoding
34260     \fontfamily
34261     \fontseries
34262     \fontshape
34263 }
34264 { \text_declare_purify_equivalent:Nn #1 { \use_none:n } }
34265 \text_declare_purify_equivalent:Nn \fontsize { \use_none:nn }
34266 \text_declare_purify_equivalent:Nn \selectfont { }
34267 \text_declare_purify_equivalent:Nn \usefont { \use_none:nmmn }
34268 \tl_map_inline:nn
34269 {
34270     \emph
34271     \text
34272     \textnormal
34273     \textrm
34274     \textsf
34275     \texttt

```

```

34276     \textbf
34277     \textmd
34278     \textit
34279     \textsl
34280     \textup
34281     \textsc
34282     \textulc
34283   }
34284   { \text_declare_purify_equivalent:Nn #1 { \use:n } }
34285 \tl_map_inline:nn
34286   {
34287     \normalfont
34288     \rmfamily
34289     \sffamily
34290     \ttfamily
34291     \bfseries
34292     \mdseries
34293     \itshape
34294     \scshape
34295     \slshape
34296     \upshape
34297     \em
34298     \Huge
34299     \LARGE
34300     \Large
34301     \footnotesize
34302     \huge
34303     \large
34304     \normalsize
34305     \scriptsize
34306     \small
34307     \tiny
34308   }
34309   { \text_declare_purify_equivalent:Nn #1 { } }
34310 \exp_args:Nc \text_declare_purify_equivalent:Nn
34311   { @protected@testopt } { \use_none:nmn }

```

Environments have to be handled by pure expansion.

`__text_end_env:n`

```

34312 \text_declare_purify_equivalent:Nn \begin { \use:c }
34313 \text_declare_purify_equivalent:Nn \end { \__text_end_env:n }
34314 \cs_new:Npn \__text_end_env:n #1 { \cs:w end #1 \cs_end: }

```

(End of definition for __text_end_env:n.)

Some common symbols and similar ideas.

```

34315 \text_declare_purify_equivalent:Nn \\ { }
34316 \tl_map_inline:nn
34317   { \{ \} \# \$ \% \_ }
34318   { \text_declare_purify_equivalent:Ne #1 { \cs_to_str:N #1 } }

```

Cross-referencing.

```

34319 \text_declare_purify_equivalent:Nn \label { \use_none:n }

```

Spaces.

```
34320 \group_begin:
34321 \char_set_catcode_active:N \~
34322 \use:n
34323   {
34324     \group_end:
34325     \text_declare_purify_equivalent:Ne ~ { \c_space_tl }
34326   }
34327 \text_declare_purify_equivalent:Nn \nobreakspace { ~ }
34328 \text_declare_purify_equivalent:Nn \ { ~ }
34329 \text_declare_purify_equivalent:Nn \, { ~ }
```

90.2 Accent and letter-like data for purifying text

In contrast to case changing, both 8-bit and Unicode engines need information for text purification to handle accents and letter-like functions: these all need to be removed. However, the results are of course engine-dependent.

For the letter-like commands, life is relatively easy: they are all simply added as standard exceptions. The only oddity is `\SS`, which gets converted to two letters. (At some stage an alternative version can presumably be added to `babel` or similar.)

```
34330 \cs_set_protected:Npn \__text_loop:Nn #1#2
34331   {
34332     \quark_if_recursion_tail_stop:N #1
34333     \text_declare_purify_equivalent:Ne #1
34334     {
34335       \codepoint_generate:nm {"#2}
34336       { \char_value_catcode:n {"#2} }
34337     }
34338     \__text_loop:Nn
34339   }
34340 \__text_loop:Nn
34341 \AA { 00C5 }
34342 \AE { 00C6 }
34343 \DH { 00D0 }
34344 \DJ { 0110 }
34345 \IJ { 0132 }
34346 \L { 0141 }
34347 \NG { 014A }
34348 \O { 00D8 }
34349 \OE { 0152 }
34350 \TH { 00DE }
34351 \aa { 00E5 }
34352 \ae { 00E6 }
34353 \dh { 00F0 }
34354 \dj { 0111 }
34355 \i { 0131 }
34356 \j { 0237 }
34357 \ij { 0132 }
34358 \l { 0142 }
34359 \ng { 014B }
34360 \o { 00F8 }
34361 \oe { 0153 }
```

```

34362 \ss { 00DF }
34363 \th { 00FE }
34364 \q_recursion_tail ?
34365 \q_recursion_stop
34366 \text_declare_purify_equivalent:Nn \SS { SS }

```

_text_purify_accent:NN Accent LICR handling is a little more complex. Accents may exist as pre-composed codepoints or as independent glyphs. The former are all saved as single token lists, whilst for the latter the combining accent needs to be re-ordered compared to the character it applies to.

```

34367 \cs_new:Npn \_text_purify_accent:NN #1#2
34368 {
34369   \cs_if_exist:cTF
34370     { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
34371     {
34372       \exp_not:v
34373         { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
34374     }
34375     {
34376       \exp_not:n {#2}
34377       \exp_not:v { c__text_purify_ \token_to_str:N #1 _tl }
34378     }
34379 }
34380 \tl_map_inline:nn { \' \' \^ \~ \= \u \. \" \r \H \v \d \c \k \b \t }
34381 { \text_declare_purify_equivalent:Nn #1 { \_text_purify_accent:NN #1 } }

```

First set up the combining accents.

```

34382 \group_begin:
34383   \cs_set_protected:Npn \_text_loop:Nn #1#2
34384     {
34385       \quark_if_recursion_tail_stop:N #1
34386       \tl_const:ce { c__text_purify_ \token_to_str:N #1 _tl }
34387       { \codepoint_generate:nn {"#2} { \char_value_catcode:n { "#2 } } }
34388       \_text_loop:Nn
34389     }
34390   \_text_loop:Nn
34391   \' { 0300 }
34392   \' { 0301 }
34393   \^ { 0302 }
34394   \~ { 0303 }
34395   \= { 0304 }
34396   \u { 0306 }
34397   \. { 0307 }
34398   \" { 0308 }
34399   \r { 030A }
34400   \H { 030B }
34401   \v { 030C }
34402   \d { 0323 }
34403   \c { 0327 }
34404   \k { 0328 }
34405   \b { 0331 }
34406   \t { 0361 }
34407   \q_recursion_tail { }
34408   \q_recursion_stop

```


Now we handle the pre-composed accents: the list here is taken from `puenc.def`. All of the precomposed cases take a single letter as their second argument. We do not try to cover the case where an accent is added to a “real” dotless-i or -j, or a æ/Æ. Rather, we assume that if the UTF-8 character is used, it will have the real accent character too.

```

34409 \cs_set_protected:Npn \__text_loop:NNn #1#2#3
34410 {
34411   \quark_if_recursion_tail_stop:N #1
34412   \tl_const:ce
34413   { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
34414   { \codepoint_generate:nn {"#3} { \char_value_catcode:n { "#3 } } }
34415   \__text_loop:NNn
34416 }
34417 \__text_loop:NNn
34418 \‘ A { 00C0 }
34419 \’ A { 00C1 }
34420 \^ A { 00C2 }
34421 \~ A { 00C3 }
34422 \" A { 00C4 }
34423 \r A { 00C5 }
34424 \c C { 00C7 }
34425 \‘ E { 00C8 }
34426 \’ E { 00C9 }
34427 \^ E { 00CA }
34428 \" E { 00CB }
34429 \‘ I { 00CC }
34430 \’ I { 00CD }
34431 \^ I { 00CE }
34432 \" I { 00CF }
34433 \~ N { 00D1 }
34434 \‘ O { 00D2 }
34435 \’ O { 00D3 }
34436 \^ O { 00D4 }
34437 \~ O { 00D5 }
34438 \" O { 00D6 }
34439 \‘ U { 00D9 }
34440 \’ U { 00DA }
34441 \^ U { 00DB }
34442 \" U { 00DC }
34443 \’ Y { 00DD }
34444 \‘ a { 00E0 }
34445 \’ a { 00E1 }
34446 \^ a { 00E2 }
34447 \~ a { 00E3 }
34448 \" a { 00E4 }
34449 \r a { 00E5 }
34450 \c c { 00E7 }
34451 \‘ e { 00E8 }
34452 \’ e { 00E9 }
34453 \^ e { 00EA }
34454 \" e { 00EB }
34455 \‘ i { 00EC }
34456 \‘ \i { 00EC }
34457 \’ i { 00ED }
34458 \’ \i { 00ED }

```

34459	\^ i	{ 00EE }
34460	\^ \i	{ 00EE }
34461	\" i	{ 00EF }
34462	\" \i	{ 00EF }
34463	\~ n	{ 00F1 }
34464	\' o	{ 00F2 }
34465	\' o	{ 00F3 }
34466	\^ o	{ 00F4 }
34467	\~ o	{ 00F5 }
34468	\" o	{ 00F6 }
34469	\' u	{ 00F9 }
34470	\' u	{ 00FA }
34471	\^ u	{ 00FB }
34472	\" u	{ 00FC }
34473	\' y	{ 00FD }
34474	\" y	{ 00FF }
34475	\= A	{ 0100 }
34476	\= a	{ 0101 }
34477	\u A	{ 0102 }
34478	\u a	{ 0103 }
34479	\k A	{ 0104 }
34480	\k a	{ 0105 }
34481	\' C	{ 0106 }
34482	\' c	{ 0107 }
34483	\^ C	{ 0108 }
34484	\^ c	{ 0109 }
34485	\. C	{ 010A }
34486	\. c	{ 010B }
34487	\v C	{ 010C }
34488	\v c	{ 010D }
34489	\v D	{ 010E }
34490	\v d	{ 010F }
34491	\= E	{ 0112 }
34492	\= e	{ 0113 }
34493	\u E	{ 0114 }
34494	\u e	{ 0115 }
34495	\. E	{ 0116 }
34496	\. e	{ 0117 }
34497	\k E	{ 0118 }
34498	\k e	{ 0119 }
34499	\v E	{ 011A }
34500	\v e	{ 011B }
34501	\^ G	{ 011C }
34502	\^ g	{ 011D }
34503	\u G	{ 011E }
34504	\u g	{ 011F }
34505	\. G	{ 0120 }
34506	\. g	{ 0121 }
34507	\c G	{ 0122 }
34508	\c g	{ 0123 }
34509	\^ H	{ 0124 }
34510	\^ h	{ 0125 }
34511	\~ I	{ 0128 }
34512	\~ i	{ 0129 }

34513 \~ \i { 0129 }
34514 \= I { 012A }
34515 \= i { 012B }
34516 \= \i { 012B }
34517 \u I { 012C }
34518 \u i { 012D }
34519 \u \i { 012D }
34520 \k I { 012E }
34521 \k i { 012F }
34522 \k \i { 012F }
34523 \. I { 0130 }
34524 \^ J { 0134 }
34525 \^ j { 0135 }
34526 \^ \j { 0135 }
34527 \c K { 0136 }
34528 \c k { 0137 }
34529 \' L { 0139 }
34530 \' l { 013A }
34531 \c L { 013B }
34532 \c l { 013C }
34533 \v L { 013D }
34534 \v l { 013E }
34535 \. L { 013F }
34536 \. l { 0140 }
34537 \' N { 0143 }
34538 \' n { 0144 }
34539 \c N { 0145 }
34540 \c n { 0146 }
34541 \v N { 0147 }
34542 \v n { 0148 }
34543 \= O { 014C }
34544 \= o { 014D }
34545 \u O { 014E }
34546 \u o { 014F }
34547 \H O { 0150 }
34548 \H o { 0151 }
34549 \' R { 0154 }
34550 \' r { 0155 }
34551 \c R { 0156 }
34552 \c r { 0157 }
34553 \v R { 0158 }
34554 \v r { 0159 }
34555 \' S { 015A }
34556 \' s { 015B }
34557 \^ S { 015C }
34558 \^ s { 015D }
34559 \c S { 015E }
34560 \c s { 015F }
34561 \v S { 0160 }
34562 \v s { 0161 }
34563 \c T { 0162 }
34564 \c t { 0163 }
34565 \v T { 0164 }
34566 \v t { 0165 }

34567	\~ U	{ 0168 }
34568	\~ u	{ 0169 }
34569	\= U	{ 016A }
34570	\= u	{ 016B }
34571	\u U	{ 016C }
34572	\u u	{ 016D }
34573	\r U	{ 016E }
34574	\r u	{ 016F }
34575	\H U	{ 0170 }
34576	\H u	{ 0171 }
34577	\k U	{ 0172 }
34578	\k u	{ 0173 }
34579	\^ W	{ 0174 }
34580	\^ w	{ 0175 }
34581	\^ Y	{ 0176 }
34582	\^ y	{ 0177 }
34583	\" Y	{ 0178 }
34584	\' Z	{ 0179 }
34585	\' z	{ 017A }
34586	\. Z	{ 017B }
34587	\. z	{ 017C }
34588	\v Z	{ 017D }
34589	\v z	{ 017E }
34590	\v A	{ 01CD }
34591	\v a	{ 01CE }
34592	\v I	{ 01CF }
34593	\v \i	{ 01D0 }
34594	\v i	{ 01D0 }
34595	\v O	{ 01D1 }
34596	\v o	{ 01D2 }
34597	\v U	{ 01D3 }
34598	\v u	{ 01D4 }
34599	\v G	{ 01E6 }
34600	\v g	{ 01E7 }
34601	\v K	{ 01E8 }
34602	\v k	{ 01E9 }
34603	\k O	{ 01EA }
34604	\k o	{ 01EB }
34605	\v \j	{ 01F0 }
34606	\v j	{ 01F0 }
34607	\' G	{ 01F4 }
34608	\' g	{ 01F5 }
34609	\' N	{ 01F8 }
34610	\' n	{ 01F9 }
34611	\' \AE	{ 01FC }
34612	\' \ae	{ 01FD }
34613	\' \O	{ 01FE }
34614	\' \o	{ 01FF }
34615	\v H	{ 021E }
34616	\v h	{ 021F }
34617	\. A	{ 0226 }
34618	\. a	{ 0227 }
34619	\c E	{ 0228 }
34620	\c e	{ 0229 }

```
34621 \. 0 { 022E }
34622 \. o { 022F }
34623 \= Y { 0232 }
34624 \= y { 0233 }
34625 \q_recursion_tail ? { }
34626 \q_recursion_stop
34627 \group_end:
(End of definition for \_text_purify_accent:NN.)
34628 </package>
```

Chapter 91

l3box implementation

```
34629 (*package)
34630 (@@=box)
```

91.1 Support code

`_box_dim_eval:w` Evaluating a dimension expression expandably. The only difference with `\dim_eval:n` is the lack of `\dim_use:N`, to produce an internal dimension rather than expand it into characters.

```
34631 \cs_new_eq:NN \_box_dim_eval:w \tex_dimexpr:D
34632 \cs_new:Npn \_box_dim_eval:n #1
34633 { \_box_dim_eval:w #1 \scan_stop: }
```

(End of definition for _box_dim_eval:w and _box_dim_eval:n.)

`_kernel_kern:n` We need kerns in a few places. At present, we don't have a module for this concept, so it goes in at first use: here. The idea is to avoid repeated use of the bare primitive.

```
34634 \cs_new_protected:Npn \_kernel_kern:n #1
34635 { \tex_kern:D \_box_dim_eval:n {#1} }
```

(End of definition for _kernel_kern:n.)

91.2 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt.

`\box_new:N` Defining a new `(box)` register: remember that box 255 is not generally available.

```
\box_new:c
34636 \cs_new_protected:Npn \box_new:N #1
34637 {
34638   \_kernel_chk_if_free_cs:N #1
34639   \cs:w newbox \cs_end: #1
34640 }
34641 \cs_generate_variant:Nn \box_new:N { c }
```

Clear a $\langle box \rangle$ register.

```

34642 \cs_new_protected:Npn \box_clear:N #1
\box_clear:N      34643   { \box_set_eq:NN #1 \c_empty_box }
\box_clear:c      34644 \cs_new_protected:Npn \box_gclear:N #1
\box_gclear:N     34645   { \box_gset_eq:NN #1 \c_empty_box }
\box_gclear:c     34646 \cs_generate_variant:Nn \box_clear:N { c }
\box_gclear:c     34647 \cs_generate_variant:Nn \box_gclear:N { c }

```

Clear or new.

```

34648 \cs_new_protected:Npn \box_clear_new:N #1
\box_clear_new:N  34649   { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\box_clear_new:c  34650 \cs_new_protected:Npn \box_gclear_new:N #1
\box_gclear_new:N 34651   { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
\box_gclear_new:c 34652 \cs_generate_variant:Nn \box_clear_new:N { c }
\box_gclear_new:c 34653 \cs_generate_variant:Nn \box_gclear_new:N { c }

```

Assigning the contents of a box to be another box.

```

34654 \cs_new_protected:Npn \box_set_eq:NN #1#2
\box_set_eq:NN    34655   { \tex_setbox:D #1 \tex_copy:D #2 }
\box_set_eq:cN    34656 \cs_new_protected:Npn \box_gset_eq:NN #1#2
\box_set_eq:cN    34657   { \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
\box_set_eq:Nc    34658 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_set_eq:cc    34659 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
\box_gset_eq:NN
\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc

```

Assigning the contents of a box to be another box, then drops the original box.

```

34660 \cs_new_protected:Npn \box_set_eq_drop:NN #1#2
\box_set_eq_drop:NN 34661   { \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_drop:cN 34662 \cs_new_protected:Npn \box_gset_eq_drop:NN #1#2
\box_set_eq_drop:cN 34663   { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_drop:Nc 34664 \cs_generate_variant:Nn \box_set_eq_drop:NN { c , Nc , cc }
\box_set_eq_drop:cc 34665 \cs_generate_variant:Nn \box_gset_eq_drop:NN { c , Nc , cc }
\box_gset_eq_drop:NN
\box_gset_eq_drop:cN
\box_gset_eq_drop:Nc
\box_gset_eq_drop:cc

```

Copies of the cs functions defined in l3basics.

```

34666 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
\box_if_exist:N    34667   { TF , T , F , p }
\box_if_exist:p:c  34668 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
\box_if_exist:c    34669   { TF , T , F , p }
\box_if_exist:NTF
\box_if_exist:cTF

```

91.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

34670 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_ht:N         34671 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_ht:c         34672 \cs_new_eq:NN \box_wd:N \tex_wd:D
\box_dp:N         34673 \cs_generate_variant:Nn \box_ht:N { c }
\box_dp:c         34674 \cs_generate_variant:Nn \box_dp:N { c }
\box_wd:N         34675 \cs_generate_variant:Nn \box_wd:N { c }
\box_wd:c

```

The $\backslash\text{box_ht:N}$ and $\backslash\text{box_dp:N}$ primitives do not expand but rather are suitable for use after $\backslash\text{the}$ or inside dimension expressions. Here we obtain the same behaviour by using $\backslash\text{_box_dim_eval:n}$ (basically $\backslash\text{dimexpr}$) rather than $\backslash\text{dim_eval:n}$ (basically $\backslash\text{the dimexpr}$).

```
\box_ht_plus_dp:N
```

```

34676 \cs_new_protected:Npn \box_ht_plus_dp:N #1
34677   { \__box_dim_eval:n { \box_ht:N #1 + \box_dp:N #1 } }
34678 \cs_generate_variant:Nn \box_ht_plus_dp:N { c }

```

Setting the size whilst respecting local scope requires copying; the same issue does not come up when working globally. When debugging, the dimension expression #2 is surrounded by parentheses to catch early termination.

```

\box_set_ht:Nn
\box_set_ht:cn 34679 \cs_new_protected:Npn \box_set_dp:Nn #1#2
\box_gset_ht:Nn 34680   {
\box_gset_ht:cn 34681     \tex_setbox:D #1 = \tex_copy:D #1
\box_set_dp:Nn 34682     \box_dp:N #1 \__box_dim_eval:n {#2}
\box_set_dp:cn 34683   }
\box_gset_dp:Nn 34684 \cs_generate_variant:Nn \box_set_dp:Nn { c }
\box_gset_dp:cn 34685 \cs_new_protected:Npn \box_gset_dp:Nn #1#2
\box_set_wd:Nn 34686   { \box_dp:N #1 \__box_dim_eval:n {#2} }
\box_set_wd:cn 34687 \cs_generate_variant:Nn \box_gset_dp:Nn { c }
\box_gset_wd:Nn 34688 \cs_new_protected:Npn \box_set_ht:Nn #1#2
\box_gset_wd:cn 34689   {
34690     \tex_setbox:D #1 = \tex_copy:D #1
34691     \box_ht:N #1 \__box_dim_eval:n {#2}
34692   }
34693 \cs_generate_variant:Nn \box_set_ht:Nn { c }
34694 \cs_new_protected:Npn \box_gset_ht:Nn #1#2
34695   { \box_ht:N #1 \__box_dim_eval:n {#2} }
34696 \cs_generate_variant:Nn \box_gset_ht:Nn { c }
34697 \cs_new_protected:Npn \box_set_wd:Nn #1#2
34698   {
34699     \tex_setbox:D #1 = \tex_copy:D #1
34700     \box_wd:N #1 \__box_dim_eval:n {#2}
34701   }
34702 \cs_generate_variant:Nn \box_set_wd:Nn { c }
34703 \cs_new_protected:Npn \box_gset_wd:Nn #1#2
34704   { \box_wd:N #1 \__box_dim_eval:n {#2} }
34705 \cs_generate_variant:Nn \box_gset_wd:Nn { c }

```

91.4 Using boxes

Using a $\langle box \rangle$. These are just T_EX primitives with meaningful names.

```

\box_use_drop:N 34706 \cs_new_eq:NN \box_use_drop:N \tex_box:D
\box_use_drop:c 34707 \cs_new_eq:NN \box_use:N \tex_copy:D
\box_use:N 34708 \cs_generate_variant:Nn \box_use_drop:N { c }
\box_use:c 34709 \cs_generate_variant:Nn \box_use:N { c }

```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

```

\box_move_left:nn 34710 \cs_new_protected:Npn \box_move_left:nn #1#2
\box_move_right:nn 34711   { \tex_moveleft:D \__box_dim_eval:n {#1} #2 }
\box_move_up:nn 34712 \cs_new_protected:Npn \box_move_right:nn #1#2
\box_move_down:nn 34713   { \tex_moveright:D \__box_dim_eval:n {#1} #2 }
34714 \cs_new_protected:Npn \box_move_up:nn #1#2
34715   { \tex_raise:D \__box_dim_eval:n {#1} #2 }
34716 \cs_new_protected:Npn \box_move_down:nn #1#2
34717   { \tex_lower:D \__box_dim_eval:n {#1} #2 }

```


91.5 Box conditionals

The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```
\if_hbox:N      34718 \cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
\if_vbox:N      34719 \cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
\if_box_empty:N 34720 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

\box_if_horizontal_p:N 34721 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
\box_if_horizontal_p:c 34722 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_horizontal:NTF 34723 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
\box_if_horizontal:cTF 34724 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_vertical_p:N  34725 \prg_generate_conditional_variant:Nnn \box_if_horizontal:N
\box_if_vertical_p:c  34726 { c } { p , T , F , TF }
\box_if_vertical:NTF  34727 \prg_generate_conditional_variant:Nnn \box_if_vertical:N
\box_if_vertical:cTF  34728 { c } { p , T , F , TF }
```

Testing if a $\langle box \rangle$ is empty/void.

```
\box_if_empty_p:N  34729 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
\box_if_empty_p:c  34730 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_empty:NTF  34731 \prg_generate_conditional_variant:Nnn \box_if_empty:N
\box_if_empty:cTF  34732 { c } { p , T , F , TF }
```

(End of definition for $\backslash box_new:N$ and others. These functions are documented on page 302.)

91.6 The last box inserted

```
\box_set_to_last:N  Set a box to the previous box.
\box_set_to_last:c  34733 \cs_new_protected:Npn \box_set_to_last:N #1
\box_gset_to_last:N 34734 { \tex_setbox:D #1 \tex_lastbox:D }
\box_gset_to_last:c 34735 \cs_new_protected:Npn \box_gset_to_last:N #1
34736 { \tex_global:D \tex_setbox:D #1 \tex_lastbox:D }
34737 \cs_generate_variant:Nn \box_set_to_last:N { c }
34738 \cs_generate_variant:Nn \box_gset_to_last:N { c }
```

(End of definition for $\backslash box_set_to_last:N$ and $\backslash box_gset_to_last:N$. These functions are documented on page 305.)

91.7 Constant boxes

```
\c_empty_box A box we never use.
34739 \box_new:N \c_empty_box
```

(End of definition for $\backslash c_empty_box$. This variable is documented on page 305.)

91.8 Scratch boxes

```
\l_tmpa_box Scratch boxes.
\l_tmpb_box 34740 \box_new:N \l_tmpa_box
\g_tmpa_box 34741 \box_new:N \l_tmpb_box
\g_tmpb_box 34742 \box_new:N \g_tmpa_box
           34743 \box_new:N \g_tmpb_box
```

(End of definition for `\l_tmpa_box` and others. These variables are documented on page 305.)

91.9 Viewing box contents

TeX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other L^AT_EX3 show functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

```
\box_show:N Essentially a wrapper around the internal function, but evaluating the breadth and depth
\box_show:c arguments now outside the group.
\box_show:Nnn 34744 \cs_new_protected:Npn \box_show:N #1
\box_show:cnn 34745 { \box_show:Nnn #1 \c_max_int \c_max_int }
              34746 \cs_generate_variant:Nn \box_show:N { c }
              34747 \cs_new_protected:Npn \box_show:Nnn #1#2#3
              34748 { \__box_show:NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }
              34749 \cs_generate_variant:Nn \box_show:Nnn { c }
```

(End of definition for `\box_show:N` and `\box_show:Nnn`. These functions are documented on page 306.)

```
\box_log:N Getting TeX to write to the log without interruption the run is done by altering the
\box_log:c interaction mode. For that, the  $\epsilon$ -TeX extensions are needed.
\box_log:Nnn 34750 \cs_new_protected:Npn \box_log:N #1
\box_log:cnn 34751 { \box_log:Nnn #1 \c_max_int \c_max_int }
\__box_log:nNnn 34752 \cs_generate_variant:Nn \box_log:N { c }
              34753 \cs_new_protected:Npn \box_log:Nnn
              34754 { \exp_args:No \__box_log:nNnn { \tex_the:D \tex_interactionmode:D } }
              34755 \cs_new_protected:Npn \__box_log:nNnn #1#2#3#4
              34756 {
              34757 \int_gset:Nn \tex_interactionmode:D { 0 }
              34758 \__box_show:NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }
              34759 \int_gset:Nn \tex_interactionmode:D {#1}
              34760 }
              34761 \cs_generate_variant:Nn \box_log:Nnn { c }
```

(End of definition for `\box_log:N`, `\box_log:Nnn`, and `__box_log:nNnn`. These functions are documented on page 306.)

```
\__box_show:NNnn The internal auxiliary to actually do the output uses a group to deal with breadth and
\__box_show:NNff depth values. The \use:n here gives better output appearance. Setting \tracingonline
and \errorcontextlines is used to control what appears in the terminal.
              34762 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
              34763 {
              34764 \box_if_exist:NTF #2
              34765 {
              34766 \group_begin:
```

```

34767         \int_set:Nn \tex_showboxbreadth:D {#3}
34768         \int_set:Nn \tex_showboxdepth:D {#4}
34769         \int_set:Nn \tex_tracingonline:D {#1}
34770         \int_set:Nn \tex_errorcontextlines:D { -1 }
34771         \tex_showbox:D \use:n {#2}
34772     \group_end:
34773 }
34774 {
34775     \msg_error:nne { kernel } { variable-not-defined }
34776     { \token_to_str:N #2 }
34777 }
34778 }
34779 \cs_generate_variant:Nn \__box_show:NNnn { NNff }

```

(End of definition for `__box_show:NNnn`.)

91.10 Horizontal mode boxes

`\hbox:n` (The test suite for this command, and others in this file, is `m3box002.lvt`.)
Put a horizontal box directly into the input stream.

```

34780 \cs_new_protected:Npn \hbox:n #1
34781 { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }

```

(End of definition for `\hbox:n`. This function is documented on page 306.)

`\hbox_set:Nn`
`\hbox_set:cn`
`\hbox_gset:Nn`
`\hbox_gset:cn`

```

34782 \cs_new_protected:Npn \hbox_set:Nn #1#2
34783 {
34784     \tex_setbox:D #1 \tex_hbox:D
34785     { \color_group_begin: #2 \color_group_end: }
34786 }
34787 \cs_new_protected:Npn \hbox_gset:Nn #1#2
34788 {
34789     \tex_global:D \tex_setbox:D #1 \tex_hbox:D
34790     { \color_group_begin: #2 \color_group_end: }
34791 }
34792 \cs_generate_variant:Nn \hbox_set:Nn { c }
34793 \cs_generate_variant:Nn \hbox_gset:Nn { c }

```

(End of definition for `\hbox_set:Nn` and `\hbox_gset:Nn`. These functions are documented on page 306.)

`\hbox_set_to_wd:Nnn` Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

`\hbox_set_to_wd:cn`
`\hbox_gset_to_wd:Nnn`
`\hbox_gset_to_wd:cn`

```

34794 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
34795 {
34796     \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
34797     { \color_group_begin: #3 \color_group_end: }
34798 }
34799 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn #1#2#3
34800 {
34801     \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
34802     { \color_group_begin: #3 \color_group_end: }
34803 }
34804 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
34805 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }

```

(End of definition for `\hbox_set_to_wd:Nnn` and `\hbox_gset_to_wd:Nnn`. These functions are documented on page 307.)

`\hbox_set:Nw` Storing material in a horizontal box. This type is useful in environment definitions.

```

\hbox_set:cw 34806 \cs_new_protected:Npn \hbox_set:Nw #1
\hbox_gset:Nw 34807 {
\hbox_gset:cw 34808   \tex_setbox:D #1 \tex_hbox:D
\hbox_set_end: 34809   \c_group_begin_token
\hbox_gset_end: 34810   \color_group_begin:
34811 }
34812 \cs_new_protected:Npn \hbox_gset:Nw #1
34813 {
34814   \tex_global:D \tex_setbox:D #1 \tex_hbox:D
34815   \c_group_begin_token
34816   \color_group_begin:
34817 }
34818 \cs_generate_variant:Nn \hbox_set:Nw { c }
34819 \cs_generate_variant:Nn \hbox_gset:Nw { c }
34820 \cs_new_protected:Npn \hbox_set_end:
34821 {
34822   \color_group_end:
34823   \c_group_end_token
34824 }
34825 \cs_new_eq:NN \hbox_gset_end: \hbox_set_end:

```

(End of definition for `\hbox_set:Nw` and others. These functions are documented on page 307.)

`\hbox_set_to_wd:Nnw` Combining the above ideas.

```

\hbox_set_to_wd:cw 34826 \cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2
\hbox_gset_to_wd:Nnw 34827 {
\hbox_gset_to_wd:cw 34828   \tex_setbox:D #1 \tex_hbox:D to \_box_dim_eval:n {#2}
34829   \c_group_begin_token
34830   \color_group_begin:
34831 }
34832 \cs_new_protected:Npn \hbox_gset_to_wd:Nnw #1#2
34833 {
34834   \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \_box_dim_eval:n {#2}
34835   \c_group_begin_token
34836   \color_group_begin:
34837 }
34838 \cs_generate_variant:Nn \hbox_set_to_wd:Nnw { c }
34839 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnw { c }

```

(End of definition for `\hbox_set_to_wd:Nnw` and `\hbox_gset_to_wd:Nnw`. These functions are documented on page 307.)

`\hbox_to_wd:n` Put a horizontal box directly into the input stream.

```

\hbox_to_zero:n 34840 \cs_new_protected:Npn \hbox_to_wd:n #1#2
34841 {
34842   \tex_hbox:D to \_box_dim_eval:n {#1}
34843   { \color_group_begin: #2 \color_group_end: }
34844 }
34845 \cs_new_protected:Npn \hbox_to_zero:n #1
34846 {
34847   \tex_hbox:D to \c_zero_dim

```

```

34848     { \color_group_begin: #1 \color_group_end: }
34849   }

```

(End of definition for `\hbox_to_wd:n` and `\hbox_to_zero:n`. These functions are documented on page 306.)

`\hbox_overlap_center:n` Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```

\hbox_overlap_left:n
\hbox_overlap_right:n
34850 \cs_new_protected:Npn \hbox_overlap_center:n #1
34851   { \hbox_to_zero:n { \tex_hss:D #1 \tex_hss:D } }
34852 \cs_new_protected:Npn \hbox_overlap_left:n #1
34853   { \hbox_to_zero:n { \tex_hss:D #1 } }
34854 \cs_new_protected:Npn \hbox_overlap_right:n #1
34855   { \hbox_to_zero:n { #1 \tex_hss:D } }

```

(End of definition for `\hbox_overlap_center:n`, `\hbox_overlap_left:n`, and `\hbox_overlap_right:n`. These functions are documented on page 307.)

`\hbox_unpack:N` Unpacking a box and if requested also clear it.

```

\hbox_unpack:c
\hbox_unpack_drop:N
\hbox_unpack_drop:c
34856 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
34857 \cs_new_eq:NN \hbox_unpack_drop:N \tex_unhbox:D
34858 \cs_generate_variant:Nn \hbox_unpack:N { c }
34859 \cs_generate_variant:Nn \hbox_unpack_drop:N { c }

```

(End of definition for `\hbox_unpack:N` and `\hbox_unpack_drop:N`. These functions are documented on page 307.)

91.11 Vertical mode boxes

TeX ends these boxes directly with the internal `end_graf` routine. This means that there is no `\par` at the end of vertical boxes unless we insert one. Thus all vertical boxes include a `\par` just before closing the color group.

`\vbox:n` The following test files are used for this code: `m3box003.lvt`.

The following test files are used for this code: `m3box003.lvt`.

`\vbox_top:n` Put a vertical box directly into the input stream.

```

34860 \cs_new_protected:Npn \vbox:n #1
34861   { \tex_vbox:D { \color_group_begin: #1 \par \color_group_end: } }
34862 \cs_new_protected:Npn \vbox_top:n #1
34863   { \tex_vtop:D { \color_group_begin: #1 \par \color_group_end: } }

```

(End of definition for `\vbox:n` and `\vbox_top:n`. These functions are documented on page 308.)

`\vbox_to_ht:n` Put a vertical box directly into the input stream.

```

\vbox_to_zero:n
\vbox_to_ht:n
\vbox_to_zero:n
34864 \cs_new_protected:Npn \vbox_to_ht:n #1#2
34865   {
34866     \tex_vbox:D to \__box_dim_eval:n {#1}
34867     { \color_group_begin: #2 \par \color_group_end: }
34868   }
34869 \cs_new_protected:Npn \vbox_to_zero:n #1
34870   {
34871     \tex_vbox:D to \c_zero_dim
34872     { \color_group_begin: #1 \par \color_group_end: }
34873   }

```

(End of definition for `\vbox_to_ht:nn` and others. These functions are documented on page 308.)

```

\vbox_set:Nn Storing material in a vertical box with a natural height.
\vbox_set:cn
\vbox_gset:Nn
\vbox_gset:cn
34874 \cs_new_protected:Npn \vbox_set:Nn #1#2
34875 {
34876   \tex_setbox:D #1 \tex_vbox:D
34877   { \color_group_begin: #2 \par \color_group_end: }
34878 }
34879 \cs_new_protected:Npn \vbox_gset:Nn #1#2
34880 {
34881   \tex_global:D \tex_setbox:D #1 \tex_vbox:D
34882   { \color_group_begin: #2 \par \color_group_end: }
34883 }
34884 \cs_generate_variant:Nn \vbox_set:Nn { c }
34885 \cs_generate_variant:Nn \vbox_gset:Nn { c }

```

(End of definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 308.)

```

\vbox_set_top:Nn Storing material in a vertical box with a natural height and reference point at the baseline
\vbox_set_top:cn of the first object in the box.
\vbox_gset_top:Nn
\vbox_gset_top:cn
34886 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
34887 {
34888   \tex_setbox:D #1 \tex_vtop:D
34889   { \color_group_begin: #2 \par \color_group_end: }
34890 }
34891 \cs_new_protected:Npn \vbox_gset_top:Nn #1#2
34892 {
34893   \tex_global:D \tex_setbox:D #1 \tex_vtop:D
34894   { \color_group_begin: #2 \par \color_group_end: }
34895 }
34896 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
34897 \cs_generate_variant:Nn \vbox_gset_top:Nn { c }

```

(End of definition for `\vbox_set_top:Nn` and `\vbox_gset_top:Nn`. These functions are documented on page 308.)

```

\vbox_set_to_ht:Nnn Storing material in a vertical box with a specified height.
\vbox_set_to_ht:cn
\vbox_gset_to_ht:Nnn
\vbox_gset_to_ht:cn
34898 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
34899 {
34900   \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
34901   { \color_group_begin: #3 \par \color_group_end: }
34902 }
34903 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn #1#2#3
34904 {
34905   \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
34906   { \color_group_begin: #3 \par \color_group_end: }
34907 }
34908 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
34909 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }

```

(End of definition for `\vbox_set_to_ht:Nnn` and `\vbox_gset_to_ht:Nnn`. These functions are documented on page 308.)

`\vbox_set:Nw` Storing material in a vertical box. This type is useful in environment definitions.

```

\vbox_set:cnw 34910 \cs_new_protected:Npn \vbox_set:Nw #1
\vbox_gset:Nw 34911 {
\vbox_gset:cnw 34912   \tex_setbox:D #1 \tex_vbox:D
\vbox_set_end: 34913   \c_group_begin_token
\vbox_gset_end: 34914   \color_group_begin:
34915 }
34916 \cs_new_protected:Npn \vbox_gset:Nw #1
34917 {
34918   \tex_global:D \tex_setbox:D #1 \tex_vbox:D
34919   \c_group_begin_token
34920   \color_group_begin:
34921 }
34922 \cs_generate_variant:Nn \vbox_set:Nw { c }
34923 \cs_generate_variant:Nn \vbox_gset:Nw { c }
34924 \cs_new_protected:Npn \vbox_set_end:
34925 {
34926   \par
34927   \color_group_end:
34928   \c_group_end_token
34929 }
34930 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

```

(End of definition for `\vbox_set:Nw` and others. These functions are documented on page 308.)

`\vbox_set_to_ht:Nnw` A combination of the above ideas.

```

\vbox_set_to_ht:cnw 34931 \cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
\vbox_gset_to_ht:Nnw 34932 {
\vbox_gset_to_ht:cnw 34933   \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
34934   \c_group_begin_token
34935   \color_group_begin:
34936 }
34937 \cs_new_protected:Npn \vbox_gset_to_ht:Nnw #1#2
34938 {
34939   \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
34940   \c_group_begin_token
34941   \color_group_begin:
34942 }
34943 \cs_generate_variant:Nn \vbox_set_to_ht:Nnw { c }
34944 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnw { c }

```

(End of definition for `\vbox_set_to_ht:Nnw` and `\vbox_gset_to_ht:Nnw`. These functions are documented on page 309.)

`\vbox_unpack:N` Unpacking a box and if requested also clear it.

```

\vbox_unpack:c 34945 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\vbox_unpack_drop:N 34946 \cs_new_eq:NN \vbox_unpack_drop:N \tex_unvbox:D
\vbox_unpack_drop:c 34947 \cs_generate_variant:Nn \vbox_unpack:N { c }
34948 \cs_generate_variant:Nn \vbox_unpack_drop:N { c }

```

(End of definition for `\vbox_unpack:N` and `\vbox_unpack_drop:N`. These functions are documented on page 309.)

```

\ vbox_set_split_to_ht:NNn Splitting a vertical box in two.
\ vbox_set_split_to_ht:cNn 34949 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
\ vbox_set_split_to_ht:Ncn 34950 { \tex_setbox:D #1 \tex_vsplit:D #2 to \_box_dim_eval:n {#3} }
\ vbox_set_split_to_ht:ccn 34951 \cs_generate_variant:Nn \vbox_set_split_to_ht:NNn { c , Nc , cc }
\ vbox_gset_split_to_ht:NNn 34952 \cs_new_protected:Npn \vbox_gset_split_to_ht:NNn #1#2#3
\ vbox_gset_split_to_ht:cNn 34953 {
\ vbox_gset_split_to_ht:Ncn 34954 \tex_global:D \tex_setbox:D #1
\ vbox_gset_split_to_ht:ccn 34955 \tex_vsplit:D #2 to \_box_dim_eval:n {#3}
34956 }
34957 \cs_generate_variant:Nn \vbox_gset_split_to_ht:NNn { c , Nc , cc }

```

(End of definition for `\vbox_set_split_to_ht:NNn` and `\vbox_gset_split_to_ht:NNn`. These functions are documented on page 309.)

91.12 Affine transformations

`\l__box_angle_fp` When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the `fp` module so that the value is tidied up properly.

```
34958 \fp_new:N \l__box_angle_fp
```

(End of definition for `\l__box_angle_fp`.)

`\l__box_cos_fp` These are used to hold the calculated sine and cosine values while carrying out a rotation.

```
\l__box_sin_fp 34959 \fp_new:N \l__box_cos_fp
```

```
34960 \fp_new:N \l__box_sin_fp
```

(End of definition for `\l__box_cos_fp` and `\l__box_sin_fp`.)

`\l__box_top_dim` These are the positions of the four edges of a box before manipulation.

```
\l__box_bottom_dim 34961 \dim_new:N \l__box_top_dim
```

```
\l__box_left_dim 34962 \dim_new:N \l__box_bottom_dim
```

```
\l__box_right_dim 34963 \dim_new:N \l__box_left_dim
```

```
34964 \dim_new:N \l__box_right_dim
```

(End of definition for `\l__box_top_dim` and others.)

`\l__box_top_new_dim` These are the positions of the four edges of a box after manipulation.

```
\l__box_bottom_new_dim 34965 \dim_new:N \l__box_top_new_dim
```

```
\l__box_left_new_dim 34966 \dim_new:N \l__box_bottom_new_dim
```

```
\l__box_right_new_dim 34967 \dim_new:N \l__box_left_new_dim
```

```
34968 \dim_new:N \l__box_right_new_dim
```

(End of definition for `\l__box_top_new_dim` and others.)

`\l__box_internal_box` Scratch space, but also needed by some parts of the driver.

```
34969 \box_new:N \l__box_internal_box
```

(End of definition for `\l__box_internal_box`.)

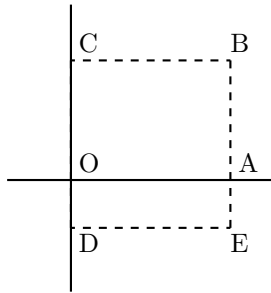


Figure 1: Coordinates of a box prior to rotation.

```

\box_rotate:Nn Rotation of a box starts with working out the relevant sine and cosine. The actual
\box_rotate:cn rotation is in an auxiliary to keep the flow slightly clearer
\box_grotate:Nn
\box_grotate:cn 34970 \cs_new_protected:Npn \box_rotate:Nn #1#2
                 34971 { \__box_rotate:NnN #1 {#2} \hbox_set:Nn }
\__box_rotate:NnN 34972 \cs_generate_variant:Nn \box_rotate:Nn { c }
\__box_rotate:N 34973 \cs_new_protected:Npn \box_grotate:Nn #1#2
\__box_rotate_xdir:nnN 34974 { \__box_rotate:NnN #1 {#2} \hbox_gset:Nn }
\__box_rotate_ydir:nnN 34975 \cs_generate_variant:Nn \box_grotate:Nn { c }
\__box_rotate_quadrant_one: 34976 \cs_new_protected:Npn \__box_rotate:NnN #1#2#3
\__box_rotate_quadrant_two: 34977 {
\__box_rotate_quadrant_three: 34978 #3 #1
\__box_rotate_quadrant_four: 34979 {
34980 \fp_set:Nn \l__box_angle_fp {#2}
34981 \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
34982 \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
34983 \__box_rotate:N #1
34984 }
34985 }

```

The edges of the box are then recorded: the left edge is always at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```

34986 \cs_new_protected:Npn \__box_rotate:N #1
34987 {
34988 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
34989 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
34990 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
34991 \dim_zero:N \l__box_left_dim

```

The next step is to work out the x and y coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices B , C , D and E is illustrated (Figure 1). The vertex O is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point P and angle α :

$$\begin{aligned}
 P'_x &= P_x - O_x \\
 P'_y &= P_y - O_y \\
 P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
 P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
 P'''_x &= P''_x + O_x + L_x \\
 P'''_y &= P''_y + O_y
 \end{aligned}$$

The “extra” horizontal translation L_x at the end is calculated so that the leftmost point of the resulting box has x -coordinate 0. This is desirable as \TeX boxes must have the reference point at the left edge of the box. (As O is always $(0,0)$, this part of the calculation is omitted here.)

```

34992     \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
34993     {
34994         \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
34995         { \__box_rotate_quadrant_one: }
34996         { \__box_rotate_quadrant_two: }
34997     }
34998     {
34999         \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
35000         { \__box_rotate_quadrant_three: }
35001         { \__box_rotate_quadrant_four: }
35002     }

```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current \TeX reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

```

35003     \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
35004     \hbox_set:Nn \l__box_internal_box
35005     {
35006         \__kernel_kern:n { -\l__box_left_new_dim }
35007         \hbox:n
35008         {
35009             \__box_backend_rotate:Nn
35010             \l__box_internal_box
35011             \l__box_angle_fp
35012         }
35013     }

```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```

35014     \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
35015     \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
35016     \box_set_wd:Nn \l__box_internal_box
35017     { \l__box_right_new_dim - \l__box_left_new_dim }
35018     \box_use_drop:N \l__box_internal_box
35019 }

```

These functions take a general point $(\#1,\#2)$ and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both x' and y' at the same time. Contrast this with the equivalent function in the `l3coffins` module, where both parts are needed.

```

35020 \cs_new_protected:Npn \__box_rotate_xdir:nnN #1#2#3
35021 {
35022     \dim_set:Nn #3
35023     {
35024         \fp_to_dim:n
35025         {
35026             \l__box_cos_fp * \dim_to_fp:n {#1}
35027             - \l__box_sin_fp * \dim_to_fp:n {#2}

```

```

35028     }
35029   }
35030 }
35031 \cs_new_protected:Npn \__box_rotate_ydir:nnN #1#2#3
35032 {
35033   \dim_set:Nn #3
35034   {
35035     \fp_to_dim:n
35036     {
35037       \l__box_sin_fp * \dim_to_fp:n {#1}
35038       + \l__box_cos_fp * \dim_to_fp:n {#2}
35039     }
35040   }
35041 }

```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting y -values, whereas the left and right edges need the x -values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```

35042 \cs_new_protected:Npn \__box_rotate_quadrant_one:
35043 {
35044   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
35045   \l__box_top_new_dim
35046   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
35047   \l__box_bottom_new_dim
35048   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
35049   \l__box_left_new_dim
35050   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
35051   \l__box_right_new_dim
35052 }
35053 \cs_new_protected:Npn \__box_rotate_quadrant_two:
35054 {
35055   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
35056   \l__box_top_new_dim
35057   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
35058   \l__box_bottom_new_dim
35059   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
35060   \l__box_left_new_dim
35061   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
35062   \l__box_right_new_dim
35063 }
35064 \cs_new_protected:Npn \__box_rotate_quadrant_three:
35065 {
35066   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
35067   \l__box_top_new_dim
35068   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
35069   \l__box_bottom_new_dim
35070   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
35071   \l__box_left_new_dim
35072   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
35073   \l__box_right_new_dim
35074 }
35075 \cs_new_protected:Npn \__box_rotate_quadrant_four:

```

```

35076 {
35077   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
35078     \l__box_top_new_dim
35079   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
35080     \l__box_bottom_new_dim
35081   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
35082     \l__box_left_new_dim
35083   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
35084     \l__box_right_new_dim
35085 }

```

(End of definition for `\box_rotate:Nn` and others. These functions are documented on page 313.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.

```

\l__box_scale_y_fp 35086 \fp_new:N \l__box_scale_x_fp
35087 \fp_new:N \l__box_scale_y_fp

```

(End of definition for `\l__box_scale_x_fp` and `\l__box_scale_y_fp`.)

`\box_resize_to_wd_and_ht_plus_dp:Nnn` Resizing a box starts by working out the various dimensions of the existing box.

```

\box_resize_to_wd_and_ht_plus_dp:cnm 35088 \cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn #1#2#3
\box_gresize_to_wd_and_ht_plus_dp:Nnn 35089 {
\box_gresize_to_wd_and_ht_plus_dp:cnm 35090   \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
\__box_resize_to_wd_and_ht_plus_dp:NnnN 35091   \hbox_set:Nn
\__box_resize_set_corners:N 35092 }
\__box_resize:N 35093 \cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
\__box_resize:NNN 35094 \cs_new_protected:Npn \box_gresize_to_wd_and_ht_plus_dp:Nnn #1#2#3
35095 {
35096   \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
35097   \hbox_gset:Nn
35098 }
35099 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht_plus_dp:Nnn { c }
35100 \cs_new_protected:Npn \__box_resize_to_wd_and_ht_plus_dp:NnnN #1#2#3#4
35101 {
35102   #4 #1
35103   {
35104     \__box_resize_set_corners:N #1

```

The x -scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```

35105   \fp_set:Nn \l__box_scale_x_fp
35106     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }

```

The y -scaling needs both the height and the depth of the current box.

```

35107   \fp_set:Nn \l__box_scale_y_fp
35108     {
35109       \dim_to_fp:n {#3}
35110       / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
35111     }

```

Hand off to the auxiliary which does the rest of the work.

```

35112   \__box_resize:N #1
35113 }
35114 }
35115 \cs_new_protected:Npn \__box_resize_set_corners:N #1
35116 {

```

```

35117 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
35118 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
35119 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
35120 \dim_zero:N \l__box_left_dim
35121 }

```

With at least one real scaling to do, the next phase is to find the new edge coordinates. In the x direction this is relatively easy: just scale the right edge. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```

35122 \cs_new_protected:Npn \__box_resize:N #1
35123 {
35124   \__box_resize:NNN \l__box_right_new_dim
35125   \l__box_scale_x_fp \l__box_right_dim
35126   \__box_resize:NNN \l__box_bottom_new_dim
35127   \l__box_scale_y_fp \l__box_bottom_dim
35128   \__box_resize:NNN \l__box_top_new_dim
35129   \l__box_scale_y_fp \l__box_top_dim
35130   \__box_resize_common:N #1
35131 }
35132 \cs_new_protected:Npn \__box_resize:NNN #1#2#3
35133 {
35134   \dim_set:Nn #1
35135   { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
35136 }

```

(End of definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` and others. These functions are documented on page 312.)

<pre> \box_resize_to_ht:Nn \box_resize_to_ht:cn \box_gresize_to_ht:Nn \box_gresize_to_ht:cn __box_resize_to_ht:NnN \box_resize_to_ht_plus_dp:Nn \box_resize_to_ht_plus_dp:cn \box_gresize_to_ht_plus_dp:Nn \box_gresize_to_ht_plus_dp:cn __box_resize_to_ht_plus_dp:NnN \box_resize_to_wd:Nn \box_resize_to_wd:cn \box_gresize_to_wd:Nn \box_gresize_to_wd:cn __box_resize_to_wd:NnN \box_resize_to_wd_and_ht:Nnn \box_resize_to_wd_and_ht:cnn \box_gresize_to_wd_and_ht:Nnn \box_gresize_to_wd_and_ht:cnn __box_resize_to_wd_ht:NnnN </pre>	<pre> 35137 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2 35138 { __box_resize_to_ht:NnN #1 {#2} \hbox_set:Nn } 35139 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c } 35140 \cs_new_protected:Npn \box_gresize_to_ht:Nn #1#2 35141 { __box_resize_to_ht:NnN #1 {#2} \hbox_gset:Nn } 35142 \cs_generate_variant:Nn \box_gresize_to_ht:Nn { c } 35143 \cs_new_protected:Npn __box_resize_to_ht:NnN #1#2#3 35144 { 35145 #3 #1 35146 { 35147 __box_resize_set_corners:N #1 35148 \fp_set:Nn \l__box_scale_y_fp 35149 { 35150 \dim_to_fp:n {#2} 35151 / \dim_to_fp:n { \l__box_top_dim } 35152 } 35153 \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp 35154 __box_resize:N #1 35155 } 35156 } 35157 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2 </pre>
--	---

```

35158 { \_box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_set:Nn }
35159 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
35160 \cs_new_protected:Npn \box_gresize_to_ht_plus_dp:Nn #1#2
35161 { \_box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_gset:Nn }
35162 \cs_generate_variant:Nn \box_gresize_to_ht_plus_dp:Nn { c }
35163 \cs_new_protected:Npn \_box_resize_to_ht_plus_dp:NnN #1#2#3
35164 {
35165   #3 #1
35166   {
35167     \_box_resize_set_corners:N #1
35168     \fp_set:Nn \l__box_scale_y_fp
35169     {
35170       \dim_to_fp:n {#2}
35171       / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
35172     }
35173     \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
35174     \_box_resize:N #1
35175   }
35176 }
35177 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
35178 { \_box_resize_to_wd:NnN #1 {#2} \hbox_set:Nn }
35179 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
35180 \cs_new_protected:Npn \box_gresize_to_wd:Nn #1#2
35181 { \_box_resize_to_wd:NnN #1 {#2} \hbox_gset:Nn }
35182 \cs_generate_variant:Nn \box_gresize_to_wd:Nn { c }
35183 \cs_new_protected:Npn \_box_resize_to_wd:NnN #1#2#3
35184 {
35185   #3 #1
35186   {
35187     \_box_resize_set_corners:N #1
35188     \fp_set:Nn \l__box_scale_x_fp
35189     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
35190     \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
35191     \_box_resize:N #1
35192   }
35193 }
35194 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
35195 { \_box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_set:Nn }
35196 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
35197 \cs_new_protected:Npn \box_gresize_to_wd_and_ht:Nnn #1#2#3
35198 { \_box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_gset:Nn }
35199 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht:Nnn { c }
35200 \cs_new_protected:Npn \_box_resize_to_wd_and_ht:NnnN #1#2#3#4
35201 {
35202   #4 #1
35203   {
35204     \_box_resize_set_corners:N #1
35205     \fp_set:Nn \l__box_scale_x_fp
35206     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
35207     \fp_set:Nn \l__box_scale_y_fp
35208     {
35209       \dim_to_fp:n {#3}
35210       / \dim_to_fp:n { \l__box_top_dim }
35211     }

```

```

35212     \_box_resize:N #1
35213     }
35214 }

```

(End of definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 311.)

`\box_scale:Nnn` When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. `\box_gscale:Nnn` Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The code here is split into two as this allows sharing with the auto-resizing functions. `_box_scale:NnnN` `_box_scale:N`

```

35215 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
35216 { \_box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn }
35217 \cs_generate_variant:Nn \box_scale:Nnn { c }
35218 \cs_new_protected:Npn \box_gscale:Nnn #1#2#3
35219 { \_box_scale:NnnN #1 {#2} {#3} \hbox_gset:Nn }
35220 \cs_generate_variant:Nn \box_gscale:Nnn { c }
35221 \cs_new_protected:Npn \_box_scale:NnnN #1#2#3#4
35222 {
35223     #4 #1
35224     {
35225         \fp_set:Nn \l__box_scale_x_fp {#2}
35226         \fp_set:Nn \l__box_scale_y_fp {#3}
35227         \_box_scale:N #1
35228     }
35229 }
35230 \cs_new_protected:Npn \_box_scale:N #1
35231 {
35232     \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
35233     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
35234     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
35235     \dim_zero:N \l__box_left_dim
35236     \dim_set:Nn \l__box_top_new_dim
35237     { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
35238     \dim_set:Nn \l__box_bottom_new_dim
35239     { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
35240     \dim_set:Nn \l__box_right_new_dim
35241     { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
35242     \_box_resize_common:N #1
35243 }

```

(End of definition for `\box_scale:Nnn` and others. These functions are documented on page 313.)

`\box_autosize_to_wd_and_ht:Nnn` Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere. `\box_autosize_to_wd_and_ht:cnn`

```

\box_gautosize_to_wd_and_ht:Nnn
\box_gautosize_to_wd_and_ht:cnn
35244 \cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
35245 { \_box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_set:Nn }
\box_autosize_to_wd_and_ht_plus_dp:Nnn
\box_autosize_to_wd_and_ht_plus_dp:cnn
35246 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
35247 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht:Nnn #1#2#3
35248 { \_box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_gset:Nn }
\box_gautosize_to_wd_and_ht_plus_dp:Nnn
\box_gautosize_to_wd_and_ht_plus_dp:cnn
35249 \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht:Nnn { c }
35250 \cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
35251 {
35252     \_box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }

```

```

35253     \hbox_set:Nn
35254   }
35255   \cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
35256   \cs_new_protected:Npn \box_gautosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
35257     {
35258       \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }
35259       \hbox_gset:Nn
35260     }
35261   \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht_plus_dp:Nnn { c }
35262   \cs_new_protected:Npn \__box_autosize:NnnnN #1#2#3#4#5
35263     {
35264       #5 #1
35265       {
35266         \fp_set:Nn \l__box_scale_x_fp { ( \dim_to_fp:n {#2} ) / \box_wd:N #1 }
35267         \fp_set:Nn \l__box_scale_y_fp
35268           { ( \dim_to_fp:n {#3} ) / ( \dim_to_fp:n {#4} ) }
35269         \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
35270           { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
35271           { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
35272         \__box_scale:N #1
35273       }
35274     }

```

(End of definition for `\box_autosize_to_wd_and_ht:Nnn` and others. These functions are documented on page 311.)

`__box_resize_common:N` The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

```

35275   \cs_new_protected:Npn \__box_resize_common:N #1
35276     {
35277       \hbox_set:Nn \l__box_internal_box
35278         {
35279           \__box_backend_scale:Nnn
35280             #1
35281             \l__box_scale_x_fp
35282             \l__box_scale_y_fp
35283         }

```

The new height and depth can be applied directly.

```

35284     \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
35285       {
35286         \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
35287         \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
35288       }
35289       {
35290         \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
35291         \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
35292       }

```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```

35293     \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
35294       {

```



```

35295     \hbox_to_wd:nn { \l__box_right_new_dim }
35296     {
35297         \__kernel_kern:n { \l__box_right_new_dim }
35298         \box_use_drop:N \l__box_internal_box
35299         \tex_hss:D
35300     }
35301 }
35302 {
35303     \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
35304     \hbox:n
35305     {
35306         \__kernel_kern:n { Opt }
35307         \box_use_drop:N \l__box_internal_box
35308         \tex_hss:D
35309     }
35310 }
35311 }

```

(End of definition for `__box_resize_common:N`.)

91.13 Viewing part of a box

`\box_set_clipped:N` A wrapper around the driver-dependent code.

```

\box_set_clipped:c 35312 \cs_new_protected:Npn \box_set_clipped:N #1
\box_gset_clipped:N 35313 { \hbox_set:Nn #1 { \__box_backend_clip:N #1 } }
\box_gset_clipped:c 35314 \cs_generate_variant:Nn \box_set_clipped:N { c }
35315 \cs_new_protected:Npn \box_gset_clipped:N #1
35316 { \hbox_gset:Nn #1 { \__box_backend_clip:N #1 } }
35317 \cs_generate_variant:Nn \box_gset_clipped:N { c }

```

(End of definition for `\box_set_clipped:N` and `\box_gset_clipped:N`. These functions are documented on page 313.)

`\box_set_trim:Nnnnn` Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```

\box_set_trim:cnnnn 35318 \cs_new_protected:Npn \box_set_trim:Nnnnn #1#2#3#4#5
\box_gset_trim:Nnnnn 35319 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
\__box_set_trim:NnnnnN 35320 \cs_generate_variant:Nn \box_set_trim:Nnnnn { c }
35321 \cs_new_protected:Npn \box_gset_trim:Nnnnn #1#2#3#4#5
35322 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
35323 \cs_generate_variant:Nn \box_gset_trim:Nnnnn { c }
35324 \cs_new_protected:Npn \__box_set_trim:NnnnnN #1#2#3#4#5#6
35325 {
35326     \hbox_set:Nn \l__box_internal_box
35327     {
35328         \__kernel_kern:n { -#2 }
35329         \box_use:N #1
35330         \__kernel_kern:n { -#4 }
35331     }

```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. `\box_move_down:nn` is used in both

cases so the resulting box always contains a `\lower` primitive. The internal box is used here as it allows safe use of `\box_set_dp:Nn`.

```

35332 \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
35333 {
35334   \hbox_set:Nn \l__box_internal_box
35335   {
35336     \box_move_down:nn \c_zero_dim
35337     { \box_use_drop:N \l__box_internal_box }
35338   }
35339   \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
35340 }
35341 {
35342   \hbox_set:Nn \l__box_internal_box
35343   {
35344     \box_move_down:nn { (#3) - \box_dp:N #1 }
35345     { \box_use_drop:N \l__box_internal_box }
35346   }
35347   \box_set_dp:Nn \l__box_internal_box \c_zero_dim
35348 }

```

Same thing, this time from the top of the box.

```

35349 \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
35350 {
35351   \hbox_set:Nn \l__box_internal_box
35352   {
35353     \box_move_up:nn \c_zero_dim
35354     { \box_use_drop:N \l__box_internal_box }
35355   }
35356   \box_set_ht:Nn \l__box_internal_box
35357   { \box_ht:N \l__box_internal_box - (#5) }
35358 }
35359 {
35360   \hbox_set:Nn \l__box_internal_box
35361   {
35362     \box_move_up:nn { (#5) - \box_ht:N \l__box_internal_box }
35363     { \box_use_drop:N \l__box_internal_box }
35364   }
35365   \box_set_ht:Nn \l__box_internal_box \c_zero_dim
35366 }
35367 #6 #1 \l__box_internal_box
35368 }

```

(End of definition for `\box_set_trim:Nnnnn`, `\box_gset_trim:Nnnnn`, and `__box_set_trim:NnnnnN`. These functions are documented on page 313.)

`\box_set_viewport:Nnnnn`
`\box_set_viewport:cnnnn`
`\box_gset_viewport:Nnnnn`
`\box_gset_viewport:cnnnn`
`__box_viewport:NnnnnN`

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```

35369 \cs_new_protected:Npn \box_set_viewport:Nnnnn #1#2#3#4#5
35370 { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
35371 \cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
35372 \cs_new_protected:Npn \box_gset_viewport:Nnnnn #1#2#3#4#5
35373 { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
35374 \cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
35375 \cs_new_protected:Npn \__box_set_viewport:NnnnnN #1#2#3#4#5#6

```

```

35376 {
35377   \hbox_set:Nn \l__box_internal_box
35378     {
35379       \__kernel_kern:n { -#2 }
35380       \box_use:N #1
35381       \__kernel_kern:n { #4 - \box_wd:N #1 }
35382     }
35383   \dim_compare:nNnTF {#3} < \c_zero_dim
35384     {
35385       \hbox_set:Nn \l__box_internal_box
35386         {
35387           \box_move_down:nn \c_zero_dim
35388             { \box_use_drop:N \l__box_internal_box }
35389         }
35390       \box_set_dp:Nn \l__box_internal_box { - \__box_dim_eval:n {#3} }
35391     }
35392     {
35393       \hbox_set:Nn \l__box_internal_box
35394         { \box_move_down:nn {#3} { \box_use_drop:N \l__box_internal_box } }
35395       \box_set_dp:Nn \l__box_internal_box \c_zero_dim
35396     }
35397   \dim_compare:nNnTF {#5} > \c_zero_dim
35398     {
35399       \hbox_set:Nn \l__box_internal_box
35400         {
35401           \box_move_up:nn \c_zero_dim
35402             { \box_use_drop:N \l__box_internal_box }
35403         }
35404       \box_set_ht:Nn \l__box_internal_box
35405         {
35406           (#5)
35407           \dim_compare:nNnT {#3} > \c_zero_dim
35408             { - (#3) }
35409         }
35410     }
35411     {
35412       \hbox_set:Nn \l__box_internal_box
35413         {
35414           \box_move_up:nn { - \__box_dim_eval:n {#5} }
35415             { \box_use_drop:N \l__box_internal_box }
35416         }
35417       \box_set_ht:Nn \l__box_internal_box \c_zero_dim
35418     }
35419   #6 #1 \l__box_internal_box
35420 }

```

(End of definition for \box_set_viewport:Nnnnn, \box_gset_viewport:Nnnnn, and __box_viewport:NnnnnN. These functions are documented on page 313.)

```

35421 </package>

```

Chapter 92

l3coffins implementation

```
35422 <*package>
35423 <@@=coffin>
```

92.1 Coffins: data structures and general variables

`\l__coffin_internal_box` Scratch variables.

```
\l__coffin_internal_dim 35424 \box_new:N \l__coffin_internal_box
\l__coffin_internal_tl 35425 \dim_new:N \l__coffin_internal_dim
35426 \tl_new:N \l__coffin_internal_tl
```

(End of definition for \l__coffin_internal_box, \l__coffin_internal_dim, and \l__coffin_internal_tl.)

`\c__coffin_corners_prop` The “corners”; of a coffin define the real content, as opposed to the \TeX bounding box. They all start off in the same place, of course.

```
35427 \prop_const_from_keyval:Nn \c__coffin_corners_prop
35428 {
35429   tl = { Opt } { Opt } ,
35430   tr = { Opt } { Opt } ,
35431   bl = { Opt } { Opt } ,
35432   br = { Opt } { Opt } ,
35433 }
```

(End of definition for \c__coffin_corners_prop.)

`\c__coffin_poles_prop` Pole positions are given for horizontal, vertical and reference-point based values.

```
35434 \prop_const_from_keyval:Nn \c__coffin_poles_prop
35435 {
35436   l = { Opt } { Opt } { Opt } { 1000pt } ,
35437   hc = { Opt } { Opt } { Opt } { 1000pt } ,
35438   r = { Opt } { Opt } { Opt } { 1000pt } ,
35439   b = { Opt } { Opt } { 1000pt } { Opt } ,
35440   vc = { Opt } { Opt } { 1000pt } { Opt } ,
35441   t = { Opt } { Opt } { 1000pt } { Opt } ,
35442   B = { Opt } { Opt } { 1000pt } { Opt } ,
35443   H = { Opt } { Opt } { 1000pt } { Opt } ,
35444   T = { Opt } { Opt } { 1000pt } { Opt } ,
35445 }
```

(End of definition for \c__coffin_poles_prop.)

`\l__coffin_slope_A_fp` Used for calculations of intersections.

`\l__coffin_slope_B_fp` 35446 `\fp_new:N \l__coffin_slope_A_fp`
35447 `\fp_new:N \l__coffin_slope_B_fp`

(End of definition for \l__coffin_slope_A_fp and \l__coffin_slope_B_fp.)

`\l__coffin_error_bool` For propagating errors so that parts of the code can work around them.

35448 `\bool_new:N \l__coffin_error_bool`

(End of definition for \l__coffin_error_bool.)

`\l__coffin_offset_x_dim` The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.

`\l__coffin_offset_y_dim`

35449 `\dim_new:N \l__coffin_offset_x_dim`
35450 `\dim_new:N \l__coffin_offset_y_dim`

(End of definition for \l__coffin_offset_x_dim and \l__coffin_offset_y_dim.)

`\l__coffin_pole_a_tl` Needed for finding the intersection of two poles.

`\l__coffin_pole_b_tl`

35451 `\tl_new:N \l__coffin_pole_a_tl`
35452 `\tl_new:N \l__coffin_pole_b_tl`

(End of definition for \l__coffin_pole_a_tl and \l__coffin_pole_b_tl.)

`\l__coffin_x_dim` For calculating intersections and so forth.

`\l__coffin_y_dim`

`\l__coffin_x_prime_dim`

`\l__coffin_y_prime_dim`

35453 `\dim_new:N \l__coffin_x_dim`
35454 `\dim_new:N \l__coffin_y_dim`
35455 `\dim_new:N \l__coffin_x_prime_dim`
35456 `\dim_new:N \l__coffin_y_prime_dim`

(End of definition for \l__coffin_x_dim and others.)

92.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

`__coffin_to_value:N` Coffins are a two-part structure and we rely on the internal nature of box allocation to make everything work. As such, we need an interface to turn coffin identifiers into numbers. For the purposes here, the signature allowed is N despite the nature of the underlying primitive.

35457 `\cs_new_eq:NN __coffin_to_value:N \tex_number:D`

(End of definition for __coffin_to_value:N.)

`\coffin_if_exist_p:N` Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```

\coffin_if_exist_p:c
\coffin_if_exist:NTF
\coffin_if_exist:cTF
35458 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
35459 {
35460   \cs_if_exist:NTF #1
35461   {
35462     \cs_if_exist:cTF { coffin ~ \__coffin_to_value:N #1 ~ poles }
35463     { \prg_return_true: }
35464     { \prg_return_false: }
35465   }
35466   { \prg_return_false: }
35467 }
35468 \prg_generate_conditional_variant:Nnn \coffin_if_exist:N
35469 { c } { p , T , F , TF }

```

(End of definition for \coffin_if_exist:NTF. This function is documented on page 315.)

`__coffin_if_exist:NT` Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. So a wrapper is provided to deal with this correctly, issuing an error on erroneous use.

```

35470 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
35471 {
35472   \coffin_if_exist:NTF #1
35473   { #2 }
35474   {
35475     \msg_error:nne { coffin } { unknown }
35476     { \token_to_str:N #1 }
35477   }
35478 }

```

(End of definition for __coffin_if_exist:NT.)

`\coffin_clear:N` Clearing coffins means emptying the box and resetting all of the structures.

```

\coffin_clear:c
\coffin_gclear:N
\coffin_gclear:c
35479 \cs_new_protected:Npn \coffin_clear:N #1
35480 {
35481   \__coffin_if_exist:NT #1
35482   {
35483     \box_clear:N #1
35484     \__coffin_reset_structure:N #1
35485   }
35486 }
35487 \cs_generate_variant:Nn \coffin_clear:N { c }
35488 \cs_new_protected:Npn \coffin_gclear:N #1
35489 {
35490   \__coffin_if_exist:NT #1
35491   {
35492     \box_gclear:N #1
35493     \__coffin_greset_structure:N #1
35494   }
35495 }
35496 \cs_generate_variant:Nn \coffin_gclear:N { c }

```

(End of definition for \coffin_clear:N and \coffin_gclear:N. These functions are documented on page 315.)

\coffin_new:N Creating a new coffin means making the underlying box and adding the data structures. The **\debug_suspend:** and **\debug_resume:** functions prevent **\prop_gclear_new:c** from writing useless information to the log file.

```

35497 \cs_new_protected:Npn \coffin_new:N #1
35498 {
35499   \box_new:N #1
35500   \debug_suspend:
35501   \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ corners }
35502   \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ poles }
35503   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
35504     \c__coffin_corners_prop
35505   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
35506     \c__coffin_poles_prop
35507   \debug_resume:
35508 }
35509 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End of definition for \coffin_new:N. This function is documented on page 315.)

\hcoffin_set:Nn Horizontal coffins are relatively easy: set the appropriate box, reset the structures then
\hcoffin_set:cn update the handle positions.

```

\hcoffin_gset:Nn
\hcoffin_gset:cn
35510 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
35511 {
35512   \__coffin_if_exist:NT #1
35513   {
35514     \hbox_set:Nn #1
35515     {
35516       \color_ensure_current:
35517       #2
35518     }
35519     \coffin_reset_poles:N #1
35520   }
35521 }
35522 \cs_generate_variant:Nn \hcoffin_set:Nn { c }
35523 \cs_new_protected:Npn \hcoffin_gset:Nn #1#2
35524 {
35525   \__coffin_if_exist:NT #1
35526   {
35527     \hbox_gset:Nn #1
35528     {
35529       \color_ensure_current:
35530       #2
35531     }
35532     \coffin_greset_poles:N #1
35533   }
35534 }
35535 \cs_generate_variant:Nn \hcoffin_gset:Nn { c }

```

(End of definition for \hcoffin_set:Nn and \hcoffin_gset:Nn. These functions are documented on page 316.)

\vcoffin_set:Nnn Setting vertical coffins is more complex. First, the material is typeset with a given width.
\vcoffin_set:cnn The default handles and poles are set as for a horizontal coffin, before finding the top
\vcoffin_gset:Nnn baseline using a temporary box. No **\color_ensure_current:** here as that would add a
\vcoffin_gset:cnn

__coffin_set_vertical:NnnNNN

__coffin_set_vertical_aux:

whatsit to the start of the vertical box and mess up the location of the T pole (see *TEX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```

35536 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
35537 {
35538   \__coffin_set_vertical:NnnNNN #1 {#2} {#3}
35539   \vbox_set:Nn \coffin_reset_poles:N \__coffin_set_pole:Nnn
35540 }
35541 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
35542 \cs_new_protected:Npn \vcoffin_gset:Nnn #1#2#3
35543 {
35544   \__coffin_set_vertical:NnnNNN #1 {#2} {#3}
35545   \vbox_gset:Nn \coffin_greset_poles:N \__coffin_gset_pole:Nnn
35546 }
35547 \cs_generate_variant:Nn \vcoffin_gset:Nnn { c }
35548 \cs_new_protected:Npn \__coffin_set_vertical:NnnNNN #1#2#3#4#5#6
35549 {
35550   \__coffin_if_exist:NT #1
35551   {
35552     #4 #1
35553     {
35554       \dim_set:Nn \tex_hsize:D {#2}
35555       \__coffin_set_vertical_aux:
35556       #3
35557     }
35558     #5 #1
35559     \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
35560     #6 #1 { T }
35561     {
35562       { Opt }
35563       {
35564         \dim_eval:n
35565         { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
35566       }
35567       { 1000pt }
35568       { Opt }
35569     }
35570     \box_clear:N \l__coffin_internal_box
35571   }
35572 }
35573 \cs_new_protected:Npe \__coffin_set_vertical_aux:
35574 {
35575   \bool_lazy_and:nnT
35576   { \cs_if_exist_p:N \fmtname }
35577   { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
35578   {
35579     \dim_set_eq:NN \exp_not:N \linewidth \tex_hsize:D
35580     \dim_set_eq:NN \exp_not:N \columnwidth \tex_hsize:D
35581   }
35582 }

```

(End of definition for `\vcoffin_set:Nnn` and others. These functions are documented on page 316.)

`\hcoffin_set:Nw` These are the “begin”/“end” versions of the above: watch the grouping!
`\hcoffin_set:cw` 35583 \cs_new_protected:Npn \hcoffin_set:Nw #1
`\hcoffin_gset:Nw`
`\hcoffin_gset:cw`
`\hcoffin_set_end:`
`\hcoffin_gset_end:`


```

35584 {
35585   \__coffin_if_exist:NT #1
35586   {
35587     \hbox_set:Nw #1 \color_ensure_current:
35588     \cs_set_protected:Npn \hcoffin_set_end:
35589     {
35590       \hbox_set_end:
35591       \coffin_reset_poles:N #1
35592     }
35593   }
35594 }
35595 \cs_generate_variant:Nn \hcoffin_set:Nw { c }
35596 \cs_new_protected:Npn \hcoffin_gset:Nw #1
35597 {
35598   \__coffin_if_exist:NT #1
35599   {
35600     \hbox_gset:Nw #1 \color_ensure_current:
35601     \cs_set_protected:Npn \hcoffin_gset_end:
35602     {
35603       \hbox_gset_end:
35604       \coffin_greset_poles:N #1
35605     }
35606   }
35607 }
35608 \cs_generate_variant:Nn \hcoffin_gset:Nw { c }
35609 \cs_new_protected:Npn \hcoffin_set_end: { }
35610 \cs_new_protected:Npn \hcoffin_gset_end: { }

```

(End of definition for `\hcoffin_set:Nw` and others. These functions are documented on page 316.)

```

\vcoffin_set:Nnw The same for vertical coffins.
\vcoffin_set:cnw 35611 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
\vcoffin_gset:Nnw 35612 {
\vcoffin_gset:cnw 35613   \__coffin_set_vertical:NnNNNNWw #1 {#2} \vbox_set:Nw
\__coffin_set_vertical:NnNNNNWw 35614   \vcoffin_set_end:
\vcoffin_set_end: 35615   \vbox_set_end: \coffin_reset_poles:N \__coffin_set_pole:Nnn
\vcoffin_gset_end: 35616 }
35617 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }
35618 \cs_new_protected:Npn \vcoffin_gset:Nnw #1#2
35619 {
35620   \__coffin_set_vertical:NnNNNNWw #1 {#2} \vbox_gset:Nw
35621   \vcoffin_gset_end:
35622   \vbox_gset_end: \coffin_greset_poles:N \__coffin_gset_pole:Nnn
35623 }
35624 \cs_generate_variant:Nn \vcoffin_gset:Nnw { c }
35625 \cs_new_protected:Npn \__coffin_set_vertical:NnNNNNWw #1#2#3#4#5#6#7
35626 {
35627   \__coffin_if_exist:NT #1
35628   {
35629     #3 #1
35630     \dim_set:Nn \tex_hsize:D {#2}
35631     \__coffin_set_vertical_aux:
35632     \cs_set_protected:Npn #4
35633     {

```

```

35634         #5
35635         #6 #1
35636         \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
35637         #7 #1 { T }
35638         {
35639             { Opt }
35640             {
35641                 \dim_eval:n
35642                 { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
35643             }
35644             { 1000pt }
35645             { Opt }
35646         }
35647         \box_clear:N \l__coffin_internal_box
35648     }
35649 }
35650 }
35651 \cs_new_protected:Npn \vcoffin_set_end: { }
35652 \cs_new_protected:Npn \vcoffin_gset_end: { }

```

(End of definition for `\vcoffin_set:Nnw` and others. These functions are documented on page 316.)

```

\coffin_set_eq:NN Setting two coffins equal is just a wrapper around other functions.
\coffin_set_eq:Nc 35653 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
\coffin_set_eq:cN 35654 {
\coffin_set_eq:cc 35655     \__coffin_if_exist:NT #1
\coffin_gset_eq:NN 35656     {
\coffin_gset_eq:Nc 35657         \box_set_eq:NN #1 #2
\coffin_gset_eq:cN 35658         \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
\coffin_gset_eq:cc 35659         { coffin ~ \__coffin_to_value:N #2 ~ corners }
\coffin_gset_eq:cc 35660         \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
\coffin_gset_eq:cc 35661         { coffin ~ \__coffin_to_value:N #2 ~ poles }
35662     }
35663 }
35664 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
35665 \cs_new_protected:Npn \coffin_gset_eq:NN #1#2
35666 {
35667     \__coffin_if_exist:NT #1
35668     {
35669         \box_gset_eq:NN #1 #2
35670         \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
35671         { coffin ~ \__coffin_to_value:N #2 ~ corners }
35672         \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
35673         { coffin ~ \__coffin_to_value:N #2 ~ poles }
35674     }
35675 }
35676 \cs_generate_variant:Nn \coffin_gset_eq:NN { c , Nc , cc }

```

(End of definition for `\coffin_set_eq:NN` and `\coffin_gset_eq:NN`. These functions are documented on page 315.)

\c_empty_coffin Special coffins: these cannot be set up earlier as they need `\coffin_new:N`. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available. The empty coffin is created entirely by hand: not everything is in place yet.

```

\l__coffin_aligned_coffin
\l__coffin_aligned_internal_coffin

```

```

35677 \coffin_new:N \c_empty_coffin
35678 \coffin_new:N \l__coffin_aligned_coffin
35679 \coffin_new:N \l__coffin_aligned_internal_coffin

```

(End of definition for `\c_empty_coffin`, `\l__coffin_aligned_coffin`, and `\l__coffin_aligned_internal_coffin`. This variable is documented on page 320.)

```

\l_tmpa_coffin The usual scratch space.
\l_tmpb_coffin 35680 \coffin_new:N \l_tmpa_coffin
\g_tmpa_coffin 35681 \coffin_new:N \l_tmpb_coffin
\g_tmpb_coffin 35682 \coffin_new:N \g_tmpa_coffin
               35683 \coffin_new:N \g_tmpb_coffin

```

(End of definition for `\l_tmpa_coffin` and others. These variables are documented on page 320.)

92.3 Measuring coffins

`\coffin_dp:N` Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```

\coffin_dp:c
\coffin_ht:N 35684 \cs_new_eq:NN \coffin_dp:N \box_dp:N
\coffin_ht:c 35685 \cs_new_eq:NN \coffin_dp:c \box_dp:c
\coffin_wd:N 35686 \cs_new_eq:NN \coffin_ht:N \box_ht:N
\coffin_wd:c 35687 \cs_new_eq:NN \coffin_ht:c \box_ht:c
             35688 \cs_new_eq:NN \coffin_wd:N \box_wd:N
             35689 \cs_new_eq:NN \coffin_wd:c \box_wd:c

```

(End of definition for `\coffin_dp:N`, `\coffin_ht:N`, and `\coffin_wd:N`. These functions are documented on page 318.)

92.4 Coffins: handle and pole management

`__coffin_get_pole:NnN` A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```

35690 \cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
35691 {
35692   \prop_get:cnNF
35693     { coffin ~ \__coffin_to_value:N #1 ~ poles } {#2} #3
35694     {
35695       \msg_error:nnee { coffin } { unknown-pole }
35696       { \exp_not:n {#2} } { \token_to_str:N #1 }
35697       \tl_set:Nn #3 { { Opt } { Opt } { Opt } { Opt } }
35698     }
35699 }

```

(End of definition for `__coffin_get_pole:NnN`.)

`__coffin_reset_structure:N` Resetting the structure is a simple copy job.

```

\__coffin_greset_structure:N
35700 \cs_new_protected:Npn \__coffin_reset_structure:N #1
35701 {
35702   \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
35703   \c__coffin_corners_prop
35704   \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
35705   \c__coffin_poles_prop

```

```

35706 }
35707 \cs_new_protected:Npn \__coffin_greset_structure:N #1
35708 {
35709   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
35710   \c__coffin_corners_prop
35711   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
35712   \c__coffin_poles_prop
35713 }

```

(End of definition for __coffin_reset_structure:N and __coffin_greset_structure:N.)

`\coffin_set_horizontal_pole:Nnn` Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```

\coffin_set_horizontal_pole:cnm
\coffin_gset_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnm
\__coffin_set_horizontal_pole:NnnN
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnm
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnm
\__coffin_set_vertical_pole:NnnN
\__coffin_set_pole:Nnn
\__coffin_gset_pole:Nnn
35714 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
35715 { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_put:cne }
35716 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
35717 \cs_new_protected:Npn \coffin_gset_horizontal_pole:Nnn #1#2#3
35718 { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_gput:cne }
35719 \cs_generate_variant:Nn \coffin_gset_horizontal_pole:Nnn { c }
35720 \cs_new_protected:Npn \__coffin_set_horizontal_pole:NnnN #1#2#3#4
35721 {
35722   \__coffin_if_exist:NT #1
35723   {
35724     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
35725     {#2}
35726     {
35727       { Opt } { \dim_eval:n {#3} }
35728       { 1000pt } { Opt }
35729     }
35730   }
35731 }
35732 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
35733 { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_put:cne }
35734 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
35735 \cs_new_protected:Npn \coffin_gset_vertical_pole:Nnn #1#2#3
35736 { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_gput:cne }
35737 \cs_generate_variant:Nn \coffin_gset_vertical_pole:Nnn { c }
35738 \cs_new_protected:Npn \__coffin_set_vertical_pole:NnnN #1#2#3#4
35739 {
35740   \__coffin_if_exist:NT #1
35741   {
35742     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
35743     {#2}
35744     {
35745       { \dim_eval:n {#3} } { Opt }
35746       { Opt } { 1000pt }
35747     }
35748   }
35749 }
35750 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
35751 {
35752   \prop_put:cne { coffin ~ \__coffin_to_value:N #1 ~ poles }
35753   {#2} {#3}

```

```

35754 }
35755 \cs_new_protected:Npn \__coffin_gset_pole:Nnn #1#2#3
35756 {
35757   \prop_gput:cne { coffin ~ \__coffin_to_value:N #1 ~ poles }
35758   {#2} {#3}
35759 }

```

(End of definition for `\coffin_set_horizontal_pole:Nnn` and others. These functions are documented on page 316.)

`\coffin_reset_poles:N`
`\coffin_greset_poles:N`

Simple shortcuts.

```

35760 \cs_new_protected:Npn \coffin_reset_poles:N #1
35761 {
35762   \__coffin_reset_structure:N #1
35763   \__coffin_update_corners:N #1
35764   \__coffin_update_poles:N #1
35765 }
35766 \cs_new_protected:Npn \coffin_greset_poles:N #1
35767 {
35768   \__coffin_greset_structure:N #1
35769   \__coffin_gupdate_corners:N #1
35770   \__coffin_gupdate_poles:N #1
35771 }

```

(End of definition for `\coffin_reset_poles:N` and `\coffin_greset_poles:N`. These functions are documented on page 317.)

`__coffin_update_corners:N`
`__coffin_gupdate_corners:N`
`__coffin_update_corners:NN`
`__coffin_update_corners:NNN`

Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying \TeX box.

```

35772 \cs_new_protected:Npn \__coffin_update_corners:N #1
35773 { \__coffin_update_corners:NN #1 \prop_put:Nne }
35774 \cs_new_protected:Npn \__coffin_gupdate_corners:N #1
35775 { \__coffin_update_corners:NN #1 \prop_gput:Nne }
35776 \cs_new_protected:Npn \__coffin_update_corners:NN #1#2
35777 {
35778   \exp_args:Nc \__coffin_update_corners:NNN
35779   { coffin ~ \__coffin_to_value:N #1 ~ corners }
35780   #1 #2
35781 }
35782 \cs_new_protected:Npn \__coffin_update_corners:NNN #1#2#3
35783 {
35784   #3 #1
35785   { tl }
35786   { { Opt } { \dim_eval:n { \box_ht:N #2 } } } }
35787   #3 #1
35788   { tr }
35789   {
35790     { \dim_eval:n { \box_wd:N #2 } }
35791     { \dim_eval:n { \box_ht:N #2 } }
35792   }
35793   #3 #1
35794   { bl }
35795   { { Opt } { \dim_eval:n { -\box_dp:N #2 } } } }
35796   #3 #1
35797   { br }

```

```

35798     {
35799         { \dim_eval:n { \box_wd:N #2 } }
35800         { \dim_eval:n { -\box_dp:N #2 } }
35801     }
35802 }

```

(End of definition for `__coffin_update_corners:N` and others.)

`__coffin_update_poles:N` This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```

\__coffin_gupdate_poles:N
\__coffin_update_poles:NN
\__coffin_update_poles:NNN
35803 \cs_new_protected:Npn \__coffin_update_poles:N #1
35804 { \__coffin_update_poles:NN #1 \prop_put:Nne }
35805 \cs_new_protected:Npn \__coffin_gupdate_poles:N #1
35806 { \__coffin_update_poles:NN #1 \prop_gput:Nne }
35807 \cs_new_protected:Npn \__coffin_update_poles:NN #1#2
35808 {
35809     \exp_args:Nc \__coffin_update_poles:NNN
35810     { coffin ~ \__coffin_to_value:N #1 ~ poles }
35811     #1 #2
35812 }
35813 \cs_new_protected:Npn \__coffin_update_poles:NNN #1#2#3
35814 {
35815     #3 #1 { hc }
35816     {
35817         { \dim_eval:n { 0.5 \box_wd:N #2 } }
35818         { Opt } { Opt } { 1000pt }
35819     }
35820     #3 #1 { r }
35821     {
35822         { \dim_eval:n { \box_wd:N #2 } }
35823         { Opt } { Opt } { 1000pt }
35824     }
35825     #3 #1 { vc }
35826     {
35827         { Opt }
35828         { \dim_eval:n { ( \box_ht:N #2 - \box_dp:N #2 ) / 2 } }
35829         { 1000pt }
35830         { Opt }
35831     }
35832     #3 #1 { t }
35833     {
35834         { Opt }
35835         { \dim_eval:n { \box_ht:N #2 } }
35836         { 1000pt }
35837         { Opt }
35838     }
35839     #3 #1 { b }
35840     {
35841         { Opt }
35842         { \dim_eval:n { -\box_dp:N #2 } }
35843         { 1000pt }
35844         { Opt }

```

```

35845     }
35846 }

```

(End of definition for `__coffin_update_poles:N` and others.)

92.5 Coffins: calculation of pole intersections

```

\__coffin_calculate_intersection:Nnn
\__coffin_calculate_intersection:nnnnnnnn
\__coffin_calculate_intersection:nnnnnn

```

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

```

35847 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
35848 {
35849   \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
35850   \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
35851   \bool_set_false:N \l__coffin_error_bool
35852   \exp_last_two_unbraced:Noo
35853     \__coffin_calculate_intersection:nnnnnnnn
35854     \l__coffin_pole_a_tl \l__coffin_pole_b_tl
35855   \bool_if:NT \l__coffin_error_bool
35856     {
35857       \msg_error:nn { coffin } { no-pole-intersection }
35858       \dim_zero:N \l__coffin_x_dim
35859       \dim_zero:N \l__coffin_y_dim
35860     }
35861 }

```

The two poles passed here each have four values (as dimensions), (a, b, c, d) and (a', b', c', d') . These are arguments 1–4 and 5–8, respectively. In both cases a and b are the coordinates of a point on the pole and c and d define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by d/c and d'/c' . However, if one of the poles is either horizontal or vertical then one or more of c, d, c' and d' are zero and a special case is needed.

```

35862 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
35863   #1#2#3#4#5#6#7#8
35864 {
35865   \dim_compare:nNnTF {#3} = \c_zero_dim

```

The case where the first pole is vertical. So the x -component of the intersection is at a . There is then a test on the second pole: if it is also vertical then there is an error.

```

35866     {
35867       \dim_set:Nn \l__coffin_x_dim {#1}
35868       \dim_compare:nNnTF {#7} = \c_zero_dim
35869         { \bool_set_true:N \l__coffin_error_bool }

```

The second pole may still be horizontal, in which case the y -component of the intersection is b' . If not,

$$y = \frac{d'}{c'}(a - a') + b'$$

with the x -component already known to be $\#1$.

```

35870     {
35871       \dim_set:Nn \l__coffin_y_dim
35872         {
35873           \dim_compare:nNnTF {#8} = \c_zero_dim

```

```

35874         {#6}
35875         {
35876             \fp_to_dim:n
35877             {
35878                 ( \dim_to_fp:n {#8} / \dim_to_fp:n {#7} )
35879                 * ( \dim_to_fp:n {#1} - \dim_to_fp:n {#5} )
35880                 + \dim_to_fp:n {#6}
35881             }
35882         }
35883     }
35884 }
35885 }

```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the x - and y -components interchanged.

```

35886 {
35887     \dim_compare:nNnTF {#4} = \c_zero_dim
35888     {
35889         \dim_set:Nn \l__coffin_y_dim {#2}
35890         \dim_compare:nNnTF {#8} = { \c_zero_dim }
35891         { \bool_set_true:N \l__coffin_error_bool }
35892     }

```

Now we deal with the case where the second pole may be vertical, or if not we have

$$x = \frac{c'}{d'}(b - b') + a'$$

which is again handled by the same auxiliary.

```

35893     \dim_set:Nn \l__coffin_x_dim
35894     {
35895         \dim_compare:nNnTF {#7} = \c_zero_dim
35896         {#5}
35897         {
35898             \fp_to_dim:n
35899             {
35900                 ( \dim_to_fp:n {#7} / \dim_to_fp:n {#8} )
35901                 * ( \dim_to_fp:n {#4} - \dim_to_fp:n {#6} )
35902                 + \dim_to_fp:n {#5}
35903             }
35904         }
35905     }
35906 }
35907 }

```

The first pole is neither horizontal nor vertical. To avoid even more complexity, we now work out both slopes and pass to an auxiliary.

```

35908 {
35909     \use:e
35910     {
35911         \__coffin_calculate_intersection:nnnnnn
35912         { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
35913         { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
35914     }
35915     {#1} {#2} {#5} {#6}
35916 }

```



```

35917     }
35918 }

```

Assuming the two poles are not parallel, then the intersection point is found in two steps. First we find the x -value with

$$x = \frac{sa - s'a' - b + b'}{s - s'}$$

and then finding the y -value with

$$y = s(x - a) + b$$

```

35919 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnn #1#2#3#4#5#6
35920 {
35921   \fp_compare:nNnTF {#1} = {#2}
35922   { \bool_set_true:N \l__coffin_error_bool }
35923   {
35924     \dim_set:Nn \l__coffin_x_dim
35925     {
35926       \fp_to_dim:n
35927       {
35928         (
35929           #1 * \dim_to_fp:n {#3}
35930           - #2 * \dim_to_fp:n {#5}
35931           - \dim_to_fp:n {#4}
35932           + \dim_to_fp:n {#6}
35933         )
35934         /
35935         ( #1 - #2 )
35936       }
35937     }
35938     \dim_set:Nn \l__coffin_y_dim
35939     {
35940       \fp_to_dim:n
35941       {
35942         #1 * ( \l__coffin_x_dim - \dim_to_fp:n {#3} )
35943         + \dim_to_fp:n {#4}
35944       }
35945     }
35946   }
35947 }

```

(End of definition for __coffin_calculate_intersection:Nnn, __coffin_calculate_intersection:nnnnnnnn, and __coffin_calculate_intersection:nnnnnn.)

92.6 Affine transformations

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.

```

\l__coffin_cos_fp
35948 \fp_new:N \l__coffin_sin_fp
35949 \fp_new:N \l__coffin_cos_fp

```

(End of definition for \l__coffin_sin_fp and \l__coffin_cos_fp.)

`\l__coffin_bounding_prop` A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```
35950 \prop_new:N \l__coffin_bounding_prop
```

(End of definition for `\l__coffin_bounding_prop`.)

`\l__coffin_corners_prop` Used to avoid needing to track scope for intermediate steps.

```
\l__coffin_poles_prop 35951 \prop_new:N \l__coffin_corners_prop
```

```
35952 \prop_new:N \l__coffin_poles_prop
```

(End of definition for `\l__coffin_corners_prop` and `\l__coffin_poles_prop`.)

`\l__coffin_bounding_shift_dim` The shift of the bounding box of a coffin from the real content.

```
35953 \dim_new:N \l__coffin_bounding_shift_dim
```

(End of definition for `\l__coffin_bounding_shift_dim`.)

`\l__coffin_left_corner_dim` These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

`\l__coffin_right_corner_dim`

`\l__coffin_bottom_corner_dim`

```
35954 \dim_new:N \l__coffin_left_corner_dim
```

```
35955 \dim_new:N \l__coffin_right_corner_dim
```

```
35956 \dim_new:N \l__coffin_bottom_corner_dim
```

```
35957 \dim_new:N \l__coffin_top_corner_dim
```

(End of definition for `\l__coffin_left_corner_dim` and others.)

`\coffin_rotate:Nn` Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set `\l__coffin_sin_fp` and `\l__coffin_cos_fp`, which are carried through unchanged for the rest of the procedure.

`\coffin_rotate:cn`

`\coffin_grotate:Nn`

`\coffin_grotate:cn`

`__coffin_rotate:NnNNN`

```
35958 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
```

```
35959 { \__coffin_rotate:NnNNN #1 {#2} \box_rotate:Nn \prop_set_eq:cN \hbox_set:Nn }
```

```
35960 \cs_generate_variant:Nn \coffin_rotate:Nn { c }
```

```
35961 \cs_new_protected:Npn \coffin_grotate:Nn #1#2
```

```
35962 { \__coffin_rotate:NnNNN #1 {#2} \box_grotate:Nn \prop_gset_eq:cN \hbox_gset:Nn }
```

```
35963 \cs_generate_variant:Nn \coffin_grotate:Nn { c }
```

```
35964 \cs_new_protected:Npn \__coffin_rotate:NnNNN #1#2#3#4#5
```

```
35965 {
```

```
35966 \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
```

```
35967 \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }
```

Use a local copy of the property lists to avoid needing to pass the name and scope around.

```
35968 \prop_set_eq:Nc \l__coffin_corners_prop
```

```
35969 { coffin ~ \__coffin_to_value:N #1 ~ corners }
```

```
35970 \prop_set_eq:Nc \l__coffin_poles_prop
```

```
35971 { coffin ~ \__coffin_to_value:N #1 ~ poles }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
35972 \prop_map_inline:Nn \l__coffin_corners_prop
```

```
35973 { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
```

```
35974 \prop_map_inline:Nn \l__coffin_poles_prop
```

```
35975 { \__coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```

35976   \__coffin_set_bounding:N #1
35977   \prop_map_inline:Nn \l__coffin_bounding_prop
35978     { \__coffin_rotate_bounding:nnn {##1} ##2 }

```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```

35979   \__coffin_find_corner_maxima:N #1
35980   \__coffin_find_bounding_shift:
35981     #3 #1 {#2}

```

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The x -direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The y -direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

```

35982   \hbox_set:Nn \l__coffin_internal_box
35983     {
35984       \__kernel_kern:n
35985         { \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim }
35986       \box_move_down:nn { \l__coffin_bottom_corner_dim }
35987         { \box_use:N #1 }
35988     }

```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```

35989   \box_set_ht:Nn \l__coffin_internal_box
35990     { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
35991   \box_set_dp:Nn \l__coffin_internal_box { Opt }
35992   \box_set_wd:Nn \l__coffin_internal_box
35993     { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
35994   #5 #1 { \box_use_drop:N \l__coffin_internal_box }

```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```

35995   \prop_map_inline:Nn \l__coffin_corners_prop
35996     { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
35997   \prop_map_inline:Nn \l__coffin_poles_prop
35998     { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }

```

Update the coffin data.

```

35999   #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
36000     \l__coffin_corners_prop
36001   #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
36002     \l__coffin_poles_prop
36003 }

```

(End of definition for \coffin_rotate:Nn, \coffin_grotate:Nn, and __coffin_rotate:NnNNN. These functions are documented on page 317.)

`__coffin_set_bounding:N` The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```

36004 \cs_new_protected:Npn \__coffin_set_bounding:N #1
36005 {
36006   \prop_put:Nne \l__coffin_bounding_prop { tl }
36007   { { Opt } { \dim_eval:n { \box_ht:N #1 } } }
36008   \prop_put:Nne \l__coffin_bounding_prop { tr }
36009   {
36010     { \dim_eval:n { \box_wd:N #1 } }
36011     { \dim_eval:n { \box_ht:N #1 } }
36012   }
36013   \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
36014   \prop_put:Nne \l__coffin_bounding_prop { bl }
36015   { { Opt } { \dim_use:N \l__coffin_internal_dim } }
36016   \prop_put:Nne \l__coffin_bounding_prop { br }
36017   {
36018     { \dim_eval:n { \box_wd:N #1 } }
36019     { \dim_use:N \l__coffin_internal_dim }
36020   }
36021 }

```

(End of definition for __coffin_set_bounding:N.)

`__coffin_rotate_bounding:nnn`
`__coffin_rotate_corner:Nnnn` Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```

36022 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
36023 {
36024   \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
36025   \prop_put:Nne \l__coffin_bounding_prop {#1}
36026   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
36027 }
36028 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
36029 {
36030   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
36031   \prop_put:Nne \l__coffin_corners_prop {#2}
36032   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
36033 }

```

(End of definition for __coffin_rotate_bounding:nnn and __coffin_rotate_corner:Nnnn.)

`__coffin_rotate_pole:Nnnnnn` Rotating a single pole simply means shifting the coordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```

36034 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
36035 {
36036   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
36037   \__coffin_rotate_vector:nnNN {#5} {#6}
36038   \l__coffin_x_prime_dim \l__coffin_y_prime_dim
36039   \prop_put:Nne \l__coffin_poles_prop {#2}
36040   {
36041     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
36042     { \dim_use:N \l__coffin_x_prime_dim }
36043     { \dim_use:N \l__coffin_y_prime_dim }
36044   }
36045 }

```

(End of definition for `_coffin_rotate_pole:Nnnnnn`.)

`_coffin_rotate_vector:nnNN` A rotation function, which needs only an input vector (as dimensions) and an output space. The values `\l__coffin_cos_fp` and `\l__coffin_sin_fp` should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```

36046 \cs_new_protected:Npn \_coffin_rotate_vector:nnNN #1#2#3#4
36047   {
36048     \dim_set:Nn #3
36049     {
36050       \fp_to_dim:n
36051       {
36052         \dim_to_fp:n {#1} * \l__coffin_cos_fp
36053         - \dim_to_fp:n {#2} * \l__coffin_sin_fp
36054       }
36055     }
36056     \dim_set:Nn #4
36057     {
36058       \fp_to_dim:n
36059       {
36060         \dim_to_fp:n {#1} * \l__coffin_sin_fp
36061         + \dim_to_fp:n {#2} * \l__coffin_cos_fp
36062       }
36063     }
36064   }

```

(End of definition for `_coffin_rotate_vector:nnNN`.)

`_coffin_find_corner_maxima:N`
`_coffin_find_corner_maxima_aux:nn` The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```

36065 \cs_new_protected:Npn \_coffin_find_corner_maxima:N #1
36066   {
36067     \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
36068     \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
36069     \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
36070     \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
36071     \prop_map_inline:Nn \l__coffin_corners_prop
36072     { \_coffin_find_corner_maxima_aux:nn ##2 }
36073   }
36074 \cs_new_protected:Npn \_coffin_find_corner_maxima_aux:nn #1#2
36075   {
36076     \dim_set:Nn \l__coffin_left_corner_dim
36077     { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
36078     \dim_set:Nn \l__coffin_right_corner_dim
36079     { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
36080     \dim_set:Nn \l__coffin_bottom_corner_dim
36081     { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
36082     \dim_set:Nn \l__coffin_top_corner_dim
36083     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
36084   }

```

(End of definition for `_coffin_find_corner_maxima:N` and `_coffin_find_corner_maxima_aux:nn`.)

`_coffin_find_bounding_shift:`
`_coffin_find_bounding_shift_aux:nn`

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```

36085 \cs_new_protected:Npn \_coffin_find_bounding_shift:
36086 {
36087   \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
36088   \prop_map_inline:Nn \l__coffin_bounding_prop
36089     { \_coffin_find_bounding_shift_aux:nn ##2 }
36090 }
36091 \cs_new_protected:Npn \_coffin_find_bounding_shift_aux:nn #1#2
36092 {
36093   \dim_set:Nn \l__coffin_bounding_shift_dim
36094     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
36095 }

```

(End of definition for _coffin_find_bounding_shift: and _coffin_find_bounding_shift_aux:nn.)

`_coffin_shift_corner:Nnnn`
`_coffin_shift_pole:Nnnnnn`

Shifting the corners and poles of a coffin means subtracting the appropriate values from the x - and y -components. For the poles, this means that the direction vector is unchanged.

```

36096 \cs_new_protected:Npn \_coffin_shift_corner:Nnnn #1#2#3#4
36097 {
36098   \prop_put:Nne \l__coffin_corners_prop {#2}
36099   {
36100     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
36101     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
36102   }
36103 }
36104 \cs_new_protected:Npn \_coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
36105 {
36106   \prop_put:Nne \l__coffin_poles_prop {#2}
36107   {
36108     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
36109     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
36110     {#5} {#6}
36111   }
36112 }

```

(End of definition for _coffin_shift_corner:Nnnn and _coffin_shift_pole:Nnnnnn.)

`\l__coffin_scale_x_fp`
`\l__coffin_scale_y_fp`

Storage for the scaling factors in x and y , respectively.

```

36113 \fp_new:N \l__coffin_scale_x_fp
36114 \fp_new:N \l__coffin_scale_y_fp

```

(End of definition for \l__coffin_scale_x_fp and \l__coffin_scale_y_fp.)

`\l__coffin_scaled_total_height_dim`
`\l__coffin_scaled_width_dim`

When scaling, the values given have to be turned into absolute values.

```

36115 \dim_new:N \l__coffin_scaled_total_height_dim
36116 \dim_new:N \l__coffin_scaled_width_dim

```

(End of definition for \l__coffin_scaled_total_height_dim and \l__coffin_scaled_width_dim.)

`\coffin_resize:Nnn` Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

```

\coffin_resize:cnn
\coffin_gresize:Nnn
\coffin_gresize:cnn
\__coffin_resize:NnnNN
36117 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
36118 {
36119   \__coffin_resize:NnnNN #1 {#2} {#3}
36120   \box_resize_to_wd_and_ht_plus_dp:Nnn
36121   \prop_set_eq:cN
36122 }
36123 \cs_generate_variant:Nn \coffin_resize:Nnn { c }
36124 \cs_new_protected:Npn \coffin_gresize:Nnn #1#2#3
36125 {
36126   \__coffin_resize:NnnNN #1 {#2} {#3}
36127   \box_gresize_to_wd_and_ht_plus_dp:Nnn
36128   \prop_gset_eq:cN
36129 }
36130 \cs_generate_variant:Nn \coffin_gresize:Nnn { c }
36131 \cs_new_protected:Npn \__coffin_resize:NnnNN #1#2#3#4#5
36132 {
36133   \fp_set:Nn \l__coffin_scale_x_fp
36134   { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
36135   \fp_set:Nn \l__coffin_scale_y_fp
36136   {
36137     \dim_to_fp:n {#3}
36138     / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
36139   }
36140   #4 #1 {#2} {#3}
36141   \__coffin_resize_common:NnnN #1 {#2} {#3} #5
36142 }

```

(End of definition for `\coffin_resize:Nnn`, `\coffin_gresize:Nnn`, and `__coffin_resize:NnnNN`. These functions are documented on page 317.)

`__coffin_resize_common:NnnN` The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

36143 \cs_new_protected:Npn \__coffin_resize_common:NnnN #1#2#3#4
36144 {
36145   \prop_set_eq:Nc \l__coffin_corners_prop
36146   { coffin ~ \__coffin_to_value:N #1 ~ corners }
36147   \prop_set_eq:Nc \l__coffin_poles_prop
36148   { coffin ~ \__coffin_to_value:N #1 ~ poles }
36149   \prop_map_inline:Nn \l__coffin_corners_prop
36150   { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
36151   \prop_map_inline:Nn \l__coffin_poles_prop
36152   { \__coffin_scale_pole:Nnnnn #1 {##1} ##2 }

```

Negative x -scaling values place the poles in the wrong location: this is corrected here.

```

36153 \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
36154 {
36155   \prop_map_inline:Nn \l__coffin_corners_prop
36156   { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
36157   \prop_map_inline:Nn \l__coffin_poles_prop
36158   { \__coffin_x_shift_pole:Nnnnn #1 {##1} ##2 }

```

```

36159     }
36160     #4 { coffin ~ \_coffin_to_value:N #1 ~ corners }
36161     \l__coffin_corners_prop
36162     #4 { coffin ~ \_coffin_to_value:N #1 ~ poles }
36163     \l__coffin_poles_prop
36164 }

```

(End of definition for _coffin_resize_common:NnnN.)

\coffin_scale:Nnn For scaling, the opposite calculation is done to find the new dimensions for the coffin.
\coffin_scale:cnn Only the total height is needed, as this is the shift required for corners and poles. The
\coffin_gscale:Nnn scaling is done the T_EX way as this works properly with floating point values without
\coffin_gscale:cnn needing to use the fp module.

```

\_coffin_scale:NnnNN
36165 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
36166 { \_coffin_scale:NnnNN #1 {#2} {#3} \box_scale:Nnn \prop_set_eq:cN }
36167 \cs_generate_variant:Nn \coffin_scale:Nnn { c }
36168 \cs_new_protected:Npn \coffin_gscale:Nnn #1#2#3
36169 { \_coffin_scale:NnnNN #1 {#2} {#3} \box_gscale:Nnn \prop_gset_eq:cN }
36170 \cs_generate_variant:Nn \coffin_gscale:Nnn { c }
36171 \cs_new_protected:Npn \_coffin_scale:NnnNN #1#2#3#4#5
36172 {
36173   \fp_set:Nn \l__coffin_scale_x_fp {#2}
36174   \fp_set:Nn \l__coffin_scale_y_fp {#3}
36175   #4 #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
36176   \dim_set:Nn \l__coffin_internal_dim
36177     { \coffin_ht:N #1 + \coffin_dp:N #1 }
36178   \dim_set:Nn \l__coffin_scaled_total_height_dim
36179     { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
36180   \dim_set:Nn \l__coffin_scaled_width_dim
36181     { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
36182   \_coffin_resize_common:NnnN #1
36183     { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
36184   #5
36185 }

```

(End of definition for \coffin_scale:Nnn, \coffin_gscale:Nnn, and _coffin_scale:NnnNN. These functions are documented on page 317.)

_coffin_scale_vector:nnNN This functions scales a vector from the origin using the pre-set scale factors in x and y . This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```

36186 \cs_new_protected:Npn \_coffin_scale_vector:nnNN #1#2#3#4
36187 {
36188   \dim_set:Nn #3
36189     { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
36190   \dim_set:Nn #4
36191     { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
36192 }

```

(End of definition for _coffin_scale_vector:nnNN.)

_coffin_scale_corner:Nnnn Scaling both corners and poles is a simple calculation using the preceding vector scaling.
_coffin_scale_pole:Nnnnnn

```

36193 \cs_new_protected:Npn \_coffin_scale_corner:Nnnn #1#2#3#4
36194 {

```



```

36195     \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
36196     \prop_put:Nne \l__coffin_corners_prop {#2}
36197     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
36198   }
36199 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
36200 {
36201     \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
36202     \prop_put:Nne \l__coffin_poles_prop {#2}
36203     {
36204         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
36205         {#5} {#6}
36206     }
36207 }

```

(End of definition for __coffin_scale_corner:Nnnn and __coffin_scale_pole:Nnnnnn.)

__coffin_x_shift_corner:Nnnn
 __coffin_x_shift_pole:Nnnnnn

These functions correct for the x displacement that takes place with a negative horizontal scaling.

```

36208 \cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
36209 {
36210     \prop_put:Nne \l__coffin_corners_prop {#2}
36211     {
36212         { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
36213     }
36214 }
36215 \cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
36216 {
36217     \prop_put:Nne \l__coffin_poles_prop {#2}
36218     {
36219         { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
36220         {#5} {#6}
36221     }
36222 }

```

(End of definition for __coffin_x_shift_corner:Nnnn and __coffin_x_shift_pole:Nnnnnn.)

92.7 Aligning and typesetting of coffins

\coffin_join:NnnNnnnn

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

\coffin_join:cnmNnnnn

\coffin_join:Nnncnnnn

\coffin_join:cnmNnnnn

\coffin_gjoin:NnnNnnnn

\coffin_gjoin:cnmNnnnn

\coffin_gjoin:Nnncnnnn

\coffin_gjoin:cnmNnnnn

__coffin_join:NnnNnnnnN

```

36223 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
36224 {
36225     \__coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
36226     \coffin_set_eq:NN
36227 }
36228 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
36229 \cs_new_protected:Npn \coffin_gjoin:NnnNnnnn #1#2#3#4#5#6#7#8
36230 {
36231     \__coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
36232     \coffin_gset_eq:NN
36233 }

```

```

36234 \cs_generate_variant:Nn \coffin_gjoin:NnnNnnnn { c , Nnnc , cnnc }
36235 \cs_new_protected:Npn \__coffin_join:NnnNnnnnN #1#2#3#4#5#6#7#8#9
36236 {
36237   \__coffin_align:NnnNnnnnN
36238   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin

```

Correct the placement of the reference point. If the x -offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the x -offset and the width of the second box. So a second kern may be needed.

```

36239   \hbox_set:Nn \l__coffin_aligned_coffin
36240   {
36241     \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
36242     { \__kernel_kern:n { -\l__coffin_offset_x_dim } }
36243     \hbox_unpack:N \l__coffin_aligned_coffin
36244     \dim_set:Nn \l__coffin_internal_dim
36245     { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
36246     \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
36247     { \__kernel_kern:n { -\l__coffin_internal_dim } }
36248   }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

36249   \__coffin_reset_structure:N \l__coffin_aligned_coffin
36250   \prop_clear:c
36251   {
36252     coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
36253     \c_space_tl corners
36254   }
36255   \__coffin_update_poles:N \l__coffin_aligned_coffin

```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```

36256   \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
36257   {
36258     \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
36259     \__coffin_offset_poles:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
36260     \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
36261     \__coffin_offset_corners:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
36262   }
36263   {
36264     \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
36265     \__coffin_offset_poles:Nnn #4
36266     { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
36267     \__coffin_offset_corners:Nnn #1 { Opt } { Opt }
36268     \__coffin_offset_corners:Nnn #4
36269     { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
36270   }
36271   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
36272   #9 #1 \l__coffin_aligned_coffin
36273 }

```

(End of definition for \coffin_join:NnnNnnnn, \coffin_gjoin:NnnNnnnn, and __coffin_join:NnnNnnnnN. These functions are documented on page 318.)

`\coffin_attach:NnnNnnnn`
`\coffin_attach:cnnNnnnn`
`\coffin_attach:Nnncnnnn`
`\coffin_attach:cnncnnnn`
`\coffin_gattach:NnnNnnnn`
`\coffin_gattach:cnnNnnnn`
`\coffin_gattach:Nnncnnnn`
`\coffin_gattach:cnncnnnn`
`__coffin_attach:NnnNnnnnN`
`__coffin_attach_mark:NnnNnnnn`

A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is hear also as it is similar but without the structure updates.

```

36274 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
36275 {
36276   \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
36277   \coffin_set_eq:NN
36278 }
36279 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
36280 \cs_new_protected:Npn \coffin_gattach:NnnNnnnn #1#2#3#4#5#6#7#8
36281 {
36282   \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
36283   \coffin_gset_eq:NN
36284 }
36285 \cs_generate_variant:Nn \coffin_gattach:NnnNnnnn { c , Nnnc , cnnc }
36286 \cs_new_protected:Npn \__coffin_attach:NnnNnnnnN #1#2#3#4#5#6#7#8#9
36287 {
36288   \__coffin_align:NnnNnnnnN
36289   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
36290   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
36291   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
36292   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
36293   \__coffin_reset_structure:N \l__coffin_aligned_coffin
36294   \prop_set_eq:cc
36295   {
36296     coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
36297     \c_space_tl corners
36298   }
36299   { coffin ~ \__coffin_to_value:N #1 ~ corners }
36300   \__coffin_update_poles:N \l__coffin_aligned_coffin
36301   \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
36302   \__coffin_offset_poles:Nnn #4
36303   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
36304   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
36305   #9 #1 \l__coffin_aligned_coffin
36306 }
36307 \cs_new_protected:Npn \__coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
36308 {
36309   \__coffin_align:NnnNnnnnN
36310   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
36311   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
36312   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
36313   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
36314   \box_set_eq:NN #1 \l__coffin_aligned_coffin
36315 }

```

(End of definition for \coffin_attach:NnnNnnnn and others. These functions are documented on page 318.)

`__coffin_align:NnnNnnnnN`

The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the ‘primed’ storage area to be used for the

second coffin. The ‘real’ box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.

```

36316 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
36317 {
36318   \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
36319   \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
36320   \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
36321   \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
36322   \dim_set:Nn \l__coffin_offset_x_dim
36323     { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
36324   \dim_set:Nn \l__coffin_offset_y_dim
36325     { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
36326   \hbox_set:Nn \l__coffin_aligned_internal_coffin
36327     {
36328     \box_use:N #1
36329     \__kernel_kern:n { -\box_wd:N #1 }
36330     \__kernel_kern:n { \l__coffin_offset_x_dim }
36331     \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
36332   }
36333   \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
36334 }

```

(End of definition for __coffin_align:NnnNnnnnN.)

```

\__coffin_offset_poles:Nnn
  \__coffin_offset_pole:Nnnnnnn

```

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping over the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a - means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that - should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```

36335 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
36336 {
36337   \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ poles }
36338     { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
36339 }
36340 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
36341 {
36342   \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
36343   \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
36344   \tl_if_in:nnTF {#2} { - }
36345     { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
36346     { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
36347   \exp_last_unbraced:NNo \__coffin_set_pole:Nnn \l__coffin_aligned_coffin
36348     { \l__coffin_internal_tl }
36349     {
36350     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
36351     {#5} {#6}
36352   }
36353 }

```

(End of definition for __coffin_offset_poles:Nnn and __coffin_offset_pole:Nnnnnnn.)

`_coffin_offset_corners:Nnn` Saving the offset corners of a coffin is very similar, except that there is no need to worry
`_coffin_offset_corner:Nnnnn` about naming: every corner can be saved here as order is unimportant.

```

36354 \cs_new_protected:Npn \_coffin_offset_corners:Nnn #1#2#3
36355 {
36356   \prop_map_inline:cn { coffin ~ \_coffin_to_value:N #1 ~ corners }
36357   { \_coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
36358 }
36359 \cs_new_protected:Npn \_coffin_offset_corner:Nnnnn #1#2#3#4#5#6
36360 {
36361   \prop_put:cne
36362   {
36363     coffin ~ \_coffin_to_value:N \l__coffin_aligned_coffin
36364     \c_space_tl corners
36365   }
36366   { #1 - #2 }
36367   {
36368     { \dim_eval:n { #3 + #5 } }
36369     { \dim_eval:n { #4 + #6 } }
36370   }
36371 }
  
```

(End of definition for `_coffin_offset_corners:Nnn` and `_coffin_offset_corner:Nnnnn`.)

`_coffin_update_vertical_poles:NNN` The T and B poles need to be recalculated after alignment. These functions find the
`_coffin_update_T:nnnnnnnnN` larger absolute value for the poles, but this is of course only logical when the poles are
`_coffin_update_B:nnnnnnnnN` horizontal.

```

36372 \cs_new_protected:Npn \_coffin_update_vertical_poles:NNN #1#2#3
36373 {
36374   \_coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
36375   \_coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
36376   \exp_last_two_unbraced:Noo \_coffin_update_T:nnnnnnnnN
36377   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
36378   \_coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
36379   \_coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
36380   \exp_last_two_unbraced:Noo \_coffin_update_B:nnnnnnnnN
36381   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
36382 }
36383 \cs_new_protected:Npn \_coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
36384 {
36385   \dim_compare:nNnTF {#2} < {#6}
36386   {
36387     \_coffin_set_pole:Nnn #9 { T }
36388     { { Opt } {#6} { 1000pt } { Opt } }
36389   }
36390   {
36391     \_coffin_set_pole:Nnn #9 { T }
36392     { { Opt } {#2} { 1000pt } { Opt } }
36393   }
36394 }
36395 \cs_new_protected:Npn \_coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
36396 {
36397   \dim_compare:nNnTF {#2} < {#6}
36398   {
36399     \_coffin_set_pole:Nnn #9 { B }
  
```

```

36400         { { Opt } {#2} { 1000pt } { Opt } }
36401     }
36402     {
36403         \__coffin_set_pole:Nnn #9 { B }
36404         { { Opt } {#6} { 1000pt } { Opt } }
36405     }
36406 }

```

(End of definition for `__coffin_update_vertical_poles:NNN`, `__coffin_update_T:nnnnnnnnN`, and `__coffin_update_B:nnnnnnnnN`.)

`\c__coffin_empty_coffin` An empty-but-horizontal coffin.

```

36407 \coffin_new:N \c__coffin_empty_coffin
36408 \tex_setbox:D \c__coffin_empty_coffin = \tex_hbox:D { }

```

(End of definition for `\c__coffin_empty_coffin`.)

`\coffin_typeset:Nnnnn` Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

`\coffin_typeset:cnnnn`

```

36409 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
36410 {
36411     \mode_leave_vertical:
36412     \__coffin_align:NnnNnnnnN \c__coffin_empty_coffin { H } { l }
36413     #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
36414     \box_use_drop:N \l__coffin_aligned_coffin
36415 }
36416 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

```

(End of definition for `\coffin_typeset:Nnnnn`. This function is documented on page 318.)

92.8 Coffin diagnostics

`\l__coffin_display_coffin` Used for printing coffins with data structures attached.

```

\l__coffin_display_coord_coffin 36417 \coffin_new:N \l__coffin_display_coffin
\l__coffin_display_pole_coffin 36418 \coffin_new:N \l__coffin_display_coord_coffin
36419 \coffin_new:N \l__coffin_display_pole_coffin

```

(End of definition for `\l__coffin_display_coffin`, `\l__coffin_display_coord_coffin`, and `\l__coffin_display_pole_coffin`.)

`\l__coffin_display_handles_prop` This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```

36420 \prop_new:N \l__coffin_display_handles_prop
36421 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
36422 { { b } { r } { -1 } { 1 } }
36423 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
36424 { { b } { hc } { 0 } { 1 } }
36425 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
36426 { { b } { l } { 1 } { 1 } }
36427 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
36428 { { vc } { r } { -1 } { 0 } }
36429 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
36430 { { vc } { hc } { 0 } { 0 } }

```

```

36431 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
36432   { { vc } { 1 } { 1 } { 0 } }
36433 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
36434   { { t } { r } { -1 } { -1 } }
36435 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
36436   { { t } { hc } { 0 } { -1 } }
36437 \prop_put:Nnn \l__coffin_display_handles_prop { br }
36438   { { t } { l } { 1 } { -1 } }
36439 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
36440   { { t } { r } { -1 } { -1 } }
36441 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
36442   { { t } { hc } { 0 } { -1 } }
36443 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
36444   { { t } { l } { 1 } { -1 } }
36445 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
36446   { { vc } { r } { -1 } { 1 } }
36447 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
36448   { { vc } { hc } { 0 } { 1 } }
36449 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
36450   { { vc } { l } { 1 } { 1 } }
36451 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
36452   { { b } { r } { -1 } { -1 } }
36453 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
36454   { { b } { hc } { 0 } { -1 } }
36455 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
36456   { { b } { l } { 1 } { -1 } }

```

(End of definition for \l__coffin_display_handles_prop.)

`\l__coffin_display_offset_dim` The standard offset for the label from the handle position when displaying handles.

```

36457 \dim_new:N \l__coffin_display_offset_dim
36458 \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

```

(End of definition for \l__coffin_display_offset_dim.)

`\l__coffin_display_x_dim` `\l__coffin_display_y_dim` As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```

36459 \dim_new:N \l__coffin_display_x_dim
36460 \dim_new:N \l__coffin_display_y_dim

```

(End of definition for \l__coffin_display_x_dim and \l__coffin_display_y_dim.)

`\l__coffin_display_poles_prop` A property list for printing poles: various things need to be deleted from this to get a “nice” output.

```

36461 \prop_new:N \l__coffin_display_poles_prop

```

(End of definition for \l__coffin_display_poles_prop.)

`\l__coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```

36462 \tl_new:N \l__coffin_display_font_tl
36463 \bool_lazy_and:nnT
36464   { \cs_if_exist_p:N \fmtname }
36465   { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
36466   {

```

```

36467 \tl_set:Nn \l__coffin_display_font_tl
36468 { \sffamily \tiny }
36469 }

```

(End of definition for \l__coffin_display_font_tl.)

__coffin_rule:nn Abstract out creation of rules here until there is a higher-level interface.

```

36470 \cs_new_protected:Npn \__coffin_rule:nn #1#2
36471 {
36472 \mode_leave_vertical:
36473 \hbox:n { \tex_vrule:D width #1 height #2 \scan_stop: }
36474 }

```

(End of definition for __coffin_rule:nn.)

\coffin_mark_handle:Nnnn Marking a single handle is relatively easy. The standard attachment function is used, **\coffin_mark_handle:cnnn** meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

```

36475 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
36476 {
36477 \hcoffin_set:Nn \l__coffin_display_pole_coffin
36478 {
36479 \color_select:n {#4}
36480 \__coffin_rule:nn { 1pt } { 1pt }
36481 }
36482 \__coffin_attach_mark:NnnNnnn #1 {#2} {#3}
36483 \l__coffin_display_pole_coffin { hc } { vc } { 0pt } { 0pt }
36484 \hcoffin_set:Nn \l__coffin_display_coord_coffin
36485 {
36486 \color_select:n {#4}
36487 \l__coffin_display_font_tl
36488 ( \tl_to_str:n { #2 , #3 } )
36489 }
36490 \prop_get:NnN \l__coffin_display_handles_prop
36491 { #2 #3 } \l__coffin_internal_tl
36492 \quark_if_no_value:NTF \l__coffin_internal_tl
36493 {
36494 \prop_get:NnN \l__coffin_display_handles_prop
36495 { #3 #2 } \l__coffin_internal_tl
36496 \quark_if_no_value:NTF \l__coffin_internal_tl
36497 {
36498 \__coffin_attach_mark:NnnNnnn #1 {#2} {#3}
36499 \l__coffin_display_coord_coffin { l } { vc }
36500 { 1pt } { 0pt }
36501 }
36502 {
36503 \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
36504 \l__coffin_internal_tl #1 {#2} {#3}
36505 }
36506 }
36507 {
36508 \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
36509 \l__coffin_internal_tl #1 {#2} {#3}

```



```

36510     }
36511   }
36512 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
36513   {
36514     \__coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
36515     \l__coffin_display_coord_coffin {#1} {#2}
36516     { #3 \l__coffin_display_offset_dim }
36517     { #4 \l__coffin_display_offset_dim }
36518   }
36519 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }

```

(End of definition for \coffin_mark_handle:Nnnn and __coffin_mark_handle_aux:nnnnNnn. This function is documented on page 319.)

\coffin_display_handles:Nn Printing the poles starts by removing any duplicates, for which the H poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```

\coffin_display_handles:cn
  \__coffin_display_handles_aux:nnnnnn
  \__coffin_display_handles_aux:nnnn
  \__coffin_display_attach:Nnnnn
36520 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
36521   {
36522     \hcoffin_set:Nn \l__coffin_display_pole_coffin
36523     {
36524       \color_select:n {#2}
36525       \__coffin_rule:mn { 1pt } { 1pt }
36526     }
36527     \prop_set_eq:Nc \l__coffin_display_poles_prop
36528     { coffin ~ \__coffin_to_value:N #1 ~ poles }
36529     \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
36530     \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
36531     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
36532     { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
36533     \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
36534     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
36535     { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
36536     \coffin_set_eq:NN \l__coffin_display_coffin #1
36537     \prop_map_inline:Nn \l__coffin_display_poles_prop
36538     {
36539       \prop_remove:Nn \l__coffin_display_poles_prop {##1}
36540       \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
36541     }
36542     \box_use_drop:N \l__coffin_display_coffin
36543   }

```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```

36544 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
36545   {
36546     \prop_map_inline:Nn \l__coffin_display_poles_prop
36547     {
36548       \bool_set_false:N \l__coffin_error_bool
36549       \__coffin_calculate_intersection:nnnnnnn {#2} {#3} {#4} {#5} ##2
36550       \bool_if:NF \l__coffin_error_bool
36551       {

```

```

36552 \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
36553 \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
36554 \__coffin_display_attach:Nnnnn
36555 \l__coffin_display_pole_coffin { hc } { vc }
36556 { Opt } { Opt }
36557 \hcoffin_set:Nn \l__coffin_display_coord_coffin
36558 {
36559 \color_select:n {#6}
36560 \l__coffin_display_font_tl
36561 ( \tl_to_str:n { #1 , ##1 } )
36562 }
36563 \prop_get:NnN \l__coffin_display_handles_prop
36564 { #1 ##1 } \l__coffin_internal_tl
36565 \quark_if_no_value:NTF \l__coffin_internal_tl
36566 {
36567 \prop_get:NnN \l__coffin_display_handles_prop
36568 { ##1 #1 } \l__coffin_internal_tl
36569 \quark_if_no_value:NTF \l__coffin_internal_tl
36570 {
36571 \__coffin_display_attach:Nnnnn
36572 \l__coffin_display_coord_coffin { l } { vc }
36573 { 1pt } { Opt }
36574 }
36575 {
36576 \exp_last_unbraced:No
36577 \__coffin_display_handles_aux:nnnn
36578 \l__coffin_internal_tl
36579 }
36580 }
36581 {
36582 \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
36583 \l__coffin_internal_tl
36584 }
36585 }
36586 }
36587 }
36588 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
36589 {
36590 \__coffin_display_attach:Nnnnn
36591 \l__coffin_display_coord_coffin {#1} {#2}
36592 { #3 \l__coffin_display_offset_dim }
36593 { #4 \l__coffin_display_offset_dim }
36594 }
36595 \cs_generate_variant:Nn \coffin_display_handles:Nn { c }

```

This is a dedicated version of `\coffin_attach:NnnNnnnn` with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.

```

36596 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
36597 {
36598 \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
36599 \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
36600 \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
36601 \dim_set:Nn \l__coffin_offset_x_dim

```

```

36602     { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
36603 \dim_set:Nn \l__coffin_offset_y_dim
36604     { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
36605 \hbox_set:Nn \l__coffin_aligned_coffin
36606     {
36607     \box_use:N \l__coffin_display_coffin
36608     \__kernel_kern:n { -\box_wd:N \l__coffin_display_coffin }
36609     \__kernel_kern:n { \l__coffin_offset_x_dim }
36610     \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
36611     }
36612 \box_set_ht:Nn \l__coffin_aligned_coffin
36613     { \box_ht:N \l__coffin_display_coffin }
36614 \box_set_dp:Nn \l__coffin_aligned_coffin
36615     { \box_dp:N \l__coffin_display_coffin }
36616 \box_set_wd:Nn \l__coffin_aligned_coffin
36617     { \box_wd:N \l__coffin_display_coffin }
36618 \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
36619 }

```

(End of definition for `\coffin_display_handles:Nn` and others. This function is documented on page 319.)

```

\coffin_show_structure:N
\coffin_show_structure:c
\coffin_log_structure:N
\coffin_log_structure:c
__coffin_show_structure:NN

```

For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```

36620 \cs_new_protected:Npn \coffin_show_structure:N
36621     { \__coffin_show_structure:NN \msg_show:nneeee }
36622 \cs_generate_variant:Nn \coffin_show_structure:N { c }
36623 \cs_new_protected:Npn \coffin_log_structure:N
36624     { \__coffin_show_structure:NN \msg_log:nneeee }
36625 \cs_generate_variant:Nn \coffin_log_structure:N { c }
36626 \cs_new_protected:Npn \__coffin_show_structure:NN #1#2
36627     {
36628     \__coffin_if_exist:NT #2
36629     {
36630     #1 { coffin } { show }
36631     { \token_to_str:N #2 }
36632     {
36633     \iow_newline: >~ ht ~~~ \dim_eval:n { \coffin_ht:N #2 }
36634     \iow_newline: >~ dp ~~~ \dim_eval:n { \coffin_dp:N #2 }
36635     \iow_newline: >~ wd ~~~ \dim_eval:n { \coffin_wd:N #2 }
36636     }
36637     {
36638     \prop_map_function:cN
36639     { coffin ~ \__coffin_to_value:N #2 ~ poles }
36640     \msg_show_item_unbraced:nn
36641     }
36642     { }
36643     }
36644     }

```

(End of definition for `\coffin_show_structure:N`, `\coffin_log_structure:N`, and `__coffin_show_structure:NN`. These functions are documented on page 319.)

```

\coffin_show:N
\coffin_show:c
\coffin_log:N
\coffin_log:c
\coffin_show:Nnn
\coffin_show:cnn
\coffin_log:Nnn
\coffin_log:cnn
__coffin_show:NNNnn

```

Essentially a combination of `\coffin_show_structure:N` and `\box_show:Nnn`, but we need to avoid having two prompts, so we use `\msg_term:nneeee` instead of `\msg_show:nneeee` in the show case.

```

36645 \cs_new_protected:Npn \coffin_show:N #1
36646   { \coffin_show:Nnn #1 \c_max_int \c_max_int }
36647 \cs_generate_variant:Nn \coffin_show:N { c }
36648 \cs_new_protected:Npn \coffin_log:N #1
36649   { \coffin_log:Nnn #1 \c_max_int \c_max_int }
36650 \cs_generate_variant:Nn \coffin_log:N { c }
36651 \cs_new_protected:Npn \coffin_show:Nnn
36652   { \__coffin_show:NNNnn \msg_term:nneeee \box_show:Nnn }
36653 \cs_generate_variant:Nn \coffin_show:Nnn { c }
36654 \cs_new_protected:Npn \coffin_log:Nnn
36655   { \__coffin_show:NNNnn \msg_log:nneeee \box_show:Nnn }
36656 \cs_generate_variant:Nn \coffin_log:Nnn { c }
36657 \cs_new_protected:Npn \__coffin_show:NNNnn #1#2#3#4#5
36658   {
36659     \__coffin_if_exist:NT #3
36660     {
36661       \__coffin_show_structure:NN #1 #3
36662       #2 #3 {#4} {#5}
36663     }
36664   }

```

(End of definition for `\coffin_show:N` and others. These functions are documented on page 319.)

92.9 Messages

```

36665 \msg_new:nnnn { coffin } { no-pole-intersection }
36666   { No~intersection~between~coffin~poles. }
36667   {
36668     LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
36669     but~they~do~not~have~a~unique~meeting~point:~
36670     the~value~(Opt,~Opt)~will~be~used.
36671   }
36672 \msg_new:nnnn { coffin } { unknown }
36673   { Unknown~coffin~'#1'. }
36674   { The~coffin~'#1'~was~never~defined. }
36675 \msg_new:nnnn { coffin } { unknown-pole }
36676   { Pole~'#1'~unknown~for~coffin~'#2'. }
36677   {
36678     LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
36679     but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
36680   }
36681 \msg_new:nnn { coffin } { show }
36682   {
36683     Size~of~coffin~#1 : #2 \\
36684     Poles~of~coffin~#1 : #3 .
36685   }
36686 </package>

```

Chapter 93

l3color implementation

```
36687 (*package)
```

```
36688 (@@=color)
```

93.1 Basics

`\l__color_current_tl` The color currently active for foreground (text, *etc.*) material. This is stored in the form of a color model followed by one or more values. There are four pre-defined models, three of which take numerical values in the range [0, 1]:

- `gray` $\langle gray \rangle$ Grayscale color with the $\langle gray \rangle$ value running from 0 (fully black) to 1 (fully white)
- `cmyk` $\langle cyan \rangle$ $\langle magenta \rangle$ $\langle yellow \rangle$ $\langle black \rangle$
- `rgb` $\langle red \rangle$ $\langle green \rangle$ $\langle blue \rangle$

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- `spot` $\langle name \rangle$ $\langle tint \rangle$ A pre-defined spot color, where the $\langle name \rangle$ should be a pre-defined string color name and the $\langle tint \rangle$ should be in the range [0, 1].

Additional models may be created to allow mixing of spot colors. The number of data entries these require will depend on the number of colors to be mixed.

TeXhackers note: The content of `\l__color_current_tl` comprises two brace groups, the first containing the color model and the second containing the value(s) applicable in that model.

(End of definition for \l__color_current_tl.)

`\color_group_begin:` Grouping for color is the same as using the basic `\group_begin:` and `\group_end:` functions. However, for semantic reasons, they are renamed here.

`\color_group_end:`

```
36689 \cs_new_eq:NN \color_group_begin: \group_begin:
```

```
36690 \cs_new_eq:NN \color_group_end: \group_end:
```

(End of definition for \color_group_begin: and \color_group_end:. These functions are documented on page 321.)

`\color_ensure_current:` A driver-independent wrapper for setting the foreground color to the current color “now”.

```
36691 \cs_new_protected:Npn \color_ensure_current:
36692   { \__color_select:N \l__color_current_tl }
```

(End of definition for `\color_ensure_current:`. This function is documented on page 321.)

`\s__color_stop` Internal scan marks.

```
36693 \scan_new:N \s__color_stop
```

(End of definition for `\s__color_stop`.)

`__color_select:N` Take an internal color specification and pass it to the driver. This code is needed to ensure the current color but will also be used by the higher-level material.

```
\__color_select_math:N
\__color_select:nn
36694 \cs_new_protected:Npn \__color_select:N #1
36695   {
36696     \exp_after:wN \__color_select:nn #1
36697     \group_insert_after:N \__color_backend_reset:
36698   }
36699 \cs_new_protected:Npn \__color_select_math:N #1
36700   { \exp_after:wN \__color_select:nn #1 }
36701 \cs_new_protected:Npn \__color_select:nn #1#2
36702   { \use:c { __color_backend_select_ #1 :n } {#2} }
```

(End of definition for `__color_select:N`, `__color_select_math:N`, and `__color_select:nn`.)

`\l__color_current_tl` The current color, with the model and

```
36703 \tl_new:N \l__color_current_tl
36704 \tl_set:Nn \l__color_current_tl { { gray } { 0 } }
```

(End of definition for `\l__color_current_tl`.)

93.2 Predefined color names

The ability to predefine colors with a name is a key part of this module and means there has to be a method for storing the results. At first sight, it seems natural to follow the usual `expl3` model and create a `color` variable type for the process. That would then allow both local and global colors, constant colors and the like. However, these names need to be accessible in some form at the user level, for selection of colors either simply by name or as part of a more complex expression. This does not require that the full name is exposed but does require that they can be looked up in a predictable way. As such, it is more useful to expose just the color names as part of the interface, with the result that only local color names can be created. (This is also seen for example in key creation in `l3keys`.) As a result, color names are declarative (no `new` functions).

Since there is no need to manipulate colors *en masse*, each is stored in a two-part structure: a `prop` for the colors themselves, and a `tl` for the default model for each color.

93.3 Setup

```
\l__color_internal_int
\l__color_internal_tl 36705 \int_new:N \l__color_internal_int
36706 \tl_new:N \l__color_internal_tl

(End of definition for \l__color_internal_int and \l__color_internal_tl.)
```

```
\s__color_mark Internal scan marks. \s__color_stop is already defined in l3color-base.
36707 \scan_new:N \s__color_mark

(End of definition for \s__color_mark.)
```

```
\l__color_ignore_error_bool Used to avoid issuing multiple errors if there is a change-of-model with input container
an error.
36708 \bool_new:N \l__color_ignore_error_bool

(End of definition for \l__color_ignore_error_bool.)
```

93.4 Utility functions

`\color_if_exist:p:n` A simple wrapper to avoid needing to have the lookup repeated in too many places. To guard against a color created in a group, we need to test for entries in the prop.

`\color_if_exist:nTF`

```
36709 \prg_new_conditional:Npnm \color_if_exist:n #1 { p , T, F, TF }
36710 {
36711   \prop_if_exist:cTF { l__color_named_ #1 _prop }
36712   {
36713     \prop_if_empty:cTF { l__color_named_ #1 _prop }
36714     \prg_return_false:
36715     \prg_return_true:
36716   }
36717   \prg_return_false:
36718 }
```

(End of definition for `\color_if_exist:nTF`. This function is documented on page 324.)

```
\__color_model:N Simple abstractions.
\__color_values:N 36719 \cs_new:Npn \__color_model:N #1 { \exp_after:wN \use_i:nn #1 }
36720 \cs_new:Npn \__color_values:N #1 { \exp_after:wN \use_ii:nn #1 }

(End of definition for \__color_model:N and \__color_values:N.)
```

```
\__color_extract:nNN Recover the values for the standard model for a color.
\__color_extract:VNN 36721 \cs_new_protected:Npn \__color_extract:nNN #1#2#3
36722 {
36723   \tl_set_eq:Nc #2 { l__color_named_ #1 _tl }
36724   \prop_get:cVN { l__color_named_ #1 _prop } #2 #3
36725 }
36726 \cs_generate_variant:Nn \__color_extract:nNN { V }
```

(End of definition for `__color_extract:nNN`.)

93.5 Model conversion

`__color_convert:nnN`
`__color_convert:VVN`
`__color_convert:nnnN`
`__color_convert:nVnN`
`__color_convert:nnVN`
`t_rgb_rgb:w_____`
`__color_convert_rgb_cmyk:w`
`__color_convert_rgb_cmyk:nnn`
`__color_convert_rgb_cmyk:nnnn`

Model conversion is carried out using standard formulae for base models, as described in the manual for *xcolor* (see also the *PostScript Language Reference Manual*). For other models direct conversion might not be defined, so we go through the fallback models if necessary.

```

36727 \cs_new_protected:Npn \__color_convert:nnN #1#2#3
36728   { \__color_convert:nnVN {#1} {#2} #3 #3 }
36729 \cs_generate_variant:Nn \__color_convert:nnN { VV }
36730 \cs_generate_variant:Nn \exp_last_unbraced:Nf { c }
36731 \cs_new_protected:Npn \__color_convert:nnnN #1#2#3#4
36732   {
36733     \tl_set:Nc #4
36734       {
36735         \cs_if_exist_use:cTF { __color_convert_ #1 _ #2 :w }
36736         { #3 \s__color_stop }
36737         {
36738           \cs_if_exist:cTF { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ #2 :
36739             {
36740               \exp_last_unbraced:cf
36741               { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ #2 :w }
36742               { \use:c { __color_convert_ #1 _ \use:c { c__color_fallback_ #1 _tl } :w }
36743                 \s__color_stop
36744             }
36745           }
36746           \exp_last_unbraced:cf
36747           { __color_convert_ \use:c { c__color_fallback_ #2 _tl } _ #2 :w }
36748           {
36749             \cs_if_exist_use:cTF { __color_convert_ #1 _ \use:c { c__color_fallback
36750               { #3 \s__color_stop }
36751               {
36752                 \exp_last_unbraced:cf
36753                 { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ \use:c
36754                   { \use:c { __color_convert_ #1 _ \use:c { c__color_fallback_ #1 _
36755                     \s__color_stop
36756                 }
36757               }
36758             \s__color_stop
36759           }
36760         }
36761       }
36762   }
36763 \cs_generate_variant:Nn \__color_convert:nnnN { nV , nnV }
36764 \cs_new:Npn \__color_convert_gray_gray:w #1 \s__color_stop
36765   { #1 }
36766 \cs_new:Npn \__color_convert_gray_rgb:w #1 \s__color_stop
36767   { #1 ~ #1 ~ #1 }
36768 \cs_new:Npn \__color_convert_gray_cmyk:w #1 \s__color_stop
36769   { 0 ~ 0 ~ 0 ~ \fp_eval:n { 1 - #1 } }

```

These rather odd values are based on NTSC television: the set are used for the cmyk conversion.

```

36770 \cs_new:Npn \__color_convert_rgb_gray:w #1 ~ #2 ~ #3 \s__color_stop
36771   { \fp_eval:n { 0.3 * #1 + 0.59 * #2 + 0.11 * #3 } }

```



```

36772 \cs_new:Npn \__color_convert_rgb_rgb:w #1 \s__color_stop
36773   { #1 }

```

The conversion from `rgb` to `cmk` is the most complex: a two-step procedure which requires *black generation* and *undercolor removal* functions. The PostScript reference describes them as device-dependent, but following `xcolor` we assume they are linear. Moreover, as the likelihood of anyone using a non-unitary matrix here is tiny, we simplify and treat those two concepts as no-ops. To allow code sharing with parsing of `cmk` values, we have an intermediate function here (`__color_convert_rgb_cmyk:nnn`) which actually takes `cmk` values as input.

```

36774 \cs_new:Npn \__color_convert_rgb_cmyk:w #1 ~ #2 ~ #3 \s__color_stop
36775   {
36776     \exp_args:Neee \__color_convert_rgb_cmyk:nnn
36777     { \fp_eval:n { 1 - #1 } }
36778     { \fp_eval:n { 1 - #2 } }
36779     { \fp_eval:n { 1 - #3 } }
36780   }
36781 \cs_new:Npn \__color_convert_rgb_cmyk:nnn #1#2#3
36782   {
36783     \exp_args:Ne \__color_convert_rgb_cmyk:nnnn
36784     { \fp_eval:n { min( #1, #2 , #3 ) } } {#1} {#2} {#3}
36785   }
36786 \cs_new:Npn \__color_convert_rgb_cmyk:nnnn #1#2#3#4
36787   {
36788     \fp_eval:n { min ( 1 , max ( 0 , #2 - #1 ) ) } \c_space_tl
36789     \fp_eval:n { min ( 1 , max ( 0 , #3 - #1 ) ) } \c_space_tl
36790     \fp_eval:n { min ( 1 , max ( 0 , #4 - #1 ) ) } \c_space_tl
36791     #1
36792   }
36793 \cs_new:Npn \__color_convert_cmyk_gray:w #1 ~ #2 ~ #3 ~ #4 \s__color_stop
36794   { \fp_eval:n { 1 - min ( 1 , 0.3 * #1 + 0.59 * #2 + 0.11 * #3 + #4 ) } }
36795 \cs_new:Npn \__color_convert_cmyk_rgb:w #1 ~ #2 ~ #3 ~ #4 \s__color_stop
36796   {
36797     \fp_eval:n { 1 - min ( 1 , #1 + #4 ) } \c_space_tl
36798     \fp_eval:n { 1 - min ( 1 , #2 + #4 ) } \c_space_tl
36799     \fp_eval:n { 1 - min ( 1 , #3 + #4 ) }
36800   }
36801 \cs_new:Npn \__color_convert_cmyk_cmyk:w #1 \s__color_stop
36802   { #1 }

```

(End of definition for `__color_convert:nnN` and others.)

93.6 Color expressions

```

\l__color_model_tl Working space to store the color data whilst doing calculations: keeping it on the stack
\l__color_value_tl is attractive but gets tricky (return is non-trivial).
\l__color_next_model_tl
\l__color_next_value_tl
36803 \tl_new:N \l__color_model_tl
36804 \tl_new:N \l__color_value_tl
36805 \tl_new:N \l__color_next_model_tl
36806 \tl_new:N \l__color_next_value_tl

```

(End of definition for `\l__color_model_tl` and others.)

```

    \_color_parse:nN
    \_color_parse_aux:nN
    \_color_parse_eq:Nn
    \_color_parse_eq:nNn
    \_color_parse:Nw
    \_color_parse_loop_init:Nnn
    \_color_parse_loop:w
    \_color_parse_loop_check:nn
    \_color_parse_loop:nn
    \_color_parse_gray:n
    \_color_parse_std:n
    \_color_parse_break:w
    \_color_parse_end:
    \_color_parse_mix:Nnnn
    \_color_parse_mix:NVn
    \_color_parse_mix:nNnn
    \_color_parse_mix_gray:nw
    \_color_parse_mix_rgb:nw
    \_color_parse_mix_cmyk:nw

```

The main function for parsing color expressions removes actives but otherwise expands, then starts working through the expression itself. At the end, we apply the payload.

```

36807 \cs_new_protected:Npe \_color_parse:nN #1#2
36808 {
36809   \tl_set:Ne \exp_not:c { l__color_named_ . _tl }
36810   { \exp_not:N \_color_model:N \exp_not:N \l__color_current_tl }
36811   \prop_put:NVe \exp_not:c { l__color_named_ . _prop }
36812   \exp_not:c { l__color_named_ . _tl }
36813   { \exp_not:N \_color_values:N \exp_not:N \l__color_current_tl }
36814   \exp_not:N \exp_args:Ne \exp_not:N \_color_parse_aux:nN
36815   { \exp_not:N \tl_to_str:n {#1} } #2
36816 }

```

Before going to all of the effort of parsing an expression, these two precursor functions look for a pre-defined name, either on its own or with a trailing ! (which is the same thing).

```

36817 \cs_new_protected:Npn \_color_parse_aux:nN #1#2
36818 {
36819   \color_if_exist:nTF {#1}
36820   { \_color_parse_set_eq:Nn #2 {#1} }
36821   { \_color_parse:Nw #2#1 ! \s__color_stop }
36822   \_color_check_model:N #2
36823 }
36824 \cs_new_protected:Npn \_color_parse_set_eq:Nn #1#2
36825 {
36826   \tl_if_empty:NTF \l_color_fixed_model_tl
36827   { \exp_args:Nv \_color_parse_set_eq:nNn { l__color_named_ #2 _tl } }
36828   { \exp_args:Nv \_color_parse_set_eq:nNn \l_color_fixed_model_tl }
36829   #1 {#2}
36830 }

```

Here, we have to allow for the case where there is a fixed model: that can't be swept up by generic conversion as we are dealing with a named color.

```

36831 \cs_new_protected:Npn \_color_parse_set_eq:nNn #1#2#3
36832 {
36833   \prop_get:cnNTF
36834   { l__color_named_ #3 _prop } {#1}
36835   \l__color_value_tl
36836   { \tl_set:Ne #2 { {#1} { \l__color_value_tl } } }
36837   {
36838     \tl_set_eq:Nc \l__color_model_tl { l__color_named_ #3 _tl }
36839     \prop_get:cVN { l__color_named_ #3 _prop } \l__color_model_tl
36840     \l__color_value_tl
36841     \_color_convert:nnN
36842     \l__color_model_tl {#1} \l__color_value_tl
36843     \tl_set:Ne #2
36844     {
36845       {#1}
36846       { \l__color_value_tl }
36847     }
36848   }
36849 }
36850 \cs_new_protected:Npn \_color_parse:Nw #1#2 ! #3 \s__color_stop
36851 {

```

```

36852 \color_if_exist:nTF {#2}
36853 {
36854   \tl_if_blank:nTF {#3}
36855     { \__color_parse_set_eq:Nn #1 {#2} }
36856     { \__color_parse_loop_init:Nnn #1 {#2} {#3} }
36857   }
36858   {
36859     \msg_error:nnn { color } { unknown-color } {#2}
36860     \tl_set:Nn \l__color_current_tl { { gray } { 0 } }
36861   }
36862 }

```

Once we establish that a full parse is needed, the next job is to get the detail of the first color. That will determine the model we use for the calculation: splitting here makes checking that a bit easier.

```

36863 \cs_new_protected:Npn \__color_parse_loop_init:Nnn #1#2#3
36864 {
36865   \group_begin:
36866     \__color_extract:nNN {#2} \l__color_model_tl \l__color_value_tl
36867     \__color_parse_loop:w #3 ! ! ! \s__color_stop
36868     \tl_set:Ne \l__color_internal_tl
36869       { { \l__color_model_tl } { \l__color_value_tl } }
36870   \exp_args:NNNV \group_end:
36871   \tl_set:Nn #1 \l__color_internal_tl
36872 }

```

This is the loop proper: there can be an open-ended set of colors to parse, separated by ! tokens. There are a few cases to look out for. At the end of the expression and with we find a mix of 100 then we simply skip the next color entirely (we can't stop the loop as there might be a further valid color to mix in). On the other hand, if we get a mix of 0 then drop everything so far and start again. There is also a trailing white to "read in" if the final explicit data is a mix. Those conditions are separate from actually looping, which is therefore sorted out by checking if we have further data to process: in contrast to xcolor, we don't allow !! so the test can be simplified.

```

36873 \cs_new_protected:Npn \__color_parse_loop:w #1 ! #2 ! #3 ! #4 ! #5 \s__color_stop
36874 {
36875   \tl_if_blank:nF {#1}
36876   {
36877     \bool_lazy_and:nnTF
36878       { \fp_compare_p:nNn {#1} > { 0 } }
36879       { \fp_compare_p:nNn {#1} < { 100 } }
36880     {
36881       \use:e
36882       {
36883         \__color_parse_loop:nn {#1}
36884         { \tl_if_blank:nTF {#2} { white } {#2} }
36885       }
36886     }
36887     { \__color_parse_loop_check:nn {#1} {#2} }
36888   }
36889   \tl_if_blank:nF {#3}
36890   { \__color_parse_loop:w #3 ! #4 ! #5 \s__color_stop }
36891   \__color_parse_end:
36892 }

```

As these are unusual cases, we accept slower performance here for clearer code: check for the error conditions, handle the boundary cases after that.

```

36893 \cs_new_protected:Npn \__color_parse_loop_check:nn #1#2
36894 {
36895   \bool_if:NF \l__color_ignore_error_bool
36896   {
36897     \bool_lazy_or:nnT
36898     { \fp_compare_p:nNn {#1} < { 0 } }
36899     { \fp_compare_p:nNn {#1} > { 100 } }
36900     { \msg_error:nnnnn { color } { out-of-range } {#1} { 0 } { 100 } }
36901   }
36902   \fp_compare:nNnF {#1} > \c_zero_fp
36903   {
36904     \tl_if_blank:nTF {#2}
36905     { \__color_extract:nNN { white } }
36906     { \__color_extract:nNN {#2} }
36907     \l__color_model_tl \l__color_value_tl
36908   }
36909 }

```

The “payload” of calculation in the loop first. If the model for the upcoming color is different from that of the existing (partial) color, convert the model. For `gray` the two are flipped round so that the outcome is something with “real” color. We are then in a position to do the actual calculation itself. The two auxiliaries here give us a way to break the loop should an invalid name be found.

```

36910 \cs_new_protected:Npn \__color_parse_loop:nn #1#2
36911 {
36912   \color_if_exist:nTF {#2}
36913   {
36914     \__color_extract:nNN {#2} \l__color_next_model_tl \l__color_next_value_tl
36915     \tl_if_eq:NnF \l__color_model_tl \l__color_next_model_tl
36916     {
36917       \str_if_eq:VnTF \l__color_model_tl { gray }
36918       { \__color_parse_gray:n {#2} }
36919       { \__color_parse_std:n {#2} }
36920     }
36921     \tl_set:Ne \l__color_value_tl
36922     {
36923       \__color_parse_mix:NVVn
36924       \l__color_model_tl \l__color_value_tl \l__color_next_value_tl {#1}
36925     }
36926   }
36927   {
36928     \msg_error:nnn { color } { unknown-color } {#2}
36929     \__color_extract:nNN { black } \l__color_model_tl \l__color_value_tl
36930     \__color_parse_break:w
36931   }
36932 }

```

The `gray` model needs special handling: the models need to be swapped: we do that using a dedicated function.

```

36933 \cs_new_protected:Npn \__color_parse_gray:n #1
36934 {
36935   \tl_set_eq:NN \l__color_model_tl \l__color_next_model_tl

```

```

36936 \tl_set:Nn \l__color_next_model_tl { gray }
36937 \exp_args:NnV \__color_convert:nnN { gray } \l__color_model_tl
36938 \l__color_value_tl
36939 \prop_get:cVN { l__color_named_ #1 _prop } \l__color_model_tl
36940 \l__color_next_value_tl
36941 }
36942 \cs_new_protected:Npn \__color_parse_std:n #1
36943 {
36944 \prop_get:cVNF { l__color_named_ #1 _prop }
36945 \l__color_model_tl
36946 \l__color_next_value_tl
36947 {
36948 \__color_convert:VVN
36949 \l__color_next_model_tl
36950 \l__color_model_tl
36951 \l__color_next_value_tl
36952 }
36953 }
36954 \cs_new_protected:Npn \__color_parse_break:w #1 \__color_parse_end: { }
36955 \cs_new_protected:Npn \__color_parse_end: { }

```

Do the vector arithmetic: mainly a question of shuffling input, along with one pre-calculation to keep down the use of division.

```

36956 \cs_new:Npn \__color_parse_mix:Nnnn #1#2#3#4
36957 {
36958 \exp_args:Nf \__color_parse_mix:nNnn
36959 { \fp_eval:n { #4 / 100 } }
36960 #1 {#2} {#3}
36961 }
36962 \cs_generate_variant:Nn \__color_parse_mix:Nnnn { NVV }
36963 \cs_new:Npn \__color_parse_mix:nNnn #1#2#3#4
36964 {
36965 \use:c { __color_parse_mix_ #2 :nw } {#1}
36966 #3 \s__color_mark #4 \s__color_stop
36967 }
36968 \cs_new:Npn \__color_parse_mix_gray:nw #1#2 \s__color_mark #3 \s__color_stop
36969 { \fp_eval:n { #2 * #1 + #3 * ( 1 - #1 ) } }
36970 \cs_new:Npn \__color_parse_mix_rgb:nw
36971 #1#2 ~ #3 ~ #4 \s__color_mark #5 ~ #6 ~ #7 \s__color_stop
36972 {
36973 \fp_eval:n { #2 * #1 + #5 * ( 1 - #1 ) } \c_space_tl
36974 \fp_eval:n { #3 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
36975 \fp_eval:n { #4 * #1 + #7 * ( 1 - #1 ) }
36976 }
36977 \cs_new:Npn \__color_parse_mix_cmyk:nw
36978 #1#2 ~ #3 ~ #4 ~ #5 \s__color_mark #6 ~ #7 ~ #8 ~ #9 \s__color_stop
36979 {
36980 \fp_eval:n { #2 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
36981 \fp_eval:n { #3 * #1 + #7 * ( 1 - #1 ) } \c_space_tl
36982 \fp_eval:n { #4 * #1 + #8 * ( 1 - #1 ) } \c_space_tl
36983 \fp_eval:n { #5 * #1 + #9 * ( 1 - #1 ) }
36984 }

```

(End of definition for __color_parse:nN and others.)

```

\__color_parse_model_gray:w Turn the input into internal form, also tidying up the number quickly.
\__color_parse_model_rgb:w 36985 \cs_new:Npn \__color_parse_model_gray:w #1 , #2 \s__color_stop
\__color_parse_model_cmyk:w 36986 { { gray } { \__color_parse_number:n {#1} } }
\__color_parse_number:n 36987 \cs_new:Npn \__color_parse_model_rgb:w #1 , #2 , #3 , #4 \s__color_stop
\__color_parse_number:w 36988 {
36989 { rgb }
36990 {
36991 \__color_parse_number:n {#1} ~
36992 \__color_parse_number:n {#2} ~
36993 \__color_parse_number:n {#3}
36994 }
36995 }
36996 \cs_new:Npn \__color_parse_model_cmyk:w #1 , #2 , #3 , #4 , #5 \s__color_stop
36997 {
36998 { cmyk }
36999 {
37000 \__color_parse_number:n {#1} ~
37001 \__color_parse_number:n {#2} ~
37002 \__color_parse_number:n {#3} ~
37003 \__color_parse_number:n {#4}
37004 }
37005 }
37006 \cs_new:Npn \__color_parse_number:n #1
37007 { \__color_parse_number:w #1 . 0 . \s__color_stop }
37008 \cs_new:Npn \__color_parse_number:w #1 . #2 . #3 \s__color_stop
37009 { \tl_if_blank:nTF {#1} { 0 } {#1} . #2 }

```

(End of definition for __color_parse_model_gray:w and others.)

```

\__color_parse_model_Gray:w
\__color_parse_model_hsb:w 37010 \cs_new:Npn \__color_parse_model_Gray:w #1 , #2 \s__color_stop
\__color_parse_model_Hsb:w 37011 { { gray } { \fp_eval:n { #1 / 15 } } }
\__color_parse_model_HSB:w 37012 \cs_new:Npn \__color_parse_model_hsb:w #1 , #2 , #3 , #4 \s__color_stop
\__color_parse_model_HTML:w 37013 { \__color_parse_model_hsb:nnn {#1} {#2} {#3} }
\__color_parse_model_RGB:w 37014 \cs_new:Npn \__color_parse_model_Hsb:w #1 , #2 , #3 , #4 \s__color_stop
\__color_parse_model_hsb:nnn 37015 {
\__color_parse_model_hsb_aux:nnn 37016 \exp_args:Ne \__color_parse_model_hsb:nnn { \fp_eval:n { #1 / 360 } }
\__color_parse_model_hsb:nnnn 37017 {#2} {#3}
\__color_parse_model_hsb:nnnnn 37018 }
\__color_parse_model_hsb_0:nnnn The conversion here is non-trivial but is described at length in the xcolor manual. For
\__color_parse_model_hsb_1:nnnn ease, we calculate the integer and fractional parts of the hue first, then use them to work
\__color_parse_model_hsb_2:nnnn out the possible values for r, g and b before putting them in the correct places.
\__color_parse_model_hsb_3:nnnn 37019 \cs_new:Npn \__color_parse_model_hsb:nnn #1#2#3
\__color_parse_model_hsb_4:nnnn 37020 {
\__color_parse_model_hsb_5:nnnn 37021 { rgb }
\__color_parse_model_wave:w 37022 {
\__color_parse_model_wave_auxi:nn 37023 \exp_args:Ne \__color_parse_model_hsb_aux:nnn
\__color_parse_model_wave_auxii:nn 37024 { \fp_eval:n { 6 * (#1) } } {#2} {#3}
\__color_parse_model_wave_rho:n 37025 }
37026 }
37027 \cs_new:Npn \__color_parse_model_hsb_aux:nnn #1#2#3
37028 {
37029 \exp_args:Nee \__color_parse_model_hsb_aux:nnnn

```

```

37030     { \fp_eval:n { floor(#1) } } { \fp_eval:n { #1 - floor(#1) } }
37031     {#2} {#3}
37032   }
37033 \cs_new:Npn \__color_parse_model_hsb_aux:nnnn #1#2#3#4
37034   {
37035     \use:e
37036     {
37037       \exp_not:N \__color_parse_model_hsb_aux:nnnnn
37038       { \__color_parse_number:n {#4} }
37039       { \fp_eval:n { round(#4 * (1 - #3), 5) } }
37040       { \fp_eval:n { round(#4 * (1 - #3 * #2), 5) } }
37041       { \fp_eval:n { round(#4 * (1 - #3 * (1 - #2)), 5) } }
37042       {#1}
37043     }
37044   }
37045 \cs_new:Npn \__color_parse_model_hsb_aux:nnnnn #1#2#3#4#5
37046   { \use:c { __color_parse_model_hsb_#5 :nnnn } {#1} {#2} {#3} {#4} }
37047 \cs_new:cpn { __color_parse_model_hsb_0:nnnn } #1#2#3#4 { #1 ~ #4 ~ #2 }
37048 \cs_new:cpn { __color_parse_model_hsb_1:nnnn } #1#2#3#4 { #3 ~ #1 ~ #2 }
37049 \cs_new:cpn { __color_parse_model_hsb_2:nnnn } #1#2#3#4 { #2 ~ #1 ~ #4 }
37050 \cs_new:cpn { __color_parse_model_hsb_3:nnnn } #1#2#3#4 { #2 ~ #3 ~ #1 }
37051 \cs_new:cpn { __color_parse_model_hsb_4:nnnn } #1#2#3#4 { #4 ~ #2 ~ #1 }
37052 \cs_new:cpn { __color_parse_model_hsb_5:nnnn } #1#2#3#4 { #1 ~ #2 ~ #3 }
37053 \cs_new:cpn { __color_parse_model_hsb_6:nnnn } #1#2#3#4 { #1 ~ #2 ~ #2 }
37054 \cs_new:Npn \__color_parse_model_HSB:w #1 , #2 , #3 , #4 \s__color_stop
37055   {
37056     \exp_args:Neee \__color_parse_model_hsb:nnn
37057     { \fp_eval:n { round((#1) / 240, 5) } }
37058     { \fp_eval:n { round((#2) / 240, 5) } }
37059     { \fp_eval:n { round((#3) / 240, 5) } }
37060   }
37061 \cs_new:Npn \__color_parse_model_HTML:w #1 , #2 \s__color_stop
37062   { \__color_parse_model_HTML_aux:w #1 0 0 0 0 0 0 \s__color_stop }
37063 \cs_new:Npn \__color_parse_model_HTML_aux:w #1#2#3#4#5#6#7 \s__color_stop
37064   {
37065     { rgb }
37066     {
37067       \fp_eval:n { round(\int_from_hex:n {#1#2} / 255, 5) } ~
37068       \fp_eval:n { round(\int_from_hex:n {#3#4} / 255, 5) } ~
37069       \fp_eval:n { round(\int_from_hex:n {#5#6} / 255, 5) }
37070     }
37071   }
37072 \cs_new:Npn \__color_parse_model_RGB:w #1 , #2 , #3 , #4 \s__color_stop
37073   {
37074     { rgb }
37075     {
37076       \fp_eval:n { round((#1) / 255, 5) } ~
37077       \fp_eval:n { round((#2) / 255, 5) } ~
37078       \fp_eval:n { round((#3) / 255, 5) }
37079     }
37080   }

```

Following the description in the xcolor manual. As we always use rgb, there is no need to find the sixth, we just pas the information straight to the hsb auxiliary defined earlier.

```

37081 \cs_new:Npn \__color_parse_model_wave:w #1 , #2 \s__color_stop
37082 {
37083   { rgb }
37084   {
37085     \fp_compare:nNnTF {#1} < { 420 }
37086     { \__color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 380) / 40 }
37087     }
37088     {
37089       \fp_compare:nNnTF {#1} > { 700 }
37090       { \__color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 780) / -80 } }
37091       { \__color_parse_model_wave_auxi:nn {#1} { 1 } }
37092     }
37093   }
37094 }
37095 \cs_new:Npn \__color_parse_model_wave_auxi:nn #1#2
37096 {
37097   \fp_compare:nNnTF {#1} < { 440 }
37098   {
37099     \__color_parse_model_wave_auxii:nn
37100     { 4 + \__color_parse_model_wave_rho:n { (#1 - 440) / -60 } }
37101     {#2}
37102   }
37103   {
37104     \fp_compare:nNnTF {#1} < { 490 }
37105     {
37106       \__color_parse_model_wave_auxii:nn
37107       { 4 - \__color_parse_model_wave_rho:n { (#1 - 440) / 50 } }
37108       {#2}
37109     }
37110     {
37111       \fp_compare:nNnTF {#1} < { 510 }
37112       {
37113         \__color_parse_model_wave_auxii:nn
37114         { 2 + \__color_parse_model_wave_rho:n { (#1 - 510) / -20 } }
37115         {#2}
37116       }
37117       {
37118         \fp_compare:nNnTF {#1} < { 580 }
37119         {
37120           \__color_parse_model_wave_auxii:nn
37121           { 2 - \__color_parse_model_wave_rho:n { (#1 - 510) / 70 } }
37122           {#2}
37123         }
37124         {
37125           \fp_compare:nNnTF {#1} < { 645 }
37126           {
37127             \__color_parse_model_wave_auxii:nn
37128             { \__color_parse_model_wave_rho:n { (#1 - 645) / -65 } }
37129             {#2}
37130           }
37131           { \__color_parse_model_wave_auxii:nn { 0 } {#2} }
37132         }
37133       }
37134     }

```



```

37135     }
37136   }
37137 \cs_new:Npn \__color_parse_model_wave_auxii:nn #1#2
37138   {
37139     \exp_args:Neee \__color_parse_model_hsb_aux:nnn
37140     { \fp_eval:n {#1} }
37141     { 1 }
37142     { \__color_parse_model_wave_rho:n {#2} }
37143   }
37144 \cs_new:Npn \__color_parse_model_wave_rho:n #1
37145   { \fp_eval:n { min(1, max(0,#1) ) } }

```

(End of definition for __color_parse_model_Gray:w and others.)

__color_parse_model_cmy:w Simply pass data to the conversion functions.

```

37146 \cs_new:Npn \__color_parse_model_cmy:w #1 , #2 , #3 , #4 \s__color_stop
37147   {
37148     { cmyk }
37149     { \__color_convert_rgb_cmyk:nnn {#1} {#2} {#3} }
37150   }

```

(End of definition for __color_parse_model_cmy:w.)

__color_parse_model_tHsb:w There are three stages to the process here: bring the tH argument into the normal range, divide through to get to hsb and finally convert that to rgb. The final stage can be delegated to the parsing function for hsb, and the conversion from Hsb to hsb is trivial, so the main focus here is the first stage. We use a simple expandable loop to do the work, and we implement the equation given in the xcolor manual (number 85 there) as a simple expression.

```

37151 \cs_new:Npn \__color_parse_model_tHsb:w #1 , #2 , #3 , #4 \s__color_stop
37152   {
37153     \exp_args:Ne \__color_parse_model_hsb:nnn
37154     { \__color_parse_model_tHsb:n {#1} } {#2} {#3}
37155   }
37156 \cs_new:Npn \__color_parse_model_tHsb:n #1
37157   {
37158     \__color_parse_model_tHsb:nw {#1}
37159     0 , 0 ;
37160     60 , 30 ;
37161     120 , 60 ;
37162     180 , 120 ;
37163     210 , 180 ;
37164     240 , 240 ;
37165     360 , 360 ;
37166     \q_recursion_tail , ;
37167     \q_recursion_stop
37168   }
37169 \cs_new:Npn \__color_parse_model_tHsb:nw #1 #2 , #3 ; #4 , #5 ;
37170   {
37171     \quark_if_recursion_tail_stop_do:nn {#4} { 0 }
37172     \fp_compare:nNnTF {#1} > {#4}
37173     { \__color_parse_model_tHsb:nw {#1} #4 , #5 ; }
37174     {
37175       \use_i_delimit_by_q_recursion_stop:nw

```

```

37176         { \fp_eval:n { ((#1 - #2) / (#4 - #2) * (#5 - #3) + #3) / 360 } }
37177     }
37178 }

```

(End of definition for `__color_parse_model_tHsb:w`, `__color_parse_model_tHsb:n`, and `__color_parse_model_tHsb:nw`.)

`__color_parse_model_&spot:w` We cannot extract data here from that passed by `xcolor`, so we fall back on a black tint.

```

37179 \cs_new:cpn { __color_parse_model_&spot:w } #1 , #2 \s__color_stop
37180   { { gray } { #1 } }

```

(End of definition for `__color_parse_model_&spot:w`.)

93.7 Selecting colors (and color models)

`\l_color_fixed_model_tl` For selecting a single fixed model.

```

37181 \tl_new:N \l_color_fixed_model_tl

```

(End of definition for `\l_color_fixed_model_tl`. This variable is documented on page 324.)

`__color_check_model:N` Check that the model in use is the one required.

```

\__color_check_model:nn
37182 \cs_new_protected:Npn \__color_check_model:N #1
37183   {
37184     \tl_if_empty:NF \l_color_fixed_model_tl
37185     {
37186       \exp_after:wN \__color_check_model:nn #1
37187       \tl_if_eq:NNF \l__color_model_tl \l_color_fixed_model_tl
37188       {
37189         \__color_convert:VWN \l__color_model_tl \l_color_fixed_model_tl
37190         \l__color_value_tl
37191       }
37192       \tl_set:Ne #1
37193       { { \l_color_fixed_model_tl } { \l__color_value_tl } }
37194     }
37195   }
37196 \cs_new_protected:Npn \__color_check_model:nn #1#2
37197   {
37198     \tl_set:Nn \l__color_model_tl {#1}
37199     \tl_set:Nn \l__color_value_tl {#2}
37200   }

```

(End of definition for `__color_check_model:N` and `__color_check_model:nn`.)

`__color_finalise_current:` A backend-neutral location for “last minute” manipulations before handing off to the backend code. We set the special `.` syntax here: this will therefore always be available. The finalisation is separate from the main function so it can also be applied to *e.g.* page color.

```

37201 \cs_new_protected:Npe \__color_finalise_current:
37202   {
37203     \tl_set:Ne \exp_not:c { l__color_named_ . _tl }
37204     { \exp_not:N \__color_model:N \exp_not:N \l__color_current_tl }
37205     \prop_clear:N \exp_not:c { l__color_named_ . _prop }
37206     \prop_put:Nve \exp_not:c { l__color_named_ . _prop }
37207     \exp_not:c { l__color_named_ . _tl }

```

```

37208     { \exp_not:N \__color_values:N \exp_not:N \l__color_current_tl }
37209   }

```

(End of definition for __color_finalise_current:.)

`\color_select:n` Parse the input expressions then get the backend to actually activate them. The main complexity here is the need to check through multiple models. That is done “locally” here as the approach is subtly different to when different models are being stored.

```

\__color_select_main:Nw
\__color_select_loop:Nw
\__color_select:nnN
\__color_select_swap:Nnn
37210 \cs_new_protected:Npn \color_select:n #1
37211 {
37212   \__color_parse:nN {#1} \l__color_current_tl
37213   \__color_finalise_current:
37214   \__color_select:N \l__color_current_tl
37215 }
37216 \cs_new_protected:Npn \color_select:nn #1#2
37217 {
37218   \__color_select_main:Nw \l__color_current_tl
37219   #1 // \s__color_mark #2 // \s__color_stop
37220   \__color_finalise_current:
37221   \__color_select:N \l__color_current_tl
37222 }

```

If the first color model is the fixed one, or if there is no fixed model, we don’t need most of the data: just set up and apply the backend function.

```

37223 \cs_new_protected:Npn \__color_select_main:Nw
37224 #1 #2 / #3 / #4 \s__color_mark #5 / #6 / #7 \s__color_stop
37225 {
37226   \__color_select:nnN {#2} {#5} #1
37227   \bool_lazy_or:nnF
37228     { \tl_if_empty_p:N \l_color_fixed_model_tl }
37229     { \str_if_eq_p:nV {#2} \l_color_fixed_model_tl }
37230     { \__color_select_loop:Nw #1 #3 / #4 \s__color_mark #6 / #7 \s__color_stop }
37231 }

```

If a fixed model applies, we need to check each possible value in order. If there is no hit at all, fall back on the generic formula-based interchange.

```

37232 \cs_new_protected:Npn \__color_select_loop:Nw
37233 #1 #2 / #3 \s__color_mark #4 / #5 \s__color_stop
37234 {
37235   \str_if_eq:nVTF {#2} \l_color_fixed_model_tl
37236     { \__color_select:nnN {#2} {#4} #1 }
37237     {
37238       \tl_if_blank:nTF {#2}
37239         { \exp_after:wN \__color_select_swap:Nnn \exp_after:wN #1 #1 }
37240         { \__color_select_loop:Nw #1 #3 \s__color_mark #5 \s__color_stop }
37241     }
37242 }
37243 \cs_new_protected:Npn \__color_select:nnN #1#2#3
37244 {
37245   \cs_if_exist:cTF { __color_parse_model_ #1 :w }
37246     {
37247       \tl_set:Ne #3
37248         { \use:c { __color_parse_model_ #1 :w } #2 , 0 , 0 , 0 , 0 \s__color_stop }
37249     }
37250     { \msg_error:nnn { color } { unknown-model } {#1} }

```

```

37251 }
37252 \cs_new_protected:Npn \__color_select_swap:Nnn #1#2#3
37253 {
37254   \__color_convert:nVnN {#2} \l_color_fixed_model_tl {#3} \l__color_value_tl
37255   \tl_set:Ne #1
37256   { { \l_color_fixed_model_tl } { \l__color_value_tl } }
37257 }

```

(End of definition for `\color_select:n` and others. These functions are documented on page 324.)

93.8 Math color

The approach here is the same as for the $\text{\LaTeX} 2_{\epsilon}$ `\mathcolor` command, but as we are working at the `expl3` level we can make some minor changes.

`\l_color_math_active_tl` Tokens representing active sub/superscripts.

```

37258 \tl_new:N \l_color_math_active_tl
37259 \tl_set:Nn \l_color_math_active_tl { ' }

```

(End of definition for `\l_color_math_active_tl`. This function is documented on page 325.)

`\g__color_math_seq` Not all engines have multiple color stacks, and at the same time we are not expecting breaking within a colored math fragment. So we track the color stack ourselves.

```

37260 \seq_new:N \g__color_math_seq

```

(End of definition for `\g__color_math_seq`.)

`\color_math:nn` The basic set up here is relatively simple: store the current color, parse the new color
`\color_math:nnn` as-normal, then switch color before inserting the tokens we are asked to change. The
`__color_math:nn` tricky part is right at the end, handling the reset.

```

37261 \cs_new_protected:Npn \color_math:nn #1#2
37262 {
37263   \__color_math:nn {#2}
37264   { \__color_parse:nN {#1} \l__color_current_tl }
37265 }
37266 \cs_new_protected:Npn \color_math:nnn #1#2#3
37267 {
37268   \__color_math:nn {#3}
37269   {
37270     \__color_select_main:Nw \l__color_current_tl
37271     #1 / / \s__color_mark #2 / / \s__color_stop
37272   }
37273 }
37274 \cs_new_protected:Npn \__color_math:nn #1#2
37275 {
37276   \seq_gpush:NV \g__color_math_seq \l__color_current_tl
37277   #2
37278   \__color_select_math:N \l__color_current_tl
37279   #1
37280   \__color_math_scan:w
37281 }

```

(End of definition for `\color_math:nn`, `\color_math:nnn`, and `__color_math:nn`. These functions are documented on page 325.)

`_color_math_scan:w` The complication when changing the color back is due to the fact that the `\color_`
`_color_math_scan_auxi:` `math:nn(n)` may be followed by `^` or `_` or the hidden superscript (for example `'`) and its
`_color_math_scan_auxii:` argument may end in a `\mathop` in which case the sub- and superscripts may be attached
`_color_math_scan_end:` as `\limits` instead of after the material. All cases need separate treatment. To avoid
repeatedly collecting the same token, we first check for an alignment tab: assuming we
don't have one of those, we can "recycle" `\l_peek_token` safely. As we have an explicit
`\c_alignment_token`, there needs to be an align-safe group present.

```

37282 \cs_new_protected:Npn \_color_math_scan:w
37283 {
37284   \peek_remove_filler:n
37285   {
37286     \group_align_safe_begin:
37287     \peek_catcode:NTF \c_alignment_token
37288     {
37289       \group_align_safe_end:
37290       \_color_math_scan_end:
37291     }
37292     {
37293       \group_align_safe_end:
37294       \_color_math_scan_auxi:
37295     }
37296   }
37297 }

```

Dealing with literal `_` and `^` is easy, and as we have exactly two cases, we can hard-code this. We use a hard-coded list for limits: these are all primitives. The `\use_none:n` here also removes the test token so it is left just in the right place.

```

37298 \cs_new_protected:Npn \_color_math_scan_auxi:
37299 {
37300   \token_case_catcode:NnTF \l_peek_token
37301   {
37302     \c_math_subscript_token { }
37303     \c_math_superscript_token { }
37304   }
37305   { \_color_math_scripts:Nw }
37306   {
37307     \token_case_meaning:NnTF \l_peek_token
37308     {
37309       \tex_limits:D { \tex_limits:D }
37310       \tex_nolimits:D { \tex_nolimits:D }
37311       \tex_displaylimits:D { \tex_displaylimits:D }
37312     }
37313     { \_color_math_scan:w \use_none:n }
37314     { \_color_math_scan_auxii: }
37315   }
37316 }

```

The one final case to handle is math-active tokens, most obviously `'`, as these won't be covered earlier.

```

37317 \cs_new_protected:Npn \_color_math_scan_auxii:
37318 {
37319   \tl_map_inline:Nn \l_color_math_active_tl
37320   {
37321     \token_if_eq_meaning:NNT \l_peek_token ##1

```

```

37322         {
37323             \tl_map_break:n
37324             {
37325                 \use_i:nn
37326                 { \_color_math_scan_auxiii:N ##1 }
37327             }
37328         }
37329     \_color_math_scan_end:
37330 }
37331 }
37332 \cs_new_protected:Npn \_color_math_scan_auxiii:N #1
37333 {
37334     \exp_after:wN \exp_after:wN \exp_after:wN \_color_math_scan:w
37335     \char_generate:nn { '#1 } { 13 }
37336 }
37337 \cs_new_protected:Npn \_color_math_scan_end:
37338 {
37339     \_color_backend_reset:
37340     \seq_gpop:NN \g\_color_math_seq \l\_color_current_tl
37341 }

```

(End of definition for `_color_math_scan:w` and others.)

```

\_color_math_scripts:Nw
\_color_math_script_aux:N

```

The tricky part of handling sub and superscripts is that we have to reset color to the one that is on the stack but reset it back to what it was before to allow for cases like

$$\left[\color{red} a + \sum_{i=1}^n \right]$$

Here, \TeX constructs a `\vbox` stacking subscript, summation sign, and superscript. So technically the superscript comes first and the `\sum` that should get colored red is the middle.

The approach here is to set up a brace group immediately after the script token, then to set the color appropriately in that argument. We need an extra group to keep the color contained, and as we need to allow for an explicit closing brace in the source, the inner group also is a brace one rather than `\group_begin:-`based. At the end of the outer group we need to insert `_color_math_scan:w` to continue the search for a second script token.

Notice that here we *don't* need to use the math-specific color selector as we can allow the `\group_insert_after:N \@_backend_reset:` to operate normally.

```

37342 \cs_new_protected:Npn \_color_math_scripts:Nw #1
37343 {
37344     #1
37345     \c_group_begin_token
37346     \c_group_begin_token
37347     \seq_get:NN \g\_color_math_seq \l\_color_current_tl
37348     \_color_select:N \l\_color_current_tl
37349     \group_insert_after:N \c_group_end_token
37350     \group_insert_after:N \_color_math_scan:w
37351     \peek_remove_filler:n
37352     {
37353         \peek_catcode_remove:NF \c_group_begin_token
37354         { \_color_math_script_aux:N }
37355     }
37356 }

```

Deal with the case where we do not have an explicit brace pair in the source.

```
37357 \cs_new_protected:Npn \__color_math_script_aux:N #1 { #1 \c_group_end_token }
```

(End of definition for `__color_math_scripts:Nw` and `__color_math_script_aux:N`.)

93.9 Fill and stroke color

```

\color_fill:n
\color_stroke:n 37358 \cs_new_protected:Npn \color_fill:n #1
\color_fill:nn 37359 {
\color_stroke:nn 37360 \__color_parse:nN {#1} \l__color_current_tl
\__color_draw:nnn 37361 \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
37362 }
37363 \cs_new_protected:Npn \color_stroke:n #1
37364 {
37365 \__color_parse:nN {#1} \l__color_current_tl
37366 \exp_after:wN \__color_draw:nnn \l__color_current_tl { stroke }
37367 }
37368 \cs_new_protected:Npn \color_fill:nn #1#2
37369 {
37370 \__color_select_main:Nw \l__color_current_tl
37371 #1 // \s__color_mark #2 // \s__color_stop
37372 \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
37373 }
37374 \cs_new_protected:Npn \color_stroke:nn #1#2
37375 {
37376 \__color_select_main:Nw \l__color_current_tl
37377 #1 // \s__color_mark #2 // \s__color_stop
37378 \exp_after:wN \__color_draw:nnn \l__color_current_tl { stroke }
37379 }
37380 \cs_new_protected:Npn \__color_draw:nnn #1#2#3
37381 {
37382 \use:c { __color_backend_ #3 _ #1 :n } {#2}
37383 \exp_args:Nc \group_insert_after:N { __color_backend_ #3 _ reset: }
37384 }

```

(End of definition for `\color_fill:n` and others. These functions are documented on page 325.)

93.10 Defining named colors

`\l__color_named_tl` Space to store the detail of the named color.

```
37385 \tl_new:N \l__color_named_tl
```

(End of definition for `\l__color_named_tl`.)

```

\color_set:nn Defining named colors means working through the model list and saving both the “main”
\__color_set:nnn color and any equivalents in other models. Even if there is only one model, we store a
\__color_set:nn prop as well as a tl, as there could be grouping weirdness, etc. When setting using an
\__color_set:nnw expression, we need to avoid any fixed model issues, which is done without a group as in
\color_set:nnn l3keys.
\__color_set_aux:nnn 37386 \cs_new_protected:Npn \color_set:nn #1#2
\__color_set_colon:nnw 37387 {
\__color_set_loop:nw
\color_set_eq:nn

```

```

37388 \exp_args:NV \_color_set:nnn
37389 \l_color_fixed_model_tl {#1} {#2}
37390 }
37391 \cs_new_protected:Npn \_color_set:nnn #1#2#3
37392 {
37393 \tl_clear:N \l_color_fixed_model_tl
37394 \_color_set:nn {#2} {#3}
37395 \tl_set:Nn \l_color_fixed_model_tl {#1}
37396 }
37397 \cs_new_protected:Npn \_color_set:nn #1#2
37398 {
37399 \str_if_eq:nnF {#1} { . }
37400 {
37401 \_color_parse:nN {#2} \l_color_named_tl
37402 \tl_clear_new:c { l_color_named_ #1 _tl }
37403 \tl_set:ce { l_color_named_ #1 _tl }
37404 { \_color_model:N \l_color_named_tl }
37405 \prop_clear_new:c { l_color_named_ #1 _prop }
37406 \prop_put:cve { l_color_named_ #1 _prop } { l_color_named_ #1 _tl }
37407 { \_color_values:N \l_color_named_tl }
37408 \_color_set:nnw {#1} {#2} #2 ! \s_color_stop
37409 }
37410 }

```

When setting an expression-based color, there could be multiple model data available for one or more of the input colors. Where that is true for the *first* named color in an expression, we re-parse the expression when they are also parameter-based: only **cm**yk, **gray** and **rgb** make any sense here. There is a bit of a performance hit but this should be rare and taking place during set-up.

```

37411 \cs_new_protected:Npn \_color_set:nnw #1#2#3 ! #4 \s_color_stop
37412 {
37413 \clist_map_inline:nn { cmyk , gray , rgb }
37414 {
37415 \prop_get:cnNT { l_color_named_ #3 _prop } {##1} \l_color_internal_tl
37416 {
37417 \prop_if_in:cnF { l_color_named_ #1 _prop } {##1}
37418 {
37419 \group_begin:
37420 \bool_set_true:N \l_color_ignore_error_bool
37421 \tl_set:cn { l_color_named_ #3 _tl } {##1}
37422 \_color_parse:nN {#2} \l_color_internal_tl
37423 \exp_args:NNNV \group_end:
37424 \tl_set:Nn \l_color_internal_tl \l_color_internal_tl
37425 \prop_put:cee { l_color_named_ #1 _prop }
37426 { \_color_model:N \l_color_internal_tl }
37427 { \_color_values:N \l_color_internal_tl }
37428 }
37429 }
37430 }
37431 }
37432 \cs_new_protected:Npn \color_set:nnn #1#2#3
37433 {
37434 \str_if_eq:nnF {#1} { . }
37435 {

```



```

37436     \tl_clear_new:c { l__color_named_ #1 _tl }
37437     \prop_clear_new:c { l__color_named_ #1 _prop }
37438     \exp_args:Ne \__color_set_aux:nnn { \tl_to_str:n {#2} }
37439     {#1} {#3}
37440   }
37441 }
37442 \cs_new_protected:Npe \__color_set_aux:nnn #1#2#3
37443 {
37444   \exp_not:N \__color_set_colon:nnw {#2} {#3}
37445   #1 \c_colon_str \c_colon_str \exp_not:N \s__color_stop
37446 }
37447 \use:e
37448 {
37449   \cs_new_protected:Npn \exp_not:N \__color_set_colon:nnw
37450   #1#2 #3 \c_colon_str #4 \c_colon_str
37451   #5 \exp_not:N \s__color_stop
37452 }
37453 {
37454   \tl_if_blank:nTF {#4}
37455   { \__color_set_loop:nw {#1} #3 }
37456   { \__color_set_loop:nw {#1} #4 }
37457   // \s__color_mark #2 // \s__color_stop
37458 }
37459 \cs_new_protected:Npn \__color_set_loop:nw
37460 #1#2 / #3 \s__color_mark #4 / #5 \s__color_stop
37461 {
37462   \tl_if_blank:nF {#2}
37463   {
37464     \__color_select:nnN {#2} {#4} \l__color_named_tl
37465     \tl_set:Ne \l__color_internal_tl { \__color_model:N \l__color_named_tl }
37466     \tl_if_empty:cT { l__color_named_ #1 _tl }
37467     { \tl_set_eq:cN { l__color_named_ #1 _tl } \l__color_internal_tl }
37468     \prop_put:cVe { l__color_named_ #1 _prop } \l__color_internal_tl
37469     { \__color_values:N \l__color_named_tl }
37470     \__color_set_loop:nw {#1} #3 \s__color_mark #5 \s__color_stop
37471   }
37472 }
37473 \cs_new_protected:Npn \color_set_eq:nn #1#2
37474 {
37475   \color_if_exist:nTF {#2}
37476   {
37477     \tl_clear_new:c { l__color_named_ #1 _tl }
37478     \prop_clear_new:c { l__color_named_ #1 _prop }
37479     \str_if_eq:nnTF {#2} { . }
37480     {
37481       \tl_set:ce { l__color_named_ #1 _tl }
37482       { \__color_model:N \l__color_current_tl }
37483       \prop_put:cve { l__color_named_ #1 _prop } { l__color_named_ #1 _tl }
37484       { \__color_values:N \l__color_current_tl }
37485     }
37486     {
37487       \tl_set_eq:cc { l__color_named_ #1 _tl } { l__color_named_ #2 _tl }
37488       \prop_set_eq:cc { l__color_named_ #1 _prop } { l__color_named_ #2 _prop }
37489     }

```

```

37490     }
37491     {
37492     \msg_error:nnn { color } { unknown-color } {#2}
37493     }
37494 }

```

(End of definition for `\color_set:nn` and others. These functions are documented on page 324.)

A small set of colors are always defined.

```

37495 \color_set:nnn { black } { gray } { 0 }
37496 \color_set:nnn { white } { gray } { 1 }
37497 \color_set:nnn { cyan } { cmyk } { 1 , 0 , 0 , 0 }
37498 \color_set:nnn { magenta } { cmyk } { 0 , 1 , 0 , 0 }
37499 \color_set:nnn { yellow } { cmyk } { 0 , 0 , 1 , 0 }
37500 \color_set:nnn { red } { rgb } { 1 , 0 , 0 }
37501 \color_set:nnn { green } { rgb } { 0 , 1 , 0 }
37502 \color_set:nnn { blue } { rgb } { 0 , 0 , 1 }

```

`\l__color_named_._prop`
`\l__color_named_._tl`

A special named color: this is always defined though not fixed in definition.

```

37503 \prop_new:c { l__color_named_._prop }
37504 \tl_new:c { l__color_named_._tl }
37505 \tl_set:ce { l__color_named_._tl } { \__color_model:N \l__color_current_tl }

```

(End of definition for `\l__color_named_._prop` and `\l__color_named_._tl`.)

93.11 Exporting colors

`\color_export:nnN`
`\color_export:nnnN`
`__color_export:nN`
`__color_export:nnnN`

```

37506 \cs_new_protected:Npn \color_export:nnN #1#2#3
37507 {
37508   \group_begin:
37509   \tl_if_exist:cT { c__color_export_ #2 _tl }
37510   { \tl_set_eq:Nc \l_color_fixed_model_tl { c__color_export_ #2 _tl } }
37511   \__color_parse:nN {#1} #3
37512   \__color_export:nN {#2} #3
37513   \exp_args:NNNV \group_end:
37514   \tl_set:Nn #3 #3
37515 }
37516 \cs_new_protected:Npn \color_export:nnnN #1#2#3#4
37517 {
37518   \__color_select_main:Nw #4
37519   #1 // \s__color_mark #2 // \s__color_stop
37520   \__color_export:nN {#3} #4
37521 }
37522 \cs_new_protected:Npn \__color_export:nN #1#2
37523 { \exp_after:wN \__color_export:nnnN #2 {#1} #2 }
37524 \cs_new:Npn \__color_export:nnnN #1#2#3#4
37525 {
37526   \cs_if_exist_use:cF { __color_export_format_ #3 :nnN }
37527   {
37528     \msg_error:nnn { color } { unknown-export-format } {#3}
37529     \use_none:nnn
37530   }
37531   {#1} {#2} #4
37532 }

```

(End of definition for `\color_export:nnN` and others. These functions are documented on page 326.)

`__color_export_format_backend:nnN`

Simple.

```
37533 \cs_new_protected:Npn \__color_export_format_backend:nnN #1#2#3
37534   { \tl_set:Nn #3 { {#1} {#2} } }
```

(End of definition for `__color_export_format_backend:nnN`.)

`__color_export:nnnNN`

A generic auxiliary for cases where only one model is appropriate.

```
37535 \cs_new_protected:Npn \__color_export:nnnNN #1#2#3#4#5
37536   {
37537     \str_if_eq:nnTF {#2} {#1}
37538       { #5 #4 #3 \s__color_stop }
37539     {
37540       \__color_convert:nnnN {#2} {#1} {#3} #4
37541       \exp_after:wN #5 \exp_after:wN #4
37542       #4 \s__color_stop
37543     }
37544   }
```

(End of definition for `__color_export:nnnNN`.)

`\c__color_export_comma-sep-cmyk_tl`

`\c__color_export_comma-sep-rgb_tl`

`\c__color_export_HTML_tl`

`\c__color_export_space-sep-cmyk_tl`

`\c__color_export_space-sep-rgb_tl`

```
37545 \tl_const:cn { c__color_export_comma-sep-cmyk_tl } { cmyk }
37546 \tl_const:cn { c__color_export_comma-sep-rgb_tl } { rgb }
37547 \tl_const:Nn \c__color_export_HTML_tl { rgb }
37548 \tl_const:cn { c__color_export_space-sep-cmyk_tl } { cmyk }
37549 \tl_const:cn { c__color_export_space-sep-rgb_tl } { rgb }
```

(End of definition for `\c__color_export_comma-sep-cmyk_tl` and others.)

`__color_export_format_comma-sep-cmyk:nnN`

`__color_export_format_comma-sep-rgb:nnN`

`__color_export_format_space-sep-cmyk:nnN`

`__color_export_format_space-sep-rgb:nnN`

```
37550 \group_begin:
37551   \cs_set_protected:Npn \__color_tmp:w #1#2
37552     {
37553       \cs_new_protected:cpe { __color_export_format_ #1 :nnN } ##1##2##3
37554       {
37555         \exp_not:N \__color_export:nnnNN {#2} {##1} {##2} ##3
37556         \exp_not:c { __color_export_ #1 :Nw }
37557       }
37558     }
37559   \__color_tmp:w { comma-sep-cmyk } { cmyk }
37560   \__color_tmp:w { comma-sep-rgb } { rgb }
37561   \__color_tmp:w { HTML } { rgb }
37562   \__color_tmp:w { space-sep-cmyk } { cmyk }
37563   \__color_tmp:w { space-sep-rgb } { rgb }
37564
37565 \group_end:
```

(End of definition for `__color_export_format_comma-sep-cmyk:nnN` and others.)

```

\__color_export_space-sep-cmyk:Nw
\__color_export_comma-sep-cmyk:Nw
37566 \cs_new_protected:cpn { __color_export_comma-sep-cmyk:Nw }
37567   #1#2 ~ #3 ~ #4 ~ #5 \s__color_stop
37568   { \tl_set:Nn #1 { #2 , #3 , #4 , #5 } }
37569 \cs_new_protected:cpn { __color_export_space-sep-cmyk:Nw } #1#2 \s__color_stop
37570   { \tl_set:Nn #1 {#2} }

```

(End of definition for __color_export_space-sep-cmyk:Nw and __color_export_comma-sep-cmyk:Nw.)

```

\__color_export_comma-sep-rgb:Nw HTML values must be given in rgb: we force conversion if required, then do some simple
\__color_export_HTML:Nw maths.
\__color_export_space-sep-rgb:Nw
\__color_export_HTML:n
37571 \cs_new_protected:cpn { __color_export_comma-sep-rgb:Nw } #1#2 ~ #3 ~ #4 \s__color_stop
37572   { \tl_set:Ne #1 { #2 , #3 , #4 } }
37573 \cs_new_protected:Npn \__color_export_HTML:Nw #1#2 ~ #3 ~ #4 \s__color_stop
37574   {
37575     \tl_set:Ne #1
37576       {
37577         \__color_export_HTML:n {#2}
37578         \__color_export_HTML:n {#3}
37579         \__color_export_HTML:n {#4}
37580       }
37581   }
37582 \cs_new:Npn \__color_export_HTML:n #1
37583   {
37584     \fp_compare:nNnTF {#1} = { 0 }
37585       { 00 }
37586       {
37587         \fp_compare:nNnT { #1 * 255 } < { 16 } { 0 }
37588         \int_to_Hex:n { \fp_to_int:n { #1 * 255 } }
37589       }
37590   }
37591 \cs_new_protected:cpn { __color_export_space-sep-rgb:Nw } #1#2 \s__color_stop
37592   { \tl_set:Nn #1 {#2} }

```

(End of definition for __color_export_comma-sep-rgb:Nw and others.)

93.12 Additional color models

```

\l__color_internal_prop
37593 \prop_new:N \l__color_internal_prop

```

(End of definition for \l__color_internal_prop.)

\g__color_model_int A tracker for the total number of new models.

```

37594 \int_new:N \g__color_model_int

```

(End of definition for \g__color_model_int.)

\c__color_fallback_cmyk_tl For every colorspace, we define one of the base colorspace as a fallback. The base colorspace themselves are their own fallback.

```

\c__color_fallback_gray_tl
\c__color_fallback_rgb_tl
37595 \tl_const:Nn \c__color_fallback_cmyk_tl { cmyk }
37596 \tl_const:Nn \c__color_fallback_gray_tl { gray }
37597 \tl_const:Nn \c__color_fallback_rgb_tl { rgb }

```

(End of definition for `\c__color_fallback_cmyk_tl`, `\c__color_fallback_gray_tl`, and `\c__color_fallback_rgb_tl`.)

`\g__color_colorants_prop` Mapping from names to colorants.

```
37598 \prop_new:N \g__color_colorants_prop
37599 \prop_gput:Nnn \g__color_colorants_prop { black } { Black }
37600 \prop_gput:Nnn \g__color_colorants_prop { blue } { Blue }
37601 \prop_gput:Nnn \g__color_colorants_prop { cyan } { Cyan }
37602 \prop_gput:Nnn \g__color_colorants_prop { green } { Green }
37603 \prop_gput:Nnn \g__color_colorants_prop { magenta } { Magenta }
37604 \prop_gput:Nnn \g__color_colorants_prop { none } { None }
37605 \prop_gput:Nnn \g__color_colorants_prop { red } { Red }
37606 \prop_gput:Nnn \g__color_colorants_prop { yellow } { Yellow }
```

(End of definition for `\g__color_colorants_prop`.)

`\c__color_model_whitepoint_CIELAB_a_tl` Whitepoint data for the CIELAB profiles.

```
\c__color_model_whitepoint_CIELAB_b_tl 37607 \tl_const:Nn \c__color_model_whitepoint_CIELAB_a_tl { 1.0985 ~ 1 ~ 0.3558 }
\c__color_model_whitepoint_CIELAB_e_tl 37608 \tl_const:Nn \c__color_model_whitepoint_CIELAB_b_tl { 0.9807 ~ 1 ~ 1.1822 }
\c__color_model_whitepoint_CIELAB_d50_tl 37609 \tl_const:Nn \c__color_model_whitepoint_CIELAB_e_tl { 1 ~ 1 ~ 1 }
\c__color_model_whitepoint_CIELAB_d55_tl 37610 \tl_const:cn { c__color_model_whitepoint_CIELAB_d50_tl } { 0.9642 ~ 1 ~ 0.8251 }
\c__color_model_whitepoint_CIELAB_d65_tl 37611 \tl_const:cn { c__color_model_whitepoint_CIELAB_d55_tl } { 0.9568 ~ 1 ~ 0.9214 }
\c__color_model_whitepoint_CIELAB_d75_tl 37612 \tl_const:cn { c__color_model_whitepoint_CIELAB_d65_tl } { 0.9504 ~ 1 ~ 1.0888 }
37613 \tl_const:cn { c__color_model_whitepoint_CIELAB_d75_tl } { 0.9497 ~ 1 ~ 1.2261 }
```

(End of definition for `\c__color_model_whitepoint_CIELAB_a_tl` and others.)

`\c__color_model_range_CIELAB_tl` The range for CIELAB color spaces.

```
37614 \tl_const:Nn \c__color_model_range_CIELAB_tl { 0 ~ 100 ~ -128 ~ 127 ~ -128 ~ 127 }
```

(End of definition for `\c__color_model_range_CIELAB_tl`.)

`\g__color_alternative_model_prop` For tracking the alternative model set up for separations, etc.

```
37615 \prop_new:N \g__color_alternative_model_prop
37616 \clist_map_inline:nn { cyan , magenta , yellow , black }
37617 { \prop_gput:Nnn \g__color_alternative_model_prop {#1} { cmyk } }
37618 \clist_map_inline:nn { red , green , blue }
37619 { \prop_gput:Nnn \g__color_alternative_model_prop {#1} { rgb } }
```

(End of definition for `\g__color_alternative_model_prop`.)

`\g__color_alternative_values_prop` Same for the values: a bit more involved.

```
37620 \prop_new:N \g__color_alternative_values_prop
37621 \prop_gput:Nnn \g__color_alternative_values_prop { cyan } { 1 , 0 , 0 , 0 }
37622 \prop_gput:Nnn \g__color_alternative_values_prop { magenta } { 0 , 1 , 0 , 0 }
37623 \prop_gput:Nnn \g__color_alternative_values_prop { yellow } { 0 , 0 , 1 , 0 }
37624 \prop_gput:Nnn \g__color_alternative_values_prop { black } { 0 , 0 , 0 , 1 }
37625 \prop_gput:Nnn \g__color_alternative_values_prop { red } { 1 , 0 , 0 }
37626 \prop_gput:Nnn \g__color_alternative_values_prop { green } { 0 , 1 , 0 }
37627 \prop_gput:Nnn \g__color_alternative_values_prop { blue } { 0 , 0 , 1 }
```

(End of definition for `\g__color_alternative_values_prop`.)

`\color_model_new:nnn` Set up a new model: in general this has to be handled by a family-dependent function.
`__color_model_new:nnn` To avoid some “interesting” questions with casing, we fold the case of the family name. The key–value list should always be present, so we convert it up-front to a `prop`, then deal with the detail on a per-family basis.

```

37628 \cs_new_protected:Npn \color_model_new:nnn #1#2#3
37629 {
37630   \exp_args:Nee \__color_model_new:nnn
37631   { \tl_to_str:n {#1} }
37632   { \str_casefold:n {#2} } {#3}
37633 }
37634 \cs_new_protected:Npn \__color_model_new:nnn #1#2#3
37635 {
37636   \cs_if_exist:cTF { __color_parse_model_ #1 :w }
37637   {
37638     \msg_error:nnn { color } { model-already-defined } {#1}
37639   }
37640   {
37641     \cs_if_exist:cTF { __color_model_ #2 :n }
37642     {
37643       \prop_set_from_keyval:Nn \l__color_internal_prop {#3}
37644       \use:c { __color_model_ #2 :n } {#1}
37645     }
37646     {
37647       \msg_error:nnn { color } { unknown-model-type } {#2}
37648     }
37649   }
37650 }

```

(End of definition for `\color_model_new:nnn` and `__color_model_new:nnn`. This function is documented on page 327.)

`__color_model_init:nnn` A shared auxiliary to do the basics of setting up a new model: reserve a number, create
`__color_model_init:nne` a white-equivalent, set up links to the backend.

```

37651 \cs_new_protected:Npn \__color_model_init:nnn #1#2#3
37652 {
37653   \int_gincr:N \g__color_model_int
37654   \clist_map_inline:nn { fill , stroke , select }
37655   {
37656     \cs_new_protected:cpe { __color_backend_ ##1 _ #1 :n } ####1
37657     {
37658       \exp_not:c { __color_backend_ ##1 _ #2 :nn }
37659       { color \int_use:N \g__color_model_int } {####1}
37660     }
37661   }
37662   \cs_new_protected:cpe { __color_model_ #1 _white: }
37663   {
37664     \prop_put:Nnn \exp_not:N \l__color_named_white_prop {#1}
37665     { \exp_not:n {#3} }
37666     \exp_not:N \int_compare:nNnF { \tex_currentgrouplevel:D } = 0
37667     { \group_insert_after:N \exp_not:c { __color_model_ #1 _white: } }
37668   }
37669   \use:c { __color_model_ #1 _white: }
37670 }
37671 \cs_generate_variant:Nn \__color_model_init:nnn { nne }

```

(End of definition for `_color_model_init:nnn`.)

Separations must have a “real” name, which is pretty easy to find.

```
\_color_model_separation:n Separations must have a “real” name, which is pretty easy to find.
\_color_model_separation:nn 37672 \cs_new_protected:Npn \_color_model_separation:n #1
    \_color_model_separation:nnn 37673 {
\_color_model_separation:w 37674   \prop_get:NnNTF \l__color_internal_prop { name }
    \_color_model_separation_cmyk:nnnnnn 37675   \l__color_internal_tl
    \_color_model_separation_gray:nnnnnn 37676   {
    \_color_model_separation_rgb:nnnnnn 37677     \exp_args:NV \_color_model_separation:nn
    \_color_model_convert:nnn 37678     \l__color_internal_tl {#1}
    \_color_model_separation_CIELAB:nnnnnn 37679   }
    \_color_model_separation_CIELAB:nnnnnn 37680   {
    37681     \msg_error:nnn { color }
    37682     { separation-requires-name } {#1}
    37683   }
    37684 }
```

We have two keys to find at this stage: the alternative space model and linked values.

```
37685 \cs_new_protected:Npn \_color_model_separation:nn #1#2
37686 {
37687   \prop_get:NnNTF \l__color_internal_prop { alternative-model }
37688   \l__color_internal_tl
37689   {
37690     \exp_args:NV \_color_model_separation:nnn
37691     \l__color_internal_tl {#2} {#1}
37692   }
37693   {
37694     \msg_error:nnn { color }
37695     { separation-alternative-model } {#2}
37696   }
37697 }
37698 \cs_new_protected:Npn \_color_model_separation:nnn #1#2#3
37699 {
37700   \cs_if_exist:cTF { __color_model_separation_ #1 :nnnnnn }
37701   {
37702     \prop_get:NnNTF \l__color_internal_prop { alternative-values }
37703     \l__color_internal_tl
37704     {
37705       \exp_after:wN \_color_model_separation:w \l__color_internal_tl
37706       , 0 , 0 , 0 , 0 \s__color_stop {#2} {#3} {#1}
37707     }
37708     {
37709       \msg_error:nnn { color }
37710       { separation-alternative-values } {#2}
37711     }
37712   }
37713   {
37714     \msg_error:nnn { color }
37715     { unknown-alternative-model } {#1}
37716   }
37717 }
```

As each alternative space leads to a different requirement for conversion, and as there are only a small number of choices, we manually split the data and then set up. Notice that mixing tints is really just the same as mixing gray. The `white` color is special, as it

allows tints to be adjusted without an additional color space. To make sure the data is set for that at all group levels, we need to work on a per-level basis. Within the output, only the set-up needs the “real” name of the colorspace: we use a simple tracking number for general usage as this is a clear namespace without issues of escaping chars.

```

37718 \cs_new_protected:Npn \__color_model_separation:w
37719   #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7#8
37720   {
37721     \__color_model_init:nnn {#6} { separation } { 0 }
37722     \cs_new_eq:cN { __color_parse_mix_ #6 :nw } \__color_parse_mix_gray:nw
37723     \cs_new:cpn { __color_parse_model_ #6 :w } ##1 , ##2 \s__color_stop
37724       { {#6} { \__color_parse_number:n {##1} } }
37725     \use:c { __color_model_separation_ #8 :nnnnnn }
37726       {#6} {#7} {#1} {#2} {#3} {#4}
37727     \prop_gput:Nnn \g__color_alternative_model_prop {#6} {#8}
37728     \prop_gput:Nne \g__color_colorants_prop {#6}
37729       { \str_convert_pdfname:n {#7} }
37730   }
37731 \cs_new_protected:Npn \__color_model_separation_cmyk:nnnnnn #1#2#3#4#5#6
37732   {
37733     \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
37734     \cs_new:cpn { __color_convert_ #1 _cmyk:w } ##1 \s__color_stop
37735       {
37736         \fp_eval:n {##1 * #3} ~
37737         \fp_eval:n {##1 * #4} ~
37738         \fp_eval:n {##1 * #5} ~
37739         \fp_eval:n {##1 * #6}
37740       }
37741     \cs_new:cpn { __color_convert_cmyk_ #1 :w } ##1 \s__color_stop { 1 }
37742     \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 , #6 }
37743     \__color_backend_separation_init:nnnnn {#2} { /DeviceCMYK } { }
37744     { 0 ~ 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 ~ #6 }
37745   }
37746 \cs_new_protected:Npn \__color_model_separation_rgb:nnnnnn #1#2#3#4#5#6
37747   {
37748     \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }
37749     \cs_new:cpn { __color_convert_ #1 _rgb:w } ##1 \s__color_stop
37750       {
37751         \fp_eval:n {##1 * #3} ~
37752         \fp_eval:n {##1 * #4} ~
37753         \fp_eval:n {##1 * #5}
37754       }
37755     \cs_new:cpn { __color_convert_rgb_ #1 :w } ##1 \s__color_stop { 1 }
37756     \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 }
37757     \__color_backend_separation_init:nnnnn {#2} { /DeviceRGB } { }
37758     { 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 }
37759   }
37760 \cs_new_protected:Npn \__color_model_separation_gray:nnnnnn #1#2#3#4#5#6
37761   {
37762     \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
37763     \cs_new:cpn { __color_convert_ #1 _gray:w } ##1 \s__color_stop
37764       { \fp_eval:n {##1 * #3} }
37765     \cs_new:cpn { __color_convert_gray_ #1 :w } ##1 \s__color_stop { 1 }
37766     \prop_gput:Nnn \g__color_alternative_values_prop {#1} {#3}
37767     \__color_backend_separation_init:nnnnn {#2} { /DeviceGray } { } { 0 } {#3}

```



```
37768 }
```

Generic model conversion *via* an alternative intermediate.

```
37769 \cs_new_protected:Npn \__color_model_convert:nnn #1#2#3
37770 {
37771   \cs_new:cpe { __color_convert_ #1 _ #3 :w } ##1 \s__color_stop
37772   {
37773     \exp_not:N \exp_args:NNe \exp_not:N \use:nn
37774     \exp_not:c { __color_convert_ #2 _ #3 :w }
37775     { \exp_not:c { __color_convert_ #1 _ #2 :w } ##1 \s__color_stop }
37776     \c_space_tl \exp_not:N \s__color_stop
37777   }
37778 }
```

Setting up for CIELAB needs a bit more work: there is the illuminant and the need for an appropriate object.

```
37779 \cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnn #1#2#3#4#5#6
37780 {
37781   \prop_get:NnNF \l__color_internal_prop { illuminant }
37782   \l__color_internal_tl
37783   {
37784     \msg_error:nnn { color }
37785     { CIELAB-requires-illuminant } {#1}
37786     \tl_set:Nn \l__color_internal_tl { d50 }
37787   }
37788   \exp_args:NV \__color_model_separation_CIELAB:nnnnnnn
37789   \l__color_internal_tl {#1} {#2} {#3} {#4} {#5} {#6}
37790 }
```

If a CIELAB space is being set up, we need the illuminant, then create the appropriate set up. At present, this doesn't include BlackPoint or Range data, but that may be added later. As CIELAB colors cannot be converted to anything else, we fallback to producing black in the gray colorspace: the user should set up a second model for colors set up this way.

```
37791 \cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnnn #1#2#3#4#5#6#7
37792 {
37793   \tl_if_exist:cTF { c__color_model_whitepoint_CIELAB_ #1 _tl }
37794   {
37795     \__color_backend_separation_init_CIELAB:nnn {#1} {#3} { #4 ~ #5 ~ #6 }
37796     \tl_const:cn { c__color_fallback_ #2 _tl } { gray }
37797     \cs_new:cpn { __color_convert_ #2 _gray:w } ##1 \s__color_stop
37798     { 0 }
37799     \cs_new:cpn { __color_convert_gray_ #2 :w } ##1 \s__color_stop
37800     { 1 }
37801   }
37802   {
37803     \msg_error:nnn { color }
37804     { unknown-CIELAB-illuminant } {#1}
37805   }
37806 }
```

(End of definition for `__color_model_separation:n` and others.)

```
\__color_model_devicen:n
\__color_model_devicen:nn
\__color_model_devicen:nnn
\__color_model_devicen:nnnn
\__color_model_devicen_parse_1:nn
\__color_model_devicen_parse_2:nn
\__color_model_devicen_parse_3:nn
\__color_model_devicen_parse_4:nn
\__color_model_devicen_parse_generic:nn
\__color_model_devicen_parse:nw
\__color_model_devicen_mix:nw
\__color_model_devicen_init:nnn
```

We require a list of component names here: one might call them colorants, but it's convenient to use T_EX names instead so we slightly adjust the terminology.

```

37807 \cs_new_protected:Npn \l__color_model_devicen:n #1
37808 {
37809   \prop_get:NnNTF \l__color_internal_prop { names }
37810   \l__color_internal_tl
37811   {
37812     \exp_args:NV \l__color_model_devicen:nn
37813     \l__color_internal_tl {#1}
37814   }
37815   {
37816     \msg_error:nnn { color }
37817     { DeviceN-requires-names } {#1}
37818   }
37819 }

```

All valid models will have an alternative listed, either hard-coded for the core device ones, or dynamically added for Separations, etc.

```

37820 \cs_new_protected:Npn \l__color_model_devicen:nn #1#2
37821 {
37822   \tl_clear:N \l__color_model_tl
37823   \clist_map_inline:nn {#1}
37824   {
37825     \prop_get:NnNTF \g__color_alternative_model_prop {##1}
37826     \l__color_internal_tl
37827     {
37828       \tl_if_empty:NTF \l__color_model_tl
37829       { \tl_set_eq:NN \l__color_model_tl \l__color_internal_tl }
37830       {
37831         \str_if_eq:VVF \l__color_model_tl \l__color_internal_tl
37832         {
37833           \msg_error:nnn { color }
37834           { DeviceN-inconsistent-alternative }
37835           {#2}
37836         }
37837         \clist_map_break:n { \use_none:n }
37838       }
37839     }
37840     {
37841       \str_if_eq:nnF {##1} { none }
37842       {
37843         \msg_error:nnn { color }
37844         { DeviceN-no-alternative }
37845         {#2}
37846       }
37847     }
37848   }
37849   \tl_if_empty:NTF \l__color_model_tl
37850   {
37851     \msg_error:nnn { color }
37852     { DeviceN-no-alternative } {#2}
37853   }
37854   { \exp_args:NV \l__color_model_devicen:nnn \l__color_model_tl {#1} {#2} }
37855 }

```

We now complete the data we require by first finding out how many colorants there are, then moving on to begin constructing the function required to map to the alternative

color space.

```

37856 \cs_new_protected:Npn \__color_model_devicen:nnn #1#2#3
37857 {
37858   \exp_args:Ne \__color_model_devicen:nnnn
37859   { \clist_count:n {#2} } {#1} {#2} {#3}
37860 }

```

At this stage, we have checked everything is in place, so we can set up the \TeX and backend data structures. As for separations, it's not really possible in general to have a fallback, so we simply provide “black” for each element.

```

37861 \cs_new_protected:Npn \__color_model_devicen:nnnn #1#2#3#4
37862 {
37863   \__color_model_init:nne {#4} { devicen }
37864   {
37865     0 \prg_replicate:nn { #1 - 1 } { ~ 0 }
37866   }
37867   \cs_if_exist_use:cF { __color_model_devicen_parse_ #1 :nn }
37868   { \__color_model_devicen_parse_generic:nn }
37869   {#4} {#1}
37870   \__color_model_devicen_init:nnn {#1} {#2} {#3}
37871   \__color_model_devicen_convert:nnne {#4} {#2} {#3}
37872   {
37873     1 \prg_replicate:nn { #1 - 1 } { ~ 1 }
37874   }
37875 }

```

For short lists of DeviceN colors, we can use hand-tuned parsing. This lines up with other models, where we allow for up to four components. For larger spaces, rather than limit artificially, we use a somewhat slow approach based on open-ended commas-lists.

```

37876 \cs_new_protected:cpn { __color_model_devicen_parse_1:nn } #1#2
37877 {
37878   \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 \s__color_stop
37879   { {#1} { \__color_parse_number:n {##1} } }
37880   \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \__color_parse_mix_gray:nw
37881 }
37882 \cs_new_protected:cpn { __color_model_devicen_parse_2:nn } #1#2
37883 {
37884   \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 , ##3 \s__color_stop
37885   { {#1} { \__color_parse_number:n {##1} ~ \__color_parse_number:n {##2} } }
37886   \cs_new:cpn { __color_parse_mix_ #1 :nw }
37887   ##1##2 ~ ##3 \s__color_mark ##4 ~ ##5 \s__color_stop
37888   {
37889     \fp_eval:n { ##2 * ##1 + ##4 * ( 1 - ##1 ) } \c_space_tl
37890     \fp_eval:n { ##3 * ##1 + ##5 * ( 1 - ##1 ) }
37891   }
37892 }
37893 \cs_new_protected:cpn { __color_model_devicen_parse_3:nn } #1#2
37894 {
37895   \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 , ##3 , ##4 \s__color_stop
37896   {
37897     {#1}
37898     {
37899       \__color_parse_number:n {##1} ~
37900       \__color_parse_number:n {##2} ~

```

```

37901         \_color_parse_number:n {##3}
37902     }
37903 }
37904 \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \_color_parse_mix_rgb:nw
37905 }
37906 \cs_new_protected:cpn { __color_model_devicen_parse_4:nn } #1#2
37907 {
37908     \cs_new:cpn { __color_parse_model_ #1 :w }
37909     ##1 , ##2 , ##3 , ##4 , ##5 \s_color_stop
37910     {
37911         {#1}
37912         {
37913             \_color_parse_number:n {##1} ~
37914             \_color_parse_number:n {##2} ~
37915             \_color_parse_number:n {##3} ~
37916             \_color_parse_number:n {##4}
37917         }
37918     }
37919     \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \_color_parse_mix_cmyk:nw
37920 }
37921 \cs_new_protected:Npn \_color_model_devicen_parse_generic:nn #1#2
37922 {
37923     \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 \s_color_stop
37924     {
37925         {#1}
37926         { \_color_model_devicen_parse:nw {#2} ##1 , ##2 , \q_nil , \s_color_stop }
37927     }
37928     \cs_new:cpe { __color_parse_mix_ #1 :nw }
37929     ##1 ##2 \s_color_mark ##3 \s_color_stop
37930     {
37931         \exp_not:N \_color_model_devicen_mix:nw {##1}
37932         ##2 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s_color_mark
37933         ##3 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s_color_stop
37934     }
37935 }
37936 \cs_new:Npn \_color_model_devicen_parse:nw #1#2 , #3 \s_color_stop
37937 {
37938     \int_compare:nNnT {#1} > 0
37939     {
37940         \quark_if_nil:nTF {#2}
37941         { \prg_replicate:nn {#1} { 0 ~ } }
37942         {
37943             \_color_parse_number:n {#2}
37944             \int_compare:nNnT {#1} > 1 { ~ }
37945             \exp_args:Nf \_color_model_devicen_parse:nw
37946             { \int_eval:n { #1 - 1 } } #3 \s_color_stop
37947         }
37948     }
37949 }
37950 \cs_new:Npn \_color_model_devicen_mix:nw #1#2 ~ #3 \s_color_mark #4 ~ #5 \s_color_stop
37951 {
37952     \fp_eval:n { #2 * #1 + #4 * ( 1 - #1 ) }
37953     \quark_if_nil:oF { \tl_head:w #3 \q_stop }
37954     {

```

```

37955     \c_space_tl
37956     \_color_model_devicen_mix:nw {#1} #3 \s_color_mark #5 \s_color_stop
37957   }
37958 }

```

To construct the tint transformation, we have to use PostScript. The aim is to have the final tint for each device colorant as

$$1 - \prod_n (1 - X_n D_{X_n})$$

where X is a DeviceN colorant and D is the amount of device colorant that the DeviceN colorant maps to. At the start of the process, the PostScript stack will contain the X_n values, whilst we have the D values on a per-DeviceN colorant basis. The more convenient approach for us is therefore to take each DeviceN colorant in turn and find the value $1 - X_n D_{X_n}$, multiplying as we go, and finalise with the subtraction. That contrasts to `colorspace`: it splits the process up by process color, which works better when you have a fixed list of colorants. (`colorspace` only supports up to 4 DeviceN colors, and only `cmymk` as the alternative space.) To set this up, we first need to know the number of values in the target color space: this is easily handled as there are a very small range of possibilities. Once we have that information, it's relatively easy to build the required PostScript using some generic code.

```

37959 \cs_new_protected:Npn \_color_model_devicen_init:nnn #1#2#3
37960 {
37961   \exp_args:Ne \_color_model_devicen_init:nnnn
37962   {
37963     \str_case:nn {#2}
37964     {
37965       { cmyk } { 4 }
37966       { gray } { 1 }
37967       { rgb } { 3 }
37968     }
37969   }
37970   {#1} {#2} {#3}
37971 }

```

As we always need to split the alternative values into parts, we use a shared auxiliary and only use a minimal difference between code paths. Construction of the tint transformation is as far as possible done using loops, which means there are some inefficiencies for device colors in the DeviceN space: we roll the stack one-at-a-time even if there is a potential shortcut. However, that way there is nothing to special-case. Once this is sorted, we can write the tint transform object, which will remain as the last object until we sort out the final step: the colorant list.

```

37972 \cs_new_protected:Npn \_color_model_devicen_init:nnnn #1#2#3#4
37973 {
37974   \tl_set:Nc \l_color_internal_tl
37975   { \prg_replicate:nn {#1} { 1.0 ~ } }
37976   \int_zero:N \l_color_internal_int
37977   \clist_map_inline:nn {#4}
37978   {
37979     \int_incr:N \l_color_internal_int
37980     \prop_get:NnN \g_color_alternative_values_prop {##1}
37981     \l_color_value_tl
37982     \exp_after:wN \_color_model_devicen_transform:w

```

```

37983         \l__color_value_tl , 0 , 0 , 0 , \s__color_stop {#1} {#2}
37984     }
37985 \tl_put_right:Ne \l__color_internal_tl
37986 {
37987     \prg_replicate:nn {#1}
37988     { neg ~ 1.0 ~ add ~ #1 ~ -1 ~ roll ~ }
37989     \int_eval:n { #2 + #1 } ~ #1 ~ roll
37990     \prg_replicate:nn {#2} { ~ pop } ~
37991     #1 ~ 1 ~ roll
37992 }
37993 \use:e
37994 {
37995     \__color_backend_devicen_init:nnn
37996     {
37997         \clist_map_function:nN {#4}
37998         \__color_model_devicen_colorant:n
37999     }
38000     {
38001         \str_case:nn {#3}
38002         {
38003             { cmyk } { /DeviceCMYK }
38004             { gray } { /DeviceGray }
38005             { rgb } { /DeviceRGB }
38006         }
38007     }
38008     { \exp_not:V \l__color_internal_tl }
38009 }
38010 }
38011 \cs_new_protected:Npn \__color_model_devicen_transform:w
38012 #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7
38013 {
38014     \use:c { __color_model_devicen_transform_ #6 :nnnnn }
38015     {#1} {#2} {#3} {#4} {#7}
38016 }
38017 \cs_new_protected:cpn { __color_model_devicen_transform_1:nnnnn } #1#2#3#4#5
38018 { \__color_model_devicen_transform:nnn {#5} { 1 } {#1} }
38019 \cs_new_protected:cpn { __color_model_devicen_transform_3:nnnnn } #1#2#3#4#5
38020 {
38021     \clist_map_inline:nn { #1 , #2 , #3 }
38022     { \__color_model_devicen_transform:nnn {#5} { 3 } {##1} }
38023 }
38024 \cs_new_protected:cpn { __color_model_devicen_transform_4:nnnnn } #1#2#3#4#5
38025 {
38026     \clist_map_inline:nn { #1 , #2 , #3 , #4 }
38027     { \__color_model_devicen_transform:nnn {#5} { 4 } {##1} }
38028 }
38029 \cs_new_protected:Npn \__color_model_devicen_transform:nnn #1#2#3
38030 {
38031     \tl_put_right:Ne \l__color_internal_tl
38032     {
38033         \fp_compare:nNnF {#3} = \c_zero_fp
38034         {
38035             \int_eval:n { #1 - \l__color_internal_int + #2 } ~ index ~
38036             -#3 ~ mul ~ 1.0 ~ add ~ mul ~

```

```

38037     }
38038     #2 ~ -1 ~ roll ~
38039   }
38040 }
38041 \cs_new:Npn \__color_model_devicen_colorant:n #1
38042 {
38043   / \prop_item:Nn \g__color_colorants_prop {#1} ~
38044 }

```

Here we need to set up conversion from the DeviceN space to the alternative at the T_EX level. This also means supplying methods for inter-converting to other parameter-based spaces. Essentially the approach is exactly the same as the PostScript, just expressed in T_EX terms.

```

38045 \cs_new_protected:Npn \__color_model_devicen_convert:nmmm #1#2#3
38046 {
38047   \use:c { __color_model_devicen_convert_ #2 :nmm } {#1} {#3}
38048 }
38049 \cs_generate_variant:Nn \__color_model_devicen_convert:nmmm { nmne }
38050 \cs_new_protected:Npn \__color_model_devicen_convert_cmyk:nmm #1#2
38051 {
38052   \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
38053   \__color_model_devicen_convert:nmmmm {#1} { cmyk } { 4 } {#2}
38054 }
38055 \cs_new_protected:Npn \__color_model_devicen_convert_gray:nmm #1#2
38056 {
38057   \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
38058   \__color_model_devicen_convert:nmmmm {#1} { gray } { 1 } {#2}
38059 }
38060 \cs_new_protected:Npn \__color_model_devicen_convert_rgb:nmm #1#2
38061 {
38062   \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }
38063   \__color_model_devicen_convert:nmmmm {#1} { rgb } { 3 } {#2}
38064 }
38065 \cs_new_protected:Npn \__color_model_devicen_convert:nmmmm #1#2#3#4#5
38066 {
38067   \cs_new:cpn { __color_convert_ #2 _ #1 :w } ##1 \s__color_stop {#5}
38068   \cs_new:cpe { __color_convert_ #1 _ #2 :w } ##1 \s__color_stop
38069   {
38070     \exp_not:c { __color_convert_devicen_ #2 : \prg_replicate:nn {#3} { n } w }
38071     \prg_replicate:nn {#3} { { 1 } }
38072     ##1 ~ \exp_not:N \s__color_mark
38073     \clist_map_function:nN {#4} \__color_model_devicen_convert:n
38074     {}
38075     \exp_not:N \s__color_stop
38076   }
38077 }
38078 \cs_new:Npn \__color_model_devicen_convert:n #1
38079 {
38080   {
38081     \exp_args:Ne \__color_model_devicen_convert_aux:n
38082     { \prop_item:Nn \g__color_alternative_values_prop {#1} }
38083   }
38084 }
38085 \cs_new:Npn \__color_model_devicen_convert_aux:n #1

```

```

38086 { \_color_model_devicen_convert_aux:w #1 , , , \s__color_stop }
38087 \cs_new:Npn \_color_model_devicen_convert_aux:w #1 , #2 , #3 , #4 , #5 \s__color_stop
38088 {
38089   {#1}
38090   \tl_if_blank:nF {#2}
38091   {
38092     {#2}
38093     \tl_if_blank:nF {#3}
38094     {
38095       {#3}
38096       \tl_if_blank:nF {#4} { {#4} }
38097     }
38098   }
38099 }
38100 \cs_new:Npn \_color_convert_devicen_cmyk:nnnw
38101 #1#2#3#4#5 ~ #6 \s__color_mark #7#8 \s__color_stop
38102 {
38103   \_color_convert_devicen_cmyk:nnnnnnnn {#5} {#1} {#2} {#3} {#4} #7
38104   #6 \s__color_mark #8 \s__color_stop
38105 }
38106 \cs_new:Npn \_color_convert_devicen_cmyk:nnnnnnnn #1#2#3#4#5#6#7#8#9
38107 {
38108   \use:e
38109   {
38110     \exp_not:N \_color_convert_devicen_cmyk_aux:nnnw
38111     { \fp_eval:n { #2 * (1 - (#1 * #6)) } }
38112     { \fp_eval:n { #3 * (1 - (#1 * #7)) } }
38113     { \fp_eval:n { #4 * (1 - (#1 * #8)) } }
38114     { \fp_eval:n { #5 * (1 - (#1 * #9)) } }
38115   }
38116 }
38117 \cs_new:Npn \_color_convert_devicen_cmyk_aux:nnnw
38118 #1#2#3#4 #5 \s__color_mark #6 \s__color_stop
38119 {
38120   \tl_if_blank:nTF {#5}
38121   {
38122     \fp_eval:n { 1 - #1 } ~
38123     \fp_eval:n { 1 - #2 } ~
38124     \fp_eval:n { 1 - #3 } ~
38125     \fp_eval:n { 1 - #4 }
38126   }
38127   {
38128     \_color_convert_devicen_cmyk:nnnw {#1} {#2} {#3} {#4}
38129     #5 \s__color_mark #6 \s__color_stop
38130   }
38131 }
38132 \cs_new:Npn \_color_convert_devicen_gray:nw
38133 #1#2 ~ #3 \s__color_mark #4#5 \s__color_stop
38134 {
38135   \_color_convert_devicen_gray:nnn {#2} {#1} #4
38136   #3 \s__color_mark #5 \s__color_stop
38137 }
38138 \cs_new:Npn \_color_convert_devicen_gray:nnn #1#2#3
38139 {

```



```

38140     \exp_args:Ne \_color_convert_devicen_gray_aux:nw
38141     { \fp_eval:n { #2 * (1 - (#1 * #3)) } }
38142   }
38143 \cs_new:Npn \_color_convert_devicen_gray_aux:nw
38144 #1 #2 \s__color_mark #3 \s__color_stop
38145 {
38146   \tl_if_blank:nTF {#2}
38147   { \fp_eval:n { 1 - #1 } }
38148   {
38149     \_color_convert_devicen_gray:nw {#1}
38150     #2 \s__color_mark #3 \s__color_stop
38151   }
38152 }
38153 \cs_new:Npn \_color_convert_devicen_rgb:nnnw
38154 #1#2#3#4 ~ #5 \s__color_mark #6#7 \s__color_stop
38155 {
38156   \_color_convert_devicen_rgb:nnnnnn {#4} {#1} {#2} {#3} #6
38157   #5 \s__color_mark #7 \s__color_stop
38158 }
38159 \cs_new:Npn \_color_convert_devicen_rgb:nnnnnn #1#2#3#4#5#6#7
38160 {
38161   \use:e
38162   {
38163     \exp_not:N \_color_convert_devicen_rgb_aux:nnnw
38164     { \fp_eval:n { #2 * (1 - (#1 * #5)) } }
38165     { \fp_eval:n { #3 * (1 - (#1 * #6)) } }
38166     { \fp_eval:n { #4 * (1 - (#1 * #7)) } }
38167   }
38168 }
38169 \cs_new:Npn \_color_convert_devicen_rgb_aux:nnnw
38170 #1#2#3 #4 \s__color_mark #5 \s__color_stop
38171 {
38172   \tl_if_blank:nTF {#4}
38173   {
38174     \fp_eval:n { 1 - #1 } ~
38175     \fp_eval:n { 1 - #2 } ~
38176     \fp_eval:n { 1 - #3 }
38177   }
38178   {
38179     \_color_convert_devicen_rgb:nnnw {#1} {#2} {#3}
38180     #4 \s__color_mark #5 \s__color_stop
38181   }
38182 }

```

(End of definition for `_color_model_devicen:n` and others.)

`\c_color_icc_colorspace_signatures_prop`

The signatures in the ICC file header indicating the underlying colorspace. We map it to three values: The number of components, the values corresponding to white, and the range.

```

38183 \prop_const_from_keyval:Nn \c_color_icc_colorspace_signatures_prop
38184 {
38185   % Gray
38186   47524159 = {1} {1} {0} {},
38187   % RGB

```

```

38188     52474220 = {3} {0~0~0} {1~1~1} {},
38189 % CMYK
38190     434D594B = {4} {0~0~0~1} {0~0~0~0} {},
38191 % Lab
38192     4C616220 = {3} {0~0~0} {100~0~0} {0~100~-128~127~-128~127}
38193 }

```

(End of definition for \c__color_icc_colorspace_signatures_prop.)

__color_model_iccbased:n For an ICC profile, we need a file name and a number of components. The file name is processed here so the backend can treat it as a string.

```

\__color_model_iccbased:nn
\__color_model_iccbased:nnn
  \__color_model_iccbased_aux:nnn
38194 \cs_new_protected:Npn \__color_model_iccbased:n #1
38195 {
38196   \prop_get:NnNTF \l__color_internal_prop { file }
38197   \l__color_internal_tl
38198   {
38199     \exp_args:NV \__color_model_iccbased:nn
38200     \l__color_internal_tl {#1}
38201   }
38202   {
38203     \msg_error:nnn { color }
38204     { ICCBased-requires-file } {#1}
38205   }
38206 }
38207 \cs_new_protected:Npn \__color_model_iccbased:nn #1#2
38208 {
38209   \prop_get:NeNTF \c__color_icc_colorspace_signatures_prop
38210   { \file_hex_dump:nnn { #1 } { 17 } { 20 } } \l__color_internal_tl
38211   {
38212     \exp_last_unbraced:NV \__color_model_iccbased_aux:nnnnnn
38213     \l__color_internal_tl { #2 } { #1 }
38214   }
38215   {
38216     \msg_error:nnn { color }
38217     { ICCBased-unsupported-colorspace } {#2}
38218   }
38219 }

```

Here, we can use the same internals as for DeviceN approach as we know the number of components. No conversion is possible, so there is no need to worry about that at all.

```

38220 \cs_new_protected:Npn \__color_model_iccbased_aux:nnnnnn #1#2#3#4#5#6
38221 {
38222   \__color_model_init:nnn {#5} { iccbased } {#3}
38223   \tl_const:cn { c__color_fallback_ #5 _tl } { gray }
38224   \cs_new:cpn { __color_convert_ #5 _gray:w } ##1 \s__color_stop { 0 }
38225   \cs_new:cpn { __color_convert_gray_ #5 :w } ##1 \s__color_stop { #2 }
38226   \use:c { __color_model_devicen_parse_ #1 :nn } {#5} {#1}
38227   \exp_args:Ne \__color_backend_iccbased_init:nnn
38228   { \file_full_name:n {#6} } {#1} {#4}
38229 }

```

(End of definition for __color_model_iccbased:n and others.)

93.13 Applying profiles

`\color_profile_apply:nn` With a limited range of outcomes, this is largely about getting data to the backend.

```

38230 \cs_new_protected:Npn \color_profile_apply:nn #1#2
38231   {
38232     \exp_args:Ne \color_profile_apply:nn
38233     { \file_full_name:n {#1} } {#2}
38234   }
38235 \cs_new_protected:Npn \__color_profile_apply:nn #1#2
38236   {
38237     \cs_if_exist_use:cF { __color_profile_apply_ \tl_to_str:n {#2} :n }
38238     {
38239       \msg_error:nnn { color } { ICC-Device-unknown } {#2}
38240       \use_none:n
38241     }
38242     {#1}
38243   }
38244 \cs_new_protected:Npn \__color_profile_apply_gray:n #1
38245   {
38246     \int_gincr:N \g__color_model_int
38247     \__color_backend_iccbased_device:nnn {#1} { Gray } { 1 }
38248   }
38249 \cs_new_protected:Npn \__color_profile_apply_rgb:n #1
38250   {
38251     \int_gincr:N \g__color_model_int
38252     \__color_backend_iccbased_device:nnn {#1} { RGB } { 3 }
38253   }
38254 \cs_new_protected:Npn \__color_profile_apply_cmyk:n #1
38255   {
38256     \int_gincr:N \g__color_model_int
38257     \__color_backend_iccbased_device:nnn {#1} { CMYK } { 4 }
38258   }

```

(End of definition for `\color_profile_apply:nn` and others. This function is documented on page 328.)

93.14 Diagnostics

`\color_show:n` Extract the information about a color and format for the user: the approach is similar to the keys module here.

`\color_log:n`

```

\__color_show:Nn 38259 \cs_new_protected:Npn \color_show:n
\__color_show:n 38260   { \__color_show:Nn \msg_show:nneeee }
38261 \cs_new_protected:Npn \color_log:n
38262   { \__color_show:Nn \msg_log:nneeee }
38263 \cs_new_protected:Npn \__color_show:Nn #1#2
38264   {
38265     #1 { color } { show }
38266     {#2}
38267     {
38268       \color_if_exist:nT {#2}
38269       {
38270         \exp_args:Nv \__color_show:n { l__color_named_ #2 _tl }
38271         \prop_map_function:cN
38272         { l__color_named_ #2 _prop }

```

```

38273             \msg_show_item_unbraced:nn
38274         }
38275     }
38276     { }
38277     { }
38278 }
38279 \cs_new:Npn \__color_show:n #1
38280 {
38281     \msg_show_item_unbraced:nn { model } {#1}
38282 }

```

(End of definition for `\color_show:n` and others. These functions are documented on page 324.)

93.15 Messages

```

38283 \msg_new:nnnn { color } { CIELAB-requires-illuminant }
38284 { CIELAB-color-space-#1-require-an-illuminant. }
38285 {
38286     LaTeX-has-been-asked-to-create-a-separation-color-space-using-
38287     CIELAB-specifications,-but-no-\\ \\
38288     \iow_indent:n { illuminant~=<basis> }
38289     \\ \\
38290     key-was-given-with-the-correct-information.-LaTeX-will-use-illuminant-
38291     'd50'-for-recovery.
38292 }
38293 \msg_new:nnnn { color } { conversion-not-available }
38294 { No-model-conversion-available-from-#1-to-#2. }
38295 {
38296     LaTeX-has-been-asked-to-convert-a-color-from-model-#1-
38297     to-model-#2,-but-there-is-no-method-available-to-do-that.
38298 }
38299 \msg_new:nnnn { color } { DeviceN-inconsistent-alternative }
38300 { DeviceN-color-spaces-require-a-single-alternative-space. }
38301 {
38302     LaTeX-has-been-asked-to-create-a-DeviceN-color-space-#1,-
38303     but-the-constituent-colors-do-not-have-a-common-alternative-
38304     color.
38305 }
38306 \msg_new:nnnn { color } { DeviceN-no-alternative }
38307 { DeviceN-color-spaces-require-an-alternative-space. }
38308 {
38309     LaTeX-has-been-asked-to-create-a-DeviceN-color-space-#1,-
38310     but-the-constituent-colors-do-not-all-have-a-device-based-alternative.
38311 }
38312 \msg_new:nnnn { color } { DeviceN-requires-names }
38313 { DeviceN-color-space-#1-require-a-list-of-names. }
38314 {
38315     LaTeX-has-been-asked-to-create-a-DeviceN-color-space,-
38316     but-no-\\ \\
38317     \iow_indent:n { names~=<names> }
38318     \\ \\
38319     key-was-given-with-the-correct-information.
38320 }
38321 \msg_new:nnnn { color } { ICC-Device-unknown }

```

```

38322 { Unknown-device-color-space-#1' . }
38323 {
38324 LaTeX-has-been-asked-to-apply-an-ICC-profile-but-the-device-color-space-
38325 #1'-is-unknown.
38326 }
38327 \msg_new:nnnn { color } { ICCBased-unsupported-colorspace }
38328 { ICCBased-color-space-#1'-uses-an-unsupported-data-color-space. }
38329 {
38330 LaTeX-has-been-asked-to-create-a-ICCBased-colorspace,~but-the-
38331 used-data-colorspace-is-not-supported.~ICC-profiles-used-for~
38332 defining-a-ICCBased-colorspace-should-use-a-Lab,~RGB,~or~
38333 CMYK-data-colorspace.~LaTeX-will-ignore-this-request.
38334 }
38335 \msg_new:nnnn { color } { ICCBased-requires-file }
38336 { ICCBased-color-space-#1'-require-an-file. }
38337 {
38338 LaTeX-has-been-asked-to-create-an-ICCBased-color-space,~but-no~\ \ \
38339 \iow_indent:n { file-~<name> }
38340 \ \ \
38341 key-was-given-with-the-correct-information.~LaTeX-will-ignore-this-
38342 request.
38343 }
38344 \msg_new:nnnn { color } { model-already-defined }
38345 { Color-model-#1'-already-defined. }
38346 {
38347 LaTeX-was-asked-to-define-a-new-color-model-called-#1',~but~
38348 this-color-model-already-exists.
38349 }
38350 \msg_new:nnnn { color } { out-of-range }
38351 { Input-value-#1-out-of-range-[#2,~#3]. }
38352 {
38353 LaTeX-was-expecting-a-value-in-the-range-[#2,~#3]~as-part-of-a-color,~
38354 but-you-gave-#1.~LaTeX-will-assume-you-meant-the-limit-of-the-range~
38355 and-continue.
38356 }
38357 \msg_new:nnnn { color } { separation-alternative-model }
38358 { Separation-color-space-#1'-require-an-alternative-model. }
38359 {
38360 LaTeX-has-been-asked-to-create-a-separation-color-space,~
38361 but-no~\ \ \
38362 \iow_indent:n { alternative-model-~<model> }
38363 \ \ \
38364 key-was-given-with-the-correct-information.
38365 }
38366 \msg_new:nnnn { color } { separation-alternative-values }
38367 { Separation-color-space-#1'-require-values-for-the-alternative-space. }
38368 {
38369 LaTeX-has-been-asked-to-create-a-separation-color-space,~
38370 but-no~\ \ \
38371 \iow_indent:n { alternative-values-~<model> }
38372 \ \ \
38373 key-was-given-with-the-correct-information.
38374 }
38375 \msg_new:nnnn { color } { separation-requires-name }

```

```

38376 { Separation~color~space~'#1'~require~a~formal~name. }
38377 {
38378 LaTeX~has~been~asked~to~create~a~separation~color~space,~
38379 but~no~\\ \\
38380 \iow_indent:n { name~=<formal~name> }
38381 \\ \\
38382 key~was~given~with~the~correct~information.
38383 }
38384 \msg_new:nnn { color } { unhandled-model }
38385 {
38386 Unhandled~color~model~in~LaTeX2e~value~"#1":
38387 \\ \\
38388 falling~back~on~grayscale.
38389 }
38390 \msg_new:nnnn { color } { unknown-color }
38391 { Unknown~color~'#1'. }
38392 {
38393 LaTeX~has~been~asked~to~use~a~color~named~'#1',~
38394 but~this~has~never~been~defined.
38395 }
38396 \msg_new:nnnn { color } { unknown-alternative-model }
38397 { Separation~color~space~'#1'~require~an~valid~alternative~space. }
38398 {
38399 LaTeX~has~been~asked~to~create~a~separation~color~space,~
38400 but~the~model~given~as\\ \\
38401 \iow_indent:n { alternative~model~=<model> }
38402 \\ \\
38403 is~unknown.
38404 }
38405 \msg_new:nnnn { color } { unknown-export-format }
38406 { Unknown~export~format~'#1'. }
38407 {
38408 LaTeX~has~been~asked~to~export~a~color~in~format~'#1',~
38409 but~this~has~never~been~defined.
38410 }
38411 \msg_new:nnnn { color } { unknown-CIELAB-illuminant }
38412 { Unknown~illuminant~model~'#1'. }
38413 {
38414 LaTeX~has~been~asked~to~use~create~a~color~space~using~CIELAB~
38415 illuminant~'#1',~but~this~does~not~exist.
38416 }
38417 \msg_new:nnnn { color } { unknown-model }
38418 { Unknown~color~model~'#1'. }
38419 {
38420 LaTeX~has~been~asked~to~use~a~color~model~called~'#1',~
38421 but~this~model~is~not~set~up.
38422 }
38423 \msg_new:nnnn { color } { unknown-model-type }
38424 { Unknown~color~model~type~'#1'. }
38425 {
38426 LaTeX~has~been~asked~to~create~a~new~color~model~called~'#1',~
38427 but~this~type~of~model~was~never~set~up.
38428 }
38429 \prop_gput:Nnn \g_msg_module_name_prop { color } { LaTeX }

```

```
38430 \prop_gput:Nnn \g_msg_module_type_prop { color } { }
38431 \msg_new:nnn { color } { show }
38432 {
38433   The~color~#1~
38434   \tl_if_empty:nTF {#2}
38435     { is~undefined. }
38436     { has~the~properties: #2 }
38437 }
38438 </package>
```

Chapter 94

l3pdf implementation

```
38439 (*package)
38440 (@@=pdf)
38441 (*tex)

\s__pdf_stop Internal scan marks.
38442 \scan_new:N \s__pdf_stop
(End of definition for \s__pdf_stop.)

\g__pdf_init_bool A boolean so we have some chance of avoiding setting things we are not allowed to. As
we are potentially early in the format, we have to work a bit harder than ideal.
38443 \bool_new:N \g__pdf_init_bool
38444 \bool_lazy_and:nnT
38445   { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
38446   { \tl_if_exist_p:N \@expl@finalise@setup@@ }
38447   {
38448     \tl_gput_right:Nn \@expl@finalise@setup@@
38449       {
38450         \tl_gput_right:Nn \@kernel@after@begindocument
38451           { \bool_gset_true:N \g__pdf_init_bool }
38452       }
38453   }
(End of definition for \g__pdf_init_bool.)
```

94.1 Compression

\pdf_uncompress: Simple to do.

```
38454 \cs_new_protected:Npn \pdf_uncompress:
38455   {
38456     \bool_if:NF \g__pdf_init_bool
38457     {
38458       \__pdf_backend_compresslevel:n { 0 }
38459       \__pdf_backend_compress_objects:n { \c_false_bool }
38460     }
38461   }
```

(End of definition for \pdf_uncompress:. This function is documented on page 331.)

94.2 Objects

`\g__pdf_backend_object_int` For returning object numbers.

```
38462 \int_new:N \g__pdf_backend_object_int
```

(End of definition for `\g__pdf_backend_object_int`.)

Simple to do: all objects create a constant int so it is not a backend-specific name.

```

\pdf_object_new:n
\pdf_object_write:nnn
\pdf_object_write:nne
\pdf_object_write:nnx
\pdf_object_ref:n
\__kernel_pdf_object_id:n
38463 \cs_new_protected:Npn \pdf_object_new:n #1
38464 {
38465   \__pdf_backend_object_new:
38466   \__pdf_object_record:nN {#1} \g__pdf_backend_object_int
38467 }
38468 \cs_new_protected:Npn \pdf_object_write:nnn #1#2#3
38469 {
38470   \exp_args:Ne \__pdf_backend_object_write:nnn
38471   { \__pdf_object_retrieve:n {#1} } {#2} {#3}
38472   \bool_gset_true:N \g__pdf_init_bool
38473 }
38474 \cs_generate_variant:Nn \pdf_object_write:nnn { nne , nnx }
38475 \cs_new:Npn \pdf_object_ref:n #1
38476 {
38477   \exp_args:Ne \__pdf_backend_object_ref:n
38478   { \__pdf_object_retrieve:n {#1} }
38479 }
38480 \cs_new:Npn \__kernel_pdf_object_id:n #1
38481 {
38482   \exp_args:Ne \__pdf_backend_object_id:n
38483   { \__pdf_object_retrieve:n {#1} }
38484 }
38485 </tex>

```

(End of definition for `\pdf_object_new:n` and others. These functions are documented on page 329.)

`__pdf_object_record:nN`
`__pdf_object_retrieve:n`
`ltx.pdf.object_id` Object mappings are tracked in Lua for LuaTeX as this makes retrieving them much easier; as a result, there is a split in approaches. In Lua we store values in a table indexed by name. The Lua function here is set up to deal with both named and indexed objects: fits the Lua idiom well.

```

38486 (*lua)
38487
38488 local scan_int = token.scan_int
38489 local scan_string = token.scan_string
38490 local cprint = tex.cprint
38491
38492 local __pdf_objects_named = {}
38493 local __pdf_objects_indexed = {}
38494
38495 luacmd('\__pdf_object_record:nN', function()
38496   local name = scan_string()
38497   local n = scan_int()
38498   __pdf_objects_named[name] = n
38499 end, 'protected', 'global')
38500
38501 local function object_id(name, index)

```

```

38502   if index then
38503     return __pdf_objects_indexed[name][index] or 0
38504   else
38505     return __pdf_objects_named[name] or 0
38506   end
38507 end
38508
38509 luacmd('__pdf_object_retrieve:n', function()
38510   local name = scan_string()
38511   return cprint(12,tostring(object_id(name)))
38512 end,'global')
38513
38514 ltx.pdf = ltx.pdf or {}
38515 ltx.pdf.object_id = object_id
38516
38517  $\langle$ /lua)

```

Whereas in \TeX we use integer constants.

```

38518  $\langle$ *tex)
38519 \sys_if_engine luatex:F
38520 {
38521   \cs_new_protected:Npn \__pdf_object_record:nN #1#2
38522     {
38523       \int_const:cn
38524         { c__pdf_object_ #1 _int } {#2}
38525     }
38526   \cs_new:Npn \__pdf_object_retrieve:n #1
38527     {
38528       \int_if_exist:cTF { c__pdf_object_ #1 _int }
38529         {
38530           \int_use:c
38531             { c__pdf_object_ #1 _int }
38532         }
38533         { 0 }
38534     }
38535 }

```

(End of definition for \backslash __pdf_object_record:nN, \backslash __pdf_object_retrieve:n, and ltx.pdf.object_id. This function is documented on page ??.)

```

\pdf_object_if_exist_p:n
\pdf_object_if_exist:nTF
38536 \prg_new_conditional:Npnn \pdf_object_if_exist:n #1 { p , T , F , TF }
38537 {
38538   \int_compare:nNnTF { \__pdf_object_retrieve:n {#1} } = 0
38539     \prg_return_false:
38540     \prg_return_true:
38541 }

```

(End of definition for \backslash pdf_object_if_exist:nTF. This function is documented on page 329.)

```

\pdf_object_new_indexed:nn
\pdf_object_write_indexed:mmnn
\pdf_object_write_indexed:mnne
\pdf_object_ref_indexed:nn
38542 \cs_new_protected:Npn \pdf_object_new_indexed:nn #1#2
38543 {

```

Again we split between the common code and the macro- or Lua-based implementation. To make life easier for the Lua route, all of the potential expressions are expanded to braced numbers.

```

38544   \__pdf_backend_object_new:
38545   \__pdf_object_record:neN {#1}
38546     { \int_eval:n {#2} } \g__pdf_backend_object_int
38547   }
38548 \cs_new_protected:Npn \pdf_object_write_indexed:nnnn #1#2#3#4
38549   {
38550     \exp_args:Ne \__pdf_backend_object_write:nnn
38551       { \__pdf_object_retrieve:ne {#1} { \int_eval:n {#2} } } {#3} {#4}
38552     \bool_gset_true:N \g__pdf_init_bool
38553   }
38554 \cs_generate_variant:Nn \pdf_object_write_indexed:nnnn { nnne }
38555 \cs_new:Npn \pdf_object_ref_indexed:nn #1#2
38556   {
38557     \exp_args:Ne \__pdf_backend_object_ref:n
38558       { \__pdf_object_retrieve:ne {#1} { \int_eval:n {#2} } }
38559   }
38560 \cs_new:Npn \__kernel_pdf_object_id_indexed:nn #1#2
38561   {
38562     \exp_args:Ne \__pdf_backend_object_id:n
38563       { \__pdf_object_retrieve:ne {#1} { \int_eval:n {#2} } }
38564   }
38565 </tex>

```

(End of definition for `\pdf_object_new_indexed:nn` and others. These functions are documented on page 330.)

`__pdf_object_record:nnN` Again we split for Lua: the same idea as above but with nested tables. As we've arranged above that the TeX code passes a braced number, we can use `tonumber(scan_string())` rather than `scan_int()` for the index.

```

\__pdf_object_record:neN
\__pdf_object_retrieve:nn
\__pdf_object_record:NnN
\__pdf_object_retrieve:Nn
38566 (*lua)
38567
38568 luacmd('\__pdf_object_record:nnN', function()
38569   local name = scan_string()
38570   local index = tonumber(scan_string())
38571   local n = scan_int()
38572   __pdf_objects_indexed[name] = __pdf_objects_indexed[name] or {}
38573   __pdf_objects_indexed[name][index] = n
38574 end, 'protected', 'global')
38575
38576 luacmd('\__pdf_object_retrieve:nn', function()
38577   local name = scan_string()
38578   local index = tonumber(scan_string())
38579   return cprint(12, tostring(object_id(name, index)))
38580 end, 'global')
38581
38582 </lua>

```

The non-Lua approach is to divide the range into blocks, and store in integer arrays that can simulate dynamic assignment.

```

38583 (*tex)
38584 \sys_if_engine luatex:F
38585   {
38586     \cs_new_protected:Npn \__pdf_object_record:nnN #1#2#3
38587       {
38588         \use:e

```

```

38589     {
38590         \__pdf_object_record:nnN
38591         \__pdf_object_index_split:nn {#1} {#2}
38592         \exp_not:N #3
38593     }
38594 }
38595 \cs_new_protected:Npn \__pdf_object_record:nnN #1#2#3
38596 {
38597     \intarray_if_exist:NF #1
38598     { \intarray_new:nn #1 \c__pdf_object_block_size_int }
38599     \intarray_gset:Nnn #1 {#2} #3
38600 }
38601 \cs_new:Npn \__pdf_object_retrieve:nn #1#2
38602 {
38603     \use:e
38604     {
38605         \exp_not:N \__pdf_object_retrieve:Nn
38606         \__pdf_object_index_split:nn {#1} {#2}
38607     }
38608 }
38609 \cs_new:Npn \__pdf_object_retrieve:Nn #1#2
38610 { \intarray_item:Nn #1 {#2} }

```

As we want blocks to start from one, and within the block for the top value to be “in” the block, we do a little bit of manipulation. By shifting down by one, we keep the values “in” the block, then we adjust the block/index number to get back on track.

```

38611 \cs_new:Npn \__pdf_object_index_split:nn #1#2
38612 {
38613     \exp_not:c
38614     {
38615         g__pdf_object_ #1 _
38616         \int_eval:n
38617         {
38618             \int_div_truncate:nn { #2 - 1 }
38619             \c__pdf_object_block_size_int + 1
38620         }
38621         _intarray
38622     }
38623     {
38624         \int_eval:n
38625         { \int_mod:nn { #2 - 1 } \c__pdf_object_block_size_int + 1 }
38626     }
38627 }

```

(End of definition for __pdf_object_record:nnN and others.)

\c__pdf_object_block_size_int Sets the block size used for managing indexed objects.

```

38628 \int_const:Nn \c__pdf_object_block_size_int { 10000 }
38629 }

```

(End of definition for \c__pdf_object_block_size_int.)

__pdf_object_record:neN Common variants.

```

\__pdf_object_retrieve:ne
38630 \cs_generate_variant:Nn \__pdf_object_record:nnN { ne }
38631 \cs_generate_variant:Nn \__pdf_object_retrieve:nn { ne }

```

(End of definition for `_pdf_object_record:neN` and `_pdf_object_retrieve:ne`.)

`\pdf_object_unnamed_write:nn` No tracking needed here.

```
\pdf_object_unnamed_write:ne 38632 \cs_new_protected:Npn \pdf_object_unnamed_write:nn #1#2
\pdf_object_unnamed_write:nx 38633 {
38634   \exp_args:Ne \_pdf_backend_object_now:nn {#1} {#2}
38635   \bool_gset_true:N \g__pdf_init_bool
38636 }
38637 \cs_generate_variant:Nn \pdf_object_unnamed_write:nn { ne , nx }
```

(End of definition for `\pdf_object_unnamed_write:nn`. This function is documented on page 330.)

`\pdf_object_ref_last:` A one-step wrapper for consistency.

```
38638 \cs_new:Npn \pdf_object_ref_last: { \_pdf_backend_object_last: }
```

(End of definition for `\pdf_object_ref_last:`. This function is documented on page 331.)

`\pdf_pageobject_ref:n`

```
38639 \cs_new:Npn \pdf_pageobject_ref:n #1
38640 { \exp_args:Ne \_pdf_backend_pageobject_ref:n {#1} }
```

(End of definition for `\pdf_pageobject_ref:n`. This function is documented on page 331.)

94.3 Version

`\pdf_version_compare_p:Nn` To compare version, we need to split the given value then deal with both major and
`\pdf_version_compare:NnTF` minor version

```
__pdf_version_compare_=:w 38641 \prg_new_conditional:Npnn \pdf_version_compare:Nn #1#2 { p , T , F , TF }
__pdf_version_compare_<:w 38642 { \use:c { __pdf_version_compare_ #1 :w } #2 . . \s__pdf_stop }
__pdf_version_compare_>:w 38643 \cs_new:cpn { __pdf_version_compare_=:w } #1 . #2 . #3 \s__pdf_stop
38644 {
38645   \bool_lazy_and:nnTF
38646   { \int_compare_p:nNn \_pdf_backend_version_major: = {#1} }
38647   { \int_compare_p:nNn \_pdf_backend_version_minor: = {#2} }
38648   { \prg_return_true: }
38649   { \prg_return_false: }
38650 }
38651 \cs_new:cpn { __pdf_version_compare_<:w } #1 . #2 . #3 \s__pdf_stop
38652 {
38653   \bool_lazy_or:nnTF
38654   { \int_compare_p:nNn \_pdf_backend_version_major: < {#1} }
38655   {
38656     \bool_lazy_and_p:nn
38657     { \int_compare_p:nNn \_pdf_backend_version_major: = {#1} }
38658     { \int_compare_p:nNn \_pdf_backend_version_minor: < {#2} }
38659   }
38660   { \prg_return_true: }
38661   { \prg_return_false: }
38662 }
38663 \cs_new:cpn { __pdf_version_compare_>:w } #1 . #2 . #3 \s__pdf_stop
38664 {
38665   \bool_lazy_or:nnTF
38666   { \int_compare_p:nNn \_pdf_backend_version_major: > {#1} }
```

```

38667     {
38668         \bool_lazy_and_p:nn
38669         { \int_compare_p:nNn \_pdf_backend_version_major: = {#1} }
38670         { \int_compare_p:nNn \_pdf_backend_version_minor: > {#2} }
38671     }
38672     { \prg_return_true: }
38673     { \prg_return_false: }
38674 }

```

(End of definition for `\pdf_version_compare:NnTF` and others. This function is documented on page 331.)

`\pdf_version_gset:n` Split the version and set.
`\pdf_version_min_gset:n`
`_pdf_version_gset:w`

```

38675 \cs_new_protected:Npn \pdf_version_gset:n #1
38676 { \_pdf_version_gset:w #1 . . \s__pdf_stop }
38677 \cs_new_protected:Npn \pdf_version_min_gset:n #1
38678 {
38679     \pdf_version_compare:NnT < {#1}
38680     { \_pdf_version_gset:w #1 . . \s__pdf_stop }
38681 }
38682 \cs_new_protected:Npn \_pdf_version_gset:w #1 . #2 . #3\s__pdf_stop
38683 {
38684     \bool_if:NF \g__pdf_init_bool
38685     {
38686         \_pdf_backend_version_major_gset:n {#1}
38687         \_pdf_backend_version_minor_gset:n {#2}
38688     }
38689 }

```

(End of definition for `\pdf_version_gset:n`, `\pdf_version_min_gset:n`, and `_pdf_version_gset:w`. These functions are documented on page 331.)

`\pdf_version:` Wrappers.
`\pdf_version_major:`
`\pdf_version_minor:`

```

38690 \cs_new:Npn \pdf_version:
38691 { \_pdf_backend_version_major: . \_pdf_backend_version_minor: }
38692 \cs_new:Npn \pdf_version_major: { \_pdf_backend_version_major: }
38693 \cs_new:Npn \pdf_version_minor: { \_pdf_backend_version_minor: }

```

(End of definition for `\pdf_version:`, `\pdf_version_major:`, and `\pdf_version_minor:`. These functions are documented on page 331.)

94.4 Page size

`\pdf_pagesize_gset:nn`

```

38694 \cs_new_protected:Npn \pdf_pagesize_gset:nn #1#2
38695 { \_pdf_backend_pagesize_gset:nn {#1} {#2} }

```

(End of definition for `\pdf_pagesize_gset:nn`. This function is documented on page 331.)

94.5 Destinations

`\pdf_destination:nn`

```
38696 \cs_new_protected:Npn \pdf_destination:nn #1#2
38697   { \__pdf_backend_destination:nn {#1} {#2} }
```

(End of definition for `\pdf_destination:nn`. This function is documented on page 332.)

`\pdf_destination:nmmn`

```
38698 \cs_new_protected:Npn \pdf_destination:nmmn #1#2#3#4
38699   {
38700     \hbox_to_zero:n
38701     { \__pdf_backend_destination:nmmn {#1} {#2} {#3} {#4} }
38702   }
```

(End of definition for `\pdf_destination:nmmn`. This function is documented on page 332.)

94.6 PDF Page size (media box)

Everything here is delayed to the start of the document so that the backend will definitely be loaded.

```
38703 \cs_if_exist:NT \@kernel@before@begindocument
38704   {
38705     \tl_gput_right:Nn \@kernel@before@begindocument
38706     {
38707       \bool_lazy_all:nT
38708         {
38709           { \cs_if_exist_p:N \stockheight }
38710           { \cs_if_exist_p:N \stockwidth }
38711           { \cs_if_exist_p:N \IfDocumentMetadataTF }
38712           { \IfDocumentMetadataTF { \c_true_bool } { \c_false_bool } }
38713           { \int_compare_p:nNn \tex_mag:D = { 1000 } }
38714         }
38715       {
38716         \bool_lazy_and:nnTF
38717           { \dim_compare_p:nNn \stockheight > { Opt } }
38718           { \dim_compare_p:nNn \stockwidth > { Opt } }
38719           {
38720             \__pdf_backend_pagesize_gset:nn
38721             \stockwidth \stockheight
38722           }
38723         {
38724           \bool_lazy_or:nnF
38725             { \dim_compare_p:nNn \stockheight < { Opt } }
38726             { \dim_compare_p:nNn \stockwidth < { Opt } }
38727             {
38728               \bool_lazy_and:nnT
38729                 { \dim_compare_p:nNn \paperheight > { Opt } }
38730                 { \dim_compare_p:nNn \paperwidth > { Opt } }
38731               {
38732                 \__pdf_backend_pagesize_gset:nn
38733                 \paperwidth \paperheight
38734               }
38735             }
38736         }
38737       }
38738     }
38739   }
```

```
38735     }
38736   }
38737 }
38738 }
38739 }
38740 </tex>
38741 </package>
```


Chapter 95

l3deprecation implementation

```
38742 (*package)
38743 (@@=deprecation)
```

95.1 Patching definitions to deprecate

```
\_kernel_patch_deprecation:nnNNpn {<date>} {<replacement>} {<definition>}
{<function>} {<parameters>} {<code>}
```

defines the *<function>* to produce an error and run its *<code>*.

We make `\debug_on:n {deprecation}` turn the *<function>* into an `\outer` error, and `\debug_off:n {deprecation}` restore whatever the behaviour was without `\debug_on:n {deprecation}`.

In the explanations below, *<definition>* *<function>* *<parameters>* *{<code>}* or assignments that only differ in the scope of the *<definition>* will be called “the standard definition”.

```
\_kernel_patch_deprecation:nnNNpn (The parameter text is grabbed using #5#.) The arguments of \_kernel_deprecation_
\_deprecation_patch_aux:nnNNnn code:nn are run upon \debug_on:n {deprecation} and \debug_off:n {deprecation},
\_deprecation_warn_once:nnNnn respectively. In both scenarios we the <function> may be \outer so we undefine it with
\_deprecation_patch_aux:Nn \tex_let:D before redefining it, with \_kernel_deprecation_error:Nnn or with some
\_deprecation_just_error:nnNN code added shortly.
```

```
38744 \cs_new_protected:Npn \_kernel_patch_deprecation:nnNNpn #1#2#3#4#5#
38745 { \_deprecation_patch_aux:nnNNnn {#1} {#2} #3 #4 {#5} }
38746 \cs_new_protected:Npn \_deprecation_patch_aux:nnNNnn #1#2#3#4#5#6
38747 {
38748   \_kernel_deprecation_code:nn
38749   {
38750     \tex_let:D #4 \scan_stop:
38751     \_kernel_deprecation_error:Nnn #4 {#2} {#1}
38752   }
38753   { \tex_let:D #4 \scan_stop: }
38754   \cs_if_eq:NNTF #3 \cs_gset_protected:Npn
38755   { \_deprecation_warn_once:nnNnn {#1} {#2} #4 {#5} {#6} }
38756   { \_deprecation_patch_aux:Nn #3 { #4 #5 {#6} } }
38757 }
```

In case we want a warning, the $\langle function \rangle$ is defined to produce such a warning without grabbing any argument, then redefine itself to the standard definition that the $\langle function \rangle$ should have, with arguments, and call that definition. The e-type expansion and $\backslash\exp_not:n$ avoid needing to double the #, which we could not do anyways. We then deal with the code for $\backslash\debug_off:n \{deprecation\}$: presumably someone doing that does not need the warning so we simply do the standard definition.

```

38758 \cs_new_protected:Npn \__deprecation_warn_once:nnNnn #1#2#3#4#5
38759 {
38760   \cs_gset_protected:Npe #3
38761   {
38762     \__kernel_if_debug:TF
38763     {
38764       \exp_not:N \msg_warning:nneee
38765       { deprecation } { deprecated-command }
38766       {#1}
38767       { \token_to_str:N #3 }
38768       { \tl_to_str:n {#2} }
38769     }
38770     { }
38771     \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
38772     \exp_not:N #3
38773   }
38774   \__kernel_deprecation_code:nn { }
38775   { \cs_set_protected:Npn #3 #4 {#5} }
38776 }

```

In case we want neither warning nor error, the $\langle function \rangle$ is given its standard definition. Here #1 is $\backslash\cs_new:Npn$ or $\backslash\cs_new_protected:Npn$ and #2 is $\langle function \rangle$ $\langle parameters \rangle \{ \langle code \rangle \}$, so #1#2 performs the assignment. For $\backslash\debug_off:n \{deprecation\}$ we want to use the same assignment but with a different scope, hence the $\backslash\cs_if_eq:NNTF$ test.

```

38777 \cs_new_protected:Npn \__deprecation_patch_aux:Nn #1#2
38778 {
38779   #1 #2
38780   \cs_if_eq:NNTF #1 \cs_gset_protected:Npn
38781   { \__kernel_deprecation_code:nn { } { \cs_set_protected:Npn #2 } }
38782   { \__kernel_deprecation_code:nn { } { \cs_set:Npn #2 } }
38783 }

```

(End of definition for $\backslash__kernel_patch_deprecation:nnNNpn$ and others.)

$\backslash__kernel_deprecation_error:Nnn$ The $\backslash\outer$ definition here ensures the command cannot appear in an argument.

```

38784 \cs_new_protected:Npn \__kernel_deprecation_error:Nnn #1#2#3
38785 {
38786   \tex_protected:D \tex_outer:D \tex_edef:D #1
38787   {
38788     \exp_not:N \msg_expandable_error:nnnnn
38789     { deprecation } { deprecated-command }
38790     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
38791     \exp_not:N \msg_error:nneee
38792     { deprecation } { deprecated-command }
38793     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
38794   }
38795 }

```

(End of definition for `_kernel_deprecation_error:Nnn`.)

```
38796 \msg_new:nnn { deprecation } { deprecated-command }
38797   {
38798     \tl_if_blank:nF {#3} { Use~ \tl_trim_spaces:n {#3} ~not~ }
38799     #2~deprecated-on~#1.
38800   }
```

95.2 Deprecated l3basics functions

38801 (@@=cs)

`\cs_argument_spec:N` For the present, do not deprecate fully as L^AT_EX 2_ε will need to catch up: one for Fall 2022.

```
38802 %\_kernel_patch_deprecation:nnNNpn { 2022-06-24 } { \cs_parameter_spec:N }
38803 \cs_new:Npn \cs_argument_spec:N { \cs_parameter_spec:N }
```

(End of definition for `\cs_argument_spec:N`.)

95.3 Deprecated l3file functions

38804 (@@=file)

`\iow_shipout_x:Nn` Previously described as x-type, but the hash behaviour is really e-type. Currently not “live” as we need to have a transition.

```
\iow_shipout_x:Nx
\iow_shipout_x:cn
\iow_shipout_x:cx
38805 %\_kernel_patch_deprecation:nnNNpn { 2023-10-10 } { \iow_shipout_e:Nn }
38806 \cs_new_protected:Npn \iow_shipout_x:Nn { \iow_shipout_e:Nn }
38807 \cs_generate_variant:Nn \iow_shipout_x:Nn { Nx , c, cx }
```

(End of definition for `\iow_shipout_x:Nn`.)

95.4 Deprecated l3keys functions

38808 (@@=keys)

```
.str_set_x:N
.str_set_x:c
38809 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:N } #1
.str_gset_x:N
38810   { \__keys_variable_set:NnnN #1 { str } { } x }
.str_gset_x:c
38811 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:c } #1
38812   { \__keys_variable_set:cnnN {#1} { str } { } x }
38813 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_x:N } #1
38814   { \__keys_variable_set:NnnN #1 { str } { g } x }
38815 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_x:c } #1
38816   { \__keys_variable_set:cnnN {#1} { str } { g } x }
```

(End of definition for `.str_set_x:N` and `.str_gset_x:N`.)

```
.tl_set_x:N
.tl_set_x:c
38817 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:N } #1
.tl_gset_x:N
38818   { \__keys_variable_set:NnnN #1 { tl } { } x }
.tl_gset_x:c
38819 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:c } #1
38820   { \__keys_variable_set:cnnN {#1} { tl } { } x }
38821 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:N } #1
```

```

38822 { \_keys_variable_set:NnnN #1 { t1 } { g } x }
38823 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:c } #1
38824 { \_keys_variable_set:cnnN {#1} { t1 } { g } x }

```

(End of definition for .tl_set_x:N and .tl_gset_x:N.)

```

\keys_set_filter:nnnN We need a transition here so for the present this is commented out: only needed for
\keys_set_filter:nnVN latex-lab code so this should not last for too long.
\keys_set_filter:nnvN 38825 %\_kernel_patch_deprecation:nnNNpn { 2024-01-10 } { \keys_set_exclude_groups:nnn }
\keys_set_filter:nnoN 38826 \cs_new_protected:Npn \keys_set_filter:nnn { \keys_set_exclude_groups:nnn }
\keys_set_filter:nnnnN 38827 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
\keys_set_filter:nnVnN 38828 %\_kernel_patch_deprecation:nnNNpn { 2024-01-10 } { \keys_set_exclude_groups:nnnN }
\keys_set_filter:nnvnN 38829 \cs_new_protected:Npn \keys_set_filter:nnnN { \keys_set_exclude_groups:nnnN }
\keys_set_filter:nnonN 38830 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
\keys_set_filter:nnn 38831 %\_kernel_patch_deprecation:nnNNpn { 2024-01-10 } { \keys_set_exclude_groups:nnnnN }
\keys_set_filter:nnV 38832 \cs_new_protected:Npn \keys_set_filter:nnnnN { \keys_set_exclude_groups:nnnnN }
\keys_set_filter:nnv 38833 \cs_generate_variant:Nn \keys_set_filter:nnnnN { nnV , nnv , nno }
\keys_set_filter:nno (End of definition for \keys_set_filter:nnnN, \keys_set_filter:nnnnN, and \keys_set_filter:nnn.)

```

95.5 Deprecated l3msg functions

```

38834 (@@=msg)
\msg_gset:nnnn
\msg_gset:nnn 38835 \_kernel_patch_deprecation:nnNNpn { 2024-02-13 } { \msg_set:nnnn }
38836 \cs_new_protected:Npn \msg_gset:nnnn { \msg_set:nnnn }
38837 \_kernel_patch_deprecation:nnNNpn { 2024-02-13 } { \msg_set:nnn }
38838 \cs_new_protected:Npn \msg_gset:nnn { \msg_set:nnn }

```

(End of definition for \msg_gset:nnnn and \msg_gset:nnn.)

95.6 Deprecated l3pdf functions

```

38839 (@@=pdf)
\g__pdf_object_prop For tracking objects.
38840 \prop_new:N \g__pdf_object_prop
(End of definition for \g__pdf_object_prop.)
\pdf_object_new:nn
\pdf_object_write:nn 38841 \_kernel_patch_deprecation:nnNNpn { 2022-08-30 } { [\pdf_object_new:n] }
\pdf_object_write:nx 38842 \cs_new_protected:Npn \pdf_object_new:nn #1#2
38843 {
38844   \prop_gput:Nnn \g__pdf_object_prop {#1} {#2}
38845   \pdf_object_new:n {#1}
38846 }
38847 \_kernel_patch_deprecation:nnNNpn { 2022-08-30 } { [\pdf_object_write:n] }
38848 \cs_new_protected:Npn \pdf_object_write:nn #1#2
38849 {
38850   \exp_args:Nee \_pdf_backend_object_write:nnn
38851   { \_pdf_object_retrieve:n {#1} }
38852   { \prop_item:Nn \g__pdf_object_prop {#1} } {#2}

```

```

38853     \bool_gset_true:N \g__pdf_init_bool
38854   }
38855 \cs_generate_variant:Nn \pdf_object_write:nn { nx }

```

(End of definition for \pdf_object_new:nn and \pdf_object_write:nn.)

95.7 Deprecated l3prg functions

```

38856 <@@=cs>

```

```

\bool_case_true:n
\bool_case_true:nTF
38857 \__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:n }
38858 \cs_new:Npn \bool_case_true:n { \bool_case:n }
38859 \__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:nT }
38860 \cs_new:Npn \bool_case_true:nT { \bool_case:nT }
38861 \__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:nF }
38862 \cs_new:Npn \bool_case_true:nF { \bool_case:nF }
38863 \__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:nTF }
38864 \cs_new:Npn \bool_case_true:nTF { \bool_case:nTF }

```

(End of definition for \bool_case_true:nTF.)

95.8 Deprecated l3str functions

```

38865 <@@=str>

```

```

\str_lower_case:n
\str_lower_case:f
\str_upper_case:n
\str_upper_case:f
\str_fold_case:n
\str_fold_case:V
38866 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:n }
38867 \cs_new:Npn \str_lower_case:n { \str_lowercase:n }
38868 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:f }
38869 \cs_new:Npn \str_lower_case:f { \str_lowercase:f }
38870 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:n }
38871 \cs_new:Npn \str_upper_case:n { \str_uppercase:n }
38872 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:f }
38873 \cs_new:Npn \str_upper_case:f { \str_uppercase:f }
38874 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:n }
38875 \cs_new:Npn \str_fold_case:n { \str_casefold:n }
38876 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:V }
38877 \cs_new:Npn \str_fold_case:V { \str_casefold:V }

```

(End of definition for \str_lower_case:n, \str_upper_case:n, and \str_fold_case:n.)

```

\str_foldcase:n
\str_foldcase:V
38878 \__kernel_patch_deprecation:nnNNpn { 2020-10-17 } { \str_casefold:n }
38879 \cs_new:Npn \str_foldcase:n { \str_casefold:n }
38880 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casefold:V }
38881 \cs_new:Npn \str_foldcase:V { \str_casefold:V }

```

(End of definition for \str_foldcase:n.)

`\str_declare_eight_bit_encoding:nmn` This command was made internal, with one more argument. There is no easy way to compute a reasonable value for that extra argument so we take a value that is big enough to accommodate all of Unicode.

```
38882 \__kernel_patch_deprecation:nmNnpn { 2020-08-20 } { }
38883 \cs_new_protected:Npn \str_declare_eight_bit_encoding:nmn #1
38884   { \__str_declare_eight_bit_encoding:nmnn {#1} { 1114112 } }
```

(End of definition for `\str_declare_eight_bit_encoding:nmn`.)

95.9 Deprecated `l3seq` functions

```
38885 <@@=seq>
```

`\seq_indexed_map_inline:Nn`
`\seq_indexed_map_function:NN`

```
38886 \__kernel_patch_deprecation:nmNnpn { 2020-06-18 } { \seq_map_indexed_inline:Nn }
38887 \cs_new_protected:Npn \seq_indexed_map_inline:Nn { \seq_map_indexed_inline:Nn }
38888 \__kernel_patch_deprecation:nmNnpn { 2020-06-18 } { \seq_map_indexed_function:NN }
38889 \cs_new:Npn \seq_indexed_map_function:NN { \seq_map_indexed_function:NN }
```

(End of definition for `\seq_indexed_map_inline:Nn` and `\seq_indexed_map_function:NN`.)

`\seq_mapthread_function:NNN`

```
38890 \__kernel_patch_deprecation:nmNnpn { 2023-05-10 } { \seq_map_pairwise_function:NNN }
38891 \cs_new:Npn \seq_mapthread_function:NNN { \seq_map_pairwise_function:NNN }
```

(End of definition for `\seq_mapthread_function:NNN`.)

`\seq_set_map_x:NNn`
`\seq_gset_map_x:NNn`

```
38892 \__kernel_patch_deprecation:nmNnpn { 2023-10-26 } { \seq_set_map_e:NNn }
38893 \cs_new_protected:Npn \seq_set_map_x:NNn { \seq_set_map_e:NNn }
38894 \__kernel_patch_deprecation:nmNnpn { 2023-10-26 } { \seq_gset_map_e:NNn }
38895 \cs_new_protected:Npn \seq_gset_map_x:NNn { \seq_gset_map_e:NNn }
```

(End of definition for `\seq_set_map_x:NNn` and `\seq_gset_map_x:NNn`.)

95.10 Deprecated `l3sys` functions

```
38896 <@@=sys>
```

`\sys_load_deprecation:`

```
38897 \__kernel_patch_deprecation:nmNnpn { 2021-01-11 } { (no-longer~required) }
38898 \cs_new_protected:Npn \sys_load_deprecation: { }
```

(End of definition for `\sys_load_deprecation:.`)

95.11 Deprecated `\text` functions

```

38899 <@@=text>

\text_titlecase:n
\text_titlecase:nn 38900 \__kernel_patch_deprecation:nnNNpn { 2023-07-08 } { \text_titlecase_first:n }
38901 \cs_new:Npn \text_titlecase:n #1
38902   { \text_titlecase_first:n { \text_lowercase:n {#1} } }
38903 \__kernel_patch_deprecation:nnNNpn { 2023-07-08 } { \text_titlecase_first:nn }
38904 \cs_new:Npn \text_titlecase:nn #1#2
38905   { \text_titlecase_first:nn {#1} { \text_lowercase:n {#2} } }

(End of definition for \text_titlecase:n and \text_titlecase:nn.)

```

95.12 Deprecated `\tl` functions

```

38906 <@@=tl>

\tl_lower_case:n
\tl_lower_case:nn 38907 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:n }
\tl_upper_case:n 38908 \cs_new:Npn \tl_lower_case:n #1
\tl_upper_case:nn 38909   { \text_lowercase:n {#1} }
\tl_mixed_case:n 38910 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:nn }
\tl_mixed_case:nn 38911 \cs_new:Npn \tl_lower_case:nn #1#2
38912   { \text_lowercase:nn {#1} {#2} }
38913 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:n }
38914 \cs_new:Npn \tl_upper_case:n #1
38915   { \text_uppercase:n {#1} }
38916 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:nn }
38917 \cs_new:Npn \tl_upper_case:nn #1#2
38918   { \text_uppercase:nn {#1} {#2} }
38919 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase_first:n }
38920 \cs_new:Npn \tl_mixed_case:n #1
38921   { \text_titlecase_first:n {#1} }
38922 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase_first:nn }
38923 \cs_new:Npn \tl_mixed_case:nn #1#2
38924   { \text_titlecase_first:nn {#1} {#2} }

(End of definition for \tl_lower_case:n and others.)

```

```

\tl_case:Nn
\tl_case:cn 38925 \__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:Nn }
\tl_case:NnTF 38926 \cs_new:Npn \tl_case:Nn { \token_case_meaning:Nn }
\tl_case:cnTF 38927 \__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:NnT }
38928 \cs_new:Npn \tl_case:NnT { \token_case_meaning:NnT }
38929 \__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:NnF }
38930 \cs_new:Npn \tl_case:NnF { \token_case_meaning:NnF }
38931 \__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:NnTF }
38932 \cs_new:Npn \tl_case:NnTF { \token_case_meaning:NnTF }
38933 \cs_generate_variant:Nn \tl_case:Nn { c }
38934 \prg_generate_conditional_variant:Nnn \tl_case:Nn
38935   { c } { T , F , TF }

(End of definition for \tl_case:NnTF.)

```

```

\tl_build_clear:N
\tl_build_gclear:N
38936 \__kernel_patch_deprecation:nnNNpn { 2023-10-18 } { \tl_build_begin:N }
38937 \cs_new_protected:Npn \tl_build_clear:N { \tl_build_begin:N }
38938 \__kernel_patch_deprecation:nnNNpn { 2023-10-18 } { \tl_build_gbegin:N }
38939 \cs_new_protected:Npn \tl_build_gclear:N { \tl_build_gbegin:N }

```

(End of definition for \tl_build_clear:N and \tl_build_gclear:N.)

```

\tl_build_get:NN
38940 \__kernel_patch_deprecation:nnNNpn { 2023-10-25 } { \tl_build_get_intermediate:NN }
38941 \cs_new_protected:Npn \tl_build_get:NN { \tl_build_get_intermediate:NN }

```

(End of definition for \tl_build_get:NN.)

95.13 Deprecated l3token functions

```

38942 <@@=char>

```

```

\char_to_utfviii_bytes:n
38943 \__kernel_patch_deprecation:nnNNpn { 2022-10-09 } { [ \codepoint_generate:nn ] }
38944 \cs_new:Npn \char_to_utfviii_bytes:n { \__kernel_codepoint_to_bytes:n }

```

(End of definition for \char_to_utfviii_bytes:n.)

```

\char_to_nfd:N
\char_to_nfd:n
38945 \__kernel_patch_deprecation:nnNNpn { 2022-10-09 } { \codepoint_to_nfd:n }
38946 \cs_new:Npn \char_to_nfd:N #1 { \codepoint_to_nfd:n { '#1' } }
38947 \__kernel_patch_deprecation:nnNNpn { 2022-10-09 } { \codepoint_to_nfd:n }
38948 \cs_new:Npn \char_to_nfd:n { \codepoint_to_nfd:n }

```

(End of definition for \char_to_nfd:N and \char_to_nfd:n.)

```

\char_lower_case:N
\char_upper_case:N
\char_mixed_case:Nn
\char_fold_case:N
\char_str_lower_case:N
\char_str_upper_case:N
\char_str_mixed_case:Nn
\char_str_fold_case:N
38949 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:n }
38950 \cs_new:Npn \char_lower_case:N { \text_lowercase:n }
38951 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:n }
38952 \cs_new:Npn \char_upper_case:N { \text_uppercase:n }
38953 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase_first:n }
38954 \cs_new:Npn \char_mixed_case:N { \text_titlecase_first:n }
38955 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:n }
38956 \cs_new:Npn \char_fold_case:N { \str_casefold:n }
38957 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:n }
38958 \cs_new:Npn \char_str_lower_case:N { \str_lowercase:n }
38959 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:n }
38960 \cs_new:Npn \char_str_upper_case:N { \str_uppercase:n }
38961 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_titlecase:n }
38962 \cs_new:Npn \char_str_mixed_case:N { \str_titlecase:n }
38963 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:n }
38964 \cs_new:Npn \char_str_fold_case:N { \str_casefold:n }

```

(End of definition for \char_lower_case:N and others.)


```

\char_lowercase:N
\char_titlecase:N
\char_uppercase:N
\char_foldcase:N
\char_str_lowercase:N
\char_str_titlecase:N
\char_str_uppercase:N
\char_str_foldcase:N
38965 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \text_lowercase:n }
38966 \cs_new:Npn \char_lowercase:N { \text_lowercase:n }
38967 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \text_uppercase:n }
38968 \cs_new:Npn \char_uppercase:N { \text_uppercase:n }
38969 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \text_titlecase_first:n }
38970 \cs_new:Npn \char_titlecase:N { \text_titlecase_first:n }
38971 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casefold:n }
38972 \cs_new:Npn \char_foldcase:N { \str_casefold:n }
38973 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_lowercase:n }
38974 \cs_new:Npn \char_str_lowercase:N { \str_lowercase:n }
38975 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 }
38976 { \tl_to_str:e { \text_titlecase_first:n } }
38977 \cs_new:Npn \char_str_titlecase:N #1
38978 { \tl_to_str:e { \text_titlecase_first:n {#1} } }
38979 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_uppercase:n }
38980 \cs_new:Npn \char_str_uppercase:N { \str_uppercase:n }
38981 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casefold:n }
38982 \cs_new:Npn \char_str_foldcase:N { \str_casefold:n }

```

(End of definition for \char_lowercase:N and others.)

A little extra fun here to deal with the expansion.

```

\peek_catcode_ignore_spaces:NTF
\peek_catcode_remove_ignore_spaces:NTF
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove_ignore_spaces:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove_ignore_spaces:NTF
38983 \tl_map_inline:nn
38984 {
38985 { catcode } { catcode_remove }
38986 { charcode } { charcode_remove }
38987 { meaning } { meaning_remove }
38988 }
38989 {
38990 \use:e
38991 {
38992 \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
38993 \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NTF } ##1##2##3
38994 {
38995 \peek_remove_spaces:n
38996 { \exp_not:c { peek_ #1 :NTF } ##1 {##2} {##3} }
38997 }
38998 \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
38999 \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NT } ##1##2
39000 {
39001 \peek_remove_spaces:n
39002 { \exp_not:c { peek_ #1 :NT } ##1 {##2} }
39003 }
39004 \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
39005 \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NF } ##1##2
39006 {
39007 \peek_remove_spaces:n
39008 { \exp_not:c { peek_ #1 :NF } ##1 {##2} }
39009 }
39010 }
39011 }

```

(End of definition for \peek_catcode_ignore_spaces:NTF and others.)

95.14 Deprecated **l3prop** functions

```
\prop_put_if_new:Nnn
\prop_put_if_new:NVn 39012 %\__kernel_patch_deprecation:nnNNpn { 2024-03-30 } { \prop_put_if_not_in:Nnn }
\prop_put_if_new:NnV 39013 \cs_new_protected:Npn \prop_put_if_new:Nnn { \prop_put_if_not_in:Nnn }
\prop_put_if_new:cnn 39014 %\__kernel_patch_deprecation:nnNNpn { 2024-03-30 } { \prop_gput_if_not_in:Nnn }
\prop_put_if_new:cVn 39015 \cs_new_protected:Npn \prop_gput_if_new:Nnn { \prop_gput_if_not_in:Nnn }
\prop_put_if_new:cnV 39016 \cs_generate_variant:Nn \prop_put_if_new:Nnn
\prop_gput_if_new:Nnn 39017 { NnV , NV , c , cnV , cV }
\prop_gput_if_new:NVn 39018 \cs_generate_variant:Nn \prop_gput_if_new:Nnn
\prop_gput_if_new:NnV 39019 { NnV , NV , c , cnV , cV }
\prop_gput_if_new:cnn (End of definition for \prop_put_if_new:Nnn and \prop_gput_if_new:Nnn.)
\prop_gput_if_new:cVn
\prop_gput_if_new:cnV 39020 </package>
```

Chapter 96

l3debug implementation

Internal kernel functions that are only defined here are listed in l3kernel-functions, see [41.1](#).

```
39021 <*package>
39022 <@@=debug>
      Standard file identification.
39023 \ProvidesExplFile{l3debug.def}{2024-05-08}{L3 Debugging support}
```

```
\s__debug_stop Internal scan marks.
39024 \scan_new:N \s__debug_stop
      (End of definition for \s__debug_stop.)
```

```
\_debug_use_i_delimit_by_s_stop:nw Functions to gobble up to a scan mark.
39025 \cs_new:Npn \_debug_use_i_delimit_by_s_stop:nw #1 #2 \s__debug_stop {#1}
      (End of definition for \_debug_use_i_delimit_by_s_stop:nw.)
```

```
\q__debug_recursion_tail Internal quarks.
\q__debug_recursion_stop 39026 \quark_new:N \q__debug_recursion_tail
39027 \quark_new:N \q__debug_recursion_stop
      (End of definition for \q__debug_recursion_tail and \q__debug_recursion_stop.)
```

```
\__debug_if_recursion_tail_stop:N Functions to query recursion quarks.
39028 \cs_new:Npn \__debug_use_none_delimit_by_q_recursion_stop:w
39029   #1 \q__debug_recursion_stop { }
39030 \__kernel_quark_new_test:N \__debug_if_recursion_tail_stop:N
      (End of definition for \__debug_if_recursion_tail_stop:N.)
```

```
      \debug_on:n
      \debug_off:n
39031 \cs_gset_protected:Npn \debug_on:n #1
39032   {
39033     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
39034     {
39035       \cs_if_exist_use:cF { __debug_ ##1 _on: }
39036       { \msg_error:nnn { debug } { debug } {##1} }
39037     }
}
```

```

39038 }
39039 \cs_gset_protected:Npn \debug_off:n #1
39040 {
39041   \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
39042   {
39043     \cs_if_exist_use:cF { __debug_ ##1 _off: }
39044     { \msg_error:nnn { debug } { debug } {##1} }
39045   }
39046 }
39047 \cs_new_protected:Npn \__debug_all_on:
39048 {
39049   \debug_on:n
39050   {
39051     check-declarations ,
39052     check-expressions ,
39053     deprecation ,
39054     log-functions ,
39055   }
39056 }
39057 \cs_new_protected:Npn \__debug_all_off:
39058 {
39059   \debug_off:n
39060   {
39061     check-declarations ,
39062     check-expressions ,
39063     deprecation ,
39064     log-functions ,
39065   }
39066 }

```

(End of definition for `\debug_on:n` and others. These functions are documented on page 30.)

`\debug_suspend:` Suspend and resume locally all debug-related errors and logging except deprecation errors.

`\debug_resume:` The `\debug_suspend:` and `\debug_resume:` pairs can be nested. We keep track of

`__debug_suspended:T` nesting in a token list containing a number of periods. At first begin with the “non-suspended” version of `__debug_suspended:T`.

```

39067 \tl_new:N \l__debug_suspended_tl { }
39068 \cs_gset_protected:Npn \debug_suspend:
39069 {
39070   \tl_put_right:Nn \l__debug_suspended_tl { . }
39071   \cs_set_eq:NN \__debug_suspended:T \use:n
39072 }
39073 \cs_gset_protected:Npn \debug_resume:
39074 {
39075   \__kernel_tl_set:Nx \l__debug_suspended_tl
39076   { \tl_tail:N \l__debug_suspended_tl }
39077   \tl_if_empty:NT \l__debug_suspended_tl
39078   {
39079     \cs_set_eq:NN \__debug_suspended:T \use_none:n
39080   }
39081 }
39082 \cs_new_eq:NN \__debug_suspended:T \use_none:n

```

(End of definition for `\debug_suspend:` and others. These functions are documented on page 30.)

`__debug_check-declarations_on:` When debugging is enabled these two functions set up functions that test their argument
`__debug_check-declarations_off:` (when `check-declarations` is active)

- `__kernel_chk_var_exist:N` and `__kernel_chk_cs_exist:N`, two functions that test that their argument is defined;
- `__kernel_chk_var_scope:NN` that checks that its argument #2 has scope #1.
- `__kernel_chk_var_local:N` and `__kernel_chk_var_global:N` that perform both checks.

```

39083 \cs_new_protected:Npn __kernel_chk_var_exist:N #1 { }
39084 \cs_new_protected:Npn __kernel_chk_cs_exist:N #1 { }
39085 \cs_generate_variant:Nn __kernel_chk_cs_exist:N { c }
39086 \cs_new:Npn __kernel_chk_flag_exist:NN { }
39087 \cs_new_protected:Npn __kernel_chk_var_local:N #1 { }
39088 \cs_new_protected:Npn __kernel_chk_var_global:N #1 { }
39089 \cs_new_protected:Npn __kernel_chk_var_scope:NN #1#2 { }
39090 \cs_new_protected:cpn { __debug_check-declarations_on: }
39091 {
39092   \cs_set_protected:Npn __kernel_chk_var_exist:N ##1
39093   {
39094     __debug_suspended:T \use_none:nnn
39095     \cs_if_exist:NF ##1
39096     {
39097       \msg_error:nne { debug } { non-declared-variable }
39098       { \token_to_str:N ##1 }
39099     }
39100   }
39101   \cs_set_protected:Npn __kernel_chk_cs_exist:N ##1
39102   {
39103     __debug_suspended:T \use_none:nnn
39104     \cs_if_exist:NF ##1
39105     {
39106       \msg_error:nne { kernel } { command-not-defined }
39107       { \token_to_str:N ##1 }
39108     }
39109   }
39110   \cs_set:Npn __kernel_chk_flag_exist:NN ##1##2
39111   {
39112     __debug_suspended:T \use_iii:nmnn
39113     \flag_if_exist:NTF ##2
39114     { ##1 ##2 }
39115     {
39116       \msg_expandable_error:nnn { kernel } { bad-variable } {##2}
39117       ##1 \l_tmpa_flag
39118     }
39119   }
39120   \cs_set_protected:Npn __kernel_chk_var_scope:NN
39121   {
39122     __debug_suspended:T \use_none:nnn
39123     __debug_chk_var_scope_aux:NN
39124   }
39125   \cs_set_protected:Npn __kernel_chk_var_local:N ##1
39126   {

```

```

39127     \_debug_suspended:T \use_none:nnnnn
39128     \_kernel_chk_var_exist:N ##1
39129     \_debug_chk_var_scope_aux:NN l ##1
39130   }
39131   \cs_set_protected:Npn \_kernel_chk_var_global:N ##1
39132   {
39133     \_debug_suspended:T \use_none:nnnnn
39134     \_kernel_chk_var_exist:N ##1
39135     \_debug_chk_var_scope_aux:NN g ##1
39136   }
39137 }
39138 \cs_new_protected:cpn { __debug_check-declarations_off: }
39139 {
39140   \cs_set_protected:Npn \_kernel_chk_var_exist:N ##1 { }
39141   \cs_set_protected:Npn \_kernel_chk_cs_exist:N ##1 { }
39142   \cs_set:Npn \_kernel_chk_flag_exist:NN { }
39143   \cs_set_protected:Npn \_kernel_chk_var_local:N ##1 { }
39144   \cs_set_protected:Npn \_kernel_chk_var_global:N ##1 { }
39145   \cs_set_protected:Npn \_kernel_chk_var_scope:NN ##1##2 { }
39146 }

```

(End of definition for `_debug_check-declarations_on:` and others.)

`_debug_chk_var_scope_aux:NN`
`_debug_chk_var_scope_aux:Nn`
`_debug_chk_var_scope_aux:NNn`

First check whether the name of the variable #2 starts with `<letter>_`. If it does then pass that letter, the `<scope>`, and the variable name to `_debug_chk_var_scope_aux:NNn`. That function compares the two letters and triggers an error if they differ (the `\scan_stop:` case is not reachable here). If the second character was not `_` then pass the same data to the same auxiliary, except for its first argument which is now a control sequence. That control sequence is actually a token list (but to avoid triggering the checking code we manipulate it using `\cs_set_nopar:Npn`) containing a single letter `<scope>` according to what the first assignment to the given variable was.

```

39147 \cs_new_protected:Npn \_debug_chk_var_scope_aux:NN #1#2
39148   { \exp_args:Nnf \_debug_chk_var_scope_aux:Nn #1 { \cs_to_str:N #2 } }
39149 \cs_new_protected:Npn \_debug_chk_var_scope_aux:Nn #1#2
39150   {
39151     \if:w _ \use_i:nn \_debug_use_i_delimit_by_s_stop:nw #2 ? ? \s__debug_stop
39152     \exp_after:wN \_debug_chk_var_scope_aux:NNn
39153     \_debug_use_i_delimit_by_s_stop:nw #2 ? \s__debug_stop
39154     #1 {#2}
39155   \else:
39156     \exp_args:Nc \_debug_chk_var_scope_aux:NNn
39157     { \_debug_chk_/ #2 }
39158     #1 {#2}
39159   \fi:
39160 }
39161 \cs_new_protected:Npn \_debug_chk_var_scope_aux:NNn #1#2#3
39162   {
39163     \if:w #1 #2
39164   \else:
39165     \if:w #1 \scan_stop:
39166       \cs_gset_nopar:Npn #1 {#2}
39167   \else:
39168     \msg_error:nnee { debug } { local-global }
39169     {#1} {#2} { \iow_char:N \ #3 }

```

```

39170     \fi:
39171     \fi:
39172   }
39173 \use:c { __debug_check-declarations_off: }

```

(End of definition for `__debug_chk_var_scope_aux:NN`, `__debug_chk_var_scope_aux:Nn`, and `__debug_chk_var_scope_aux:NNn`.)

`__debug_log-functions_on:` These two functions (corresponding to the `expl3` option `log-functions`) control whether
`__debug_log-functions_off:` `__kernel_debug_log:e` writes to the log file or not. By default, logging is off.

```

\__kernel_debug_log:e
39174 \cs_new_protected:cpn { __debug_log-functions_on: }
39175   {
39176     \cs_set_protected:Npn \__kernel_debug_log:e
39177       { \__debug_suspended:T \use_none:n \iow_log:e }
39178   }
39179 \cs_new_protected:cpn { __debug_log-functions_off: }
39180   { \cs_set_protected:Npn \__kernel_debug_log:e { \use_none:n } }
39181 \cs_new_protected:Npn \__kernel_debug_log:e { \use_none:n }

```

(End of definition for `__debug_log-functions_on:`, `__debug_log-functions_off:`, and `__kernel_debug_log:e`.)

`__debug_check-expressions_on:` When debugging is enabled these two functions set `__kernel_chk_expr:nNn` to test or
`__debug_check-expressions_off:` not whether the given expression is valid. The idea is to evaluate the expression within
`__kernel_chk_expr:nNn` a brace group (to catch trailing `\use_none:n` or similar), then test that the result is
`__debug_chk_expr_aux:nNn` what we expect. This is done by turning it to an integer and hitting that with `\tex_roman`
`__debug_chk_expr_aux:nNn` numeral:D after replacing the first character by `-0`. If all goes well, that primitive
finds a non-positive integer and gives an empty output. If the original expression evaluation
stopped early it leaves a trailing `\tex_relax:D`, which stops the second evaluation
(used to convert to integer) before it encounters the final `\tex_relax:D`. Since `\tex_roman`
numeral:D does not absorb `\tex_relax:D` the output will be nonempty. Note
that `#3` is empty except for `mu` expressions for which it is `\tex_mutoglu`
to avoid an “incompatible glue units” error. Note also that if we had omitted the first `\tex_`
relax:D then for instance `1+2\relax+3` would incorrectly be accepted as a valid integer
expression.

```

39182 \cs_new_protected:cpn { __debug_check-expressions_on: }
39183   {
39184     \cs_set:Npn \__kernel_chk_expr:nNn ##1##2
39185       {
39186         \__debug_suspended:T { ##1 \use_none:nnnnnn }
39187         \exp_after:wN \__debug_chk_expr_aux:nNn
39188         \exp_after:wN { \tex_the:D ##2 ##1 \scan_stop: }
39189         ##2
39190       }
39191   }
39192 \cs_new_protected:cpn { __debug_check-expressions_off: }
39193   { \cs_set:Npn \__kernel_chk_expr:nNn ##1##2##3##4 {##1} }
39194 \cs_new:Npn \__kernel_chk_expr:nNn #1#2#3#4 {#1}
39195 \cs_new:Npn \__debug_chk_expr_aux:nNn #1#2#3#4
39196   {
39197     \tl_if_empty:oF
39198       {
39199         \tex_romannumeral:D - 0
39200         \exp_after:wN \use_none:n

```

```

39201         \int_value:w #3 #2 #1 \scan_stop:
39202     }
39203     {
39204         \msg_expandable_error:nmmn
39205         { debug } { expr } {#4} {#1}
39206     }
39207     #1
39208 }

```

(End of definition for `__debug_check-expressions_on:` and others.)

`__debug_deprecation_on:` Make deprecated commands throw errors if the user requests it. This relies on two token lists, filled up in `l3deprecation` by calls to `__kernel_deprecation_code:nn`.

```

39209 \cs_new_protected:Npn \__debug_deprecation_on:
39210 { \g__debug_deprecation_on_tl }
39211 \cs_new_protected:Npn \__debug_deprecation_off:
39212 { \g__debug_deprecation_off_tl }

```

(End of definition for `__debug_deprecation_on:` and `__debug_deprecation_off:.`)

`\l__debug_internal_tl` For patching.

```

\l__debug_tmpa_tl
\l__debug_tmpb_tl
39213 \tl_new:N \l__debug_internal_tl
39214 \tl_new:N \l__debug_tmpa_tl
39215 \tl_new:N \l__debug_tmpb_tl

```

(End of definition for `\l__debug_internal_tl`, `\l__debug_tmpa_tl`, and `\l__debug_tmpb_tl`.)

`__debug_generate_parameter_list:NNN` Some functions don't take the arguments their signature indicates. For instance, `\clist_concat:NNN` doesn't take (directly) any argument, so patching it with something that uses `#1`, `#2`, or `#3` results in "Illegal parameter number in definition of `\clist_concat:NNN`".

Instead of changing *the* definition of the macros, we'll create a copy of such macros, say, `__debug_clist_concat:NNN` which will be defined as `<debug code with #1, #2 and #3>\clist_#1{#2}#3{#4}`. For that we need to identify the signature of every function and build the appropriate parameter list.

`__debug_generate_parameter_list:NNN` takes a function in `#1` and returns two parameter lists: `#2` contains the simple `#1#2#3` as would be used in the *(parameter text)* of the definition and `#3` contains the same parameters but with braces where necessary.

With the current implementation the resulting `#3` is, for example for `\some_function:NnNn`, `#1{#2}#3{#4}`. While this is correct, it might be unnecessary. Bracing everything will usually have the same outcome (unless the function was misused in the first place). What should be done?

```

39216 \cs_new_protected:Npn \__debug_generate_parameter_list:NNN #1#2#3
39217 {
39218     \__kernel_tl_set:Nx \l__debug_internal_tl
39219     { \exp_last_unbraced:Nf \use_ii:nmm \cs_split_function:N #1 }
39220     \__kernel_tl_set:Nx #2
39221     { \exp_args:NV \__debug_build_parm_text:n \l__debug_internal_tl }
39222     \__kernel_tl_set:Nx #3
39223     { \exp_args:NV \__debug_build_arg_list:n \l__debug_internal_tl }
39224 }
39225 \cs_new:Npn \__debug_build_parm_text:n #1
39226 {

```



```

39227     \__debug_arg_list_from_signature:nNN { 1 } \c_false_bool #1
39228     \q__debug_recursion_tail \q__debug_recursion_stop
39229   }
39230 \cs_new:Npn \__debug_build_arg_list:n #1
39231   {
39232     \__debug_arg_list_from_signature:nNN { 1 } \c_true_bool #1
39233     \q__debug_recursion_tail \q__debug_recursion_stop
39234   }
39235 \cs_new:Npn \__debug_arg_list_from_signature:nNN #1 #2 #3
39236   {
39237     \__debug_if_recursion_tail_stop:N #3
39238     \__debug_arg_check_invalid:N #3
39239     \bool_if:NT #2 { \__debug_arg_if_braced:NT #3 { \use_none:n } }
39240     \use:n { \c_hash_str \int_eval:n {#1} }
39241     \exp_args:Nf \__debug_arg_list_from_signature:nNN
39242       { \int_eval:n {#1+1} } #2
39243   }

```

Argument types w, p, T, and F shouldn't be included in the parameter lists, so we abort the loop if either is found.

```

39244 \cs_new:Npn \__debug_arg_check_invalid:N #1
39245   {
39246     \if:w w #1 \__debug_parm_terminate:w \else:
39247     \if:w p #1 \__debug_parm_terminate:w \else:
39248     \if:w T #1 \__debug_parm_terminate:w \else:
39249     \if:w F #1 \__debug_parm_terminate:w \else:
39250     \exp:w
39251     \fi:
39252     \fi:
39253     \fi:
39254     \fi:
39255     \exp_end:
39256   }
39257 \cs_new:Npn \__debug_parm_terminate:w
39258   { \exp_after:wN \__debug_use_none_delimit_by_q_recursion_stop:w \exp:w }
39259 \prg_new_conditional:Npnn \__debug_arg_if_braced:N #1 { T }
39260   { \exp_args:Nf \__debug_arg_if_braced:n { \__debug_get_base_form:N #1 } }
39261 \cs_new:Npn \__debug_arg_if_braced:n #1
39262   {
39263     \if:w n #1 \prg_return_true: \else:
39264     \if:w N #1 \prg_return_false: \else:
39265     \msg_expandable_error:nnn
39266       { debug } { bad-arg-type } {#1}
39267     \fi:
39268     \fi:
39269   }
39270 \msg_new:nnn { debug } { bad-arg-type }
39271   { Wrong~argument~type~#1. }

```

The macro below gets the base form of an argument type given a variant. It serves only to differentiate arguments which should be braced from ones which shouldn't. If all were to be braced this would be unnecessary. I moved the n and N variants to the beginning of the test as they are much more common here.

```

39272 \cs_new:Npn \__debug_get_base_form:N #1
39273   {

```

```

39274 \if:w n #1 \__debug_arg_return:N n \else:
39275 \if:w N #1 \__debug_arg_return:N N \else:
39276 \if:w c #1 \__debug_arg_return:N N \else:
39277 \if:w o #1 \__debug_arg_return:N n \else:
39278 \if:w V #1 \__debug_arg_return:N n \else:
39279 \if:w v #1 \__debug_arg_return:N n \else:
39280 \if:w f #1 \__debug_arg_return:N n \else:
39281 \if:w e #1 \__debug_arg_return:N n \else:
39282 \if:w x #1 \__debug_arg_return:N n \else:
39283 \__debug_arg_return:N \scan_stop:
39284 \fi:
39285 \fi:
39286 \fi:
39287 \fi:
39288 \fi:
39289 \fi:
39290 \fi:
39291 \fi:
39292 \fi:
39293 \exp_stop_f:
39294 }
39295 \cs_new:Npn \__debug_arg_return:N #1
39296 { \exp_after:wN #1 \exp:w \exp_end_continue_f:w }

```

(End of definition for __debug_generate_parameter_list:NNN and others.)

```

\__kernel_patch:nnn
\__kernel_patch_aux:nnn
\__debug_setup_debug_code:Nnn
\__debug_add_to_debug_code:Nnn
\__debug_insert_debug_code:Nnn
\__kernel_patch_weird:nnn
\__kernel_patch_weird_aux:nnn
\__debug_patch_weird:Nnn

```

Simple patching by adding material at the start and end of (a collection of) functions is straight-forward as we know the catcode set up. The approach is essentially that in `etoolbox`. Notice the need to worry about spaces: those are otherwise lost as normally in `expl3` code they would be `~`.

As discussed above, some functions don't take arguments, so we can't patch something that uses an argument in them. For these functions `__kernel_patch:nnn` is used. It starts by creating a copy of the function (say, `\clist_concat:NNN`) with a `__debug_` prefix in the name. This copy won't be changed. The code redefines the original function to take the exact same arguments as advertised in its signature (see `__debug_generate_parameter_list:NNN` above). The redefined function also contains the debug code in the proper position. If a function with the same name and the `__debug_` prefix was already defined, then the macro patches that definition by adding more debug code to it.

```

39297 \group_begin:
39298 \cs_set_protected:Npn \__kernel_patch:nnn
39299 {
39300 \group_begin:
39301 \char_set_catcode_other:N \#
39302 \__kernel_patch_aux:nnn
39303 }
39304 \cs_set_protected:Npn \__kernel_patch_aux:nnn #1#2#3
39305 {
39306 \char_set_catcode_parameter:N \#
39307 \char_set_catcode_space:N \%
39308 \tex_endlinechar:D -1 \scan_stop:
39309 \tl_map_inline:nn {#3}
39310 {

```

```

39311         \cs_if_exist:cTF { __debug_ \cs_to_str:N ##1 }
39312         { \__debug_add_to_debug_code:Nnn }
39313         { \__debug_setup_debug_code:Nnn }
39314         ##1 {#1} {#2}
39315     }
39316 \group_end:
39317 }
39318 \cs_set_protected:Npn \__debug_setup_debug_code:Nnn #1#2#3
39319 {
39320     \cs_gset_eq:cN { __debug_ \cs_to_str:N #1 } #1
39321     \__debug_generate_parameter_list:NNN #1 \l__debug_tmpa_tl \l__debug_tmpb_tl
39322     \exp_args:Ne \tex_scantokens:D
39323     {
39324         \tex_global:D \cs_prefix_spec:N #1
39325         \tex_def:D \exp_not:N #1
39326         \tl_use:N \l__debug_tmpa_tl
39327         {
39328             \tl_to_str:n {#2}
39329             \exp_not:c { __debug_ \cs_to_str:N #1 }
39330             \tl_use:N \l__debug_tmpb_tl
39331             \tl_to_str:n {#3}
39332         }
39333     }
39334 }
39335 \cs_set_protected:Npn \__debug_add_to_debug_code:Nnn #1#2#3
39336 {
39337     \use:e
39338     {
39339         \cs_set:Npn \exp_not:N \__debug_tmp:w
39340             ##1 \tl_to_str:n { macro: }
39341             ##2 \tl_to_str:n { -> }
39342             ##3 \c_backslash_str \tl_to_str:n { __debug_ }
39343                 \cs_to_str:N #1
39344             ##4 \s__debug_stop
39345         {
39346             \exp_not:N \exp_args:Ne \exp_not:N \tex_scantokens:D
39347             {
39348                 \tex_global:D ##1
39349                 \tex_def:D \exp_not:N #1 ##2
39350                 {
39351                     ##3 \tl_to_str:n {#2}
39352                     \c_backslash_str __debug_ \cs_to_str:N #1
39353                     ##4 \tl_to_str:n {#3}
39354                 }
39355             }
39356         }
39357     }
39358     \exp_after:wN \__debug_tmp:w \cs_meaning:N #1 \s__debug_stop
39359 }

```

Some functions, however, won't work with the signature reading setup above because their signature contains weird arguments. These functions need to be patched using `__kernel_patch_weird:nnn`, which won't make a copy of the function, rather it will patch the debug code directly into it. This means that whatever argument the debug

code uses must be actually used by the patched function.

```

39360 \cs_set_protected:Npn \__kernel_patch_weird:nnn
39361 {
39362   \group_begin:
39363     \char_set_catcode_other:N \#
39364     \__kernel_patch_weird_aux:nnn
39365   }
39366 \cs_set_protected:Npn \__kernel_patch_weird_aux:nnn #1#2#3
39367 {
39368   \char_set_catcode_parameter:N \#
39369   \char_set_catcode_space:N \ %
39370   \tex_endlinechar:D -1 \scan_stop:
39371   \tl_map_inline:mn {#3}
39372     { \__debug_patch_weird:Nnn ##1 {#1} {#2} }
39373   \group_end:
39374 }
39375 \cs_set_protected:Npn \__debug_patch_weird:Nnn #1#2#3
39376 {
39377   \use:e
39378   {
39379     \tex_endlinechar:D -1 \scan_stop:
39380     \exp_not:N \tex_scantokens:D
39381     {
39382       \tex_global:D \cs_prefix_spec:N #1
39383       \tex_def:D \exp_not:N #1
39384       \cs_parameter_spec:N #1
39385       {
39386         \tl_to_str:n {#2}
39387         \cs_replacement_spec:N #1
39388         \tl_to_str:n {#3}
39389       }
39390     }
39391   }
39392 }

```

(End of definition for __kernel_patch:nnn and others.)

Patching the second argument to ensure it exists. This happens before we alter #1 so the ordering is correct. For many variable types such as `int` a low-level error occurs when #2 is unknown, so adding a check is not needed.

```

39393 \__kernel_patch:nnn
39394 { \__kernel_chk_var_exist:N #2 }
39395 { }
39396 {
39397   \bool_set_eq:NN
39398   \bool_gset_eq:NN
39399   \clist_set_eq:NN
39400   \clist_gset_eq:NN
39401   \fp_set_eq:NN
39402   \fp_gset_eq:NN
39403   \prop_set_eq:NN
39404   \prop_gset_eq:NN
39405   \seq_set_eq:NN
39406   \seq_gset_eq:NN
39407   \str_set_eq:NN

```

```

39408     \str_gset_eq:NN
39409     \tl_set_eq:NN
39410     \tl_gset_eq:NN
39411 }

```

Patching both second and third arguments.

```

39412 \__kernel_patch:nnn
39413 {
39414     \__kernel_chk_var_exist:N #2
39415     \__kernel_chk_var_exist:N #3
39416 }
39417 { }
39418 {
39419     \clist_concat:NNN
39420     \clist_gconcat:NNN
39421     \prop_concat:NNN
39422     \prop_gconcat:NNN
39423     \seq_concat:NNN
39424     \seq_gconcat:NNN
39425     \str_concat:NNN
39426     \str_gconcat:NNN
39427     \tl_concat:NNN
39428     \tl_gconcat:NNN
39429 }
39430 \cs_gset_protected:Npn \__kernel_tl_set:Nx { \cs_set_nopar:Npe }
39431 \cs_gset_protected:Npn \__kernel_tl_gset:Nx { \cs_gset_nopar:Npe }

```

Patching where the first argument to a function needs scope-checking: either local or global (so two lists).

```

39432 \__kernel_patch:nnn
39433 { \__kernel_chk_var_local:N #1 }
39434 { }
39435 {
39436     \bool_set:Nn
39437     \bool_set_eq:NN
39438     \bool_set_true:N
39439     \bool_set_false:N
39440     \box_set_eq:NN
39441     \box_set_eq_drop:NN
39442     \box_set_to_last:N
39443     \clist_clear:N
39444     \clist_set_eq:NN
39445     \dim_zero:N
39446     \dim_set:Nn
39447     \dim_set_eq:NN
39448     \dim_add:Nn
39449     \dim_sub:Nn
39450     \fp_set_eq:NN
39451     \int_zero:N
39452     \int_set_eq:NN
39453     \int_add:Nn
39454     \int_sub:Nn
39455     \int_incr:N
39456     \int_decr:N

```

39457 \int_set:Nn
 39458 \hbox_set:Nn
 39459 \hbox_set_to_wd:Nnn
 39460 \hbox_set:Nw
 39461 \hbox_set_to_wd:Nnw
 39462 \muskip_zero:N
 39463 \muskip_set:Nn
 39464 \muskip_add:Nn
 39465 \muskip_sub:Nn
 39466 \muskip_set_eq:NN
 39467 \prop_clear:N
 39468 \prop_concat:NNN
 39469 \prop_pop:NnN
 39470 \prop_pop:NnNT
 39471 \prop_pop:NnNF
 39472 \prop_pop:NnNTF
 39473 \prop_put:Nnn
 39474 \prop_put_if_not_in:Nnn
 39475 \prop_put_from_keyval:Nn
 39476 \prop_remove:Nn
 39477 \prop_set_eq:NN
 39478 \prop_set_from_keyval:Nn
 39479 \seq_set_eq:NN
 39480 \skip_zero:N
 39481 \skip_set:Nn
 39482 \skip_set_eq:NN
 39483 \skip_add:Nn
 39484 \skip_sub:Nn
 39485 \str_clear:N
 39486 \str_set_eq:NN
 39487 \str_put_left:Nn
 39488 \str_put_right:Nn
 39489 __kernel_tl_set:Nx
 39490 \tl_clear:N
 39491 \tl_set_eq:NN
 39492 \tl_put_left:Nn
 39493 \tl_put_left:Nv
 39494 \tl_put_left:Nv
 39495 \tl_put_left:Ne
 39496 \tl_put_left:No
 39497 \tl_put_right:Nn
 39498 \tl_put_right:Nv
 39499 \tl_put_right:Nv
 39500 \tl_put_right:Ne
 39501 \tl_put_right:No
 39502 \tl_build_begin:N
 39503 \tl_build_put_right:Nn
 39504 \tl_build_put_left:Nn
 39505 \vbox_set:Nn
 39506 \vbox_set_top:Nn
 39507 \vbox_set_to_ht:Nnn
 39508 \vbox_set:Nw
 39509 \vbox_set_to_ht:Nnw
 39510 \vbox_set_split_to_ht:NNn

```

39511     }
39512 \__kernel_patch:nnn
39513 { \__kernel_chk_var_global:N #1 }
39514 { }
39515 {
39516   \bool_gset:Nn
39517   \bool_gset_eq:NN
39518   \bool_gset_true:N
39519   \bool_gset_false:N
39520   \box_gset_eq:NN
39521   \box_gset_eq_drop:NN
39522   \box_gset_to_last:N
39523   \cctab_gset:Nn
39524   \clist_gclear:N
39525   \clist_gset_eq:NN
39526   \dim_gset_eq:NN
39527   \dim_gzero:N
39528   \dim_gset:Nn
39529   \dim_gadd:Nn
39530   \dim_gsub:Nn
39531   \fp_gset_eq:NN
39532   \int_gzero:N
39533   \int_gset_eq:NN
39534   \int_gadd:Nn
39535   \int_gsub:Nn
39536   \int_gincr:N
39537   \int_gdecr:N
39538   \int_gset:Nn
39539   \hbox_gset:Nn
39540   \hbox_gset_to_wd:Nnn
39541   \hbox_gset:Nw
39542   \hbox_gset_to_wd:Nnw
39543   \muskip_gzero:N
39544   \muskip_gset:Nn
39545   \muskip_gadd:Nn
39546   \muskip_gsub:Nn
39547   \muskip_gset_eq:NN
39548   \prop_gclear:N
39549   \prop_gconcat:NNN
39550   \prop_gpop:NnN
39551   \prop_gpop:NnNT
39552   \prop_gpop:NnNF
39553   \prop_gpop:NnNTF
39554   \prop_gput:Nnn
39555   \prop_gput_if_not_in:Nnn
39556   \prop_gput_from_keyval:Nn
39557   \prop_gremove:Nn
39558   \prop_gset_eq:NN
39559   \prop_gset_from_keyval:Nn
39560   \seq_gset_eq:NN
39561   \skip_gzero:N
39562   \skip_gset:Nn
39563   \skip_gset_eq:NN
39564   \skip_gadd:Nn

```

```

39565     \skip_gsub:Nn
39566     \str_gclear:N
39567     \str_gset_eq:NN
39568     \str_gput_left:Nn
39569     \str_gput_right:Nn
39570     \__kernel_tl_gset:Nx
39571     \tl_gclear:N
39572     \tl_gset_eq:NN
39573     \tl_gput_left:Nn
39574     \tl_gput_left:NV
39575     \tl_gput_left:Nv
39576     \tl_gput_left:Ne
39577     \tl_gput_left:No
39578     \tl_gput_right:Nn
39579     \tl_gput_right:NV
39580     \tl_gput_right:Nv
39581     \tl_gput_right:Ne
39582     \tl_gput_right:No
39583     \tl_build_gbegin:N
39584     \tl_build_gput_right:Nn
39585     \tl_build_gput_left:Nn
39586     \vbox_gset:Nn
39587     \vbox_gset_top:Nn
39588     \vbox_gset_to_ht:Nnn
39589     \vbox_gset:Nw
39590     \vbox_gset_to_ht:Nnw
39591     \vbox_gset_split_to_ht:NNn
39592   }

```

Scoping for constants.

```

39593   \__kernel_patch:nnn
39594     { \__kernel_chk_var_scope:NN c #1 }
39595     { }
39596     {
39597       \bool_const:Nn
39598       \cctab_const:Nn
39599       \dim_const:Nn
39600       \int_const:Nn
39601       \intarray_const_from_clist:Nn
39602       \muskip_const:Nn
39603       \prop_const_from_keyval:Nn
39604       \prop_const_linked_from_keyval:Nn
39605       \skip_const:Nn
39606       \str_const:Nn
39607       \tl_const:Nn
39608     }

```

Flag functions.

```

39609   \__kernel_patch:nnn
39610     { \__kernel_chk_flag_exist:NN }
39611     { }
39612     {
39613       \flag_ensure_raised:N
39614       \flag_height:N
39615       \flag_if_raised:NT

```



```

39616     \flag_if_raised:N
39617     \flag_if_raised:NTF
39618     \flag_if_raised_p:N
39619     \flag_raise:N
39620 }

```

Various one-offs.

```

39621 \__kernel_patch:nnn
39622 { \__kernel_chk_cs_exist:N #1 }
39623 { }
39624 { \cs_generate_variant:Nn }
39625 \__kernel_patch:nnn
39626 { \__kernel_chk_var_scope:NN g #1 }
39627 { }
39628 { \cctab_new:N }
39629 \__kernel_patch:nnn
39630 { \__kernel_chk_var_scope:NN l #1 }
39631 { }
39632 { \flag_new:N }
39633 \__kernel_patch:nnn
39634 {
39635     \__kernel_chk_var_scope:NN l #1
39636     \__kernel_chk_flag_exist:NN
39637 }
39638 { }
39639 { \flag_clear:N }
39640 \__kernel_patch:nnn
39641 { \__kernel_chk_var_scope:NN g #1 }
39642 { }
39643 { \intarray_new:Nn }
39644 \__kernel_patch:nnn
39645 { \__kernel_chk_var_scope:NN q #1 }
39646 { }
39647 { \quark_new:N }
39648 \__kernel_patch:nnn
39649 { \__kernel_chk_var_scope:NN s #1 }
39650 { }
39651 { \scan_new:N }

```

Patch various internal commands to log definitions of functions. First, a kernel internal. Then internals from the cs, keys and msg modules.

```

39652 \__kernel_patch:nnn
39653 { }
39654 {
39655     \__kernel_debug_log:e
39656     { Defining~\token_to_str:N #1~ \msg_line_context: }
39657 }
39658 { \__kernel_chk_if_free_cs:N }
39659 <@@=cs>
39660 \__kernel_patch_weird:nnn
39661 {
39662     \cs_if_free:NF #4
39663     {
39664         \__kernel_debug_log:e
39665         {

```

```

39666         Variant~\token_to_str:N #4~%
39667         already~defined;~ not~ changing~ it~ \msg_line_context:
39668     }
39669 }
39670 }
39671 { }
39672 { \__cs_generate_variant:wwNN }
39673 (@@=keys)
39674 \__kernel_patch:nnn
39675 {
39676     \cs_if_exist:cF { \c__keys_code_root_str #1 }
39677     { \__kernel_debug_log:e { Defining~key~#1~\msg_line_context: } }
39678 }
39679 { }
39680 { \__keys_cmd_set_direct:nn }
39681 (@@=msg)
39682 \__kernel_patch:nnn
39683 { }
39684 {
39685     \__kernel_debug_log:e
39686     { Defining~message~ #1 / #2 ~\msg_line_context: }
39687 }
39688 { \__msg_chk_free:nn }
39689 (@@=prg)

```

Internal functions from prg module.

```

39690 \__kernel_patch_weird:nnn
39691 { \__kernel_chk_cs_exist:c { #5_p : #6 } }
39692 { }
39693 { \__prg_set_eq_conditional_p_form:wNnnnn }
39694 \__kernel_patch_weird:nnn
39695 { \__kernel_chk_cs_exist:c { #5 : #6 TF } }
39696 { }
39697 { \__prg_set_eq_conditional_TF_form:wNnnnn }
39698 \__kernel_patch_weird:nnn
39699 { \__kernel_chk_cs_exist:c { #5 : #6 T } }
39700 { }
39701 { \__prg_set_eq_conditional_T_form:wNnnnn }
39702 \__kernel_patch_weird:nnn
39703 { \__kernel_chk_cs_exist:c { #5 : #6 F } }
39704 { }
39705 { \__prg_set_eq_conditional_F_form:wNnnnn }
39706 (@@=regex)

```

Internal functions from regex module.

```

39707 \__kernel_patch:nnn
39708 {
39709     \__regex_trace_push:nnN { regex } { 1 } \__regex_escape_use:nnnn
39710     \group_begin:
39711         \__kernel_tl_set:Nx \l__regex_internal_a_tl
39712         { \__regex_trace_pop:nnN { regex } { 1 } \__regex_escape_use:nnnn }
39713         \use_none:nnn
39714     }
39715 { }

```

```

39716     { \__regex_escape_use:nnn }
39717 \__kernel_patch:nnn
39718   { \__regex_trace_push:nnN { regex } { 1 } \__regex_build:N }
39719   {
39720     \__regex_trace_states:n { 2 }
39721     \__regex_trace_pop:nnN { regex } { 1 } \__regex_build:N
39722   }
39723   { \__regex_build:N }
39724 \__kernel_patch:nnn
39725   { \__regex_trace_push:nnN { regex } { 1 } \__regex_build_for_cs:n }
39726   {
39727     \__regex_trace_states:n { 2 }
39728     \__regex_trace_pop:nnN { regex } { 1 } \__regex_build_for_cs:n
39729   }
39730   { \__regex_build_for_cs:n }
39731 \__kernel_patch:nnn
39732   {
39733     \__regex_trace:nne { regex } { 2 }
39734     {
39735       regex-new~state~
39736       L=\int_use:N \l__regex_left_state_int ~ -> ~
39737       R=\int_use:N \l__regex_right_state_int ~ -> ~
39738       M=\int_use:N \l__regex_max_state_int ~ -> ~
39739       \int_eval:n { \l__regex_max_state_int + 1 }
39740     }
39741   }
39742   { }
39743   { \__regex_build_new_state: }
39744 \__kernel_patch:nnn
39745   { \__regex_trace_push:nnN { regex } { 1 } \__regex_group_aux:nnnnN }
39746   { \__regex_trace_pop:nnN { regex } { 1 } \__regex_group_aux:nnnnN }
39747   { \__regex_group_aux:nnnnN }
39748 \__kernel_patch:nnn
39749   { \__regex_trace_push:nnN { regex } { 1 } \__regex_branch:n }
39750   { \__regex_trace_pop:nnN { regex } { 1 } \__regex_branch:n }
39751   { \__regex_branch:n }
39752 \__kernel_patch:nnn
39753   {
39754     \__regex_trace_push:nnN { regex } { 1 } \__regex_match:n
39755     \__regex_trace:nne { regex } { 1 } { analyzing~query~token~list }
39756   }
39757   { \__regex_trace_pop:nnN { regex } { 1 } \__regex_match:n }
39758   { \__regex_match:n }
39759 \__kernel_patch:nnn
39760   {
39761     \__regex_trace_push:nnN { regex } { 1 } \__regex_match_cs:n
39762     \__regex_trace:nne { regex } { 1 } { analyzing~query~token~list }
39763   }
39764   { \__regex_trace_pop:nnN { regex } { 1 } \__regex_match_cs:n }
39765   { \__regex_match_cs:n }
39766 \__kernel_patch:nnn
39767   { \__regex_trace:nne { regex } { 1 } { initializing } }
39768   { }
39769   { \__regex_match_init: }

```

```

39770 \__kernel_patch:nnn
39771 {
39772   \__regex_trace:nne { regex } { 2 }
39773   { state~\int_use:N \l__regex_curr_state_int }
39774 }
39775 { }
39776 { \__regex_use_state: }
39777 \__kernel_patch:nnn
39778 { \__regex_trace_push:nnN { regex } { 1 } \__regex_replacement:n }
39779 { \__regex_trace_pop:nnN { regex } { 1 } \__regex_replacement:n }
39780 { \__regex_replacement:n }
39781 \group_end:
39782 (@@=debug)

```

Patching arguments is a bit more involved: we do these one at a time. The basic idea is the same, using a # token that is a string.

```

39783 \group_begin:
39784 \cs_set_protected:Npn \__kernel_patch:Nn #1
39785 {
39786   \group_begin:
39787   \char_set_catcode_other:N \#
39788   \__kernel_patch_aux:Nn #1
39789 }
39790 \cs_set_protected:Npn \__kernel_patch_aux:Nn #1#2
39791 {
39792   \char_set_catcode_parameter:N \#
39793   \tex_endlinechar:D -1 \scan_stop:
39794   \exp_args:Ne \tex_scantokens:D
39795   {
39796     \tex_global:D \cs_prefix_spec:N #1 \tex_def:D \exp_not:N #1
39797     \cs_parameter_spec:N #1
39798     { \exp_args:No \tl_to_str:n { #1 #2 } }
39799   }
39800   \group_end:
39801 }

```

The functions here can get a bit repetitive, so we define a helper which can reuse the same patch code repeatedly. The main part of the patch is the same, so we just have to deal with the part which varies depending on the type of expression.

```

39802 \cs_set_protected:Npn \__kernel_patch_eval:nn #1#2
39803 {
39804   \tl_map_inline:nn {#1}
39805   {
39806     \exp_args:NNe \__kernel_patch:Nn ##1
39807     {
39808       { \c_hash_str 1 }
39809       {
39810         \exp_not:N \__kernel_chk_expr:nNn { \c_hash_str 2 }
39811         \exp_not:n {#2}
39812         \exp_not:N ##1
39813       }
39814     }
39815   }
39816 }

```

```

39817 <@@=dim>
39818 \__kernel_patch_eval:nn
39819 {
39820   \dim_set:Nn
39821   \dim_gset:Nn
39822   \dim_add:Nn
39823   \dim_gadd:Nn
39824   \dim_sub:Nn
39825   \dim_gsub:Nn
39826   \dim_const:Nn
39827 }
39828 { \__dim_eval:w { } }
39829 <@@=int>
39830 \__kernel_patch_eval:nn
39831 {
39832   \int_set:Nn
39833   \int_gset:Nn
39834   \int_add:Nn
39835   \int_gadd:Nn
39836   \int_sub:Nn
39837   \int_gsub:Nn
39838   \int_const:Nn
39839 }
39840 { \__int_eval:w { } }
39841 \__kernel_patch_eval:nn
39842 {
39843   \muskip_set:Nn
39844   \muskip_gset:Nn
39845   \muskip_add:Nn
39846   \muskip_gadd:Nn
39847   \muskip_sub:Nn
39848   \muskip_gsub:Nn
39849   \muskip_const:Nn
39850 }
39851 { \tex_muexpr:D { \tex_mutogluue:D } }
39852 \__kernel_patch_eval:nn
39853 {
39854   \skip_set:Nn
39855   \skip_gset:Nn
39856   \skip_add:Nn
39857   \skip_gadd:Nn
39858   \skip_sub:Nn
39859   \skip_gsub:Nn
39860   \skip_const:Nn
39861 }
39862 { \tex_glueexpr:D { } }

```

Patching expandable expressions, first the one-argument versions, then the two-argument ones.

```

39863 \cs_set_protected:Npn \__kernel_patch_eval:nn #1#2
39864 {
39865   \tl_map_inline:nn {#1}
39866   {
39867     \exp_args:NNe \__kernel_patch:Nn ##1

```

```

39868         {
39869         {
39870         \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
39871         \exp_not:n {#2}
39872         \exp_not:N ##1
39873         }
39874     }
39875 }
39876 }
39877 <@@=box>
39878 \__kernel_patch_eval:nn
39879 { \__box_dim_eval:n }
39880 { \__box_dim_eval:w { } }
39881 <@@=dim>
39882 \__kernel_patch_eval:nn
39883 {
39884     \dim_eval:n
39885     \dim_to_decimal:n
39886     \dim_to_decimal_in_sp:n
39887     \dim_abs:n
39888     \dim_sign:n
39889 }
39890 { \__dim_eval:w { } }
39891 <@@=int>
39892 \__kernel_patch_eval:nn
39893 {
39894     \int_eval:n
39895     \int_abs:n
39896     \int_sign:n
39897 }
39898 { \__int_eval:w { } }
39899 \__kernel_patch_eval:nn
39900 {
39901     \skip_eval:n
39902     \skip_horizontal:n
39903     \skip_vertical:n
39904 }
39905 { \tex_glueexpr:D { } }
39906 \__kernel_patch_eval:nn
39907 {
39908     \muskip_eval:n
39909 }
39910 { \tex_muexpr:D { \tex_mutogluue:D } }
39911 \cs_set_protected:Npn \__kernel_patch_eval:nn #1#2
39912 {
39913     \tl_map_inline:nn {#1}
39914     {
39915         \exp_args:NNe \__kernel_patch:Nn ##1
39916         {
39917             {
39918                 \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
39919                 \exp_not:n {#2}
39920                 \exp_not:N ##1
39921             }

```

```

39922         {
39923             \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 2 }
39924             \exp_not:n {#2}
39925             \exp_not:N ##1
39926         }
39927     }
39928 }
39929 }
39930 <@@=dim>
39931 \__kernel_patch_eval:nn
39932 {
39933     \dim_max:nn
39934     \dim_min:nn
39935 }
39936 { \__dim_eval:w { } }
39937 <@@=int>
39938 \__kernel_patch_eval:nn
39939 {
39940     \int_max:nn
39941     \int_min:nn
39942     \int_div_truncate:nn
39943     \int_mod:nn
39944 }
39945 { \__int_eval:w { } }

```

Conditionals: three argument ones then one argument ones

```

39946 \cs_set_protected:Npn \__kernel_patch_cond:nn #1#2
39947 {
39948     \clist_map_inline:nn { :nNnT , :nNnF , :nNnTF , _p:nNn }
39949     {
39950         \exp_args:Nce \__kernel_patch:Nn { #1 ##1 }
39951         {
39952             {
39953                 \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
39954                 \exp_not:n {#2}
39955                 \exp_not:c { #1 ##1 }
39956             }
39957             { \c_hash_str 2 }
39958             {
39959                 \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 3 }
39960                 \exp_not:n {#2}
39961                 \exp_not:c { #1 ##1 }
39962             }
39963         }
39964     }
39965 }
39966 <@@=dim>
39967 \__kernel_patch_cond:nn { dim_compare } { \__dim_eval:w { } }
39968 <@@=int>
39969 \__kernel_patch_cond:nn { int_compare } { \__int_eval:w { } }
39970 \cs_set_protected:Npn \__kernel_patch_cond:nn #1#2
39971 {
39972     \clist_map_inline:nn { :nT , :nF , :nTF , _p:n }
39973     {
39974         \exp_args:Nce \__kernel_patch:Nn { #1 ##1 }

```

```

39975         {
39976         {
39977         \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
39978         \exp_not:n {#2}
39979         \exp_not:c { #1 ##1 }
39980         }
39981         }
39982     }
39983 }
39984 <@@=int>
39985 \__kernel_patch_cond:nn { int_if_even } { \__int_eval:w { } }
39986 \__kernel_patch_cond:nn { int_if_odd } { \__int_eval:w { } }

```

Step functions.

```

39987 <@@=dim>
39988 \__kernel_patch:Nn \dim_step_function:nnnN
39989 {
39990     {
39991         \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { }
39992         \dim_step_function:nnnN
39993     }
39994     {
39995         \__kernel_chk_expr:nNnN {#2} \__dim_eval:w { }
39996         \dim_step_function:nnnN
39997     }
39998     {
39999         \__kernel_chk_expr:nNnN {#3} \__dim_eval:w { }
40000         \dim_step_function:nnnN
40001     }
40002 }
40003 <@@=int>
40004 \__kernel_patch:Nn \int_step_function:nnnN
40005 {
40006     {
40007         \__kernel_chk_expr:nNnN {#1} \__int_eval:w { }
40008         \int_step_function:nnnN
40009     }
40010     {
40011         \__kernel_chk_expr:nNnN {#2} \__int_eval:w { }
40012         \int_step_function:nnnN
40013     }
40014     {
40015         \__kernel_chk_expr:nNnN {#3} \__int_eval:w { }
40016         \int_step_function:nnnN
40017     }
40018 }

```

Odds and ends

```

40019 \__kernel_patch:Nn \dim_to_fp:n { { (#1) } }
40020 \group_end:
40021 <@@=skip>

```

This one has catcode changes so must be done by hand.

```

40022 \cs_set_protected:Npn \__skip_tmp:w #1

```



```

40023 {
40024   \prg_set_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
40025   {
40026     \exp_after:wN \__skip_if_finite:wwNw
40027     \skip_use:N \tex_glueexpr:D
40028     \__kernel_chk_expr:nNnN
40029     {##1} \tex_glueexpr:D { } \skip_if_finite:n
40030     ; \prg_return_false:
40031     #1 ; \prg_return_true: \s__skip_stop
40032   }
40033 }
40034 \exp_args:No \__skip_tmp:w { \tl_to_str:n { fil } }
40035 <@@=msg>
      Messages.
40036 \msg_new:nnnn { debug } { debug }
40037 { The-debugging-option~'#1'~does~not~exist~\msg_line_context:. }
40038 {
40039   The-functions~'\iow_char:N\debug_on:n'~and~
40040   '\iow_char:N\debug_off:n'~only~accept~the~arguments~
40041   'all',~'check-declarations',~'check-expressions',~
40042   'deprecation',~'log-functions',~not~'#1'.
40043 }
40044 \msg_new:nnn { debug } { expr } { '#2'~in~#1 }
40045 \msg_new:nnnn { debug } { local-global }
40046 { Inconsistent~local/global~assignment }
40047 {
40048   \c__msg_coding_error_text_tl
40049   \if:w l #2 Local
40050   \else:
40051     \if:w g #2 Global \else: Constant \fi:
40052   \fi:
40053   \ %
40054   assignment~to~a~
40055   \if:w l #1 local
40056   \else:
40057     \if:w g #1 global \else: constant \fi:
40058   \fi:
40059   \ %
40060   variable~'#3'.
40061 }
40062 \msg_new:nnnn { debug } { non-declared-variable }
40063 { The-variable~#1~has~not~been~declared~\msg_line_context:. }
40064 {
40065   \c__msg_coding_error_text_tl
40066   Checking-is~active,~and~you~have~tried~do~so~something~like: \\
40067   \\ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\
40068   without~first~having: \\
40069   \\ \tl_new:N ~ #1 \\
40070   \\
40071   LaTeX~will~continue,~creating~the~variable~where~it~is~the~one~being~set.
40072 }

```

__kernel_if_debug:TF Flip the switch for deprecated code.

40073 \cs_set_protected:Npn __kernel_if_debug:TF #1#2 {#1}

(End of definition for __kernel_if_debug:TF.)

40074 \endpackage

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
$\backslash!$	19942, 19958
$!$	<i>274</i>
$\backslash"$	19158, 19161, 32057, 34380, 34398, 34422, 34428, 34432, 34438, 34442, 34448, 34454, 34461, 34462, 34468, 34472, 34474, 34583
$\backslash\#$..	10674, 14143, 19158, 34317, 39301, 39306, 39363, 39368, 39787, 39792
$\backslash\$$	5260, 14142, 19158, 19161, 34317
$\backslash\%$	10676, 14144, 19158, 34317
$\backslash\&$	9378, 14135, 19158, 19161
$\&\&$	<i>273</i>
\backslash'	32057, 34380, 34392, 34419, 34426, 34430, 34435, 34440, 34443, 34445, 34452, 34457, 34458, 34465, 34470, 34473, 34481, 34482, 34529, 34530, 34537, 34538, 34549, 34550, 34555, 34556, 34584, 34585, 34607, 34608, 34611, 34612, 34613, 34614
$\backslash($	31669
$\backslash)$	31669
$\backslash*$	9139, 9151, 13735, 13758, 19339, 19341, 19345, 19353
$*$	<i>274</i>
$**$	<i>274</i>
$+$	<i>274</i>
$\backslash,$	21471, 34329
$\backslash-$	150
$-$	<i>274</i>
$\backslash.$	32057, 34380, 34397, 34485, 34486, 34495, 34496, 34505, 34506, 34523, 34535, 34536, 34586, 34587, 34617, 34618, 34621, 34622
$\backslash/$	149, 4143
$/$	<i>274</i>
$\backslash:$	14141
$\backslash::$	<i>43</i> , <i>406</i> , <i>425</i> , 2392, 2393, <u>2394</u> , 2395, 2396, 2397, 2398, 2400, 2402, 2403, 2404, 2411, 2414, 2417, 2423, 2579, 2581, 2586, 2591, 2593, 2598, 2650, 2651, 2652, 2653, 2654, 2665
$\backslash::N$	<i>43</i> , <u>2396</u> , 2654
$\backslash::V$	<i>43</i> , <u>2417</u>
$\backslash::V_{\text{unbraced}}$	<i>43</i> , <u>2578</u>
$\backslash::c$	<i>43</i> , <u>2398</u>
$\backslash::e$	<i>43</i> , <u>2402</u>
$\backslash::e_{\text{unbraced}}$	<i>43</i> , <u>2578</u>
$\backslash::f$	<i>43</i> , <u>2404</u> , 2653
$\backslash::f_{\text{unbraced}}$	<i>43</i> , <u>2578</u> , 2651
$\backslash::n$	<i>43</i> , <i>815</i> , <u>2395</u> , 2650, 2651, 2654
$\backslash::o$	<i>43</i> , <u>2400</u> , 2652
$\backslash::o_{\text{unbraced}}$ <i>43</i> , <u>2578</u> , 2650, 2652, 2653, 2654
$\backslash::p$	<i>43</i> , <i>406</i> , <u>2397</u>
$\backslash::v$	<i>43</i> , <u>2417</u>
$\backslash::v_{\text{unbraced}}$	<i>43</i> , <u>2578</u>
$\backslash::x$	<i>43</i> , <u>2411</u>
$\backslash::x_{\text{unbraced}}$	<i>43</i> , <u>2578</u> , 2665
$<$	<i>274</i>
$\backslash=$	21472, 32057, 34380, 34395, 34475, 34476, 34491, 34492, 34514, 34515, 34516, 34543, 34544, 34569, 34570, 34623, 34624
$=$	<i>274</i>
$>$	<i>274</i>
$?$	<i>274</i>
$?:$	<i>273</i>
$\backslash???$	<i>88</i> , <i>636</i>
$\backslash\backslash$..	2310, 3491, 3494, 3495, 3519, 3520, 3527, 3528, 4044, 4273, 4597, 4598, 5994, 6001, 6002, 6003, 6127, 7953, 7957, 7962, 7996, 8005, 8009, 8014, 8034, 8036, 8037, 8039, 8042, 8044, 8049, 8051, 8053, 8058, 8062, 8065, 8069, 8071, 8075, 8077, 8083, 8085, 8089, 8091, 8095, 8100, 8102, 8144, 8146, 8151, 8153, 8159, 8164, 8165, 8169, 8173, 8183, 8186, 8190, 8191, 8195, 8203, 8229, 8274, 9281, 9299, 9301, 9306, 9307, 9331, 9341, 9348, 9363, 9784, 9792, 9799, 9811, 9812, 9827, 9828, 9835, 9855, 9858, 9859, 9891, 9919, 9952, 9953, 9966, 10023, 10024, 10032, 10047, 10048, 10077, 10088, 10092, 10098, 10105, 10128, 10279, 10679, 11665, 11669, 11670, 11672, 11678, 11680, 11685, 11686, 11688, 11689, 11691, 11693, 11705, 11715, 11717, 11718, 11719, 11817, 11818, 14137, 14694, 14695, 14698, 15020, 15023, 15024, 15025, 15026,

- 15031, 15037, 15042, 15049, 15208,
15211, 15212, 15213, 15215, 15221,
15226, 15231, 15382, 15389, 19158,
22727, 22739, 22745, 23734, 23737,
23738, 23739, 23746, 23749, 23750,
30349, 30350, 30357, 30735, 30737,
30738, 30741, 30743, 30744, 30747,
30749, 30750, 30751, 30755, 30762,
34315, 36683, 38287, 38289, 38316,
38318, 38338, 38340, 38361, 38363,
38370, 38372, 38379, 38381, 38387,
38400, 38402, 39169, 40039, 40040,
40066, 40067, 40068, 40069, 40070
- \backslash { 4546, 7957, 7962,
8009, 8051, 8053, 8065, 8102, 8191,
8195, 10673, 14138, 14699, 19158,
31709, 31710, 31711, 34317, 40067
- \backslash } 64, 7956, 7962, 8066, 8102, 8191, 8195,
10675, 14139, 14699, 19158, 31709,
31710, 31711, 31712, 34317, 40067
- \backslash ⌊ 64, 66, 69, 71, 148, 1804,
3541, 3732, 4104, 4491, 4496, 4540,
4550, 4735, 7213, 9126, 9829, 10009,
10680, 11689, 13735, 13758, 14699,
15023, 15024, 15025, 19158, 19279,
19282, 30780, 30786, 30797, 30823,
31207, 31707, 31708, 34328, 39307,
39369, 40053, 40059, 40067, 40069
- \backslash ˆ 59, 1958, 2688, 3591, 3611,
3688, 3691, 4492, 4497, 4498, 4499,
4500, 4503, 4514, 4551, 4605, 4607,
4609, 4611, 4613, 4615, 5259, 7164,
7167, 7181, 7184, 7193, 7196, 7199,
7202, 7216, 7219, 8619, 8622, 9385,
10588, 10627, 14140, 14813, 14814,
15149, 15150, 15331, 15332, 15333,
19158, 19161, 19163, 19169, 19216,
27674, 32057, 34380, 34393, 34420,
34427, 34431, 34436, 34441, 34446,
34453, 34459, 34460, 34466, 34471,
34483, 34484, 34501, 34502, 34509,
34510, 34524, 34525, 34526, 34557,
34558, 34579, 34580, 34581, 34582
- \backslash ˘ 274
- \backslash ⌋ 14146, 19158, 19161, 34317
- \backslash ˘ 32057,
34380, 34391, 34418, 34425, 34429,
34434, 34439, 34444, 34451, 34455,
34456, 34464, 34469, 34609, 34610
- \backslash || 273
- \backslash ˘ 64, 4536, 4540, 4546,
10677, 12424, 14145, 19158, 19161,
32057, 34321, 34380, 34394, 34421,
34433, 34437, 34447, 34463, 34467,
34511, 34512, 34513, 34567, 34568
- ### A
- \backslash A 13736, 13759
- \backslash AA 32061, 33645, 34341
- \backslash aa 32061, 33645, 34351
- \backslash above 151
- \backslash abovedisplaysshortskip 152
- \backslash abovedisplayskip 153
- \backslash abovewithdelims 154
- abs 274
- \backslash accent 155
- acos 277
- acosc 277
- acot 278
- acotd 278
- acsc 277
- acscd 277
- \backslash adjdemerits 156
- \backslash adjustspacing 932
- \backslash advance 157
- \backslash AE 32062, 33646, 34342, 34611
- \backslash ae 32062, 33646, 34352, 34612
- \backslash afterassignment 158
- \backslash aftergroup 159
- \backslash alignmark 780
- \backslash aligntab 781
- asec 277
- asecd 277
- asin 277
- asind 277
- atan 278
- atand 278
- \backslash AtBeginDocument 677, 11577
- \backslash atop 160
- \backslash atopwithdelims 161
- \backslash attribute 782
- \backslash attributedef 783
- \backslash automaticdiscretionary 784
- \backslash automatichyphenmode 786
- \backslash automatichyphenpenalty 787
- \backslash autospacing 1134
- \backslash autoxspacing 1135
- ### B
- \backslash b 32057, 34380, 34405
- \backslash babelshorthand 31664
- \backslash badness 162
- \backslash baselineskip 163
- \backslash batchmode 164
- \backslash begin 10129, 31662, 31672, 34312
- \backslash begincsname 789
- \backslash begingroup 3, 7, 12, 16, 35, 63, 68, 142, 165
- \backslash beginL 473

- \beginR 474
- \belowdisplayskip 166
- \belowdisplayskip 167
- \bfseries 34291
- \binoppenalty 168
- bitset commands:
 - \bitset_addto_named_index:Nn ...
..... 286, 30180, 30180
 - \bitset_clear:N
..... 287, 30270, 30270, 30278
 - \bitset_gclear:N
..... 287, 30270, 30274, 30279
 - \bitset_gset_false:Nn
..... 287, 30236, 30242, 30269
 - \bitset_gset_true:Nn
..... 287, 30236, 30238, 30267
 - \bitset_if_exist:N 30186, 30188
 - \bitset_if_exist:NTF 287, 30185
 - \bitset_if_exist_p:N 287, 30185
 - \bitset_item:Nn
..... 287, 30300, 30300, 30315
 - \bitset_log:N 288, 30316, 30318, 30319
 - \bitset_log_named_index:N
..... 288, 30331, 30334, 30336
 - \bitset_new:N 286, 30163, 30163, 30178
 - \bitset_new:Nn 286, 30163, 30169, 30179
 - \bitset_set_false:Nn
..... 287, 30236, 30240, 30268
 - \bitset_set_true:Nn
..... 287, 30236, 30236, 30266
 - \bitset_show:N
..... 287, 288, 30316, 30316, 30317
 - \bitset_show_named_index:N
..... 288, 30331, 30331, 30333
 - \bitset_to_arabic:N 285,
288, 1254, 30280, 30280, 30298, 30327
 - \bitset_to_bin:N
..... 286, 288, 30280, 30294, 30299, 30326
- bitset internal commands:
 - __bitset_gset_false:Nn
..... 30189, 30195, 30243
 - __bitset_gset_true:Nn
..... 30189, 30191, 30239
 - \l__bitset_internal_int
..... 30220, 30224, 30228
 - __bitset_set:NNn
.. 30237, 30239, 30241, 30243, 30244
 - __bitset_set:NNnN 30189,
30190, 30192, 30194, 30196, 30197
 - __bitset_set_aux:NNn 30236
 - __bitset_set_false:Nn
..... 30189, 30193, 30241
 - __bitset_set_true:Nn
..... 30189, 30189, 30237
- __bitset_show:NN 30316, 30318, 30320
- __bitset_show_named_index:NN ...
..... 30332, 30335, 30337
- __bitset_test_digits:n 30221
- __bitset_test_digits:nTF
..... 30221, 30254
- __bitset_test_digits:w
..... 30221, 30223, 30235
- __bitset_test_digits_end:
..... 30225, 30227, 30234, 30235
- __bitset_test_digits_end:n .. 30221
- __bitset_to_int:nN
..... 30280, 30285, 30289, 30292
- \bodydir 790
- \bodydirection 791
- bool commands:
 - \bool_case:n
..... 71, 8540, 8546, 38857, 38858
 - \bool_case:nTF
71, 8540, 8540, 8542, 8544, 38859,
38860, 38861, 38862, 38863, 38864
 - \bool_case_true:n 38857, 38858
 - \bool_case_true:nTF
..... 38857, 38860, 38862, 38864
 - \bool_const:Nn
..... 66, 8283, 8283, 8288, 39597
 - \bool_do_until:Nn
..... 70, 8506, 8508, 8509, 8511
 - \bool_do_until:nn 70, 8512, 8533, 8536
 - \bool_do_while:Nn
..... 70, 8506, 8506, 8507, 8510
 - \bool_do_while:nn 70, 8512, 8520, 8523
 - .bool_gset:N 241, 22015
 - \bool_gset:Nn
..... 66, 8305, 8310, 8316, 39516
 - \bool_gset_eq:NN 66, 4482,
6648, 8301, 8302, 8304, 39398, 39517
 - \bool_gset_false:N 66, 6596, 8289,
8295, 8300, 8321, 14351, 14360, 39519
 - .bool_gset_inverse:N 241, 22023
 - \bool_gset_inverse:N
..... 66, 8317, 8320, 8322
 - \bool_gset_true:N
..... 66, 6661, 8289, 8293,
8299, 8321, 8854, 14341, 38451,
38472, 38552, 38635, 38853, 39518
 - \bool_if:N 8328, 8336
 - \bool_if:n 8379
 - \bool_if:NTF 67, 108, 2154,
5375, 5384, 5825, 5998, 6084, 6102,
6120, 6271, 6490, 6498, 6730, 7340,
7363, 7436, 7663, 7830, 7836, 7877,
8247, 8252, 8318, 8321, 8328, 8339,
8398, 8501, 8503, 8507, 8509, 8852,

- 10894, 10901, 14355, 14364, 20165,
21508, 21603, 21692, 21936, 21945,
21991, 22236, 22347, 22388, 22404,
22406, 22411, 22418, 22482, 22488,
22523, 22533, 22561, 32495, 35855,
36550, 36895, 38456, 38684, 39239
- \bool_if:nTF 66, 69–71, 913,
6087, 8345, 8379, 8451, 8458, 8477,
8484, 8493, 8514, 8523, 8527, 8536,
8555, 8634, 11775, 12119, 12124, 16784
- \bool_if_exist:N 8375, 8377
- \bool_if_exist:NTF . . . 67, 8375, 21715
- \bool_if_exist_p:N 67, 8375
- \bool_if_p:N 67, 8328
- \bool_if_p:n
. 69, 592, 8286, 8308, 8313, 8379,
8387, 8387, 8458, 8484, 8490, 8494
- \bool_lazy_all:n 8440
- \bool_lazy_all:nTF
. 68, 69, 5658, 8438, 38707
- \bool_lazy_all_p:n 69, 8438
- \bool_lazy_and:nn 8455
- \bool_lazy_and:nnTF 68,
69, 8455, 8706, 8993, 10441, 11650,
30554, 31401, 31673, 31827, 31934,
31996, 32526, 32672, 33440, 33503,
33541, 33790, 34197, 35575, 36463,
36877, 38444, 38645, 38716, 38728
- \bool_lazy_and_p:nn
. 69, 8455, 32484, 33595, 38656, 38668
- \bool_lazy_any:n 8466
- \bool_lazy_any:nTF 68, 69,
8464, 11253, 14193, 14467, 14491,
14513, 14683, 31534, 31687, 33213
- \bool_lazy_any_p:n
. 69, 8464, 31830, 33447
- \bool_lazy_or:nn 8481
- \bool_lazy_or:nnTF
. 68, 69, 3597, 3619, 8481,
8689, 8817, 11198, 15467, 30684,
30710, 30774, 30954, 31436, 31545,
31585, 31987, 32126, 32481, 32581,
32774, 32838, 32963, 33282, 33522,
33593, 33697, 33844, 34012, 34236,
36897, 37227, 38653, 38665, 38724
- \bool_lazy_or_p:nn 69, 8481, 31999,
32675, 33442, 33505, 33544, 34200
- \bool_log:N 67, 8352, 8354, 8355
- \bool_log:n 67, 8348, 8350
- \bool_new:N 66, 4341, 4782,
6565, 6566, 6568, 6569, 6570, 8281,
8281, 8282, 8371, 8372, 8373, 8374,
8849, 10622, 14204, 21347, 21575,
21576, 21583, 21584, 21589, 21592,
21715, 32077, 35448, 36708, 38443
- \bool_not_p:n 69, 8490, 8490
- .bool_set:N 241, 22015
- \bool_set:Nn
. 66, 584, 588, 8305, 8305, 8315, 39436
- \bool_set_eq:NN 66, 4476, 6809, 8301,
8301, 8303, 20171, 20173, 39397, 39437
- \bool_set_false:N 66, 122,
5349, 5554, 6540, 6611, 6625, 6687,
6729, 8289, 8291, 8298, 8318, 10728,
10870, 10878, 10886, 10896, 10903,
21633, 22252, 22253, 22254, 22264,
22265, 22280, 22300, 22301, 22320,
22330, 22395, 35851, 36548, 39439
- .bool_set_inverse:N 241, 22023
- \bool_set_inverse:N
. 66, 8317, 8317, 8319
- \bool_set_true:N
. 66, 135, 5354, 5558, 6534,
6727, 6808, 8289, 8289, 8297, 8318,
10856, 21628, 22263, 22281, 22282,
22302, 22317, 22325, 22400, 32078,
35869, 35891, 35922, 37420, 39438
- \bool_show:N 67, 8352, 8352, 8353
- \bool_show:n 67, 8348, 8348
- \bool_to_str:N . . . 67, 8337, 8337, 8342
- \bool_to_str:n
. 67, 8337, 8343, 8349, 8351
- \bool_until_do:Nn
. 70, 8500, 8502, 8503, 8505
- \bool_until_do:nn 70, 8512, 8525, 8530
- \bool_while_do:Nn
. 70, 8500, 8500, 8501, 8504
- \bool_while_do:nn 71, 8512, 8512, 8517
- \bool_xor:nn 8491
- \bool_xor:nnTF 70, 8491
- \bool_xor_p:nn 70, 8491
- \c_false_bool 66, 67,
388, 418, 570, 585, 588–590, 1653,
1705, 1706, 1737, 1761, 1766, 1798,
1817, 2091, 2098, 2745, 3005, 5037,
5055, 5247, 5294, 5593, 5795, 5812,
5825, 6021, 6157, 6658, 7765, 7774,
7783, 7793, 7855, 7863, 8281, 8292,
8296, 8363, 8398, 8429, 8452, 8458,
8476, 8645, 20167, 21959, 21961,
21968, 21973, 38459, 38712, 39227
- \g_tmpa_bool 68, 8371
- \l_tmpa_bool 67, 8371
- \g_tmpb_bool 68, 8371
- \l_tmpb_bool 67, 8371
- \c_true_bool 66, 67, 388, 585, 588–
590, 708, 1705, 1737, 1798, 1816,

- 2112, 4348, 4479, 4920, 4994, 5051,
5237, 5239, 5241, 5243, 5245, 5255,
5293, 5300, 5793, 5803, 5825, 5826,
6019, 6140, 6142, 6165, 6257, 6428,
6439, 6454, 6615, 7387, 7504, 7617,
8290, 8294, 8362, 8398, 8430, 8431,
8450, 8478, 8484, 8551, 8639, 20166,
20171, 21966, 21975, 38712, 39232
- bool internal commands:
- _bool_!:Nw 8409
 - _bool_&_0: 8421
 - _bool_&_1: 8421
 - _bool_&_2: 8421
 - _bool_(:Nw 8414
 - _bool_)_0: 8421
 - _bool_)_1: 8421
 - _bool_)_2: 8421
 - _bool_case:NnTF 8540
 - _bool_case:nTF
..... 8541, 8543, 8545, 8547, 8548
 - _bool_case:w 8540, 8550, 8553, 8557
 - _bool_case_end:nw 8556, 8559
 - _bool_choose:NNN
..... 8416, 8420, 8421, 8421
 - _bool_get_next:NN
589, 8395, 8399, 8399, 8411, 8417,
8432, 8433, 8434, 8435, 8436, 8437
 - _bool_if_p:n 8387, 8387, 8388
 - _bool_if_p_aux:w
..... 588, 8387, 8390, 8397
 - _bool_if_recursion_tail_stop_-
do:nn 8327, 8327, 8450, 8476
 - _bool_lazy_all:n
..... 8438, 8439, 8448, 8453
 - _bool_lazy_any:n
..... 8464, 8465, 8474, 8479
 - _bool_p:Nw 8419
 - _bool_show:NN 8352, 8352, 8354, 8356
 - _bool_use_i_delimit_by_q_-
recursion_stop:nw
..... 8325, 8325, 8452, 8478
 - _bool_|_0: 8421
 - _bool_|_1: 8421
 - _bool_|_2: 8421
- \botmark 169
- \botmarks 475
- \boundary 792
- \box 170
- box commands:
- \box_autosize_to_wd_and_ht:Nnn ..
..... 311, 35244, 35244, 35246
 - \box_autosize_to_wd_and_ht_plus_-
dp:Nnn ... 311, 35244, 35250, 35255
 - \box_clear:N 302, 303, 34642, 34642,
34646, 34649, 35483, 35570, 35647
 - \box_clear_new:N
..... 303, 34648, 34648, 34652
 - \box_dp:N 304, 1359,
24222, 34670, 34671, 34674, 34677,
34682, 34686, 34989, 35118, 35233,
35252, 35258, 35332, 35339, 35344,
35684, 35685, 35795, 35800, 35828,
35842, 36013, 36291, 36312, 36615
 - \box_gautosize_to_wd_and_ht:Nnn ..
..... 311, 35244, 35247, 35249
 - \box_gautosize_to_wd_and_ht_-
plus_dp:Nnn 311, 35244, 35256, 35261
 - \box_gclear:N
302, 34642, 34644, 34647, 34651, 35492
 - \box_gclear_new:N
..... 303, 34648, 34650, 34653
 - \box_gresize_to_ht:Nn
..... 311, 35137, 35140, 35142
 - \box_gresize_to_ht_plus_dp:Nn ...
..... 312, 35137, 35160, 35162
 - \box_gresize_to_wd:Nn
..... 312, 35137, 35180, 35182
 - \box_gresize_to_wd_and_ht:Nnn ...
..... 312, 35137, 35197, 35199
 - \box_gresize_to_wd_and_ht_plus_-
dp:Nnn
..... 312, 35088, 35094, 35099, 36127
 - \box_grotate:Nn
..... 313, 34970, 34973, 34975, 35962
 - \box_gscale:Nnn
..... 313, 35215, 35218, 35220, 36169
 - \box_gset_clipped:N
..... 313, 35312, 35315, 35317
 - \box_gset_dp:Nn
..... 304, 34679, 34685, 34687
 - \box_gset_eq:NN
..... 303, 34645, 34654, 34656,
34659, 35322, 35373, 35669, 39520
 - \box_gset_eq_drop:NN
..... 310, 34660, 34662, 34665, 39521
 - \box_gset_ht:Nn
..... 304, 34679, 34694, 34696
 - \box_gset_to_last:N
..... 305, 34733, 34735, 34738, 39522
 - \box_gset_trim:Nnnnn
..... 313, 35318, 35321, 35323
 - \box_gset_viewport:Nnnnn
..... 313, 35369, 35372, 35374
 - \box_gset_wd:Nn
..... 304, 34679, 34703, 34705
 - \box_ht:N 304, 1359,
24221, 34670, 34670, 34673, 34677,

- 34691, 34695, 34988, 35117, 35232,
 35245, 35248, 35252, 35258, 35349,
 35357, 35362, 35565, 35642, 35686,
 35687, 35786, 35791, 35828, 35835,
 36007, 36011, 36290, 36311, 36613
 \box_ht_plus_dp:N
 304, 34676, 34676, 34678
 \box_if_empty:N 34729, 34731
 \box_if_empty:NTF 305, 34729
 \box_if_empty_p:N 305, 34729
 \box_if_exist:N 34666, 34668
 \box_if_exist:NTF
 303, 34649, 34651, 34666, 34764
 \box_if_exist_p:N 303, 34666
 \box_if_horizontal:N .. 34721, 34725
 \box_if_horizontal:NTF ... 305, 34721
 \box_if_horizontal_p:N ... 305, 34721
 \box_if_vertical:N 34723, 34727
 \box_if_vertical:NTF 305, 34721
 \box_if_vertical_p:N 305, 34721
 \box_log:N ... 306, 34750, 34750, 34752
 \box_log:Nnn
 306, 34750, 34751, 34753, 34761
 \box_move_down:nn
 303, 1377, 34710, 34716,
 35336, 35344, 35387, 35394, 35986
 \box_move_left:nn .. 303, 34710, 34710
 \box_move_right:nn . 303, 34710, 34712
 \box_move_up:nn
 303, 34710, 34714, 35353,
 35362, 35401, 35414, 36331, 36610
 \box_new:N
 .. 302, 303, 34636, 34636, 34641,
 34649, 34651, 34739, 34740, 34741,
 34742, 34743, 34969, 35424, 35499
 \box_resize_to_ht:Nn
 311, 35137, 35137, 35139
 \box_resize_to_ht_plus_dp:Nn ...
 312, 35137, 35157, 35159
 \box_resize_to_wd:Nn
 312, 35137, 35177, 35179
 \box_resize_to_wd_and_ht:Nnn ...
 312, 35137, 35194, 35196
 \box_resize_to_wd_and_ht_plus_
 dp:Nnn
 312, 35088, 35088, 35093, 36120
 \box_rotate:Nn
 313, 34970, 34970, 34972, 35959
 \box_scale:Nnn
 313, 35215, 35215, 35217, 36166
 \box_set_clipped:N
 313, 35312, 35312, 35314
 \box_set_dp:Nn 304,
 1378, 34679, 34679, 34684, 35015,
 35287, 35290, 35339, 35347, 35390,
 35395, 35991, 36291, 36312, 36614
 \box_set_eq:NN 303,
 34643, 34654, 34654, 34658, 35319,
 35370, 35657, 36314, 36618, 39440
 \box_set_eq_drop:NN
 310, 34660, 34660, 34664, 39441
 \box_set_ht:Nn
 . 304, 34679, 34688, 34693, 35014,
 35286, 35291, 35356, 35365, 35404,
 35417, 35989, 36290, 36311, 36612
 \box_set_to_last:N
 305, 34733, 34733, 34737, 39442
 \box_set_trim:Nnnnn
 313, 35318, 35318, 35320
 \box_set_viewport:Nnnnn
 313, 35369, 35369, 35371
 \box_set_wd:Nn
 . 304, 34679, 34697, 34702, 35016,
 35303, 35992, 36292, 36313, 36616
 \box_show:N
 .. 306, 309, 319, 34744, 34744, 34746
 \box_show:Nnn 306, 320, 1411, 34744,
 34745, 34747, 34749, 36652, 36655
 \box_use:N 303, 34706,
 34707, 34709, 35003, 35329, 35380,
 35987, 36328, 36331, 36607, 36610
 \box_use_drop:N 310, 34706, 34706,
 34708, 35018, 35298, 35307, 35337,
 35345, 35354, 35363, 35388, 35394,
 35402, 35415, 35994, 36414, 36542
 \box_wd:N 304, 24220, 34670,
 34672, 34675, 34700, 34704, 34990,
 35119, 35234, 35266, 35381, 35688,
 35689, 35790, 35799, 35817, 35822,
 36010, 36018, 36212, 36219, 36245,
 36292, 36313, 36329, 36608, 36617
 \c_empty_box
 302, 305, 34643, 34645, 34739
 \g_tmpa_box 305, 34740
 \l_tmpa_box 305, 34740
 \g_tmpb_box 305, 34740
 \l_tmpb_box 305, 34740
 box internal commands:
 \l_box_angle_fp
 .. 34958, 34980, 34981, 34982, 35011
 __box_autosize:NnnN ... 35244,
 35245, 35248, 35252, 35258, 35262
 __box_backend_clip:N . 35313, 35316
 __box_backend_rotate:Nn 35009
 __box_backend_scale:Nnn 35279
 \l_box_bottom_dim 34961,
 34989, 35046, 35050, 35055, 35061,

- 35066, 35070, 35079, 35081, 35110,
 35118, 35127, 35171, 35233, 35239
 \l_box_bottom_new_dim
 34965, 35015, 35047, 35058, 35069,
 35080, 35126, 35238, 35287, 35291
 \l_box_cos_fp 34959,
 34982, 34994, 34999, 35026, 35038
 _box_dim_eval:n
 1359, 34631, 34632, 34635, 34677,
 34682, 34686, 34691, 34695, 34700,
 34704, 34711, 34713, 34715, 34717,
 34796, 34801, 34828, 34834, 34842,
 34866, 34900, 34905, 34933, 34939,
 34950, 34955, 35390, 35414, 39879
 _box_dim_eval:w
 34631, 34631, 34633, 39880
 \l_box_internal_box 34969, 35003,
 35004, 35010, 35014, 35015, 35016,
 35018, 35277, 35286, 35287, 35290,
 35291, 35298, 35303, 35307, 35326,
 35334, 35337, 35339, 35342, 35345,
 35347, 35349, 35351, 35354, 35356,
 35357, 35360, 35362, 35363, 35365,
 35367, 35377, 35385, 35388, 35390,
 35393, 35394, 35395, 35399, 35402,
 35404, 35412, 35415, 35417, 35419
 \l_box_left_dim ... 34961, 34991,
 35046, 35048, 35057, 35061, 35066,
 35072, 35077, 35081, 35120, 35235
 \l_box_left_new_dim 34965, 35006,
 35017, 35049, 35060, 35071, 35082
 _box_log:nNnn . 34750, 34754, 34755
 _box_resize:N ... 35088, 35112,
 35122, 35154, 35174, 35191, 35212
 _box_resize:NNN
 .. 35088, 35124, 35126, 35128, 35132
 _box_resize_common:N
 35130, 35242, 35275, 35275
 _box_resize_set_corners:N
 35088, 35104,
 35115, 35147, 35167, 35187, 35204
 _box_resize_to_ht:NnN
 35137, 35138, 35141, 35143
 _box_resize_to_ht_plus_dp:NnN .
 35137, 35158, 35161, 35163
 _box_resize_to_wd:NnN
 35137, 35178, 35181, 35183
 _box_resize_to_wd_and_ht:NnnN .
 35195, 35198, 35200
 _box_resize_to_wd_and_ht_plus_
 dp:NnnN . 35088, 35090, 35096, 35100
 _box_resize_to_wd_ht:NnnN .. 35137
 \l_box_right_dim .. 34961, 34990,
 35044, 35050, 35055, 35059, 35068,
 35070, 35079, 35083, 35106, 35119,
 35125, 35189, 35206, 35234, 35241
 \l_box_right_new_dim ... 34965,
 35017, 35051, 35062, 35073, 35084,
 35124, 35240, 35295, 35297, 35303
 _box_rotate:N . 34970, 34983, 34986
 _box_rotate:Nnn
 34970, 34971, 34974, 34976
 _box_rotate_quadrant_four: ...
 34970, 35001, 35075
 _box_rotate_quadrant_one: ...
 34970, 34995, 35042
 _box_rotate_quadrant_three: ...
 34970, 35000, 35064
 _box_rotate_quadrant_two: ...
 34970, 34996, 35053
 _box_rotate_xdir:nnN
 34970, 35020, 35048, 35050, 35059,
 35061, 35070, 35072, 35081, 35083
 _box_rotate_ydir:nnN
 34970, 35031, 35044, 35046, 35055,
 35057, 35066, 35068, 35077, 35079
 _box_scale:N
 35215, 35227, 35230, 35272
 _box_scale:NnnN
 35215, 35216, 35219, 35221
 \l_box_scale_x_fp 35086,
 35105, 35125, 35153, 35173, 35188,
 35190, 35205, 35225, 35241, 35266,
 35269, 35270, 35271, 35281, 35293
 \l_box_scale_y_fp
 35086, 35107, 35127, 35129,
 35148, 35153, 35168, 35173, 35190,
 35207, 35226, 35237, 35239, 35267,
 35269, 35270, 35271, 35282, 35284
 _box_set_trim:NnnnnN
 35318, 35319, 35322, 35324
 _box_set_viewport:NnnnnN
 35370, 35373, 35375
 _box_show:Nnn
 .. 34748, 34758, 34762, 34762, 34779
 \l_box_sin_fp
 .. 34959, 34981, 34992, 35027, 35037
 \l_box_top_dim 34961, 34988, 35044,
 35048, 35057, 35059, 35068, 35072,
 35077, 35083, 35110, 35117, 35129,
 35151, 35171, 35210, 35232, 35237
 \l_box_top_new_dim
 34965, 35014, 35045, 35056, 35067,
 35078, 35128, 35236, 35286, 35290
 _box_viewport:NnnnnN 35369
 \boxdir 793
 \boxdirection 794
 \boxmaxdepth 171

- bp 280
 \breakafterdirmode 795
 \brokenpenalty 172
- C**
- \c 32057,
 34380, 34403, 34424, 34450, 34507,
 34508, 34527, 34528, 34531, 34532,
 34539, 34540, 34551, 34552, 34559,
 34560, 34563, 34564, 34619, 34620
 \catcode 66, 85, 86, 87, 88, 89, 90, 91, 92,
 96, 97, 98, 99, 100, 101, 102, 103, 173
 \catcodetable 796
 cc 280
 cctab commands:
 \cctab_begin:N .. 290, 1257, 1258,
 1260, 1262–1265, 30519, 30519, 30532
 \cctab_const:Nn
 ... 289, 290, 1258, 30652, 30652,
 30662, 30664, 30671, 30715, 39598
 \cctab_end: 290,
 1257, 1258, 1261–1265, 30533, 30533
 \cctab_gsave_current:N
 ... 289, 30455, 30455, 30460
 \cctab_gset:Nn 289,
 290, 1267, 30443, 30443, 30454, 39523
 \cctab_if_exist:N 30607, 30609
 \cctab_if_exist:NTF 290, 30607, 30614
 \cctab_if_exist_p:N 290, 30607
 \cctab_item:Nn 290, 30591, 30591, 30606
 \cctab_new:N 289, 1257,
 1258, 1267, 30374, 30376, 30396,
 30415, 30663, 30726, 30727, 39628
 \cctab_select:N 125, 289,
 290, 14424, 30448, 30471, 30471,
 30473, 30657, 30666, 30673, 30717
 \c_code_cctab 290, 14424, 30676
 \c_document_cctab ... 290, 1260, 30676
 \c_initex_cctab
 ... 291, 30448, 30657, 30663
 \c_other_cctab 291, 30663
 \g_tmpa_cctab 291, 30726
 \g_tmpb_cctab 291, 30726
 cctab internal commands:
 \g__cctab_allocate_int
 ... 30370, 30512, 30514, 30516
 __cctab_begin_aux:
 ... 1262, 30500, 30502, 30510, 30524
 __cctab_chk_group_begin:n
 ... 1263, 30525, 30544, 30544, 30550
 __cctab_chk_group_end:n
 ... 1263, 30538, 30544, 30551
 __cctab_chk_if_valid:N 30611
 __cctab_chk_if_valid:NTF
 ... 30445, 30457, 30472, 30521, 30611
 __cctab_chk_if_valid_aux:NTF ...
 ... 30611, 30616, 30632, 30638, 30645
 \g__cctab_endlinechar_prop
 ... 1259, 30373, 30424, 30426, 30479
 \g__cctab_group_seq
 ... 30369, 30546, 30553
 __cctab_gset:n 30416, 30418, 30432,
 30450, 30458, 30528, 30659, 30713
 __cctab_gset_aux:n
 ... 30416, 30419, 30420
 __cctab_gstore:Nnn
 30374, 30394, 30403, 30404, 30405,
 30406, 30408, 30409, 30411, 30412
 \l__cctab_internal_a_tl
 . 1262, 1263, 30371, 30479, 30480,
 30505, 30515, 30523, 30526, 30527,
 30528, 30535, 30537, 30539, 30540
 \l__cctab_internal_b_tl
 ... 30371, 30553, 30557, 30564
 \g__cctab_internal_cctab 30461
 __cctab_internal_cctab_name: ...
 ... 30461,
 30464, 30482, 30483, 30484, 30485
 __cctab_item:nN . 30592, 30595, 30599
 __cctab_nesting_number:N
 ... 30526, 30539, 30569, 30570, 30572
 __cctab_nesting_number:w
 ... 30569, 30574, 30579
 __cctab_new:N . 1258, 1262, 30374,
 30379, 30381, 30388, 30399, 30463,
 30483, 30504, 30513, 30655, 30680
 \g__cctab_next_cctab 30500
 __cctab_select:N ... 1261, 30471,
 30472, 30476, 30489, 30529, 30540
 \g__cctab_stack_seq
 ... 1257, 30367, 30527, 30535, 30587
 \g__cctab_unused_seq
 1257, 1262, 1263, 30367, 30523, 30537
 ceil 276
 \char 174, 19515
 char commands:
 \l_char_active_seq ... 90, 200, 19156
 \char_fold_case:N 38949, 38956
 \char_foldcase:N 38965, 38972
 \char_generate:nn 124, 196,
 446, 467, 553, 702, 765, 896, 3544,
 3545, 3546, 3547, 3549, 3550, 3551,
 4108, 4182, 4198, 4210, 4630, 5668,
 6042, 12405, 12421, 14265, 14522,
 14538, 19183, 19183, 19282, 19958,
 30782, 30790, 30809, 30812, 30815,

- 30817, 30829, 30860, 31440, 31484,
 33355, 33366, 33527, 33550, 37335
 \char_gset_active_eq:NN
 196, 19162, 19179
 \char_gset_active_eq:nN
 196, 19162, 19181
 \char_lower_case:N 38949, 38950
 \char_lowercase:N 38965, 38966
 \char_mixed_case:N 38954
 \char_mixed_case:Nn 38949
 \char_set_active_eq:NN
 196, 3541, 4104, 19162, 19178
 \char_set_active_eq:nN
 196, 4145, 4146, 19162, 19180
 \char_set_catcode:nn .. 198, 112,
 113, 114, 115, 116, 117, 118, 119,
 19062, 19062, 19069, 19071, 19073,
 19075, 19077, 19079, 19081, 19083,
 19085, 19087, 19089, 19091, 19093,
 19095, 19097, 19099, 19101, 19103,
 19105, 19107, 19109, 19111, 19113,
 19115, 19117, 19119, 19121, 19123,
 19125, 19127, 19129, 19131, 30493
 \char_set_catcode_active:N
 197, 3591,
 3611, 7164, 9378, 19068, 19094,
 19163, 19216, 19278, 19353, 34321
 \char_set_catcode_active:n
 198, 9126,
 19100, 19126, 19226, 21471, 21472,
 30687, 30694, 30712, 30723, 31408
 \char_set_catcode_alignment:N ...
 197, 7216, 19068, 19076, 19341
 \char_set_catcode_alignment:n ...
 198, 19100, 19108, 19241, 30700
 \char_set_catcode_comment:N
 197, 19068, 19096
 \char_set_catcode_comment:n
 198, 19100, 19128, 30699
 \char_set_catcode_end_line:N ...
 197, 19068, 19078
 \char_set_catcode_end_line:n ...
 198, 19100, 19110, 30695
 \char_set_catcode_escape:N
 197, 19068, 19068
 \char_set_catcode_escape:n
 198, 19100, 19100, 30702
 \char_set_catcode_group_begin:N .
 197, 3688, 7167, 19068, 19070
 \char_set_catcode_group_begin:n .
 198, 19100, 19102, 19247, 30705
 \char_set_catcode_group_end:N ...
 197, 3691, 7184, 19068, 19072
 \char_set_catcode_group_end:n ...
 198, 19100, 19104, 19245, 30707
 \char_set_catcode_ignore:N
 197, 19068, 19086
 \char_set_catcode_ignore:n . 198,
 126, 127, 19100, 19118, 30692, 30696
 \char_set_catcode_invalid:N
 197, 19068, 19098
 \char_set_catcode_invalid:n
 198, 19100, 19130, 30683, 30686, 30709
 \char_set_catcode_letter:N
 197, 7193, 19068, 19090, 25898, 25899
 \char_set_catcode_letter:n
 198, 129, 131, 19100, 19122,
 19230, 30689, 30691, 30701, 30704
 \char_set_catcode_math_subscript:N
 197, 7181, 19068, 19084, 19345
 \char_set_catcode_math_subscript:n
 198, 19100, 19116, 19234, 30722
 \char_set_catcode_math_superscript:N
 197, 7219, 19068, 19082
 \char_set_catcode_math_superscript:n
 . 198, 130, 19100, 19114, 19236, 30703
 \char_set_catcode_math_toggle:N .
 197, 7196, 19068, 19074, 19339
 \char_set_catcode_math_toggle:n .
 198, 19100, 19106, 19243, 30698
 \char_set_catcode_other:N .. 197,
 1260, 3853, 7199, 14813, 14814,
 15149, 15150, 15331, 15332, 15333,
 19068, 19092, 39301, 39363, 39787
 \char_set_catcode_other:n
 198, 128, 132, 9121, 9123,
 9125, 19100, 19124, 19228, 30669,
 30688, 30690, 30693, 30706, 30721
 \char_set_catcode_parameter:N ...
 197, 7202,
 19068, 19080, 39306, 39368, 39792
 \char_set_catcode_parameter:n ...
 198, 19100, 19112, 19238, 30697
 \char_set_catcode_space:N
 197, 19068, 19088, 39307, 39369
 \char_set_catcode_space:n
 ... 198, 133, 11595, 19100, 19120,
 30674, 30708, 30719, 30720, 31207
 \char_set_lccode:nn
 198, 9374, 9375, 9376, 9377, 19132,
 19138, 19169, 19251, 19252, 19279
 \char_set_mathcode:nn
 199, 19132, 19132
 \char_set_sfcode:nn 199, 19132, 19150
 \char_set_uccode:nn 199, 19132, 19144
 \char_show_value_catcode:n
 198, 19062, 19066

- `\char_show_value_lccode:n`
..... [199](#), [19132](#), [19142](#)
- `\char_show_value_mathcode:n`
..... [199](#), [19132](#), [19136](#)
- `\char_show_value_sfcode:n`
..... [200](#), [19132](#), [19154](#)
- `\char_show_value_uccode:n`
..... [199](#), [19132](#), [19148](#)
- `\l_char_special_seq` [200](#), [19156](#)
- `\char_str_fold_case:N` . [38949](#), [38964](#)
- `\char_str_foldcase:N` .. [38965](#), [38982](#)
- `\char_str_lower_case:N` . [38949](#), [38958](#)
- `\char_str_lowercase:N` . [38965](#), [38974](#)
- `\char_str_mixed_case:N` [38962](#)
- `\char_str_mixed_case:Nn` [38949](#)
- `\char_str_titlecase:N` . [38965](#), [38977](#)
- `\char_str_upper_case:N` . [38949](#), [38960](#)
- `\char_str_uppercase:N` . [38965](#), [38980](#)
- `\char_titlecase:N` [38965](#), [38970](#)
- `\char_to_nfd:N` [38945](#), [38946](#)
- `\char_to_nfd:n` [38945](#), [38948](#)
- `\char_to_utfviii_bytes:n` [38943](#), [38944](#)
- `\char_upper_case:N` [38949](#), [38952](#)
- `\char_uppercase:N` [38965](#), [38968](#)
- `\char_value_catcode:n` . [198](#), [1265](#),
[112](#), [113](#), [114](#), [115](#), [116](#), [117](#), [118](#),
[119](#), [12417](#), [12421](#), [19062](#), [19064](#),
[19067](#), [30437](#), [30603](#), [30953](#), [30962](#),
[31442](#), [32572](#), [32576](#), [32595](#), [32653](#),
[32700](#), [32907](#), [34336](#), [34387](#), [34414](#)
- `\char_value_lccode:n`
..... [199](#), [19132](#), [19140](#), [19143](#)
- `\char_value_mathcode:n`
..... [199](#), [19132](#), [19134](#), [19137](#)
- `\char_value_sfcode:n`
..... [200](#), [19132](#), [19152](#), [19155](#)
- `\char_value_uccode:n`
..... [199](#), [19132](#), [19146](#), [19149](#)
- char internal commands:
 - `__char_generate_aux:nn` [19183](#)
 - `__char_generate_aux:nnw`
..... [19183](#), [19208](#), [19219](#), [19261](#)
 - `__char_generate_aux:w` . [19185](#), [19189](#)
 - `__char_generate_auxii:nnw` ... [19183](#)
 - `__char_generate_invalid_catcode:` [19183](#)
 - `__char_int_to_roman:w`
..... [19182](#), [19182](#), [19256](#), [19271](#)
 - `__char_quark_if_no_value:N` .. [19061](#)
 - `__char_quark_if_no_value:NNTF` . [19061](#)
 - `__char_quark_if_no_value_p:N` . [19061](#)
 - `__char_tmp:n` [19249](#), [19260](#)
 - `__char_tmp:nN` .. [19164](#), [19175](#), [19176](#)
 - `\l_char_tmp_tl` [19183](#)
- `\chardef` [1266](#), [94](#), [105](#), [175](#)
- choice commands:
 - `.choice:` [241](#), [22031](#)
- choices commands:
 - `.choices:nn` [241](#), [22033](#)
- `\cite` [1292](#), [31662](#), [31672](#)
- `\cleaders` [176](#)
- `\clearmarks` [797](#)
- clist commands:
 - `\clist_clear:N`
..... [185](#), [18434](#), [18434](#), [18435](#),
[18451](#), [18608](#), [22214](#), [31163](#), [39443](#)
 - `\clist_clear_new:N`
..... [185](#), [18438](#), [18438](#), [18439](#)
 - `\clist_concat:NnN`
..... [186](#), [1480](#), [1482](#), [18477](#),
[18477](#), [18490](#), [18505](#), [18518](#), [39419](#)
 - `\clist_const:Nn`
[185](#), [18430](#), [18430](#), [18432](#), [18433](#), [31356](#)
 - `\clist_count:N` [190](#), [193](#),
[18853](#), [18853](#), [18861](#), [18887](#), [18952](#),
[19019](#), [19030](#), [31074](#), [31106](#), [31115](#)
 - `\clist_count:n` . [190](#), [18853](#), [18865](#),
[18882](#), [18983](#), [19010](#), [19031](#), [37859](#)
 - `\clist_gclear:N`
[185](#), [18434](#), [18436](#), [18437](#), [18453](#), [39524](#)
 - `\clist_gclear_new:N`
..... [185](#), [18438](#), [18440](#), [18441](#)
 - `\clist_gconcat:NnN` ... [186](#), [18477](#),
[18479](#), [18491](#), [18507](#), [18520](#), [39420](#)
 - `\clist_get:NN`
[192](#), [18534](#), [18534](#), [18544](#), [18571](#), [18580](#)
 - `\clist_get:NNTF` [192](#), [18571](#)
 - `\clist_gpop:NN`
[192](#), [18545](#), [18547](#), [18570](#), [18583](#), [18595](#)
 - `\clist_gpop:NNTF` [193](#), [18571](#)
 - `\clist_gpush:Nn`
..... [193](#), [18596](#), [18598](#), [18599](#)
 - `\clist_gput_left:Nn`
[186](#), [18504](#), [18506](#), [18515](#), [18516](#), [18598](#)
 - `\clist_gput_right:Nn`
..... [186](#), [18517](#), [18519](#), [18530](#), [18532](#)
 - `\clist_gremove_all:Nn`
..... [187](#), [18624](#), [18626](#), [18662](#)
 - `\clist_gremove_duplicates:N`
..... [187](#), [18602](#), [18604](#), [18623](#)
 - `\clist_greverse:N`
..... [187](#), [18663](#), [18665](#), [18668](#)
 - `.clist_gset:N` [241](#), [22045](#)
 - `\clist_gset:Nn`
[186](#), [14551](#), [18496](#), [18498](#), [18502](#), [18503](#)
 - `\clist_gset_eq:NN`
..... [185](#), [18442](#), [18446](#), [18447](#),
[18448](#), [18449](#), [18605](#), [39400](#), [39525](#)

- \clist_gset_from_seq:NN [185](#), [3256](#),
[18450](#), [18452](#), [18475](#), [18476](#), [18627](#)
- \clist_gsort:Nn
..... [188](#), [3241](#), [3253](#), [3258](#), [18681](#)
- \clist_if_empty:N [18681](#), [18683](#)
- \clist_if_empty:n [18685](#)
- \clist_if_empty:NTF
. [188](#), [18486](#), [18615](#), [18648](#), [18681](#),
[18738](#), [18778](#), [18810](#), [19018](#), [21813](#)
- \clist_if_empty:nTF [188](#), [18685](#)
- \clist_if_empty_p:N [188](#), [18681](#)
- \clist_if_empty_p:n [188](#), [18685](#)
- \clist_if_exist:N [18492](#), [18494](#)
- \clist_if_exist:NTF
.... [186](#), [11442](#), [11563](#), [18492](#), [18885](#)
- \clist_if_exist_p:N [186](#), [18492](#)
- \clist_if_in:Nn [18699](#), [18732](#)
- \clist_if_in:nn [18703](#), [18734](#)
- \clist_if_in:NnTF
.. [185](#), [188](#), [995](#), [18611](#), [18699](#), [22398](#)
- \clist_if_in:nnTF .. [188](#), [18699](#), [23607](#)
- \clist_item:Nn
[193](#), [887](#), [18949](#), [18949](#), [18979](#), [19019](#)
- \clist_item:nn
[193](#), [887](#), [18980](#), [18980](#), [18988](#), [19014](#)
- \clist_log:N . [194](#), [19022](#), [19024](#), [19025](#)
- \clist_log:n [194](#), [19044](#), [19045](#)
- \clist_map_break: [189](#),
[18743](#), [18755](#), [18764](#), [18765](#), [18788](#),
[18815](#), [18828](#), [18836](#), [18837](#), [18849](#),
[18849](#), [18850](#), [18852](#), [22401](#), [22446](#)
- \clist_map_break:n [190](#), [3249](#), [3255](#),
[18719](#), [18849](#), [18851](#), [22475](#), [37836](#)
- \clist_map_function:NN
.. [189](#), [882](#), [16704](#), [16714](#), [18722](#),
[18736](#), [18736](#), [18759](#), [18858](#), [19035](#)
- \clist_map_function:nN
..... [189](#), [883](#), [14554](#),
[16709](#), [16719](#), [16730](#), [18760](#), [18760](#),
[18767](#), [19049](#), [22589](#), [37997](#), [38073](#)
- \clist_map_inline:Nn .. [189](#), [3249](#),
[3255](#), [9212](#), [18609](#), [18776](#), [18776](#),
[18795](#), [18797](#), [22396](#), [22437](#), [22466](#)
- \clist_map_inline:nn .. [189](#), [3043](#),
[10176](#), [11751](#), [11783](#), [11795](#), [18776](#),
[18792](#), [21768](#), [21900](#), [22934](#), [23118](#),
[29897](#), [30984](#), [31167](#), [37413](#), [37616](#),
[37618](#), [37654](#), [37823](#), [37977](#), [38021](#),
[38026](#), [39033](#), [39041](#), [39948](#), [39972](#)
- \clist_map_tokens:Nn
[189](#), [882](#), [18799](#), [18808](#), [18808](#), [18832](#)
- \clist_map_tokens:nn [189](#), [18833](#), [18833](#)
- \clist_map_variable:NNn
.... [189](#), [18798](#), [18798](#), [18800](#), [18806](#)
- \clist_map_variable:nNn
..... [189](#), [18798](#), [18803](#), [18803](#)
- \clist_new:N [185](#), [869](#), [18428](#),
[18428](#), [18429](#), [18600](#), [19051](#), [19052](#),
[19053](#), [19054](#), [21570](#), [21571](#), [21587](#)
- \clist_pop:NN
[192](#), [18545](#), [18545](#), [18569](#), [18581](#), [18594](#)
- \clist_pop:NNTF [192](#), [18571](#)
- \clist_push:Nn [193](#), [18596](#), [18596](#), [18597](#)
- \clist_put_left:Nn
[186](#), [18504](#), [18504](#), [18513](#), [18514](#), [18596](#)
- \clist_put_right:Nn .. [186](#), [18517](#),
[18517](#), [18526](#), [18528](#), [21992](#), [22520](#),
[22530](#), [22558](#), [31073](#), [31114](#), [31131](#)
- \clist_rand_item:N
..... [193](#), [19009](#), [19016](#), [19021](#)
- \clist_rand_item:n
..... [77](#), [193](#), [19009](#), [19009](#)
- \clist_remove_all:Nn
[187](#), [9227](#), [18624](#), [18624](#), [18661](#), [21993](#)
- \clist_remove_duplicates:N
..... [185](#), [187](#), [18602](#), [18602](#), [18622](#)
- \clist_reverse:N
..... [187](#), [18663](#), [18663](#), [18667](#)
- \clist_reverse:n
[187](#), [878](#), [18664](#), [18666](#), [18669](#), [18669](#)
- .clist_set:N [241](#), [22045](#)
- \clist_set:Nn [186](#),
[192](#), [18496](#), [18496](#), [18500](#), [18501](#),
[18505](#), [18507](#), [18518](#), [18520](#), [18705](#),
[18794](#), [18805](#), [21812](#), [21826](#), [22215](#)
- \clist_set_eq:NN [185](#), [18442](#),
[18442](#), [18443](#), [18444](#), [18445](#), [18603](#),
[22228](#), [22383](#), [31145](#), [39399](#), [39444](#)
- \clist_set_from_seq:NN . [185](#), [3250](#),
[18450](#), [18450](#), [18473](#), [18474](#), [18625](#)
- \clist_show:N
..... [193](#), [194](#), [19022](#), [19022](#), [19023](#)
- \clist_show:n [194](#), [19044](#), [19044](#)
- \clist_sort:Nn
..... [188](#), [3241](#), [3247](#), [3252](#), [18681](#)
- \clist_use:Nn
..... [191](#), [192](#), [18883](#), [18913](#), [18915](#)
- \clist_use:nn [192](#), [18916](#), [18948](#)
- \clist_use:Nnnn [191](#),
[192](#), [838](#), [18883](#), [18883](#), [18906](#), [18914](#)
- \clist_use:nnnn
..... [192](#), [18916](#), [18916](#), [18948](#)
- \c_empty_clist
[194](#), [18375](#), [18536](#), [18551](#), [18573](#), [18587](#)
- \g_tmpa_clist [194](#), [19051](#)
- \l_tmpa_clist [194](#), [19051](#)
- \g_tmpb_clist [194](#), [19051](#)
- \l_tmpb_clist [194](#), [19051](#)

- clist internal commands:
- __clist_concat:NNNN
..... [18477](#), [18478](#), [18480](#), [18481](#)
 - __clist_count:n . [18853](#), [18858](#), [18862](#)
 - __clist_count:w
..... [18853](#), [18870](#), [18874](#), [18878](#)
 - __clist_get:wN
..... [18534](#), [18539](#), [18542](#), [18576](#)
 - __clist_if_empty_n:w
..... [18685](#), [18687](#), [18692](#), [18695](#)
 - __clist_if_empty_n:wNw
..... [18685](#), [18696](#), [18698](#)
 - __clist_if_in_return:nnN
..... [18699](#), [18701](#), [18706](#), [18709](#)
 - __clist_if_wrap:n [18402](#)
 - __clist_if_wrap:nTF . [870](#), [18402](#),
[18427](#), [18469](#), [18616](#), [18630](#), [18711](#)
 - __clist_if_wrap:w
..... [870](#), [18402](#), [18406](#), [18425](#)
 - \l__clist_internal_clist
..... [873](#), [18376](#), [18510](#),
[18511](#), [18523](#), [18524](#), [18705](#), [18706](#),
[18707](#), [18794](#), [18795](#), [18805](#), [18806](#)
 - \l__clist_internal_remove_clist .
..... [18600](#),
[18608](#), [18611](#), [18613](#), [18615](#), [18620](#)
 - \l__clist_internal_remove_seq ...
..... [18600](#), [18632](#), [18633](#), [18634](#)
 - __clist_item:nnnN
.. [18949](#), [18951](#), [18957](#), [18972](#), [18982](#)
 - __clist_item_n:nw [18980](#), [18986](#), [18989](#)
 - __clist_item_n_end:n
..... [18980](#), [18997](#), [19005](#)
 - __clist_item_N_loop:nw
..... [18949](#), [18955](#), [18973](#), [18977](#)
 - __clist_item_n_loop:nw
.. [18980](#), [18990](#), [18991](#), [18994](#), [18999](#)
 - __clist_item_n_strip:n
..... [18980](#), [19006](#), [19007](#)
 - __clist_item_n_strip:w
..... [18980](#), [19007](#), [19008](#)
 - __clist_map_function:Nw
[880](#), [18736](#), [18740](#), [18746](#), [18751](#), [18783](#)
 - __clist_map_function_end:w
.... [880](#), [18736](#), [18749](#), [18753](#), [18757](#)
 - __clist_map_function_n:Nn
.... [881](#), [18760](#), [18762](#), [18768](#), [18772](#)
 - __clist_map_tokens:nw
..... [18808](#), [18812](#), [18818](#), [18824](#)
 - __clist_map_tokens_end:w
..... [18808](#), [18821](#), [18826](#), [18830](#)
 - __clist_map_tokens_n:nw
..... [18833](#), [18835](#), [18839](#), [18847](#)
 - __clist_map_unbrace:wN
[881](#), [18760](#), [18771](#), [18775](#), [18845](#), [18933](#)
 - __clist_map_variable:Nnn
..... [882](#), [18798](#), [18799](#), [18801](#)
 - __clist_pop:NNN
..... [18545](#), [18546](#), [18548](#), [18549](#)
 - __clist_pop:wN . [18545](#), [18562](#), [18568](#)
 - __clist_pop:wwNNN
.... [875](#), [18545](#), [18554](#), [18557](#), [18590](#)
 - __clist_pop_TF:NNN
..... [18571](#), [18582](#), [18584](#), [18585](#)
 - __clist_put_left:NNNn
..... [18504](#), [18505](#), [18507](#), [18508](#)
 - __clist_put_right:NNNn
..... [18517](#), [18518](#), [18520](#), [18521](#)
 - __clist_rand_item:nn
..... [19009](#), [19010](#), [19011](#)
 - __clist_remove_all:
..... [18624](#), [18641](#), [18645](#), [18658](#)
 - __clist_remove_all:NNNn
..... [18624](#), [18625](#), [18627](#), [18628](#)
 - __clist_remove_all:w
..... [877](#), [18624](#), [18659](#), [18660](#)
 - __clist_remove_duplicates:NN ...
..... [18602](#), [18603](#), [18605](#), [18606](#)
 - __clist_reverse:wwNww
.... [878](#), [18669](#), [18671](#), [18672](#), [18676](#)
 - __clist_reverse_end:ww
..... [878](#), [18669](#), [18673](#), [18679](#)
 - __clist_sanitize:n
.. [18389](#), [18389](#), [18431](#), [18497](#), [18499](#)
 - __clist_sanitize:Nn
.... [870](#), [18389](#), [18391](#), [18395](#), [18399](#)
 - __clist_set_from_seq:n
..... [18450](#), [18462](#), [18466](#)
 - __clist_set_from_seq:NNNN
..... [18450](#), [18451](#), [18453](#), [18454](#)
 - __clist_show:NN
..... [19022](#), [19022](#), [19024](#), [19026](#)
 - __clist_show:Nn
..... [19044](#), [19044](#), [19045](#), [19046](#)
 - __clist_tmp:w [877](#),
[18382](#), [18382](#), [18637](#), [18659](#), [18713](#),
[18722](#), [18726](#), [18728](#), [18863](#), [18881](#)
 - __clist_trim_next:w
..... [870](#), [881](#), [18383](#), [18383](#),
[18386](#), [18392](#), [18400](#), [18763](#), [18773](#)
 - __clist_use:Nw [885](#), [18916](#), [18918](#),
[18919](#), [18920](#), [18926](#), [18929](#), [18945](#)
 - __clist_use:nwn [18883](#), [18897](#), [18911](#)
 - __clist_use:nwwwnwn
.... [884](#), [18883](#), [18894](#), [18896](#), [18908](#)
 - __clist_use:wN
..... [18883](#), [18890](#), [18891](#), [18907](#)

- __clist_use_end:w
 [885](#), [18916](#), [18920](#), [18939](#), [18945](#)
- __clist_use_i_delimit_by_s_-
 stop:nw [18379](#), [18381](#), [18976](#)
- __clist_use_more:w
 [885](#), [18916](#), [18921](#), [18942](#), [18945](#)
- __clist_use_none_delimit_by_s_-
 mark:w [18379](#), [18379](#), [18931](#)
- __clist_use_none_delimit_by_s_-
 stop:w
 . [877](#), [18379](#), [18380](#), [18397](#), [18640](#),
 [18748](#), [18755](#), [18770](#), [18820](#), [18828](#),
 [18843](#), [18876](#), [18918](#), [18962](#), [18967](#)
- __clist_use_one:w [18916](#), [18919](#), [18937](#)
- __clist_wrap_item:w
 [870](#), [18398](#), [18426](#), [18426](#)
- \closein [177](#)
- \closeout [178](#)
- \clubpenalties [476](#)
- \clubpenalty [179](#)
- cm [280](#)
- code commands:
 .code:n [242](#), [22043](#)
- codepoint commands:
 \codepoint_generate:nn [294](#), [30774](#),
 [30784](#), [30821](#), [30974](#), [32472](#), [32488](#),
 [32489](#), [32567](#), [32571](#), [32575](#), [32653](#),
 [32660](#), [32699](#), [32842](#), [32846](#), [32906](#),
 [33231](#), [33320](#), [33322](#), [33336](#), [33338](#),
 [33403](#), [33405](#), [33407](#), [33420](#), [33457](#),
 [33483](#), [33564](#), [33579](#), [33603](#), [33611](#),
 [33623](#), [34335](#), [34387](#), [34414](#), [38943](#)
- \codepoint_str_generate:n .. [294](#),
 [14124](#), [14127](#), [14129](#), [30774](#), [30778](#),
 [30795](#), [31025](#), [31065](#), [31252](#), [31263](#),
 [31291](#), [31315](#), [31337](#), [32622](#), [32638](#)
- \codepoint_to_category:n
 [295](#), [30941](#), [30941](#), [32499](#)
- \codepoint_to_nfd:n
 [295](#), [30950](#), [30950](#), [32688](#),
 [33244](#), [38945](#), [38946](#), [38947](#), [38948](#)
- codepoint internal commands:
 __codepoint_add:nn
 [31059](#), [31060](#), [31061](#), [31071](#)
- \c_codepoint_block_size_int ...
 [30981](#), [30991](#), [31075](#), [31105](#),
 [31116](#), [31119](#), [31124](#), [31127](#), [31130](#),
 [31178](#), [31192](#), [31224](#), [31229](#), [31241](#)
- __codepoint_case:nn [31312](#),
 [31326](#), [31327](#), [31328](#), [31329](#), [31330](#)
- __codepoint_case:nnn
 [31312](#), [31314](#), [31317](#)
- __codepoint_casefold:n [31312](#), [31329](#)
- \l_codepoint_category_Cn_tl . [31083](#)
- __codepoint_data:nn
 [31218](#), [31220](#), [31238](#)
- __codepoint_data_auxi:w
 [30995](#), [31000](#), [31002](#), [31012](#),
 [31213](#), [31245](#), [31273](#), [31278](#), [31308](#)
- __codepoint_data_auxii:w
 [31018](#), [31022](#), [31257](#),
 [31261](#), [31281](#), [31282](#), [31284](#), [31286](#)
- __codepoint_data_auxiii:w
 [31020](#), [31031](#)
- __codepoint_data_auxiv:w
 [31036](#), [31052](#)
- __codepoint_data_auxv:nnnw ...
 [31056](#), [31078](#)
- __codepoint_data_category:n ...
 [31038](#), [31044](#)
- \g_codepoint_data_ior
 [30982](#), [31205](#), [31208](#), [31244](#),
 [31270](#), [31276](#), [31277](#), [31299](#), [31310](#)
- __codepoint_data_offset:nn
 [31039](#), [31040](#), [31046](#), [31062](#)
- __codepoint_finalise_blocks: ...
 [31165](#), [31215](#)
- __codepoint_finalise_blocks:n ..
 [31170](#), [31173](#)
- __codepoint_finalise_blocks:nnn
 [31181](#), [31189](#)
- __codepoint_finalise_blocks:nnnw
 [31191](#), [31196](#), [31202](#)
- __codepoint_generate:n .. [30774](#),
 [30846](#), [30847](#), [30850](#), [30852](#), [30857](#)
- __codepoint_generate:nnnn
 [30774](#), [30834](#), [30840](#)
- __codepoint_lowercase:n [31312](#), [31327](#)
- \l_codepoint_matched_block_tl ..
 .. [30994](#), [31135](#), [31140](#), [31143](#), [31161](#)
- \l_codepoint_next_codepoint_-
 fint_tl . [30993](#), [31054](#), [31068](#), [31096](#)
- __codepoint_nfd:n [30968](#), [31336](#), [31336](#)
- __codepoint_nfd:nn
 [31336](#), [31337](#), [31338](#)
- __codepoint_range:nn
 [31082](#), [31084](#), [31085](#),
 [31088](#), [31089](#), [31090](#), [31093](#), [31169](#)
- __codepoint_range:nnnn [31100](#), [31111](#)
- __codepoint_range_aux:nn
 [31095](#), [31098](#)
- __codepoint_save_blocks:nn
 [31076](#), [31117](#), [31126](#), [31133](#)
- __codepoint_str_generate:nnnn ..
 [30774](#), [30802](#), [30807](#)
- __codepoint_titlecase:n [31312](#), [31328](#)
- \l_codepoint_tmpa_tl
 [31208](#), [31210](#), [31213](#)

- __codepoint_to_bytes_auxi:n 30863, 30865, 30868
- __codepoint_to_bytes_auxii:Nnn 30863, 30873, 30879, 30890, 30912
- __codepoint_to_bytes_auxiii:n 30863, 30875, 30882, 30886, 30895, 30900, 30904, 30914
- __codepoint_to_bytes_end: 30863, 30910, 30917, 30920, 30923, 30929, 30937, 30940
- __codepoint_to_bytes_output:nnn 30863, 30918, 30921, 30925, 30931, 30934, 30939
- __codepoint_to_bytes_outputi:nw 30863, 30872, 30878, 30888, 30908, 30916
- __codepoint_to_bytes_outputii:nw 30863, 30874, 30880, 30893, 30919
- __codepoint_to_bytes_outputiii:nw 30863, 30885, 30898, 30922
- __codepoint_to_bytes_outputiv:nw 30863, 30903, 30928
- __codepoint_to_nfd:n 30950, 30951, 30952, 30958
- __codepoint_to_nfd:nn 30950, 30953, 30961, 30962, 30965, 30976, 30978
- __codepoint_to_nfd:nnn 30950, 30967, 30970
- __codepoint_to_nfd:nnnn 30950, 30970, 30971
- __codepoint_uppercase:n 31312, 31326
- coffin commands:
- \coffin_attach:NnnNnnnn 318, 1410, 36274, 36274, 36279
- \coffin_clear:N 315, 35479, 35479, 35487
- \coffin_display_handles:Nn 319, 36520, 36520, 36595
- \coffin_dp:N 318, 35684, 35684, 35685, 36138, 36177, 36634
- \coffin_gattach:NnnNnnnn 318, 36274, 36280, 36285
- \coffin_gclear:N 315, 35479, 35488, 35496
- \coffin_gjoin:NnnNnnnn 318, 36223, 36229, 36234
- \coffin_greset_poles:N 317, 35532, 35545, 35604, 35622, 35760, 35766
- \coffin_gresize:Nnn 317, 36117, 36124, 36130
- \coffin_grotate:Nn 317, 35958, 35961, 35963
- \coffin_gscale:Nnn 317, 36165, 36168, 36170
- \coffin_gset_eq:NN 315, 35653, 35665, 35676, 36232, 36283
- \coffin_gset_horizontal_pole:Nnn 316, 35714, 35717, 35719
- \coffin_gset_vertical_pole:Nnn 317, 35714, 35735, 35737
- \coffin_ht:N 319, 35684, 35686, 35687, 36138, 36177, 36633
- \coffin_if_exist:N 35458, 35468
- \coffin_if_exist:NTF 315, 35458, 35472
- \coffin_if_exist_p:N 315, 35458
- \coffin_join:NnnNnnnn 318, 36223, 36223, 36228
- \coffin_log:N 319, 36645, 36648, 36650
- \coffin_log:Nnn 320, 36645, 36649, 36654, 36656
- \coffin_log_structure:N 319, 36620, 36623, 36625
- \coffin_mark_handle:Nnnn 319, 36475, 36475, 36519
- \coffin_new:N 315, 1386, 35497, 35497, 35509, 35677, 35678, 35679, 35680, 35681, 35682, 35683, 36407, 36417, 36418, 36419
- \coffin_reset_poles:N 317, 35519, 35539, 35591, 35615, 35760, 35760
- \coffin_resize:Nnn 317, 36117, 36117, 36123
- \coffin_rotate:Nn 317, 35958, 35958, 35960
- \coffin_scale:Nnn 317, 36165, 36165, 36167
- \coffin_set_eq:NN 315, 35653, 35653, 35664, 36226, 36277, 36333, 36536
- \coffin_set_horizontal_pole:Nnn 316, 35714, 35714, 35716
- \coffin_set_vertical_pole:Nnn 317, 35714, 35732, 35734
- \coffin_show:N 319, 36645, 36645, 36647
- \coffin_show:Nnn 320, 36645, 36646, 36651, 36653
- \coffin_show_structure:N 319, 320, 1411, 36620, 36620, 36622
- \coffin_typeset:Nnnnn 318, 36409, 36409, 36416
- \coffin_wd:N 319, 35684, 35688, 35689, 36134, 36181, 36635
- \c_empty_coffin 320, 35677
- \g_tmpa_coffin 320, 35680
- \l_tmpa_coffin 320, 35680
- \g_tmpb_coffin 320, 35680
- \l_tmpb_coffin 320, 35680

- coffin internal commands:
- __coffin_align:NnnNnnnnN 36237, 36288, 36309, 36316, 36316, 36412
 - \l__coffin_aligned_coffin 35677, 36238, 36239, 36243, 36249, 36252, 36255, 36271, 36272, 36289, 36290, 36291, 36292, 36293, 36296, 36300, 36304, 36305, 36310, 36311, 36312, 36313, 36314, 36347, 36363, 36413, 36414, 36605, 36612, 36614, 36616, 36618
 - \l__coffin_aligned_internal_coffin 35677, 36326, 36333
 - __coffin_attach:NnnNnnnnN 36274, 36276, 36282, 36286
 - __coffin_attach_mark:NnnNnnnn 36274, 36307, 36482, 36498, 36514
 - \l__coffin_bottom_corner_dim ... 35954, 35986, 35990, 36069, 36080, 36081, 36101, 36109
 - \l__coffin_bounding_prop 35950, 35977, 36006, 36008, 36014, 36016, 36025, 36088
 - \l__coffin_bounding_shift_dim ... 35953, 35985, 36087, 36093, 36094
 - __coffin_calculate_intersection:Nnn ... 35847, 35847, 36318, 36321, 36598
 - __coffin_calculate_intersection:nnnnnn 35847, 35911, 35919
 - __coffin_calculate_intersection:nnnnnnnn 35847, 35853, 35862, 36549
 - \c__coffin_corners_prop 35427, 35504, 35703, 35710
 - \l__coffin_corners_prop 35951, 35968, 35972, 35995, 36000, 36031, 36071, 36098, 36145, 36149, 36155, 36161, 36196, 36210
 - \l__coffin_cos_fp 1394, 1397, 35948, 35967, 36052, 36061
 - __coffin_display_attach:Nnnnnn ... 36520, 36554, 36571, 36590, 36596
 - \l__coffin_display_coffin 36417, 36536, 36542, 36607, 36608, 36613, 36615, 36617, 36618
 - \l__coffin_display_coord_coffin . 36417, 36484, 36499, 36515, 36557, 36572, 36591
 - \l__coffin_display_font_tl 36462, 36487, 36560
 - __coffin_display_handles_ aux:nnnn 36520, 36577, 36582, 36588
 - __coffin_display_handles_ aux:nnnnnn 36520, 36540, 36544
 - \l__coffin_display_handles_prop . 36420, 36490, 36494, 36563, 36567
 - \l__coffin_display_offset_dim ... 36457, 36516, 36517, 36592, 36593
 - \l__coffin_display_pole_coffin .. 36417, 36477, 36483, 36522, 36555
 - \l__coffin_display_poles_prop ... 36461, 36527, 36532, 36535, 36537, 36539, 36546
 - \l__coffin_display_x_dim 36459, 36552, 36602
 - \l__coffin_display_y_dim 36459, 36553, 36604
 - \c__coffin_empty_coffin 36407, 36412
 - \l__coffin_error_bool 35448, 35851, 35855, 35869, 35891, 35922, 36548, 36550
 - __coffin_find_bounding_shift: .. 35980, 36085, 36085
 - __coffin_find_bounding_shift_ aux:nn 36085, 36089, 36091
 - __coffin_find_corner_maxima:N .. 35979, 36065, 36065
 - __coffin_find_corner_maxima_ aux:nn 36065, 36072, 36074
 - __coffin_get_pole:NnN ... 35690, 35690, 35849, 35850, 36374, 36375, 36378, 36379, 36529, 36530, 36533
 - __coffin_greset_structure:N ... 35493, 35700, 35707, 35768
 - __coffin_gset_pole:Nnn 35545, 35622, 35714, 35755
 - __coffin_gupdate_corners:N 35769, 35772, 35774
 - __coffin_gupdate_poles:N 35770, 35803, 35805
 - __coffin_if_exist:Ntf ... 35470, 35470, 35481, 35490, 35512, 35525, 35550, 35585, 35598, 35627, 35655, 35667, 35722, 35740, 36628, 36659
 - \l__coffin_internal_box 35424, 35559, 35565, 35570, 35636, 35642, 35647, 35982, 35989, 35991, 35992, 35994
 - \l__coffin_internal_dim 35424, 36013, 36015, 36019, 36176, 36179, 36244, 36246, 36247
 - \l__coffin_internal_tl ... 35424, 36345, 36346, 36348, 36491, 36492, 36495, 36496, 36504, 36509, 36564, 36565, 36568, 36569, 36578, 36583
 - __coffin_join:NnnNnnnnN 36223, 36225, 36231, 36235

- \l__coffin_left_corner_dim 35954, 35985, 35993, 36070, 36076, 36077, 36100, 36108
- __coffin_mark_handle_aux:nnnnNnn 36475, 36503, 36508, 36512
- __coffin_offset_corner:Nnnnn 36354, 36357, 36359
- __coffin_offset_corners:Nnn 36260, 36261, 36267, 36268, 36354, 36354
- __coffin_offset_pole:Nnnnnnn 36335, 36338, 36340
- __coffin_offset_poles:Nnn 36258, 36259, 36264, 36265, 36301, 36302, 36335, 36335
- \l__coffin_offset_x_dim 35449, 36241, 36242, 36245, 36256, 36258, 36260, 36266, 36269, 36303, 36322, 36330, 36601, 36609
- \l__coffin_offset_y_dim 35449, 36259, 36261, 36266, 36269, 36303, 36324, 36331, 36603, 36610
- \l__coffin_pole_a_tl 35451, 35849, 35854, 36374, 36377, 36378, 36381, 36529, 36531, 36534
- \l__coffin_pole_b_tl 35451, 35850, 35854, 36375, 36377, 36379, 36381, 36530, 36531, 36533, 36534
- \c__coffin_poles_prop 35434, 35506, 35705, 35712
- \l__coffin_poles_prop 35951, 35970, 35974, 35997, 36002, 36039, 36106, 36147, 36151, 36157, 36163, 36202, 36217
- __coffin_reset_structure:N 35484, 35700, 35700, 35762, 36249, 36293
- __coffin_resize:NnnNN 36117, 36119, 36126, 36131
- __coffin_resize_common:NnnN 36141, 36143, 36143, 36182
- \l__coffin_right_corner_dim 35954, 35993, 36068, 36078, 36079
- __coffin_rotate:NnNNN 35958, 35959, 35962, 35964
- __coffin_rotate_bounding:nnn 35978, 36022, 36022
- __coffin_rotate_corner:Nnnn 35973, 36022, 36028
- __coffin_rotate_pole:Nnnnnn 35975, 36034, 36034
- __coffin_rotate_vector:nnNN 36024, 36030, 36036, 36037, 36046, 36046
- __coffin_rule:nn 36470, 36470, 36480, 36525
- __coffin_scale:NnnNN 36165, 36166, 36169, 36171
- __coffin_scale_corner:Nnnn 36150, 36193, 36193
- __coffin_scale_pole:Nnnnnn 36152, 36193, 36199
- __coffin_scale_vector:nnNN 36186, 36186, 36195, 36201
- \l__coffin_scale_x_fp 36113, 36133, 36153, 36173, 36175, 36181, 36189
- \l__coffin_scale_y_fp ... 36113, 36135, 36174, 36175, 36179, 36191
- \l__coffin_scaled_total_height_dim 36115, 36178, 36183
- \l__coffin_scaled_width_dim 36115, 36180, 36183
- __coffin_set_bounding:N 35976, 36004, 36004
- __coffin_set_horizontal_pole:NnnN 35714, 35715, 35718, 35720
- __coffin_set_pole:Nnn 35539, 35615, 35714, 35750, 36347, 36387, 36391, 36399, 36403
- __coffin_set_vertical:NnnNNN 35536, 35538, 35544, 35548
- __coffin_set_vertical:NnNNNNnw 35611, 35613, 35620, 35625
- __coffin_set_vertical_aux: 35536, 35555, 35573, 35631
- __coffin_set_vertical_pole:NnnN 35714, 35733, 35736, 35738
- __coffin_shift_corner:Nnnn 35996, 36096, 36096
- __coffin_shift_pole:Nnnnnn 35998, 36096, 36104
- __coffin_show:NNNnn 36645, 36652, 36655, 36657
- __coffin_show_structure:NN 36620, 36621, 36624, 36626, 36661
- \l__coffin_sin_fp 1394, 1397, 35948, 35966, 36053, 36060
- \l__coffin_slope_A_fp 35446
- \l__coffin_slope_B_fp 35446
- __coffin_to_value:N 35457, 35457, 35462, 35501, 35502, 35503, 35505, 35658, 35659, 35660, 35661, 35670, 35671, 35672, 35673, 35693, 35702, 35704, 35709, 35711, 35724, 35742, 35752, 35757, 35779, 35810, 35969, 35971, 35999, 36001, 36146, 36148, 36160, 36162, 36252, 36296, 36299, 36337, 36356, 36363, 36528, 36639

- \l_coffin_top_corner_dim
- .. [35954](#), [35990](#), [36067](#), [36082](#), [36083](#)
- __coffin_update_B:nnnnnnnnN
- [36372](#), [36380](#), [36395](#)
- __coffin_update_corners:N
- [35763](#), [35772](#), [35772](#)
- __coffin_update_corners:NN
- [35772](#), [35773](#), [35775](#), [35776](#)
- __coffin_update_corners:NNN
- [35772](#), [35778](#), [35782](#)
- __coffin_update_poles:N
- .. [35764](#), [35803](#), [35803](#), [36255](#), [36300](#)
- __coffin_update_poles:NN
- [35803](#), [35804](#), [35806](#), [35807](#)
- __coffin_update_poles:NNN
- [35803](#), [35809](#), [35813](#)
- __coffin_update_T:nnnnnnnnN
- [36372](#), [36376](#), [36383](#)
- __coffin_update_vertical_-
- poles:NNN [36271](#), [36304](#), [36372](#), [36372](#)
- \l_coffin_x_dim [35453](#), [35858](#),
- [35867](#), [35893](#), [35924](#), [35942](#), [36024](#),
- [36026](#), [36030](#), [36032](#), [36036](#), [36041](#),
- [36195](#), [36197](#), [36201](#), [36204](#), [36319](#),
- [36323](#), [36342](#), [36350](#), [36552](#), [36599](#)
- \l_coffin_x_prime_dim
- [35453](#), [36038](#),
- [36042](#), [36319](#), [36323](#), [36599](#), [36602](#)
- __coffin_x_shift_corner:Nnnn
- [36156](#), [36208](#), [36208](#)
- __coffin_x_shift_pole:Nnnnnn
- [36158](#), [36208](#), [36215](#)
- \l_coffin_y_dim [35453](#),
- [35859](#), [35871](#), [35889](#), [35938](#), [36024](#),
- [36026](#), [36030](#), [36032](#), [36036](#), [36041](#),
- [36195](#), [36197](#), [36201](#), [36204](#), [36320](#),
- [36325](#), [36343](#), [36350](#), [36553](#), [36600](#)
- \l_coffin_y_prime_dim
- [35453](#), [36038](#),
- [36043](#), [36320](#), [36325](#), [36600](#), [36604](#)
- color commands:
- color.sc [325](#)
- \color_ensure_current:
- [321](#), [1383](#), [35516](#),
- [35529](#), [35587](#), [35600](#), [36691](#), [36691](#)
- \color_export:nnN [326](#), [37506](#), [37506](#)
- \color_export:nnnN [326](#), [37506](#), [37516](#)
- \color_fill:n [325](#), [37358](#), [37358](#)
- \color_fill:nn [325](#), [37358](#), [37368](#)
- \l_color_fixed_model_tl [324](#), [36826](#),
- [36828](#), [37181](#), [37184](#), [37187](#), [37189](#),
- [37193](#), [37228](#), [37229](#), [37235](#), [37254](#),
- [37256](#), [37389](#), [37393](#), [37395](#), [37510](#)
- \color_group_begin:
- [321](#), [34781](#), [34785](#),
- [34790](#), [34797](#), [34802](#), [34810](#), [34816](#),
- [34830](#), [34836](#), [34843](#), [34848](#), [34861](#),
- [34863](#), [34867](#), [34872](#), [34877](#), [34882](#),
- [34889](#), [34894](#), [34901](#), [34906](#), [34914](#),
- [34920](#), [34935](#), [34941](#), [36689](#), [36689](#)
- \color_group_end: [321](#), [34781](#),
- [34785](#), [34790](#), [34797](#), [34802](#), [34822](#),
- [34843](#), [34848](#), [34861](#), [34863](#), [34867](#),
- [34872](#), [34877](#), [34882](#), [34889](#), [34894](#),
- [34901](#), [34906](#), [34927](#), [36689](#), [36690](#)
- \color_if_exist:n [36709](#)
- \color_if_exist:nTF [324](#), [36709](#),
- [36819](#), [36852](#), [36912](#), [37475](#), [38268](#)
- \color_if_exist_p:n [324](#), [36709](#)
- \color_log:n [324](#), [38259](#), [38261](#)
- \color_math:nn [325](#), [37261](#), [37261](#)
- \color_math:nn(n) [1429](#)
- \color_math:nnn [325](#), [37261](#), [37266](#)
- \l_color_math_active_tl
- [325](#), [37258](#), [37319](#)
- \color_model_new:nnn [327](#), [37628](#), [37628](#)
- \color_profile_apply:nn
- [328](#), [38230](#), [38230](#)
- \color_select:n [324](#), [36479](#),
- [36486](#), [36524](#), [36559](#), [37210](#), [37210](#)
- \color_select:nn [324](#), [37210](#), [37216](#)
- \color_set:nn [324](#), [37386](#), [37386](#)
- \color_set:nnn [324](#),
- [37386](#), [37432](#), [37495](#), [37496](#), [37497](#),
- [37498](#), [37499](#), [37500](#), [37501](#), [37502](#)
- \color_set_eq:nn [324](#), [37386](#), [37473](#)
- \color_show:n [324](#), [38259](#), [38259](#)
- \color_stroke:n [325](#), [37358](#), [37363](#)
- \color_stroke:nn [325](#), [37358](#), [37374](#)
- color internal commands:
- \g_color_alternative_model_prop
- [37615](#), [37727](#), [37825](#)
- \g_color_alternative_values_-
- prop [37620](#),
- [37742](#), [37756](#), [37766](#), [37980](#), [38082](#)
- __color_backend_devicen_-
- init:nnn [37995](#)
- __color_backend_iccbased_-
- device:nnn [38247](#), [38252](#), [38257](#)
- __color_backend_iccbased_-
- init:nnn [38227](#)
- __color_backend_reset: [36697](#), [37339](#)
- __color_backend_separation_-
- init:nnnnn [37743](#), [37757](#), [37767](#)
- __color_backend_separation_-
- init_CIELAB:nnn [37795](#)

- _color_check_model:N
- [36822](#), [37182](#), [37182](#)
- _color_check_model:nn
- [37182](#), [37186](#), [37196](#)
- \g_color_colorants_prop
- [37598](#), [37728](#), [38043](#)
- _color_convert:nnN [36727](#), [36727](#),
- [36729](#), [36841](#), [36937](#), [36948](#), [37189](#)
- _color_convert:nnnN . . . [36727](#),
- [36728](#), [36731](#), [36763](#), [37254](#), [37540](#)
- _color_convert_cmyk_cmyk:w . [36801](#)
- _color_convert_cmyk_gray:w . [36793](#)
- _color_convert_cmyk_rgb:w . [36795](#)
- _color_convert_devicen_-
- cmyk:nnnnnnnn [37807](#), [38103](#), [38106](#)
- _color_convert_devicen_-
- cmyk:nnnw . . . [37807](#), [38100](#), [38128](#)
- _color_convert_devicen_cmyk_-
- aux:nnnw . . . [37807](#), [38110](#), [38117](#)
- _color_convert_devicen_-
- gray:nnn [37807](#), [38135](#), [38138](#)
- _color_convert_devicen_gray:nw
- [37807](#), [38132](#), [38149](#)
- _color_convert_devicen_gray_-
- aux:nw [37807](#), [38140](#), [38143](#)
- _color_convert_devicen_-
- rgb:nnnnnn . . . [37807](#), [38156](#), [38159](#)
- _color_convert_devicen_-
- rgb:nnnw [37807](#), [38153](#), [38179](#)
- _color_convert_devicen_rgb_-
- aux:nnnw [37807](#), [38163](#), [38169](#)
- _color_convert_gray_cmyk:w . [36768](#)
- _color_convert_gray_gray:w . [36764](#)
- _color_convert_gray_gray:wuuuuu_-
- _color_convert_gray_rgb:wuuuuu_-
- _color_convert_gray_cmyk:wuuuuu_-
- _color_convert_cmyk_gray:wuuuuu_-
- _color_convert_cmyk_rgb:wuuuuu_-
- _color_convert_cmyk_cmyk:wuuuuu_-
- _color_convert_rgb_gray:wuuuuu_-
- _color_convert_rgb_rgb:wuuuuu_-
- _color_convert_rgb_cmyk:w . [36727](#)
- _color_convert_gray_rgb:w . [36766](#)
- _color_convert_rgb_cmyk:nnn . . .
- . . . [1417](#), [36727](#), [36776](#), [36781](#), [37149](#)
- _color_convert_rgb_cmyk:nnnn . .
- [36727](#), [36783](#), [36786](#)
- _color_convert_rgb_cmyk:w . . [36774](#)
- _color_convert_rgb_gray:w . . [36770](#)
- _color_convert_rgb_rgb:w . . [36772](#)
- \l_color_current_tl [1413](#),
- [36689](#), [36692](#), [36703](#), [36810](#), [36813](#),
- [36860](#), [37204](#), [37208](#), [37212](#), [37214](#),
- [37218](#), [37221](#), [37264](#), [37270](#), [37276](#),
- [37278](#), [37340](#), [37347](#), [37348](#), [37360](#),
- [37361](#), [37365](#), [37366](#), [37370](#), [37372](#),
- [37376](#), [37378](#), [37482](#), [37484](#), [37505](#)
- _color_draw:nnn [37358](#),
- [37361](#), [37366](#), [37372](#), [37378](#), [37380](#)
- _color_export:nN
- [37506](#), [37512](#), [37520](#), [37522](#)
- _color_export:nnnN
- [37506](#), [37523](#), [37524](#)
- _color_export:nnnNN
- [37535](#), [37535](#), [37555](#)
- _color_export_comma-sep-cmyk:Nw
- [37566](#)
- \c_color_export_comma-sep-cmyk_-
- tl [37545](#)
- _color_export_comma-sep-rgb:Nw
- [37571](#)
- \c_color_export_comma-sep-rgb_-
- tl [37545](#)
- _color_export_format_backend:nnN
- [37533](#), [37533](#)
- _color_export_format_comma-sep-cmyk:nnN
- [37550](#)
- _color_export_format_comma-sep-rgb:nnN
- [37550](#)
- _color_export_format_space-sep-cmyk:nnN
- [37550](#)
- _color_export_format_space-sep-rgb:nnN
- [37550](#)
- _color_export_HTML:n
- . . . [37571](#), [37577](#), [37578](#), [37579](#), [37582](#)
- _color_export_HTML:Nw [37571](#), [37573](#)
- \c_color_export_HTML_tl [37545](#)
- _color_export_space-sep-cmyk:Nw
- [37566](#)
- \c_color_export_space-sep-cmyk_-
- tl [37545](#)
- _color_export_space-sep-rgb:Nw
- [37571](#)
- \c_color_export_space-sep-rgb_-
- tl [37545](#)
- _color_extract:nnN
- [36721](#), [36721](#), [36726](#),
- [36866](#), [36905](#), [36906](#), [36914](#), [36929](#)
- \c_color_fallback_cmyk_tl . . . [37595](#)
- \c_color_fallback_gray_tl . . . [37595](#)
- \c_color_fallback_rgb_tl . . . [37595](#)
- _color_finalise_current:
- [37201](#), [37201](#), [37213](#), [37220](#)
- \c_color_icc_colorspace_-
- signatures_prop [38183](#), [38209](#)
- \l_color_ignore_error_bool
- [36708](#), [36895](#), [37420](#)

- \l__color_internal_int
..... [36705](#), [37976](#), [37979](#), [38035](#)
- \l__color_internal_prop
..... [37593](#), [37643](#), [37674](#),
[37687](#), [37702](#), [37781](#), [37809](#), [38196](#)
- \l__color_internal_tl
..... [36705](#), [36868](#), [36871](#),
[37415](#), [37422](#), [37424](#), [37426](#), [37427](#),
[37465](#), [37467](#), [37468](#), [37675](#), [37678](#),
[37688](#), [37691](#), [37703](#), [37705](#), [37782](#),
[37786](#), [37789](#), [37810](#), [37813](#), [37826](#),
[37829](#), [37831](#), [37974](#), [37985](#), [38008](#),
[38031](#), [38197](#), [38200](#), [38210](#), [38213](#)
- __color_math:nn
..... [37261](#), [37263](#), [37268](#), [37274](#)
- __color_math_scan:w [1430](#), [37280](#),
[37282](#), [37282](#), [37313](#), [37334](#), [37350](#)
- __color_math_scan_auxi:
..... [37282](#), [37294](#), [37298](#)
- __color_math_scan_auxii:
..... [37282](#), [37314](#), [37317](#)
- __color_math_scan_auxiii:N
..... [37326](#), [37332](#)
- __color_math_scan_end:
..... [37282](#), [37290](#), [37329](#), [37337](#)
- __color_math_script_aux:N
..... [37342](#), [37354](#), [37357](#)
- __color_math_scripts:Nw
..... [37305](#), [37342](#), [37342](#)
- \g__color_math_seq
..... [37260](#), [37276](#), [37340](#), [37347](#)
- __color_model:N
.... [36719](#), [36719](#), [36810](#), [37204](#),
[37404](#), [37426](#), [37465](#), [37482](#), [37505](#)
- __color_model_convert:nnn
..... [37672](#), [37769](#)
- __color_model_devicen:n [37807](#), [37807](#)
- __color_model_devicen:nn
..... [37807](#), [37812](#), [37820](#)
- __color_model_devicen:nnn
..... [37807](#), [37854](#), [37856](#)
- __color_model_devicen:nnnn
..... [37807](#), [37858](#), [37861](#)
- __color_model_devicen_colorant:n
..... [37807](#), [37998](#), [38041](#)
- __color_model_devicen_convert:n
..... [37807](#), [38073](#), [38078](#)
- __color_model_devicen_convert:nnn
..... [37807](#)
- __color_model_devicen_convert:nnnn
..... [37807](#), [37871](#), [38045](#), [38049](#)
- __color_model_devicen_convert:nnnnn
..... [38053](#), [38058](#), [38063](#), [38065](#)
- __color_model_devicen_convert:w
..... [37807](#)
- __color_model_devicen_convert_-
aux:n [37807](#), [38081](#), [38085](#)
- __color_model_devicen_convert_-
aux:w [38086](#), [38087](#)
- __color_model_devicen_convert_-
cmyk:n [37807](#)
- __color_model_devicen_convert_-
cmyk:nnn [38050](#)
- __color_model_devicen_convert_-
gray:n [37807](#)
- __color_model_devicen_convert_-
gray:nnn [38055](#)
- __color_model_devicen_convert_-
rgb:n [37807](#)
- __color_model_devicen_convert_-
rgb:nnn [38060](#)
- __color_model_devicen_init:nnn .
..... [37807](#), [37870](#), [37959](#)
- __color_model_devicen_init:nnnn
..... [37807](#), [37961](#), [37972](#)
- __color_model_devicen_mix:nw ...
..... [37807](#), [37931](#), [37950](#), [37956](#)
- __color_model_devicen_parse:nw .
..... [37807](#), [37926](#), [37936](#), [37945](#)
- __color_model_devicen_parse_-
1:nn [37807](#)
- __color_model_devicen_parse_-
2:nn [37807](#)
- __color_model_devicen_parse_-
3:nn [37807](#)
- __color_model_devicen_parse_-
4:nn [37807](#)
- __color_model_devicen_parse_-
generic:nn ... [37807](#), [37868](#), [37921](#)
- __color_model_devicen_transform:nnn
..... [37807](#)
- __color_model_devicen_transform:w
..... [37807](#)
- __color_model_devicen_transform_-
1:nnnnn [37807](#)
- __color_model_devicen_transform_-
3:nnnnn [37807](#)
- __color_model_devicen_transform_-
4:nnnnn [37807](#)
- __color_model_devicen_transform:nnn
..... [38018](#), [38022](#), [38027](#), [38029](#)
- __color_model_devicen_transform:w
..... [37982](#), [38011](#)
- __color_model_iccbased:n
..... [38194](#), [38194](#)
- __color_model_iccbased:nn
..... [38194](#), [38199](#), [38207](#)

- _color_model_iccbased:nnn .. [38194](#)
- _color_model_iccbased_aux:nnn .
..... [38194](#)
- _color_model_iccbased_aux:nnnnnn
..... [38212](#), [38220](#)
- _color_model_init:nnn .. [37651](#),
[37651](#), [37671](#), [37721](#), [37863](#), [38222](#)
- \g_color_model_int [37594](#),
[37653](#), [37659](#), [38246](#), [38251](#), [38256](#)
- _color_model_new:nnn
..... [37628](#), [37630](#), [37634](#)
- \c_color_model_range_CIELAB_t1 .
..... [37614](#)
- _color_model_separation:n
..... [37672](#), [37672](#)
- _color_model_separation:nn
..... [37672](#), [37677](#), [37685](#)
- _color_model_separation:nnn
..... [37672](#), [37690](#), [37698](#)
- _color_model_separation:w
..... [37672](#), [37705](#), [37718](#)
- _color_model_separation_-
CIELAB:nnnnnn [37672](#), [37779](#)
- _color_model_separation_-
CIELAB:nnnnnnn [37672](#), [37788](#), [37791](#)
- _color_model_separation_-
cmyk:nnnnnn [37672](#), [37731](#)
- _color_model_separation_-
gray:nnnnnn [37672](#), [37760](#)
- _color_model_separation_-
rgb:nnnnnn [37672](#), [37746](#)
- \l_color_model_t1
[36803](#), [36838](#), [36839](#), [36842](#), [36866](#),
[36869](#), [36907](#), [36915](#), [36917](#), [36924](#),
[36929](#), [36935](#), [36937](#), [36939](#), [36945](#),
[36950](#), [37187](#), [37189](#), [37198](#), [37822](#),
[37828](#), [37829](#), [37831](#), [37849](#), [37854](#)
- \c_color_model_whitepoint_-
CIELAB_a_t1 [37607](#)
- \c_color_model_whitepoint_-
CIELAB_b_t1 [37607](#)
- \c_color_model_whitepoint_-
CIELAB_d50_t1 [37607](#)
- \c_color_model_whitepoint_-
CIELAB_d55_t1 [37607](#)
- \c_color_model_whitepoint_-
CIELAB_d65_t1 [37607](#)
- \c_color_model_whitepoint_-
CIELAB_d75_t1 [37607](#)
- \c_color_model_whitepoint_-
CIELAB_e_t1 [37607](#)
- \l_color_named._prop [37503](#)
- \l_color_named._t1 [37503](#)
- \l_color_named_t1 . [37385](#), [37401](#),
[37404](#), [37407](#), [37464](#), [37465](#), [37469](#)
- \l_color_named_white_prop ... [37664](#)
- \l_color_next_model_t1 .. [36803](#),
[36914](#), [36915](#), [36935](#), [36936](#), [36949](#)
- \l_color_next_value_t1 .. [36803](#),
[36914](#), [36924](#), [36940](#), [36946](#), [36951](#)
- _color_parse:nN
..... [36807](#), [36807](#), [37212](#), [37264](#),
[37360](#), [37365](#), [37401](#), [37422](#), [37511](#)
- _color_parse:Nw [36807](#), [36821](#), [36850](#)
- _color_parse_aux:nN
..... [36807](#), [36814](#), [36817](#)
- _color_parse_break:w
..... [36807](#), [36930](#), [36954](#)
- _color_parse_end:
..... [36807](#), [36891](#), [36954](#), [36955](#)
- _color_parse_eq:Nn [36807](#)
- _color_parse_eq:nNn [36807](#)
- _color_parse_gray:n
..... [36807](#), [36918](#), [36933](#)
- _color_parse_loop:nn
..... [36807](#), [36883](#), [36910](#)
- _color_parse_loop:w
..... [36807](#), [36867](#), [36873](#), [36890](#)
- _color_parse_loop_check:nn ...
..... [36807](#), [36887](#), [36893](#)
- _color_parse_loop_init:Nnn ...
..... [36807](#), [36856](#), [36863](#)
- _color_parse_mix:Nnnn
..... [36807](#), [36923](#), [36956](#), [36962](#)
- _color_parse_mix:nNnn
..... [36807](#), [36958](#), [36963](#)
- _color_parse_mix_cmyk:nw
..... [36807](#), [36977](#), [37919](#)
- _color_parse_mix_gray:nw
..... [36807](#), [36968](#), [37722](#), [37880](#)
- _color_parse_mix_rgb:nw
..... [36807](#), [36970](#), [37904](#)
- _color_parse_model_&spot:w . [37179](#)
- _color_parse_model_cmy:w
..... [37146](#), [37146](#)
- _color_parse_model_cmyk:w
..... [36985](#), [36996](#)
- _color_parse_model_Gray:w ...
..... [37010](#), [37010](#)
- _color_parse_model_gray:w ...
..... [36985](#), [36985](#)
- _color_parse_model_hsb:nnn ...
..... [37010](#),
[37013](#), [37016](#), [37019](#), [37056](#), [37153](#)
- _color_parse_model_hsb:nnnn . [37010](#)
- _color_parse_model_hsb:nnnnn [37010](#)

- _color_parse_model_HSB:w 37879, 37885, 37899, 37900, 37901, 37913, 37914, 37915, 37916, 37943
- _color_parse_model_Hsb:w 37010, 37054
- _color_parse_model_hsb:w 37010, 37014
- _color_parse_model_hsb:w 37010, 37012
- _color_parse_model_hsb_0:nmnn 37010
- _color_parse_model_hsb_1:nmnn 37010
- _color_parse_model_hsb_2:nmnn 37010
- _color_parse_model_hsb_3:nmnn 37010
- _color_parse_model_hsb_4:nmnn 37010
- _color_parse_model_hsb_5:nmnn 37010
- _color_parse_model_hsb_aux:nmnn 37010, 37023, 37027, 37139
- _color_parse_model_hsb_ aux:nmnn 37029, 37033
- _color_parse_model_hsb_ aux:nmnn 37037, 37045
- _color_parse_model_HTML:w 37010, 37061
- _color_parse_model_HTML_aux:w 37062, 37063
- _color_parse_model_RGB:w 37010, 37072
- _color_parse_model_rgb:w 36985, 36987
- _color_parse_model_tHsb:n 37151, 37154, 37156
- _color_parse_model_tHsb:nw 37151, 37158, 37169, 37173
- _color_parse_model_tHsb:w 37151, 37151
- _color_parse_model_wave:w 37010, 37081
- _color_parse_model_wave_ auxi:nmnn 37010, 37086, 37090, 37091, 37095
- _color_parse_model_wave_ auxii:nmnn 37010, 37099, 37106, 37113, 37120, 37127, 37131, 37137
- _color_parse_model_wave_rho:n 37010, 37100, 37107, 37114, 37121, 37128, 37142, 37144
- _color_parse_number:n 36985, 36986, 36991, 36992, 36993, 37000, 37001, 37002, 37003, 37006, 37038, 37724, 37879, 37885, 37899, 37900, 37901, 37913, 37914, 37915, 37916, 37943
- _color_parse_number:w 36985, 37007, 37008
- _color_parse_set_eq:Nn 36820, 36824, 36855
- _color_parse_set_eq:nNn 36827, 36828, 36831
- _color_parse_std:n 36807, 36919, 36942
- _color_profile_apply:nmnn 38230, 38232, 38235
- _color_profile_apply_cmyk:n 38230, 38254
- _color_profile_apply_gray:n 38230, 38244
- _color_profile_apply_rgb:n 38230, 38249
- _color_select:N 36692, 36694, 36694, 37214, 37221, 37348
- _color_select:nmnn 36694, 36696, 36700, 36701
- _color_select:nmN 37210, 37226, 37236, 37243, 37464
- _color_select_loop:Nw 37210, 37230, 37232, 37240
- _color_select_main:Nw 37210, 37218, 37223, 37270, 37370, 37376, 37518
- _color_select_math:N 36694, 36699, 37278
- _color_select_swap:Nmnn 37210, 37239, 37252
- _color_set:nmnn 37386, 37394, 37397
- _color_set:nmnn 37386, 37388, 37391
- _color_set:nmw 37386, 37408, 37411
- _color_set_aux:nmnn 37386, 37438, 37442
- _color_set_colon:nmw 37386, 37444, 37449
- _color_set_loop:nw 37386, 37455, 37456, 37459, 37470
- _color_show:n 38259, 38270, 38279
- _color_show:Nn 38259, 38260, 38262, 38263
- _color_tmp:w 37551, 37559, 37560, 37561, 37562, 37563
- \l_color_value_tl 36803, 36835, 36836, 36840, 36842, 36846, 36866, 36869, 36907, 36921, 36924, 36929, 36938, 37190, 37193, 37199, 37254, 37256, 37981, 37983

- `__color_values:N` 21275, 21610, 21867, 21868, 22512,
..... 36719, 36720, 36813,
37208, 37407, 37427, 37469, 37484
- `\columnwidth` 35580
- `\compoundhyphenmode` 798
- `\contextversion` 10200, 10230, 10453, 10473
- `\copy` 180
- `\copyfont` 933
- `cos` 276
- `cosd` 277
- `cot` 276
- `cotd` 277
- `\count` 181, 19524
- `\countdef` 182
- `\cr` 183
- `\crampeddisplaystyle` 800
- `\crampedscriptscriptstyle` 801
- `\crampedtextstyle` 803
- `\crrcr` 804
- `\creationdate` 184
- cs commands:
 - `\cs:w` 21, 22, 551, 575, 663, 865, 867,
918, 1407, 1409, 1429, 1431, 1492,
1866, 1910, 1941, 2140, 2217, 2399,
2441, 2450, 2452, 2456, 2457, 2458,
2506, 2512, 2518, 2524, 2558, 2560,
2565, 2572, 2573, 2627, 2631, 2671,
2992, 4235, 7094, 7097, 8570, 8572,
10982, 11121, 12103, 14276, 14282,
17553, 17641, 18343, 18346, 19271,
20260, 20307, 20459, 20755, 21051,
21184, 21275, 21867, 21868, 22512,
23381, 23400, 23467, 24278, 24467,
24499, 24913, 24939, 24952, 24986,
25028, 25591, 25607, 27303, 28394,
29459, 29477, 30943, 34314, 34639
 - `\cs_argument_spec:N` ... 38802, 38803
 - `\cs_end:` 21,
425, 575, 663, 664, 865, 867, 918,
1407, 1410, 1429, 1431, 1435, 1492,
1842, 1855, 1866, 1884, 1899, 1910,
1924, 1935, 1941, 2068, 2140, 2217,
2399, 2441, 2450, 2452, 2456, 2457,
2458, 2506, 2512, 2518, 2524, 2558,
2560, 2565, 2572, 2573, 2627, 2631,
2671, 2992, 4236, 4241, 6903, 7110,
8567, 8573, 8575, 8577, 8579, 8581,
8583, 8585, 8587, 8589, 8591, 8593,
10982, 10997, 11000, 11001, 11110,
11121, 12103, 14282, 14285, 17553,
17641, 18298, 18323, 18334, 18343,
18346, 19271, 20258, 20260, 20305,
20307, 20459, 20755, 21051, 21184,
21275, 21610, 21867, 21868, 22512,
23384, 23400, 23475, 24281, 24471,
24503, 24919, 24945, 24958, 24989,
25031, 25597, 25613, 27303, 28397,
29459, 29483, 30948, 34314, 34639
- `\cs_generate_from_arg_count:NNnn`
..... 19, 2120, 2120, 2130,
2131, 2132, 2133, 2163, 3108, 3108
- `\cs_generate_variant:Nn`
..... 15, 32–34, 65, 418, 419,
2705, 2705, 2718, 2719, 3034, 3036,
3038, 3040, 3108, 3109, 3219, 3221,
3243, 3246, 3252, 3258, 4004, 5017,
6168, 6918, 6959, 7288, 7289, 7312,
7314, 8282, 8288, 8297, 8298, 8299,
8300, 8303, 8304, 8315, 8316, 8319,
8322, 8342, 8353, 8355, 8504, 8505,
8510, 8511, 8940, 8972, 9112, 9137,
9265, 9268, 9477, 9479, 9481, 9483,
9751, 10170, 10171, 10172, 10173,
10211, 10216, 10250, 10275, 10293,
10295, 10297, 10468, 10489, 10501,
10509, 10525, 10537, 10539, 10541,
10568, 10571, 10572, 10585, 10591,
10592, 10595, 10598, 10702, 11059,
11102, 11210, 11224, 11227, 11240,
11250, 11294, 11312, 11315, 11318,
11321, 11351, 11419, 11426, 11439,
11452, 11482, 11504, 11510, 11557,
12137, 12143, 12144, 12149, 12150,
12155, 12156, 12161, 12162, 12179,
12180, 12197, 12198, 12199, 12200,
12201, 12202, 12203, 12204, 12267,
12268, 12269, 12270, 12271, 12272,
12273, 12274, 12275, 12276, 12277,
12278, 12335, 12336, 12337, 12338,
12339, 12340, 12341, 12342, 12343,
12344, 12345, 12346, 12361, 12395,
12396, 12397, 12398, 12462, 12464,
12466, 12468, 12470, 12472, 12474,
12476, 12538, 12539, 12544, 12545,
12546, 12547, 12695, 12725, 12735,
12756, 12761, 12763, 12772, 12784,
12785, 12822, 12825, 12830, 12831,
12882, 12893, 13093, 13104, 13105,
13128, 13135, 13137, 13214, 13235,
13237, 13256, 13272, 13326, 13332,
13355, 13358, 13419, 13434, 13435,
13438, 13439, 13469, 13470, 13471,
13472, 13473, 13474, 13475, 13484,
13485, 13486, 13487, 13520, 13521,
13526, 13527, 13606, 13641, 13670,
13688, 13714, 13728, 13775, 13836,
13914, 13933, 13971, 13986, 14003,

14004, 14005, 14018, 14114, 14159,
 14166, 16390, 16391, 16461, 16468,
 16492, 16680, 16683, 16686, 16689,
 16692, 16721, 16722, 16723, 16724,
 16725, 16726, 16732, 16772, 16773,
 16774, 16775, 16776, 16777, 16792,
 16793, 16815, 16816, 16817, 16818,
 16823, 16824, 16825, 16826, 16843,
 16844, 16869, 16870, 16871, 16872,
 16878, 16879, 16955, 16956, 17004,
 17005, 17055, 17068, 17069, 17087,
 17113, 17114, 17166, 17172, 17200,
 17229, 17239, 17262, 17263, 17318,
 17362, 17385, 17399, 17401, 17402,
 17404, 17405, 17419, 17421, 17555,
 17558, 17579, 17594, 17595, 17600,
 17601, 17603, 17605, 17618, 17619,
 17620, 17621, 17630, 17631, 17632,
 17633, 17638, 17639, 17899, 17920,
 18257, 18261, 18287, 18295, 18307,
 18309, 18311, 18341, 18344, 18347,
 18432, 18433, 18473, 18474, 18475,
 18476, 18490, 18491, 18500, 18501,
 18502, 18503, 18513, 18514, 18515,
 18516, 18526, 18528, 18530, 18532,
 18544, 18569, 18570, 18597, 18599,
 18622, 18623, 18661, 18662, 18667,
 18668, 18759, 18767, 18797, 18800,
 18832, 18861, 18882, 18906, 18915,
 18972, 18979, 18988, 19021, 19023,
 19025, 19178, 19179, 19180, 19181,
 19912, 19975, 19997, 20000, 20003,
 20032, 20035, 20038, 20041, 20044,
 20047, 20104, 20122, 20136, 20139,
 20159, 20162, 20182, 20188, 20194,
 20200, 20234, 20235, 20236, 20287,
 20353, 20354, 20367, 20368, 20369,
 20370, 20395, 20400, 20402, 20407,
 20409, 20414, 20416, 20421, 20423,
 20430, 20565, 20580, 20603, 20617,
 20637, 20639, 20757, 20763, 20767,
 20768, 20773, 20774, 20783, 20784,
 20787, 20790, 20798, 20799, 20807,
 20808, 21166, 21186, 21192, 21195,
 21196, 21201, 21202, 21211, 21212,
 21214, 21216, 21221, 21222, 21227,
 21228, 21256, 21257, 21259, 21277,
 21283, 21288, 21289, 21294, 21295,
 21304, 21305, 21307, 21309, 21314,
 21315, 21320, 21321, 21327, 21475,
 21476, 21618, 21625, 21709, 21712,
 21728, 21783, 21795, 21897, 22008,
 22014, 22257, 22268, 22271, 22274,
 22285, 22288, 22294, 22305, 22308,
 22314, 22352, 22792, 22844, 22877,
 22888, 22926, 22929, 22939, 23021,
 23023, 23053, 23095, 23104, 23113,
 23122, 23185, 23187, 23724, 23727,
 25419, 25426, 25427, 25428, 25431,
 25432, 25435, 25436, 25441, 25442,
 25449, 25450, 25451, 25452, 25454,
 25456, 25757, 25809, 28892, 28946,
 29024, 29069, 29084, 29138, 29993,
 30013, 30042, 30096, 30104, 30112,
 30178, 30179, 30266, 30267, 30268,
 30269, 30278, 30279, 30298, 30299,
 30315, 30317, 30319, 30333, 30336,
 30415, 30454, 30460, 30473, 30532,
 30550, 30606, 30662, 30939, 31385,
 31730, 32055, 32146, 32342, 32546,
 34256, 34641, 34646, 34647, 34652,
 34653, 34658, 34659, 34664, 34665,
 34673, 34674, 34675, 34678, 34684,
 34687, 34693, 34696, 34702, 34705,
 34708, 34709, 34737, 34738, 34746,
 34749, 34752, 34761, 34779, 34792,
 34793, 34804, 34805, 34818, 34819,
 34838, 34839, 34858, 34859, 34884,
 34885, 34896, 34897, 34908, 34909,
 34922, 34923, 34943, 34944, 34947,
 34948, 34951, 34957, 34972, 34975,
 35093, 35099, 35139, 35142, 35159,
 35162, 35179, 35182, 35196, 35199,
 35217, 35220, 35246, 35249, 35255,
 35261, 35314, 35317, 35320, 35323,
 35371, 35374, 35487, 35496, 35509,
 35522, 35535, 35541, 35547, 35595,
 35608, 35617, 35624, 35664, 35676,
 35716, 35719, 35734, 35737, 35960,
 35963, 36123, 36130, 36167, 36170,
 36228, 36234, 36279, 36285, 36416,
 36519, 36595, 36622, 36625, 36647,
 36650, 36653, 36656, 36726, 36729,
 36730, 36763, 36962, 37671, 38049,
 38474, 38554, 38630, 38631, 38637,
 38807, 38827, 38830, 38833, 38855,
 38933, 39016, 39018, 39085, 39624
`\cs_gset:Nn` 19, 2135, 2212
`.cs_gset:Np` 242, 22053
`\cs_gset:Npe` 17, 1454,
 1459, 1461, 2002, 2021, 2025, 17209
`\cs_gset:Npn` . . . 14, 17, 1454, 1457,
 1492, 1499, 1501, 1502, 1503, 1504,
 1505, 1506, 1507, 1508, 1509, 1510,
 1511, 1512, 1513, 1514, 1515, 1516,
 1517, 1518, 1519, 1520, 1521, 1522,
 1523, 1524, 1525, 1526, 1527, 1528,
 1529, 1530, 1531, 1532, 1533, 1534,

- 1535, 1536, 1537, 1538, 1539, 1540,
 1541, 1542, 1543, 1544, 1545, 1546,
 1547, 1548, 1549, 1550, 1551, 1552,
 1553, 1554, 1555, 1556, 1558, 1559,
 1560, 1561, 1562, 1563, 1564, 1565,
 1566, 1591, 1593, 1595, 1600, 1672,
 1675, 1737, 1738, 1739, 1740, 1790,
 1792, 1794, 1796, 1801, 1807, 1808,
 1812, 1819, 1822, 1849, 1865, 1893,
 1909, 1912, 1914, 1916, 1918, 1922,
 1929, 1933, 1940, 1943, 1950, 1952,
 1954, 1973, 2001, 2021, 2024, 6935,
 9260, 9262, 9311, 11789, 16542,
 16543, 16581, 17204, 22062, 22064,
 24158, 29928, 29933, 30958, 32551
- `\cs_gset:Npx`
 17, 1454, 1461, 2003, 2021, 2026
- `\cs_gset_eq:NN` 20, 1744,
2048, 2052, 2053, 2054, 2055, 2065,
 8294, 8296, 9164, 10290, 10534,
 12114, 12116, 12135, 14435, 14439,
 16678, 16981, 17214, 17219, 19176,
 19973, 20002, 20046, 20138, 20151,
 20155, 20185, 20226, 20350, 20364,
 20380, 20568, 20576, 29764, 29834,
 30166, 30172, 31175, 31185, 39320
- `\cs_gset_nopar:Nn` 19, 2135, 2212
- `\cs_gset_nopar:Npe`
 17, 694, 730, 735, 1454, 1455,
 1999, 2010, 2016, 12131, 12141,
 13309, 13329, 13357, 13447, 19984,
 19991, 20002, 20046, 20138, 20161,
 20185, 20192, 20198, 20350, 20364,
 20380, 20390, 20394, 22780, 39431
- `\cs_gset_nopar:Npn`
 17, 1454, 1454, 1491, 1583, 1584,
 1998, 2010, 2015, 16350, 16649, 39166
- `\cs_gset_nopar:Npx`
 17, 1454, 1456, 2000, 2010, 2017
- `\cs_gset_protected:Nn` 19, 2135, 2212
- `.cs_gset_protected:Np` 242, 22053
- `\cs_gset_protected:Npe`
 17, 1454, 1469, 1471, 2008,
2039, 2043, 17884, 21016, 25821, 38760
- `\cs_gset_protected:Npn`
 17, 1454, 1467, 1475, 1477,
 1480, 1482, 1485, 1487, 1493, 1568,
 1569, 1575, 1581, 1582, 1585, 1597,
 1599, 1601, 1603, 1605, 1606, 1607,
 1609, 1618, 1620, 1622, 1624, 1626,
 1627, 1628, 1630, 1639, 1651, 1677,
 1694, 1713, 1721, 1729, 1741, 1743,
 1745, 1747, 1759, 1773, 1810, 1956,
 1969, 1971, 1975, 1977, 1979, 1987,
 1992, 2007, 2039, 2042, 3170, 3178,
 3191, 3601, 3623, 3898, 7540, 8862,
 10233, 10404, 10476, 11798, 12713,
 13442, 13674, 17272, 17873, 18781,
 20152, 20571, 21009, 22066, 22068,
 25814, 30638, 34047, 38754, 38771,
 38780, 38993, 38999, 39005, 39031,
 39039, 39068, 39073, 39430, 39431
- `\cs_gset_protected:Npx`
 17, 1454, 1471, 2009, 2039, 2044
- `\cs_gset_protected_nopar:Nn`
 19, 2135, 2212
- `\cs_gset_protected_nopar:Npe`
 17, 1454, 1464, 1466, 2005, 2030, 2034
- `\cs_gset_protected_nopar:Npn`
 17, 1454, 1462, 2004, 2030, 2033
- `\cs_gset_protected_nopar:Npx`
 17, 1454, 1466, 2006, 2030, 2035
- `\cs_if_eq:NN` 2256, 12592, 19415
- `\cs_if_eq:NNTF` 28,
1466, 2256, 2262, 2263, 2264, 2266,
 2267, 2268, 2270, 2271, 2272, 5643,
 9210, 9321, 20050, 20142, 21919,
 23923, 23933, 23959, 23961, 23963,
 24163, 29719, 29850, 31849, 31899,
 31919, 32345, 34186, 38754, 38780
- `\cs_if_eq_p:NN`
 28, 2256, 2261, 2265, 2269, 5660,
 31402, 31988, 31989, 34237, 34238
- `\cs_if_exist:N` 1826, 1840, 1853,
 8375, 8377, 12181, 12182, 16794,
 16796, 17606, 17608, 18317, 18319,
 18492, 18494, 20496, 20498, 20775,
 20777, 21203, 21205, 21296, 21298,
 23180, 23182, 25547, 25548, 30151,
 30153, 30607, 30609, 34666, 34668
- `\cs_if_exist:NTF`
 21, 28, 365, 624, 762,
903, 1572, 1578, 1826, 1838, 1880,
 1913, 1915, 1917, 1919, 1920, 2276,
 3168, 3186, 3189, 4289, 4295, 4301,
 5021, 5371, 7080, 8650, 8651, 8652,
 8654, 8658, 8687, 8731, 8752, 8856,
 8991, 9019, 9050, 9208, 9246, 9845,
 10200, 10204, 10230, 10453, 10457,
 10473, 10950, 11131, 11575, 11628,
 11749, 14041, 14042, 14051, 14406,
 14415, 14419, 14433, 17581, 17582,
 17583, 17584, 18269, 18270, 19488,
 19507, 21639, 21735, 21832, 21837,
 21865, 22381, 22420, 22428, 22440,
 22452, 22458, 22469, 22485, 22490,
 22584, 22645, 22654, 22764, 23063,
 24156, 24330, 25409, 29721, 29757,

- 29827, 29852, 29866, 29893, 30158,
 30482, 30583, 30636, 30676, 31319,
 31967, 32131, 32333, 32398, 32406,
 33673, 33680, 34369, 35460, 35462,
 36738, 37245, 37636, 37641, 37700,
 38703, 39095, 39104, 39311, 39676
 \cs_if_exist_p:N [28](#), [364](#), [1826](#), 9066,
 30559, 31935, 31997, 32127, 34198,
 35576, 36464, 38709, 38710, 38711
 \cs_if_exist_use:N [21](#), [391](#),
[1912](#), 1918, 1954, 6032, 9631, 9649,
 10671, 22454, 22473, 32139, 32208
 \cs_if_exist_use:NTF
 . [21](#), [1912](#), 1912, 1914, 1916, 1922,
 1933, 1950, 1951, 1952, 1953, 1955,
 2954, 3023, 4583, 4590, 4980, 4985,
 5029, 5439, 5525, 7016, 9179, 16495,
 23605, 24288, 24290, 32206, 32449,
 32455, 32517, 32519, 36735, 36749,
 37526, 37867, 38237, 39035, 39043
 \cs_if_free:N 1868, 1882, 1897
 \cs_if_free:NTF [28](#), [63](#), [624](#),
[1868](#), 1981, 2880, 2907, 9621, 39662
 \cs_if_free_p:N [27](#), [28](#), [63](#), [1868](#)
 \cs_log:N [21](#), [402](#), [2301](#), 2304, 2305, 2306
 \cs_meaning:N
 . [20](#), [376](#), [1416](#), 1417, [1432](#), 1433,
 1440, 1443, 2313, 8988, 30648, 39358
 \cs_new:Nn [17](#), [64](#), [2135](#), [2212](#)
 \cs_new:Npe [15](#), [40](#),
[417](#), [1990](#), 2002, [2021](#), 2028, 2731,
 3806, 4081, 4604, 4606, 4608, 4610,
 4612, 4614, 8337, 8343, 9601, 9603,
 9605, 9612, 10644, 10656, 11182,
 11483, 14047, 15484, 22685, 23365,
 24189, 24773, 25980, 27989, 27995,
 28607, 29436, 29575, 31552, 31605,
 31929, 32112, 37771, 37928, 38068
 \cs_new:Npn [14](#), [15](#), [19](#),
[63](#), [64](#), [423](#), [438](#), [1466](#), 1602, 1623,
[1990](#), 2001, [2021](#), 2027, 2106, 2108,
 2110, 2118, 2171, 2261, 2262, 2263,
 2264, 2265, 2266, 2267, 2268, 2269,
 2270, 2271, 2272, 2338, 2342, 2351,
 2360, 2369, 2372, 2381, 2382, 2392,
 2393, 2394, 2395, 2396, 2397, 2398,
 2400, 2402, 2404, 2417, 2423, 2429,
 2440, 2442, 2449, 2451, 2453, 2460,
 2461, 2463, 2465, 2467, 2469, 2474,
 2479, 2485, 2491, 2497, 2503, 2509,
 2515, 2521, 2527, 2534, 2541, 2548,
 2555, 2562, 2569, 2578, 2579, 2581,
 2586, 2591, 2593, 2603, 2604, 2606,
 2608, 2610, 2612, 2614, 2620, 2626,
 2628, 2634, 2636, 2643, 2650, 2651,
 2652, 2653, 2654, 2655, 2657, 2666,
 2668, 2671, 2672, 2673, 2675, 2677,
 2682, 2692, 2695, 2700, 2701, 2702,
 2703, 2772, 2793, 2815, 2818, 2826,
 2839, 2854, 2865, 2897, 2988, 2990,
 3233, 3389, 3402, 3407, 3413, 3414,
 3421, 3428, 3435, 3442, 3449, 3450,
 3452, 3459, 3465, 3559, 3564, 3565,
 3573, 3579, 3780, 3785, 3792, 3828,
 3834, 3839, 3854, 3877, 3882, 3934,
 3940, 3958, 3963, 3978, 3980, 3982,
 3989, 4005, 4007, 4010, 4016, 4274,
 4307, 4312, 4314, 4321, 4323, 4324,
 4329, 4335, 4357, 4359, 4361, 4581,
 4587, 4598, 4603, 4616, 4621, 4633,
 4648, 4658, 4670, 4693, 4698, 4799,
 4814, 4820, 4837, 4847, 5359, 5641,
 5656, 5690, 5706, 5753, 5791, 5822,
 5828, 5834, 5842, 5847, 5853, 5858,
 5872, 5887, 5896, 5904, 5906, 5958,
 5967, 6038, 6074, 6281, 6695, 6799,
 6802, 6823, 6825, 6831, 6833, 6840,
 6850, 7049, 7439, 7555, 7561, 7600,
 7607, 8240, 8325, 8387, 8388, 8397,
 8399, 8409, 8414, 8419, 8421, 8429,
 8430, 8431, 8432, 8433, 8434, 8435,
 8436, 8437, 8438, 8448, 8464, 8474,
 8490, 8500, 8502, 8506, 8508, 8512,
 8520, 8525, 8533, 8540, 8542, 8544,
 8546, 8548, 8553, 8559, 8562, 8569,
 8571, 8573, 8574, 8576, 8578, 8580,
 8582, 8584, 8586, 8588, 8590, 8592,
 8594, 8599, 8600, 8601, 8602, 8603,
 8604, 8605, 8606, 8607, 8608, 8620,
 8623, 9026, 9046, 9052, 9056, 9097,
 9241, 9310, 9345, 9407, 9412, 9417,
 9419, 9421, 9423, 9429, 9437, 9443,
 9449, 9582, 9584, 9619, 9752, 9774,
 9776, 9778, 10139, 10140, 10149,
 10162, 10164, 10166, 10168, 10388,
 10390, 10465, 10599, 10607, 10645,
 10662, 10717, 10768, 10777, 10796,
 10797, 10805, 10811, 10819, 10829,
 10834, 10840, 10846, 10919, 10921,
 10923, 10971, 10979, 10985, 10992,
 10993, 10999, 11001, 11008, 11013,
 11021, 11031, 11033, 11042, 11044,
 11045, 11047, 11097, 11103, 11108,
 11116, 11125, 11140, 11146, 11150,
 11156, 11158, 11169, 11170, 11171,
 11173, 11191, 11193, 11222, 11225,
 11228, 11233, 11238, 11241, 11243,
 11251, 11265, 11275, 11285, 11292,

11297, 11304, 11374, 11480, 11499,
11505, 11511, 11516, 11522, 11536,
11574, 11642, 11765, 11766, 11770,
11771, 12390, 12451, 12531, 12568,
12653, 12656, 12657, 12658, 12659,
12671, 12686, 12693, 12696, 12703,
12709, 12726, 12733, 12736, 12744,
12757, 12759, 12762, 12764, 12773,
12778, 12783, 12786, 12797, 12798,
12799, 12800, 12807, 12814, 12816,
12823, 12834, 12846, 12855, 12861,
12867, 12869, 12872, 12877, 12883,
12884, 12885, 12886, 12894, 12936,
12945, 12964, 12966, 12979, 12986,
13002, 13013, 13021, 13027, 13030,
13035, 13047, 13053, 13054, 13056,
13064, 13070, 13077, 13079, 13081,
13094, 13096, 13098, 13106, 13114,
13120, 13127, 13129, 13134, 13136,
13138, 13139, 13147, 13159, 13168,
13177, 13182, 13188, 13211, 13212,
13213, 13215, 13269, 13392, 13400,
13407, 13408, 13584, 13589, 13594,
13599, 13604, 13613, 13619, 13624,
13629, 13634, 13639, 13643, 13649,
13651, 13659, 13661, 13663, 13689,
13715, 13717, 13719, 13727, 13729,
13740, 13749, 13752, 13763, 13772,
13774, 13776, 13784, 13786, 13793,
13814, 13824, 13829, 13834, 13835,
13837, 13845, 13847, 13855, 13861,
13867, 13886, 13888, 13897, 13903,
13910, 13912, 13915, 13925, 13932,
13934, 13942, 13947, 13952, 13963,
13970, 13972, 13978, 13980, 13985,
13987, 13993, 13994, 13999, 14000,
14001, 14002, 14006, 14011, 14016,
14019, 14021, 14029, 14034, 14044,
14062, 14073, 14075, 14081, 14090,
14105, 14115, 14119, 14133, 14134,
14214, 14222, 14229, 14270, 14272,
14278, 14284, 14286, 14291, 14296,
14312, 14328, 14342, 14441, 14447,
14473, 14479, 14511, 14521, 14532,
14558, 14565, 14625, 14632, 14654,
14664, 14733, 14743, 14780, 14842,
14883, 14885, 14902, 14908, 14931,
14961, 14982, 14991, 15070, 15090,
15110, 15133, 15140, 15167, 15270,
15284, 15311, 15320, 15322, 15343,
15348, 15354, 15359, 15427, 15450,
15462, 15471, 15477, 15482, 16360,
16366, 16374, 16381, 16388, 16392,
16398, 16431, 16441, 16444, 16548,
16557, 16604, 16609, 16611, 16620,
16626, 16631, 16659, 16666, 16764,
16770, 16814, 16827, 16925, 16932,
16950, 17023, 17053, 17081, 17086,
17108, 17143, 17145, 17153, 17159,
17167, 17173, 17175, 17177, 17188,
17230, 17240, 17264, 17279, 17289,
17297, 17299, 17305, 17311, 17339,
17357, 17361, 17363, 17386, 17387,
17388, 17395, 17397, 17437, 17457,
17462, 17464, 17465, 17471, 17479,
17485, 17487, 17495, 17503, 17511,
17519, 17532, 17534, 17541, 17543,
17641, 17648, 17662, 17667, 17673,
17684, 17689, 17696, 17698, 17700,
17702, 17704, 17706, 17708, 17726,
17731, 17736, 17741, 17746, 17748,
17754, 17772, 17780, 17788, 17794,
17800, 17808, 17816, 17822, 17828,
17835, 17851, 17861, 17863, 17900,
17914, 17921, 17953, 17985, 17987,
17989, 17995, 18001, 18013, 18021,
18033, 18041, 18074, 18107, 18109,
18111, 18113, 18115, 18120, 18125,
18130, 18135, 18136, 18137, 18138,
18139, 18140, 18141, 18142, 18143,
18144, 18145, 18146, 18147, 18148,
18149, 18150, 18151, 18160, 18161,
18170, 18176, 18178, 18187, 18194,
18200, 18202, 18204, 18220, 18231,
18254, 18331, 18332, 18340, 18342,
18345, 18351, 18352, 18353, 18354,
18355, 18356, 18357, 18358, 18359,
18360, 18361, 18379, 18380, 18381,
18383, 18389, 18395, 18425, 18426,
18466, 18568, 18568, 18660, 18669,
18676, 18679, 18692, 18698, 18736,
18746, 18753, 18760, 18768, 18775,
18808, 18818, 18826, 18833, 18839,
18849, 18851, 18853, 18862, 18865,
18874, 18883, 18907, 18908, 18911,
18913, 18916, 18926, 18937, 18939,
18942, 18948, 18949, 18957, 18973,
18980, 18989, 18991, 19005, 19007,
19008, 19009, 19011, 19016, 19064,
19134, 19140, 19146, 19152, 19183,
19189, 19219, 19261, 19286, 19288,
19436, 19466, 19613, 19625, 19626,
19634, 19643, 19652, 19661, 19663,
19665, 19667, 19669, 19671, 19673,
19675, 19677, 19679, 19681, 19683,
19685, 19692, 19698, 19705, 19706,
19707, 19708, 19711, 19805, 19813,
19815, 19817, 19827, 19837, 19923,

19926, 19928, 19929, 19935, 19949,
19960, 19967, 20216, 20255, 20268,
20270, 20282, 20338, 20340, 20509,
20536, 20543, 20553, 20581, 20591,
20604, 20606, 20608, 20616, 20618,
20634, 20663, 20669, 20751, 20809,
20814, 20816, 20824, 20832, 20840,
20842, 20854, 20860, 20873, 20875,
20877, 20879, 20881, 20889, 20894,
20899, 20904, 20909, 20911, 20917,
20919, 20927, 20935, 20941, 20947,
20955, 20963, 20969, 20975, 20982,
20996, 21030, 21032, 21038, 21051,
21052, 21059, 21067, 21072, 21086,
21095, 21100, 21110, 21120, 21138,
21144, 21150, 21159, 21164, 21243,
21246, 21251, 21254, 21322, 21351,
21362, 21368, 21370, 21372, 21382,
21388, 21393, 21400, 21408, 21412,
21419, 21421, 21429, 21433, 21440,
21450, 21458, 21466, 21480, 21489,
21501, 21502, 21503, 21504, 21506,
21522, 21533, 21541, 21546, 21552,
21655, 21672, 21676, 21705, 21997,
22233, 22506, 22508, 22573, 22582,
22588, 22590, 22595, 22599, 22603,
22607, 22613, 22625, 22629, 22633,
22696, 22708, 22915, 22917, 22927,
22950, 23022, 23024, 23026, 23037,
23096, 23098, 23105, 23111, 23129,
23137, 23146, 23156, 23206, 23207,
23208, 23209, 23210, 23211, 23212,
23213, 23214, 23215, 23225, 23249,
23251, 23253, 23262, 23264, 23271,
23283, 23284, 23286, 23296, 23306,
23316, 23326, 23334, 23336, 23343,
23345, 23346, 23351, 23358, 23372,
23374, 23390, 23391, 23399, 23401,
23410, 23412, 23424, 23429, 23433,
23438, 23440, 23442, 23444, 23446,
23453, 23455, 23463, 23465, 23477,
23479, 23481, 23483, 23507, 23509,
23511, 23512, 23513, 23515, 23517,
23519, 23521, 23539, 23554, 23555,
23561, 23577, 23583, 23711, 23712,
23713, 23714, 23715, 23716, 23717,
23722, 23725, 23771, 23773, 23775,
23777, 23783, 23787, 23789, 23798,
23799, 23808, 23821, 23834, 23841,
23855, 23871, 23883, 23894, 23904,
23910, 23921, 23931, 23957, 23968,
23985, 23996, 24001, 24021, 24023,
24034, 24039, 24052, 24075, 24076,
24080, 24097, 24098, 24122, 24130,
24148, 24177, 24203, 24207, 24210,
24212, 24218, 24230, 24242, 24249,
24255, 24263, 24286, 24301, 24320,
24328, 24343, 24358, 24369, 24379,
24389, 24394, 24403, 24420, 24433,
24438, 24444, 24446, 24453, 24483,
24511, 24527, 24538, 24543, 24561,
24579, 24590, 24605, 24610, 24621,
24631, 24641, 24657, 24701, 24706,
24713, 24721, 24727, 24732, 24736,
24753, 24761, 24793, 24810, 24824,
24843, 24851, 24860, 24869, 24880,
24882, 24896, 24906, 24907, 24924,
24931, 24936, 24949, 24962, 24967,
24995, 25009, 25037, 25038, 25042,
25059, 25081, 25083, 25094, 25126,
25130, 25145, 25162, 25186, 25188,
25190, 25192, 25202, 25207, 25218,
25230, 25241, 25254, 25274, 25292,
25294, 25306, 25312, 25320, 25334,
25341, 25352, 25359, 25373, 25463,
25472, 25476, 25501, 25512, 25518,
25543, 25545, 25562, 25584, 25589,
25600, 25617, 25644, 25645, 25646,
25647, 25663, 25674, 25682, 25694,
25700, 25706, 25714, 25722, 25728,
25734, 25742, 25750, 25758, 25771,
25793, 25841, 25847, 25858, 25882,
25884, 25886, 25888, 25896, 25900,
25907, 25914, 25915, 25916, 25917,
25918, 25919, 25922, 25924, 25953,
25961, 25972, 25974, 25976, 25978,
25985, 26009, 26011, 26021, 26036,
26045, 26059, 26067, 26075, 26082,
26089, 26097, 26107, 26121, 26132,
26133, 26139, 26156, 26163, 26165,
26172, 26177, 26194, 26195, 26196,
26215, 26221, 26231, 26243, 26250,
26264, 26272, 26310, 26319, 26340,
26342, 26344, 26353, 26364, 26376,
26391, 26404, 26417, 26425, 26443,
26461, 26468, 26476, 26486, 26487,
26496, 26497, 26506, 26516, 26530,
26540, 26551, 26559, 26561, 26572,
26578, 26613, 26634, 26636, 26638,
26640, 26647, 26656, 26661, 26668,
26675, 26695, 26700, 26717, 26728,
26733, 26743, 26745, 26755, 26763,
26765, 26771, 26773, 26775, 26779,
26798, 26799, 26804, 26812, 26813,
26836, 26849, 26856, 26864, 26865,
26866, 26867, 26868, 26869, 26877,
26883, 26885, 26887, 26909, 26914,
26924, 26934, 26945, 26958, 26969,

26974, 26981, 26990, 26992, 27001,
27010, 27024, 27026, 27028, 27041,
27051, 27056, 27065, 27073, 27080,
27086, 27095, 27097, 27109, 27114,
27122, 27127, 27137, 27143, 27149,
27156, 27163, 27165, 27170, 27172,
27177, 27179, 27193, 27203, 27215,
27220, 27227, 27237, 27239, 27241,
27252, 27266, 27280, 27300, 27313,
27315, 27320, 27333, 27338, 27346,
27351, 27361, 27373, 27403, 27404,
27405, 27407, 27409, 27411, 27425,
27431, 27440, 27459, 27465, 27475,
27494, 27502, 27535, 27541, 27550,
27552, 27566, 27625, 27633, 27651,
27668, 27669, 27674, 27699, 27722,
27750, 27766, 27776, 27787, 27808,
27823, 27828, 27833, 27835, 27849,
27855, 27870, 27878, 27888, 27898,
27911, 27929, 27935, 27949, 27964,
28002, 28004, 28006, 28008, 28010,
28025, 28040, 28055, 28070, 28085,
28100, 28108, 28122, 28124, 28130,
28142, 28150, 28157, 28383, 28390,
28427, 28435, 28436, 28447, 28454,
28456, 28462, 28473, 28483, 28490,
28497, 28512, 28551, 28564, 28595,
28601, 28608, 28628, 28630, 28647,
28662, 28675, 28682, 28687, 28689,
28698, 28711, 28714, 28735, 28748,
28763, 28781, 28796, 28806, 28815,
28828, 28844, 28861, 28874, 28880,
28882, 28887, 28888, 28889, 28890,
28893, 28898, 28904, 28909, 28911,
28934, 28942, 28944, 28947, 28952,
28958, 28963, 28965, 28988, 29013,
29022, 29023, 29025, 29030, 29032,
29037, 29039, 29049, 29057, 29065,
29067, 29070, 29075, 29080, 29082,
29083, 29085, 29090, 29095, 29097,
29102, 29109, 29123, 29128, 29130,
29140, 29142, 29144, 29146, 29148,
29159, 29169, 29171, 29174, 29179,
29181, 29189, 29190, 29204, 29211,
29217, 29218, 29231, 29246, 29252,
29274, 29289, 29299, 29320, 29329,
29352, 29370, 29381, 29386, 29397,
29414, 29419, 29447, 29451, 29456,
29463, 29469, 29474, 29486, 29505,
29512, 29514, 29530, 29536, 29543,
29551, 29563, 29583, 29593, 29599,
29604, 29617, 29629, 29637, 29646,
29669, 29679, 29864, 29944, 29959,
29960, 30008, 30014, 30028, 30097,
30105, 30113, 30119, 30126, 30131,
30137, 30149, 30280, 30289, 30294,
30300, 30464, 30570, 30572, 30579,
30591, 30595, 30599, 30778, 30784,
30795, 30807, 30821, 30840, 30857,
30863, 30868, 30912, 30914, 30916,
30919, 30922, 30928, 30934, 30940,
30941, 30950, 30952, 30965, 30970,
30971, 31218, 31238, 31312, 31317,
31326, 31327, 31328, 31329, 31330,
31336, 31338, 31391, 31397, 31399,
31409, 31424, 31434, 31451, 31465,
31482, 31495, 31497, 31498, 31549,
31567, 31578, 31580, 31582, 31595,
31629, 31644, 31645, 31647, 31649,
31713, 31721, 31728, 31731, 31733,
31739, 31750, 31762, 31767, 31776,
31790, 31801, 31811, 31816, 31822,
31847, 31861, 31866, 31871, 31882,
31884, 31889, 31895, 31909, 31915,
31941, 31951, 31961, 31963, 31974,
31980, 31985, 31993, 31994, 32009,
32010, 32023, 32028, 32038, 32045,
32079, 32081, 32083, 32085, 32087,
32089, 32091, 32093, 32095, 32097,
32107, 32119, 32124, 32136, 32144,
32147, 32149, 32155, 32166, 32168,
32174, 32188, 32202, 32213, 32215,
32221, 32226, 32232, 32246, 32257,
32266, 32273, 32280, 32290, 32299,
32304, 32315, 32317, 32331, 32340,
32343, 32350, 32355, 32360, 32365,
32376, 32383, 32388, 32390, 32394,
32396, 32416, 32423, 32431, 32447,
32453, 32459, 32466, 32477, 32493,
32506, 32513, 32515, 32524, 32536,
32541, 32558, 32562, 32585, 32589,
32600, 32606, 32646, 32651, 32652,
32654, 32670, 32709, 32714, 32725,
32742, 32753, 32766, 32784, 32789,
32824, 32834, 32851, 32863, 32876,
32887, 32903, 32909, 32943, 32951,
32961, 32974, 33003, 33037, 33115,
33133, 33150, 33175, 33191, 33201,
33211, 33223, 33238, 33251, 33257,
33267, 33280, 33288, 33298, 33303,
33312, 33314, 33330, 33346, 33350,
33361, 33372, 33385, 33413, 33426,
33433, 33438, 33463, 33476, 33489,
33496, 33501, 33513, 33520, 33532,
33539, 33555, 33572, 33586, 33591,
33617, 33706, 33708, 33714, 33725,
33736, 33742, 33760, 33773, 33783,
33799, 33804, 33809, 33827, 33828,

- 33834, 33837, 33842, 33857, 33862,
 33880, 33882, 33884, 33886, 33888,
 33890, 33892, 33894, 33896, 33898,
 33903, 33909, 33916, 33923, 33933,
 33950, 33955, 33962, 33967, 33985,
 33987, 33988, 33993, 33999, 34005,
 34010, 34020, 34027, 34038, 34040,
 34042, 34058, 34067, 34074, 34076,
 34078, 34084, 34095, 34096, 34101,
 34106, 34113, 34124, 34129, 34131,
 34133, 34138, 34143, 34154, 34165,
 34170, 34177, 34182, 34193, 34195,
 34213, 34214, 34223, 34229, 34234,
 34246, 34314, 34367, 34632, 36719,
 36720, 36764, 36766, 36768, 36770,
 36772, 36774, 36781, 36786, 36793,
 36795, 36801, 36956, 36963, 36968,
 36970, 36977, 36985, 36987, 36996,
 37006, 37008, 37010, 37012, 37014,
 37019, 37027, 37033, 37045, 37047,
 37048, 37049, 37050, 37051, 37052,
 37053, 37054, 37061, 37063, 37072,
 37081, 37095, 37137, 37144, 37146,
 37151, 37156, 37169, 37179, 37524,
 37582, 37723, 37734, 37741, 37749,
 37755, 37763, 37765, 37797, 37799,
 37878, 37884, 37886, 37895, 37908,
 37923, 37936, 37950, 38041, 38067,
 38078, 38085, 38087, 38100, 38106,
 38117, 38132, 38138, 38143, 38153,
 38159, 38169, 38224, 38225, 38279,
 38475, 38480, 38526, 38555, 38560,
 38601, 38609, 38611, 38638, 38639,
 38643, 38651, 38663, 38690, 38692,
 38693, 38803, 38858, 38860, 38862,
 38864, 38867, 38869, 38871, 38873,
 38875, 38877, 38879, 38881, 38889,
 38891, 38901, 38904, 38908, 38911,
 38914, 38917, 38920, 38923, 38926,
 38928, 38930, 38932, 38944, 38946,
 38948, 38950, 38952, 38954, 38956,
 38958, 38960, 38962, 38964, 38966,
 38968, 38970, 38972, 38974, 38977,
 38980, 38982, 39025, 39028, 39086,
 39194, 39195, 39225, 39230, 39235,
 39244, 39257, 39261, 39272, 39295
 \cs_new:Npx . 15, 1990, 2003, 2021, 2029
 \cs_new_eq:NN 20, 65, 393,
 395, 898, 1746, 2048, 2056, 2061,
 2062, 2063, 2371, 2380, 2410, 2670,
 2698, 2699, 2938, 3537, 3538, 4271,
 4277, 4290, 4296, 4302, 4425, 4426,
 4427, 4526, 4534, 4555, 4594, 4595,
 4596, 4597, 4796, 6567, 6832, 7229,
 7761, 8279, 8281, 8301, 8302, 8636,
 8637, 8638, 8639, 8642, 8643, 8644,
 8645, 8982, 10210, 10232, 10322,
 10467, 10475, 10600, 11096, 11363,
 11406, 11467, 11473, 11760, 11761,
 11762, 12130, 12131, 12532, 12533,
 13418, 13432, 13433, 13436, 13437,
 13515, 13538, 13609, 13610, 13611,
 13612, 14148, 14153, 14160, 14490,
 14506, 14508, 15498, 16358, 16645,
 16648, 16673, 16693, 16694, 16695,
 16696, 16697, 16698, 16699, 16700,
 16873, 17400, 17403, 17406, 17407,
 17408, 17409, 17410, 17411, 17450,
 17451, 17452, 17453, 17454, 17585,
 17586, 17589, 17640, 17898, 18256,
 18260, 18375, 18428, 18429, 18434,
 18435, 18436, 18437, 18438, 18439,
 18440, 18441, 18442, 18443, 18444,
 18445, 18446, 18447, 18448, 18449,
 18596, 18598, 19182, 19340, 19342,
 19343, 19344, 19346, 19349, 19350,
 19701, 19702, 19703, 19905, 20746,
 20747, 20748, 21050, 21165, 21169,
 21170, 21193, 21194, 21248, 21249,
 21250, 21253, 21258, 21262, 21263,
 21324, 21325, 21326, 21330, 21331,
 21361, 22996, 22997, 23203, 23204,
 23205, 23409, 23576, 23854, 23882,
 23890, 23891, 23892, 23901, 23903,
 24700, 24819, 24820, 24821, 25418,
 25429, 25430, 29137, 29139, 29554,
 29557, 29728, 29859, 30234, 32186,
 32392, 32421, 32549, 32602, 32604,
 32668, 32707, 33199, 33348, 33630,
 33632, 33836, 33915, 33922, 33998,
 34004, 34631, 34670, 34671, 34672,
 34706, 34707, 34718, 34719, 34720,
 34825, 34856, 34857, 34930, 34945,
 34946, 35457, 35684, 35685, 35686,
 35687, 35688, 35689, 36689, 36690,
 37722, 37880, 37904, 37919, 39082
 \cs_new_nopar:Nn 17, 2135, 2212
 \cs_new_nopar:Npe
 15, 1990, 1999, 2010, 2019
 \cs_new_nopar:Npn
 . 15, 393, 394, 1990, 1998, 2010, 2018
 \cs_new_nopar:Npx
 15, 1990, 2000, 2010, 2020
 \cs_new_protected:Nn . 18, 2135, 2212
 \cs_new_protected:Npe
 . . 15, 417, 422, 1990, 2008, 2039,
 2046, 2720, 2724, 2729, 2888, 2892,
 4105, 5199, 5213, 5215, 9469, 9471,

- 9473, 9475, 10251, 11072, 11440,
 18286, 19172, 19852, 35573, 36807,
 37201, 37442, 37553, 37656, 37662
- \cs_new_protected:Npn
 15, 423, 1466,
 1608, 1629, 1990, 2007, 2039, 2045,
 2048, 2049, 2050, 2051, 2052, 2053,
 2054, 2055, 2056, 2061, 2062, 2063,
 2064, 2066, 2075, 2096, 2120, 2130,
 2132, 2143, 2152, 2274, 2283, 2285,
 2287, 2289, 2291, 2299, 2301, 2302,
 2304, 2305, 2307, 2315, 2317, 2319,
 2383, 2411, 2576, 2598, 2665, 2689,
 2705, 2718, 2736, 2740, 2743, 2752,
 2876, 2893, 2903, 2912, 2936, 2942,
 2950, 2961, 2963, 2965, 2967, 2969,
 2980, 2982, 2995, 3003, 3014, 3033,
 3035, 3037, 3039, 3041, 3128, 3147,
 3154, 3166, 3199, 3218, 3220, 3222,
 3241, 3244, 3247, 3253, 3259, 3275,
 3284, 3304, 3314, 3325, 3335, 3345,
 3346, 3353, 3359, 3369, 3379, 3475,
 3481, 3483, 3498, 3581, 3592, 3612,
 3634, 3644, 3646, 3670, 3677, 3689,
 3692, 3695, 3705, 3713, 3720, 3729,
 3744, 3761, 3772, 3887, 3889, 3896,
 3905, 3911, 3913, 3915, 3925, 3927,
 3929, 4017, 4036, 4047, 4066, 4089,
 4101, 4126, 4137, 4151, 4159, 4166,
 4168, 4178, 4188, 4219, 4225, 4227,
 4232, 4248, 4272, 4275, 4278, 4280,
 4292, 4298, 4304, 4363, 4365, 4366,
 4371, 4377, 4385, 4395, 4409, 4428,
 4446, 4448, 4456, 4468, 4487, 4489,
 4494, 4502, 4504, 4511, 4516, 4518,
 4520, 4527, 4535, 4537, 4539, 4541,
 4548, 4553, 4556, 4562, 4808, 4866,
 4878, 4889, 4902, 4935, 4964, 4973,
 4978, 4983, 4988, 5009, 5018, 5025,
 5034, 5039, 5046, 5059, 5061, 5063,
 5065, 5071, 5093, 5106, 5133, 5138,
 5156, 5170, 5205, 5226, 5228, 5236,
 5238, 5240, 5242, 5244, 5246, 5250,
 5261, 5277, 5290, 5296, 5307, 5320,
 5326, 5345, 5365, 5396, 5407, 5422,
 5435, 5453, 5461, 5466, 5468, 5470,
 5487, 5506, 5508, 5531, 5543, 5563,
 5584, 5591, 5598, 5609, 5616, 5622,
 5680, 5732, 5741, 5754, 5769, 5778,
 5797, 5816, 5973, 6044, 6054, 6056,
 6058, 6065, 6110, 6123, 6139, 6141,
 6143, 6148, 6163, 6169, 6192, 6212,
 6227, 6234, 6241, 6243, 6245, 6252,
 6266, 6282, 6291, 6305, 6317, 6334,
 6343, 6345, 6357, 6366, 6378, 6391,
 6398, 6418, 6449, 6483, 6501, 6510,
 6516, 6522, 6528, 6571, 6580, 6594,
 6613, 6639, 6644, 6654, 6666, 6704,
 6713, 6725, 6732, 6734, 6736, 6756,
 6761, 6767, 6778, 6783, 6788, 6804,
 6856, 6874, 6876, 6919, 6943, 6960,
 6966, 6968, 6988, 7014, 7025, 7034,
 7043, 7076, 7090, 7101, 7107, 7116,
 7124, 7159, 7165, 7168, 7176, 7182,
 7185, 7194, 7197, 7200, 7203, 7208,
 7217, 7220, 7223, 7228, 7234, 7239,
 7244, 7249, 7250, 7251, 7259, 7260,
 7261, 7284, 7286, 7290, 7298, 7300,
 7302, 7306, 7307, 7326, 7343, 7345,
 7347, 7349, 7366, 7368, 7370, 7385,
 7393, 7403, 7414, 7423, 7440, 7452,
 7462, 7471, 7511, 7548, 7550, 7579,
 7584, 7615, 7637, 7655, 7657, 7684,
 7686, 7715, 7732, 7743, 7748, 7762,
 7768, 7770, 7771, 7777, 7779, 7780,
 7786, 7788, 7790, 7796, 7798, 7800,
 7808, 7828, 7834, 7840, 7846, 7854,
 7856, 7858, 7860, 7862, 7864, 7866,
 7868, 7870, 7875, 7903, 7913, 7918,
 7927, 7929, 7943, 8256, 8258, 8260,
 8267, 8281, 8283, 8289, 8291, 8293,
 8295, 8305, 8310, 8317, 8320, 8348,
 8350, 8352, 8354, 8356, 8632, 8774,
 8791, 8844, 8850, 8869, 8880, 8903,
 8933, 8937, 8965, 8969, 8973, 8978,
 9030, 9088, 9090, 9092, 9113, 9143,
 9146, 9148, 9155, 9165, 9171, 9249,
 9257, 9266, 9269, 9276, 9317, 9346,
 9366, 9371, 9382, 9458, 9460, 9493,
 9516, 9572, 9586, 9629, 9651, 9652,
 9665, 9670, 9696, 9705, 9707, 9709,
 9726, 9753, 9755, 9757, 9758, 9760,
 9762, 9764, 9766, 9768, 9770, 9772,
 10210, 10214, 10228, 10239, 10260,
 10266, 10282, 10294, 10296, 10298,
 10310, 10311, 10312, 10339, 10341,
 10352, 10354, 10373, 10375, 10377,
 10392, 10394, 10396, 10402, 10409,
 10418, 10420, 10422, 10427, 10467,
 10471, 10483, 10490, 10502, 10510,
 10516, 10526, 10538, 10540, 10542,
 10554, 10555, 10556, 10566, 10569,
 10573, 10579, 10586, 10593, 10596,
 10608, 10639, 10650, 10668, 10703,
 10726, 10738, 10757, 10761, 10850,
 10866, 10875, 10883, 10892, 10899,
 10905, 11054, 11090, 11205, 11310,
 11313, 11316, 11319, 11338, 11346,

11414, 11420, 11427, 11428, 11433,
11453, 11468, 11474, 11546, 11551,
11558, 11559, 11560, 11587, 11600,
11612, 11622, 11624, 11626, 11635,
11643, 11767, 11768, 11773, 12109,
12111, 12113, 12115, 12117, 12122,
12132, 12138, 12145, 12147, 12151,
12153, 12157, 12159, 12163, 12171,
12189, 12191, 12193, 12195, 12205,
12210, 12215, 12220, 12228, 12236,
12241, 12246, 12251, 12259, 12279,
12281, 12286, 12291, 12299, 12307,
12309, 12314, 12319, 12327, 12355,
12362, 12364, 12366, 12368, 12380,
12399, 12414, 12432, 12454, 12456,
12458, 12460, 12478, 12500, 12506,
12534, 12536, 12540, 12542, 12626,
12627, 12628, 12710, 12723, 12750,
12752, 12754, 12826, 12828, 13100,
13102, 13234, 13236, 13238, 13254,
13257, 13270, 13273, 13306, 13308,
13310, 13312, 13321, 13327, 13333,
13350, 13351, 13353, 13356, 13359,
13370, 13375, 13380, 13388, 13390,
13440, 13444, 13449, 13454, 13459,
13464, 13476, 13478, 13480, 13482,
13488, 13503, 13516, 13518, 13522,
13524, 13671, 13686, 13695, 13705,
13707, 14154, 14161, 14170, 14302,
14318, 14334, 14340, 14344, 14346,
14366, 14386, 14394, 14404, 14413,
14497, 14505, 14507, 14509, 14519,
14525, 14549, 14560, 14566, 14569,
14571, 14576, 14602, 14608, 14614,
14642, 14716, 14764, 14817, 14900,
14906, 14929, 14959, 14980, 15055,
15151, 15156, 15158, 15160, 15236,
15238, 15240, 15245, 15255, 15334,
15339, 15341, 15394, 15396, 15398,
15403, 15413, 16347, 16449, 16451,
16462, 16469, 16475, 16493, 16502,
16507, 16512, 16517, 16522, 16528,
16533, 16540, 16546, 16555, 16565,
16567, 16569, 16574, 16579, 16591,
16636, 16675, 16681, 16684, 16687,
16690, 16701, 16706, 16711, 16716,
16727, 16733, 16735, 16737, 16739,
16741, 16778, 16780, 16782, 16788,
16790, 16798, 16806, 16819, 16821,
16829, 16831, 16833, 16845, 16847,
16849, 16874, 16876, 16886, 16892,
16905, 16914, 16939, 16941, 16943,
16968, 16969, 16970, 16995, 17027,
17035, 17045, 17056, 17058, 17060,
17062, 17070, 17088, 17090, 17092,
17201, 17206, 17211, 17217, 17223,
17252, 17269, 17319, 17321, 17323,
17329, 17331, 17333, 17418, 17420,
17422, 17550, 17556, 17559, 17592,
17593, 17596, 17598, 17602, 17604,
17610, 17612, 17614, 17616, 17622,
17624, 17626, 17628, 17634, 17636,
17642, 17865, 17867, 17869, 17876,
17878, 17880, 17892, 18258, 18262,
18285, 18290, 18296, 18305, 18308,
18310, 18312, 18348, 18349, 18350,
18362, 18363, 18364, 18382, 18430,
18450, 18452, 18454, 18477, 18479,
18481, 18496, 18498, 18504, 18506,
18508, 18517, 18519, 18521, 18534,
18542, 18545, 18547, 18549, 18557,
18585, 18602, 18604, 18606, 18624,
18626, 18628, 18663, 18665, 18709,
18776, 18792, 18798, 18801, 18803,
19022, 19024, 19026, 19044, 19045,
19046, 19062, 19066, 19068, 19070,
19072, 19074, 19076, 19078, 19080,
19082, 19084, 19086, 19088, 19090,
19092, 19094, 19096, 19098, 19100,
19102, 19104, 19106, 19108, 19110,
19112, 19114, 19116, 19118, 19120,
19122, 19124, 19126, 19128, 19130,
19132, 19136, 19138, 19142, 19144,
19148, 19150, 19154, 19166, 19712,
19714, 19716, 19721, 19728, 19737,
19748, 19753, 19767, 19775, 19793,
19795, 19797, 19799, 19801, 19803,
19863, 19880, 19885, 19892, 19897,
19899, 19914, 19915, 19921, 19924,
19943, 19970, 19976, 19981, 19998,
20001, 20004, 20010, 20017, 20022,
20030, 20033, 20036, 20039, 20042,
20045, 20048, 20061, 20067, 20077,
20086, 20092, 20105, 20110, 20123,
20134, 20137, 20140, 20149, 20157,
20160, 20163, 20169, 20175, 20177,
20183, 20189, 20195, 20201, 20206,
20217, 20222, 20223, 20229, 20248,
20288, 20302, 20314, 20322, 20341,
20347, 20355, 20361, 20387, 20389,
20391, 20393, 20437, 20455, 20462,
20470, 20486, 20566, 20636, 20638,
20640, 20676, 20683, 20705, 20712,
20720, 20730, 20752, 20758, 20764,
20765, 20769, 20771, 20779, 20781,
20785, 20788, 20791, 20793, 20800,
20802, 20883, 21005, 21012, 21024,
21167, 21171, 21181, 21187, 21197,

21199, 21207, 21209, 21213, 21215,
21217, 21219, 21223, 21225, 21260,
21264, 21272, 21278, 21284, 21286,
21290, 21292, 21300, 21302, 21306,
21308, 21310, 21312, 21316, 21318,
21328, 21332, 21601, 21608, 21616,
21619, 21626, 21631, 21636, 21649,
21659, 21684, 21690, 21707, 21710,
21713, 21729, 21731, 21733, 21749,
21760, 21762, 21764, 21781, 21784,
21786, 21796, 21810, 21823, 21830,
21848, 21850, 21852, 21871, 21879,
21884, 21898, 21907, 21934, 21943,
21952, 21985, 21998, 22009, 22015,
22017, 22019, 22021, 22023, 22025,
22027, 22029, 22031, 22033, 22035,
22037, 22039, 22041, 22043, 22045,
22047, 22049, 22051, 22053, 22055,
22057, 22059, 22061, 22063, 22065,
22067, 22069, 22071, 22073, 22075,
22077, 22079, 22081, 22083, 22085,
22087, 22089, 22091, 22093, 22095,
22097, 22099, 22101, 22103, 22105,
22107, 22109, 22111, 22113, 22115,
22117, 22119, 22121, 22123, 22125,
22127, 22129, 22131, 22133, 22135,
22137, 22139, 22141, 22143, 22145,
22147, 22149, 22151, 22153, 22155,
22157, 22159, 22161, 22163, 22165,
22167, 22169, 22171, 22173, 22175,
22177, 22179, 22181, 22183, 22185,
22187, 22189, 22191, 22193, 22195,
22197, 22199, 22201, 22203, 22205,
22212, 22239, 22241, 22247, 22258,
22269, 22272, 22275, 22286, 22289,
22295, 22306, 22309, 22315, 22323,
22328, 22333, 22353, 22358, 22362,
22368, 22373, 22379, 22393, 22416,
22435, 22450, 22464, 22480, 22516,
22540, 22572, 22659, 22661, 22663,
22776, 22782, 22867, 22869, 22930,
22965, 22991, 23000, 23009, 23044,
23046, 23054, 23065, 23073, 23080,
23086, 23114, 23123, 23165, 23170,
23184, 23186, 23188, 23216, 23219,
23330, 23603, 23620, 23622, 23624,
23626, 23654, 23656, 23658, 23660,
23680, 23682, 23684, 23686, 23688,
23690, 23692, 23694, 23696, 25417,
25420, 25422, 25424, 25433, 25434,
25437, 25439, 25443, 25444, 25445,
25446, 25447, 25453, 25455, 25457,
25528, 25530, 25810, 25817, 25829,
28640, 29497, 29706, 29717, 29733,
29737, 29743, 29748, 29752, 29768,
29772, 29820, 29822, 29837, 29842,
29883, 29888, 29965, 29967, 29981,
29994, 30033, 30043, 30055, 30060,
30070, 30078, 30084, 30163, 30169,
30180, 30189, 30191, 30193, 30195,
30197, 30235, 30236, 30238, 30240,
30242, 30244, 30270, 30274, 30316,
30318, 30320, 30331, 30334, 30337,
30376, 30381, 30388, 30394, 30396,
30418, 30420, 30432, 30443, 30455,
30471, 30476, 30489, 30502, 30510,
30519, 30533, 30544, 30551, 30632,
30645, 30652, 32050, 32608, 32613,
32615, 32617, 32619, 32624, 32629,
32631, 32633, 32635, 32640, 34044,
34251, 34634, 34636, 34642, 34644,
34648, 34650, 34654, 34656, 34660,
34662, 34676, 34679, 34685, 34688,
34694, 34697, 34703, 34710, 34712,
34714, 34716, 34733, 34735, 34744,
34747, 34750, 34753, 34755, 34762,
34780, 34782, 34787, 34794, 34799,
34806, 34812, 34820, 34826, 34832,
34840, 34845, 34850, 34852, 34854,
34860, 34862, 34864, 34869, 34874,
34879, 34886, 34891, 34898, 34903,
34910, 34916, 34924, 34931, 34937,
34949, 34952, 34970, 34973, 34976,
34986, 35020, 35031, 35042, 35053,
35064, 35075, 35088, 35094, 35100,
35115, 35122, 35132, 35137, 35140,
35143, 35157, 35160, 35163, 35177,
35180, 35183, 35194, 35197, 35200,
35215, 35218, 35221, 35230, 35244,
35247, 35250, 35256, 35262, 35275,
35312, 35315, 35318, 35321, 35324,
35369, 35372, 35375, 35470, 35479,
35488, 35497, 35510, 35523, 35536,
35542, 35548, 35583, 35596, 35609,
35610, 35611, 35618, 35625, 35651,
35652, 35653, 35665, 35690, 35700,
35707, 35714, 35717, 35720, 35732,
35735, 35738, 35750, 35755, 35760,
35766, 35772, 35774, 35776, 35782,
35803, 35805, 35807, 35813, 35847,
35862, 35919, 35958, 35961, 35964,
36004, 36022, 36028, 36034, 36046,
36065, 36074, 36085, 36091, 36096,
36104, 36117, 36124, 36131, 36143,
36165, 36168, 36171, 36186, 36193,
36199, 36208, 36215, 36223, 36229,
36235, 36274, 36280, 36286, 36307,
36316, 36335, 36340, 36354, 36359,

- 36372, 36383, 36395, 36409, 36470,
36475, 36512, 36520, 36544, 36588,
36596, 36620, 36623, 36626, 36645,
36648, 36651, 36654, 36657, 36691,
36694, 36699, 36701, 36721, 36727,
36731, 36817, 36824, 36831, 36850,
36863, 36873, 36893, 36910, 36933,
36942, 36954, 36955, 37182, 37196,
37210, 37216, 37223, 37232, 37243,
37252, 37261, 37266, 37274, 37282,
37298, 37317, 37332, 37337, 37342,
37357, 37358, 37363, 37368, 37374,
37380, 37386, 37391, 37397, 37411,
37432, 37449, 37459, 37473, 37506,
37516, 37522, 37533, 37535, 37566,
37569, 37571, 37573, 37591, 37628,
37634, 37651, 37672, 37685, 37698,
37718, 37731, 37746, 37760, 37769,
37779, 37791, 37807, 37820, 37856,
37861, 37876, 37882, 37893, 37906,
37921, 37959, 37972, 38011, 38017,
38019, 38024, 38029, 38045, 38050,
38055, 38060, 38065, 38194, 38207,
38220, 38230, 38235, 38244, 38249,
38254, 38259, 38261, 38263, 38454,
38463, 38468, 38521, 38542, 38548,
38586, 38595, 38632, 38675, 38677,
38682, 38694, 38696, 38698, 38744,
38746, 38758, 38777, 38784, 38806,
38809, 38811, 38813, 38815, 38817,
38819, 38821, 38823, 38826, 38829,
38832, 38836, 38838, 38842, 38848,
38883, 38887, 38893, 38895, 38898,
38937, 38939, 38941, 39013, 39015,
39047, 39057, 39083, 39084, 39087,
39088, 39089, 39090, 39138, 39147,
39149, 39161, 39174, 39179, 39181,
39182, 39192, 39209, 39211, 39216
- `\cs_new_protected:Npx`
15, 1990, 2009, 2039, 2047, 2137, 2214
- `\cs_new_protected_nopar:Nn`
. 18, 2135, 2212
- `\cs_new_protected_nopar:Npe`
. 16, 1990, 2005, 2030, 2037
- `\cs_new_protected_nopar:Npn`
. 16, 1990, 2004, 2011, 2030, 2036
- `\cs_new_protected_nopar:Npx`
. 16, 1990, 2006, 2030, 2038
- `\cs_parameter_spec:N`
. 23, 2336, 2351, 13243,
13278, 38802, 38803, 39384, 39797
- `\cs_prefix_spec:N` . 22, 2336, 2342,
13243, 13278, 39324, 39382, 39796
- `\cs_replacement_spec:N` . . 23, 2336,
2360, 3108, 3109, 22674, 31693, 39387
- `\cs_set:Nn` 18, 398, 2135, 2212
- `.cs_set:Np` 242, 22053
- `\cs_set:Npe`
. 16, 1472, 1477, 1479, 2021, 2022,
6268, 10673, 10674, 10675, 10676,
10677, 12511, 14616, 14644, 18713,
19723, 19740, 19780, 19786, 29844
- `\cs_set:Npn` . . . 14, 16, 63, 64, 384,
394, 398, 926, 1472, 1475, 1598,
1619, 1621, 1990, 2010, 2021, 2021,
2135, 2212, 3214, 4566, 4567, 4568,
4897, 4898, 5474, 5476, 5493, 5495,
5735, 5736, 6000, 6001, 6002, 6003,
6029, 6616, 6921, 8986, 9271, 9273,
10615, 10937, 12440, 12509, 12635,
13505, 15261, 15418, 16602, 16624,
18637, 18726, 19217, 19725, 19739,
20208, 22054, 22056, 22611, 23629,
23637, 23646, 23663, 23671, 23699,
29708, 29914, 31044, 31046, 38782,
39110, 39142, 39184, 39193, 39339
- `\cs_set:Npx`
. 16, 406, 1472, 1479, 2021, 2023
- `\cs_set_eq:NN` 20, 65, 395, 585, 920,
923, 929, 1742, 2048, 2048, 2049,
2050, 2051, 2052, 2059, 2724, 2742,
2892, 3477, 3478, 3482, 3673, 3716,
3741, 4107, 4147, 4423, 5992, 6026,
6622, 6671, 7515, 7543, 7843, 7881,
7882, 7884, 7885, 7886, 7907, 8290,
8292, 8666, 10679, 10680, 10681,
10682, 10684, 10686, 10687, 12110,
12112, 14353, 14362, 15163, 16945,
16946, 16948, 17094, 17095, 17106,
18301, 19175, 19377, 19719, 19778,
19785, 19999, 20043, 20107, 20128,
20135, 20179, 20344, 20358, 20374,
21614, 21819, 21827, 21913, 29745,
29746, 29762, 29777, 29885, 29894,
29972, 29973, 30548, 39071, 39079
- `\cs_set_nopar:Nn` 18, 2135, 2212
- `\cs_set_nopar:Npe`
. 16, 463, 730, 920, 922,
923, 929, 932, 976, 979, 994, 1472,
1473, 2010, 2013, 2413, 2600, 4068,
12130, 13307, 13323, 13354, 19999,
20043, 20107, 20108, 20128, 20135,
20158, 20179, 20344, 20358, 20374,
20388, 20392, 21663, 21681, 21804,
22365, 22370, 29905, 31366, 39430
- `\cs_set_nopar:Npn` 15, 16,
201, 394, 1478, 1472, 1472, 2010,
2012, 20688, 21751, 22377, 29779,

- 30987, 30988, 30989, 30993, 30994,
30998, 31348, 31349, 31358, 31360
- `\cs_set_nopar:Npx`
16, 1472, 1474, 1495, 2010, 2014, 2156
- `\cs_set_protected:Nn` 18, 2135, 2212
- `.cs_set_protected:Np` 242, 22053
- `\cs_set_protected:Npe`
. 16, 110, 1472, 1487,
1489, 2039, 2040, 9637, 21788, 31078
- `\cs_set_protected:Npn`
. 15, 16, 395, 123, 1472,
1485, 1604, 1625, 2039, 2039, 2910,
3057, 3479, 4021, 5211, 5248, 5977,
5986, 5988, 5990, 5993, 5995, 6004,
6006, 6011, 6013, 6018, 6020, 6022,
6024, 6027, 6780, 6781, 7304, 9390,
9455, 10137, 10594, 10597, 10736,
10817, 10917, 10933, 12642, 12832,
13019, 13414, 14762, 14815, 17011,
18863, 19164, 19249, 19462, 19481,
19861, 21070, 21235, 21349, 21478,
21520, 21785, 22058, 22060, 22546,
23582, 24078, 24154, 24734, 24808,
24822, 25040, 25057, 25092, 25128,
25143, 25160, 26777, 29549, 29581,
30995, 31012, 31022, 31031, 31052,
31071, 31093, 31098, 31111, 31133,
31165, 31173, 31189, 31196, 31245,
31261, 31278, 31286, 31350, 31364,
31682, 33635, 33668, 34330, 34383,
34409, 35588, 35601, 35632, 37551,
38775, 38781, 39092, 39101, 39120,
39125, 39131, 39140, 39141, 39143,
39144, 39145, 39176, 39180, 39298,
39304, 39318, 39335, 39360, 39366,
39375, 39784, 39790, 39802, 39863,
39911, 39946, 39970, 40022, 40073
- `\cs_set_protected:Npx`
. 16, 1472, 1489, 2039, 2041
- `\cs_set_protected_nopar:Nn`
. 18, 2135, 2212
- `\cs_set_protected_nopar:Npe`
. 16, 1472, 1482, 1484, 2030, 2031
- `\cs_set_protected_nopar:Npn`
. 16, 394, 1472, 1480, 2030, 2030
- `\cs_set_protected_nopar:Npx`
. 16, 1472, 1484, 2030, 2032
- `\cs_show:N` 20,
21, 28, 402, 2301, 2301, 2302, 2303
- `\cs_split_function:N` 22, 1614,
1635, 1752, 1753, 1810, 1812, 2107,
2148, 2711, 3000, 16465, 16486, 39219
- `\cs_to_str:N`
. 6, 22, 115, 129, 388, 732,
753, 1801, 1801, 1816, 4309, 5807,
5943, 10600, 13311, 13373, 13378,
13386, 14135, 14136, 14137, 14138,
14139, 14140, 14141, 14142, 14143,
14144, 14145, 14146, 16607, 16629,
18286, 23580, 30167, 30173, 30175,
30183, 30246, 30250, 30257, 30302,
30307, 30343, 31959, 34318, 39148,
39311, 39320, 39329, 39343, 39352
- `\cs_undefine:N` 20, 865, 975, 982,
2064, 2064, 2066, 2072, 9506, 9507,
9508, 10235, 10478, 11763, 11764,
13046, 13302, 29761, 29831, 29832,
29999, 30506, 30567, 31176, 31187
- cs internal commands:
- `__cs_count_signature:N`
. 387, 2106, 2106, 2118, 2119
- `__cs_count_signature:n`
. 2106, 2107, 2108
- `__cs_count_signature:nnN`
. 2106, 2109, 2110
- `__cs_generate_from_signature:n`
. 2157, 2171
- `__cs_generate_from_signature:NNn`
. 2139, 2143
- `__cs_generate_from_signature:nnNNn`
. 2147, 2152
- `__cs_generate_internal_c:NN` 2963
- `__cs_generate_internal_end:w`
. 2946, 2980
- `__cs_generate_internal_long:nnnNNn`
. 2984, 2988
- `__cs_generate_internal_long:w`
. 2947, 2982
- `__cs_generate_internal_loop:wnnnw`
. 2944,
2950, 2962, 2964, 2966, 2968, 2971
- `__cs_generate_internal_N:NN` 2961
- `__cs_generate_internal_n:NN` 2965
- `__cs_generate_internal_one_-
go:NNn` 423, 2933, 2942
- `__cs_generate_internal_other:NN`
. 2955, 2969
- `__cs_generate_internal_test:Nw`
. 2918, 2938
- `__cs_generate_internal_test_-
aux:w` 2920, 2936, 2939
- `__cs_generate_internal_variant:n`
. 426, 2883, 2888, 2888, 3054, 3060
- `__cs_generate_internal_variant:NNn`
. 423, 2908, 2912
- `__cs_generate_internal_variant:wnnNwn`
. 2890, 2903

- __cs_generate_internal_variant_-
 loop:n . 2888, 2928, 2985, 2990, 2993
- __cs_generate_internal_x:NN . 2967
- __cs_generate_variant:N
 2707, 2720, 2720
- __cs_generate_variant:n 2995
- __cs_generate_variant:nnNN
 2710, 2743, 2743
- __cs_generate_variant:nnNnn
 2995, 2999, 3003
- __cs_generate_variant:Nnnw
 2750, 2752, 2752, 2770
- __cs_generate_variant:w
 2995, 3010, 3014, 3031
- __cs_generate_variant:ww
 2720, 2726, 2736
- __cs_generate_variant:wwNN
 419, 420, 2759, 2876, 2876, 39672
- __cs_generate_variant:wwNw
 2720, 2738, 2740
- __cs_generate_variant_F_-
 form:nnn 2995, 3037
- __cs_generate_variant_loop:nNwN
 419, 420, 2760, 2772, 2772, 2791
- __cs_generate_variant_loop_-
 base:N 2772, 2777, 2780, 2793
- __cs_generate_variant_loop_-
 end:nwwwNNnn
 419, 420, 2762, 2772, 2818
- __cs_generate_variant_loop_-
 invalid:NNwNNnn
 419, 2772, 2784, 2839
- __cs_generate_variant_loop_-
 long:wNNnn . . 420, 2765, 2772, 2826
- __cs_generate_variant_loop_-
 same:w 419, 2772, 2775, 2815
- __cs_generate_variant_loop_-
 special:NNwNNnn
 2772, 2782, 2854, 2871
- __cs_generate_variant_p_-
 form:nnn 2995, 3033
- __cs_generate_variant_same:N
 419, 2817, 2865, 2865
- __cs_generate_variant_T_-
 form:nnn 2995, 3035
- __cs_generate_variant_TF_-
 form:nnn 2995, 3039
- __cs_if_exist_c_aux: 1826, 1843, 1849
- __cs_if_exist_c_aux:w
 1826, 1856, 1865
- __cs_if_exist_use_aux:Nnn
 1912, 1930, 1941, 1943
- __cs_if_exist_use_aux:w
 1912, 1925, 1929, 1936, 1940
- __cs_if_free_c_aux:w
 1885, 1893, 1900, 1909
- __cs_parm_from_arg_count_-
 test:nnTF 2075, 2077, 2096
- __cs_split_function_auxi:w
 1810, 1815, 1819
- __cs_split_function_auxii:w
 1810, 1821, 1822
- __cs_tmp:w 387, 417, 422, 426, 1810,
 1825, 1990, 1990, 1998, 1999, 2000,
 2001, 2002, 2003, 2004, 2005, 2006,
 2007, 2008, 2009, 2010, 2012, 2013,
 2014, 2015, 2016, 2017, 2018, 2019,
 2020, 2021, 2022, 2023, 2024, 2025,
 2026, 2027, 2028, 2029, 2030, 2031,
 2032, 2033, 2034, 2035, 2036, 2037,
 2038, 2039, 2040, 2041, 2042, 2043,
 2044, 2045, 2046, 2047, 2135, 2156,
 2158, 2161, 2176, 2177, 2178, 2179,
 2180, 2181, 2182, 2183, 2184, 2185,
 2186, 2187, 2188, 2189, 2190, 2191,
 2192, 2193, 2194, 2195, 2196, 2197,
 2198, 2199, 2200, 2201, 2202, 2203,
 2204, 2205, 2206, 2207, 2208, 2209,
 2210, 2211, 2212, 2220, 2221, 2222,
 2223, 2224, 2225, 2226, 2227, 2228,
 2229, 2230, 2231, 2232, 2233, 2234,
 2235, 2236, 2237, 2238, 2239, 2240,
 2241, 2242, 2243, 2244, 2245, 2246,
 2247, 2248, 2249, 2250, 2251, 2252,
 2253, 2254, 2255, 2724, 2742, 2884,
 2892, 2910, 2941, 3057, 3064, 3065,
 3066, 3067, 3068, 3069, 3070, 3071,
 3072, 3073, 3074, 3075, 3076, 3077,
 3078, 3079, 3080, 3081, 3082, 3083,
 3084, 3085, 3086, 3087, 3088, 3089,
 3090, 3091, 3092, 3093, 3094, 3095,
 3096, 3097, 3098, 3099, 3100, 3101,
 3102, 3103, 3104, 3105, 3106, 3107
- __cs_to_str:N
 388, 1801, 1805, 1807, 1808
- __cs_to_str:w . 388, 1801, 1804, 1808
- __cs_use_i_delimit_by_s_stop:nw
 2701, 2702, 3008
- __cs_use_none_delimit_by_q_-
 recursion_stop:w
 2701, 2703, 2748, 2755, 3021
- __cs_use_none_delimit_by_s_-
 stop:w 2701, 2701, 3012
- csc 276
- cscd 277
- \csname 663,
 4, 8, 13, 17, 30, 53, 54, 61, 94, 185
- \csstring 805

- \currentcjktoken 1136, 1200
 - \currentgrouplevel 477
 - \currentgrouptype 478
 - \currentifbranch 479
 - \currentiflevel 480
 - \currentifttype 481
 - \currentspacingmode 1137
 - \currentxspacingmode 1138
- D**
- \d 32057, 34380, 34402
 - \date 366
 - \day 186, 1290, 9010
 - dd 280
 - \deadcycles 187
 - debug commands:
 - \debug_off:n . 30, 366, 1465, 1466, 1569, 1575, 1579, 39031, 39039, 39059
 - \debug_on:n ... 30, 366, 694, 1465, 1569, 1569, 1573, 39031, 39031, 39049
 - \debug_resume: 30, 1258, 1383, 1476, 1581, 1582, 30392, 35507, 39067, 39073
 - \debug_suspend: 30, 1258, 1383, 1476, 1581, 1581, 30390, 35500, 39067, 39068
 - debug internal commands:
 - __debug_add_to_debug_code:Nnn .. 39297, 39312, 39335
 - __debug_all_off: 39031, 39057
 - __debug_all_on: 1572, 1578, 39031, 39047
 - __debug_arg_check_invalid:N ... 39216, 39238, 39244
 - __debug_arg_if_braced:N 39259
 - __debug_arg_if_braced:n 39216, 39260, 39261
 - __debug_arg_if_braced:NTF 39216, 39239
 - __debug_arg_list_from_signature:nNN .. 39216, 39227, 39232, 39235, 39241
 - __debug_arg_return:N 39216, 39274, 39275, 39276, 39277, 39278, 39279, 39280, 39281, 39282, 39283, 39295
 - __debug_build_arg_list:n 39216, 39223, 39230
 - __debug_build_parm_text:n 39216, 39221, 39225
 - __debug_check-declarations_off: 39083
 - __debug_check-declarations_on: . 39083
 - __debug_check-expressions_off: . 39182
 - __debug_check-expressions_on: 39182
 - __debug_chk_expr_aux:nNnN 39182, 39187, 39195
 - __debug_chk_var_scope_aux:NN ... 39123, 39129, 39135, 39147, 39147
 - __debug_chk_var_scope_aux:Nn ... 39147, 39148, 39149
 - __debug_chk_var_scope_aux:NNn .. 1478, 39147, 39152, 39156, 39161
 - __debug_deprecation_off: 39209, 39211
 - \g__debug_deprecation_off_tl ... 1583, 39212
 - __debug_deprecation_on: 39209, 39209
 - \g__debug_deprecation_on_tl ... 1583, 39210
 - __debug_generate_parameter_-list:NNN 1480, 1482, 39216, 39216, 39321
 - __debug_get_base_form:N 39216, 39260, 39272
 - __debug_if_recursion_tail_-stop:N 39028, 39030, 39237
 - __debug_insert_debug_code:Nnn 39297
 - \l__debug_internal_tl 39213, 39218, 39221, 39223
 - __debug_log-functions_off: .. 39174
 - __debug_log-functions_on: ... 39174
 - __debug_parm_terminate:w 39216, 39246, 39247, 39248, 39249, 39257
 - __debug_patch_weird:Nnn 39297, 39372, 39375
 - __debug_setup_debug_code:Nnn ... 39297, 39313, 39318
 - __debug_suspended:TF 1476, 39067, 39071, 39079, 39082, 39094, 39103, 39112, 39122, 39127, 39133, 39177, 39186
 - \l__debug_suspended_tl 39067
 - __debug_tmp:w 39339, 39358
 - \l__debug_tmpa_tl 39213, 39321, 39326
 - \l__debug_tmpb_tl 39213, 39321, 39330
 - __debug_use_i_delimit_by_s_-stop:nw . 39025, 39025, 39151, 39153
 - __debug_use_none_delimit_by_q_-recursion_stop:w ... 39028, 39258
 - \def 36, 37, 38, 60, 62, 67, 68, 70, 84, 106, 143, 188
 - default commands:
 - .default:n 242, 22069
 - \defaultshyphenchar 189
 - \defaultskewchar 190
 - \deferred 806
 - deg 279
 - \delcode 191

- `\delimiter` 192
- `\delimiterfactor` 193
- `\delimitershortfall` 194
- deprecation internal commands:
 - `_deprecation_just_error:nnNN` [38744](#)
 - `_deprecation_patch_aux:Nn`
..... [38744](#), [38756](#), [38777](#)
 - `_deprecation_patch_aux:nnNnn` .
..... [38744](#), [38745](#), [38746](#)
 - `_deprecation_warn_once:nnNnn` ..
..... [38744](#), [38755](#), [38758](#)
- `\detokenize` 30, 94, 482
- `\DH` [32063](#), [33647](#), [34343](#)
- `\dh` [32063](#), [33647](#), [34353](#)
- dim commands:
 - `\dim_abs:n` [224](#), [951](#), [20809](#), [20809](#), [39887](#)
 - `\dim_add:Nn`
[224](#), [20791](#), [20791](#), [20798](#), [39448](#), [39822](#)
 - `\dim_case:nn` [227](#), [20889](#), [20904](#)
 - `\dim_case:nnTF`
..... [227](#), [20889](#), [20889](#), [20894](#), [20899](#)
 - `\dim_compare:n` [20849](#)
 - `\dim_compare:nNn` [20844](#)
 - `\dim_compare:nNnTF`
..... [225–228](#), [263](#), [20844](#),
[20913](#), [20949](#), [20957](#), [20966](#), [20972](#),
[20984](#), [20987](#), [20998](#), [21152](#), [35332](#),
[35349](#), [35383](#), [35397](#), [35407](#), [35865](#),
[35868](#), [35873](#), [35887](#), [35890](#), [35895](#),
[36241](#), [36246](#), [36256](#), [36385](#), [36397](#)
 - `\dim_compare:nTF` [225](#), [226](#),
[228](#), [20849](#), [20921](#), [20929](#), [20938](#), [20944](#)
 - `\dim_compare_p:n` [226](#), [20849](#)
 - `\dim_compare_p:nNn`
..... [225](#), [20844](#), [38717](#),
[38718](#), [38725](#), [38726](#), [38729](#), [38730](#)
 - `\dim_const:Nn`
[223](#), [944](#), [958](#), [20758](#), [20758](#), [20763](#),
[21173](#), [21174](#), [22998](#), [39599](#), [39826](#)
 - `\dim_do_until:nn`
..... [228](#), [20919](#), [20941](#), [20945](#)
 - `\dim_do_until:nNnn`
..... [227](#), [20947](#), [20969](#), [20973](#)
 - `\dim_do_while:nn`
..... [228](#), [20919](#), [20935](#), [20939](#)
 - `\dim_do_while:nNnn`
..... [227](#), [20947](#), [20963](#), [20967](#)
 - `\dim_eval:n` [225](#),
[226](#), [229](#), [944](#), [1358](#), [1359](#), [20761](#),
[20892](#), [20897](#), [20902](#), [20907](#), [21002](#),
[21030](#), [21030](#), [21168](#), [21172](#), [35564](#),
[35641](#), [35727](#), [35745](#), [35786](#), [35790](#),
[35791](#), [35795](#), [35799](#), [35800](#), [35817](#),
[35822](#), [35828](#), [35835](#), [35842](#), [36007](#),
[36010](#), [36011](#), [36018](#), [36100](#), [36101](#),
[36108](#), [36109](#), [36212](#), [36219](#), [36368](#),
[36369](#), [36633](#), [36634](#), [36635](#), [39884](#)
 - `\dim_gadd:Nn`
[224](#), [20791](#), [20793](#), [20799](#), [39529](#), [39823](#)
 - `.dim_gset:N` [242](#), [22079](#)
 - `\dim_gset:Nn` [224](#),
[944](#), [20779](#), [20781](#), [20784](#), [39528](#), [39821](#)
 - `\dim_gset_eq:NN`
..... [224](#), [20785](#), [20788](#), [20790](#), [39526](#)
 - `\dim_gsub:Nn`
[224](#), [20791](#), [20802](#), [20808](#), [39530](#), [39825](#)
 - `\dim_gzero:N` [223](#), [20764](#),
[20765](#), [20768](#), [20772](#), [21194](#), [39527](#)
 - `\dim_gzero_new:N`
..... [223](#), [20769](#), [20771](#), [20774](#)
 - `\dim_if_exist:N` [20775](#), [20777](#)
 - `\dim_if_exist:NTF`
..... [224](#), [20770](#), [20772](#), [20775](#)
 - `\dim_if_exist_p:N` [224](#), [20775](#)
 - `\dim_log:N` ... [231](#), [21169](#), [21169](#), [21170](#)
 - `\dim_log:n` [232](#), [21169](#), [21171](#)
 - `\dim_max:nn`
[224](#), [20809](#), [20816](#), [36079](#), [36083](#), [39933](#)
 - `\dim_min:nn` [224](#), [20809](#),
[20824](#), [36077](#), [36081](#), [36094](#), [39934](#)
 - `\dim_new:N` [223](#),
[20752](#), [20752](#), [20757](#), [20760](#), [20770](#),
[20772](#), [21175](#), [21176](#), [21177](#), [21178](#),
[34961](#), [34962](#), [34963](#), [34964](#), [34965](#),
[34966](#), [34967](#), [34968](#), [35425](#), [35449](#),
[35450](#), [35453](#), [35454](#), [35455](#), [35456](#),
[35953](#), [35954](#), [35955](#), [35956](#), [35957](#),
[36115](#), [36116](#), [36457](#), [36459](#), [36460](#)
 - `\dim_ratio:nn` [225](#), [20840](#), [20840](#)
 - `.dim_set:N` [242](#), [22079](#)
 - `\dim_set:Nn` [224](#),
[20779](#), [20779](#), [20783](#), [34988](#), [34989](#),
[34990](#), [35022](#), [35033](#), [35117](#), [35118](#),
[35119](#), [35134](#), [35232](#), [35233](#), [35234](#),
[35236](#), [35238](#), [35240](#), [35554](#), [35630](#),
[35867](#), [35871](#), [35889](#), [35893](#), [35924](#),
[35938](#), [36013](#), [36048](#), [36056](#), [36067](#),
[36068](#), [36069](#), [36070](#), [36076](#), [36078](#),
[36080](#), [36082](#), [36087](#), [36093](#), [36176](#),
[36178](#), [36180](#), [36188](#), [36190](#), [36244](#),
[36319](#), [36320](#), [36322](#), [36324](#), [36342](#),
[36343](#), [36458](#), [36552](#), [36553](#), [36599](#),
[36600](#), [36601](#), [36603](#), [39446](#), [39820](#)
 - `\dim_set_eq:NN` [224](#), [20785](#),
[20785](#), [20787](#), [35579](#), [35580](#), [39447](#)
 - `\dim_show:N` .. [231](#), [21165](#), [21165](#), [21166](#)
 - `\dim_show:n` ... [231](#), [957](#), [21167](#), [21167](#)
 - `\dim_sign:n` .. [229](#), [21032](#), [21032](#), [39888](#)

- \dim_step_function:nnnN
..... 228, 950, 20975, 20975,
21027, 39988, 39992, 39996, 40000
- \dim_step_inline:nnnn
..... 228, 21005, 21005
- \dim_step_variable:nnnNn
..... 229, 21005, 21012
- \dim_sub:Nn
224, 20791, 20800, 20807, 39449, 39824
- \dim_to_decimal:n
..... 229, 954, 21052, 21052,
21088, 21116, 21153, 21162, 39885
- \dim_to_decimal_in_bp:n .. 230, 21069
- \dim_to_decimal_in_cc:n .. 230, 21069
- \dim_to_decimal_in_cm:n .. 230, 21069
- \dim_to_decimal_in_dd:n .. 230, 21069
- \dim_to_decimal_in_in:n .. 230, 21069
- \dim_to_decimal_in_mm:n .. 230, 21069
- \dim_to_decimal_in_pc:n .. 230, 21069
- \dim_to_decimal_in_sp:n
..... 231, 1063, 21067,
21067, 24216, 24253, 24847, 39886
- \dim_to_decimal_in_unit:nn
..... 231, 21095, 21095
- \dim_to_fp:n 231, 1063, 1082,
21067, 29102, 29102, 35026, 35027,
35037, 35038, 35106, 35109, 35110,
35135, 35150, 35151, 35170, 35171,
35189, 35206, 35209, 35210, 35266,
35268, 35878, 35879, 35880, 35900,
35901, 35902, 35912, 35913, 35929,
35930, 35931, 35932, 35942, 35943,
36052, 36053, 36060, 36061, 36134,
36137, 36138, 36189, 36191, 40019
- \dim_until_do:nn
..... 228, 20919, 20927, 20932
- \dim_until_do:nNnn
..... 227, 20947, 20955, 20960
- \dim_use:N 229,
1358, 20812, 20818, 20819, 20820,
20826, 20827, 20828, 20852, 20871,
21031, 21035, 21050, 21050, 21051,
21055, 21248, 21249, 21324, 21325,
36015, 36019, 36026, 36032, 36041,
36042, 36043, 36197, 36204, 36350
- \dim_while_do:nn
..... 228, 20919, 20919, 20924
- \dim_while_do:nNnn
..... 228, 20947, 20947, 20952
- \dim_zero:N 223, 20764,
20764, 20767, 20770, 21193, 34991,
35120, 35235, 35858, 35859, 39445
- \dim_zero_new:N
..... 223, 20769, 20769, 20773
- \c_max_dim ... 230, 232, 235, 1012,
21173, 21267, 23025, 23067, 23075,
36067, 36068, 36069, 36070, 36087
- \g_tmpa_dim 232, 21175
- \l_tmpa_dim 232, 21175
- \g_tmpb_dim 232, 21175
- \l_tmpb_dim 232, 21175
- \c_zero_dim 232, 20984, 20987, 21040,
21173, 21266, 23092, 34847, 34871,
35336, 35347, 35353, 35365, 35383,
35387, 35395, 35397, 35401, 35407,
35417, 35865, 35868, 35873, 35887,
35890, 35895, 36241, 36246, 36256
- dim internal commands:
 - __dim_abs:N 20809, 20811, 20814
 - __dim_branch_unit:w
..... 954, 21105, 21110, 21110
 - __dim_case:nnTF 20889,
20892, 20897, 20902, 20907, 20909
 - __dim_case:nw
..... 20889, 20910, 20911, 20915
 - __dim_case_end:nw 20889, 20914, 20917
 - __dim_chk_unit:w
..... 954, 21097, 21100, 21100
 - __dim_compare:w . 20849, 20851, 20854
 - __dim_compare:wNN
..... 946, 20849, 20857, 20860, 20870
 - __dim_compare_!:w 20849
 - __dim_compare_<:w 20849
 - __dim_compare_=:w 20849
 - __dim_compare_>:w 20849
 - __dim_compare_end:w .. 20857, 20881
 - __dim_compare_error:
946, 20849, 20852, 20854, 20883, 20887
 - __dim_convert_remainder:w
..... 955, 21140, 21144, 21144
 - __dim_eval:w 20746, 20747, 20780,
20782, 20792, 20796, 20801, 20805,
20812, 20818, 20819, 20820, 20826,
20827, 20828, 20843, 20846, 20852,
20871, 20876, 20978, 20979, 20980,
21031, 21035, 21055, 21068, 21075,
21098, 21106, 21153, 39828, 39890,
39936, 39967, 39991, 39995, 39999
 - __dim_eval_end:
20746, 20748, 20780, 20782, 20792,
20796, 20801, 20805, 20812, 20822,
20830, 20843, 20846, 21031, 21035,
21055, 21068, 21075, 21098, 21153
 - __dim_get_quotient:w
..... 955, 21117, 21120, 21120
 - __dim_get_remainder:w
..... 955, 21127, 21132, 21138, 21138

<code>__dim_maxmin:wwN</code>	
.....	20809, 20818, 20826, 20832
<code>__dim_parse_decimal:w</code>	
....	956, 21154, 21156, 21159, 21159
<code>__dim_parse_decimal_aux:w</code>	
.....	956, 21159, 21161, 21164
<code>__dim_ratio:n</code> ..	20840, 20841, 20842
<code>__dim_sign:Nw</code> ..	21032, 21034, 21038
<code>__dim_step:NnnnN</code>	
..	20975, 20985, 20992, 20996, 21001
<code>__dim_step:NNnnnn</code>	
.....	21005, 21008, 21015, 21024
<code>__dim_step:wwwN</code> .	20975, 20977, 20982
<code>__dim_test_candidate:w</code>	
.....	956, 21146, 21150, 21150
<code>__dim_tmp:w</code> .	21070, 21078, 21079, 21080, 21081, 21082, 21083, 21084
<code>__dim_to_decimal:w</code>	
.....	21052, 21055, 21059
<code>__dim_to_decimal_aux:w</code>	
.....	953, 954, 21069, 21074, 21086, 21113
<code>__dim_use_none_delimit_by_s_- stop:w</code>	20751, 20751, 20867
<code>\dimen</code>	195, 19523
<code>\dimendef</code>	196
<code>\dimexpr</code>	483
<code>\directlua</code>	21, 23, 809
<code>\disablecjktoken</code>	1201
<code>\discretionary</code>	197
<code>\discretionaryligaturemode</code>	807
<code>\disinhibitglue</code>	1139
<code>\displayindent</code>	198
<code>\displaylimits</code>	199
<code>\displaystyle</code>	200
<code>\displaywidowpenalties</code>	484
<code>\displaywidowpenalty</code>	201
<code>\displaywidth</code>	202
<code>\divide</code>	203
<code>\DJ</code>	32064, 33648, 34344
<code>\dj</code>	32064, 33648, 34354
<code>\do</code>	1254
<code>\doublehyphendemerits</code>	204
<code>\dp</code>	205
<code>\draftmode</code>	934
draw commands:	
<code>\draw_begin:</code>	325
<code>\draw_end:</code>	325
<code>\dtou</code>	1140
<code>\dump</code>	206
<code>\dviextension</code>	810
<code>\dvifedback</code>	811
<code>\divvariable</code>	812
	E
<code>\edef</code>	73, 82, 207
<code>\efcode</code>	671
<code>\elapsedtime</code>	770
<code>\else</code>	9, 11, 18, 54, 55, 56, 208
else commands:	
<code>\else:</code>	
.....	28, 65, 72, 99, 179, 180, 238, 314, 381, 383, 389, 417, 591, 719, 1109, 1392, 1395, 1437, 1665, 1673, 1699, 1830, 1834, 1845, 1850, 1861, 1866, 1871, 1876, 1889, 1905, 2069, 2091, 2100, 2114, 2173, 2174, 2259, 2435, 2691, 2725, 2776, 2777, 2779, 2783, 2795, 2796, 2797, 2798, 2799, 2800, 2801, 2802, 2803, 2867, 2868, 2870, 2919, 3022, 3159, 3160, 3651, 3654, 3657, 3667, 3682, 3709, 3724, 3751, 3767, 3801, 3809, 3811, 3813, 3815, 3817, 3819, 3821, 3823, 3845, 3866, 3870, 3946, 3950, 4062, 4085, 4096, 4129, 4131, 4155, 4194, 4205, 4238, 4413, 4414, 4418, 4419, 4435, 4442, 4642, 4652, 4702, 4711, 4724, 4725, 4727, 4729, 4732, 4733, 4737, 4742, 4753, 4757, 4761, 4768, 4833, 4840, 4850, 4852, 4862, 4869, 4871, 4882, 5002, 5116, 5159, 5164, 5176, 5181, 5271, 5417, 5430, 5519, 5548, 5587, 5605, 5718, 5774, 5808, 5838, 6260, 6278, 6297, 6331, 6384, 6431, 6435, 6442, 6463, 6474, 6621, 6733, 6843, 6886, 6889, 7009, 7020, 7029, 7057, 7069, 7095, 7112, 7120, 7389, 7643, 7923, 7934, 8332, 8383, 8404, 8426, 8444, 8460, 8470, 8486, 8496, 8611, 8613, 8615, 8617, 10329, 10332, 10335, 11112, 11119, 11380, 11389, 11400, 12105, 12552, 12562, 12577, 12586, 12605, 12619, 12649, 12667, 12682, 12903, 12921, 12941, 12949, 12959, 12975, 12998, 13009, 13015, 13161, 13173, 13222, 13225, 13228, 13544, 13549, 13554, 13561, 13566, 13818, 13874, 13877, 13880, 13892, 13907, 14046, 14235, 14243, 14251, 14400, 14451, 14452, 14456, 14461, 14484, 14537, 14637, 14888, 14918, 14921, 14951, 14954, 14971, 14974, 15077, 15082, 15100, 15119, 15122, 15171, 15176, 15179, 15294, 15306, 15315, 15437, 15442, 16370, 16408, 16416, 16427, 16437, 16456, 16480, 16484, 16526, 16586, 16597, 16616,

- 16961, 17031, 17040, 17475, 17486,
17507, 17523, 17526, 17547, 17588,
17687, 17714, 17722, 17760, 17768,
18071, 18104, 18155, 18272, 18298,
18325, 18334, 18538, 18553, 18575,
18589, 19198, 19204, 19207, 19225,
19292, 19295, 19298, 19301, 19304,
19307, 19310, 19313, 19316, 19319,
19359, 19364, 19369, 19374, 19381,
19388, 19393, 19398, 19403, 19408,
19413, 19420, 19425, 19447, 19453,
19456, 19492, 19495, 19630, 19639,
19647, 19656, 19732, 19757, 19761,
19771, 19809, 19823, 19832, 19842,
19876, 20261, 20309, 20318, 20466,
20504, 20514, 20815, 20836, 20847,
20857, 20882, 21042, 21045, 21091,
21611, 23030, 23257, 23274, 23275,
23290, 23300, 23395, 23471, 23533,
23536, 23550, 23568, 23572, 23812,
23825, 23845, 23873, 23874, 23896,
23917, 23940, 23941, 23974, 23991,
24009, 24044, 24048, 24084, 24101,
24107, 24111, 24115, 24274, 24307,
24315, 24348, 24352, 24364, 24374,
24384, 24415, 24428, 24463, 24473,
24492, 24505, 24518, 24522, 24533,
24556, 24573, 24585, 24599, 24612,
24616, 24624, 24626, 24636, 24647,
24663, 24677, 24683, 24686, 24693,
24715, 24745, 24768, 24796, 24799,
24973, 24977, 24984, 25003, 25015,
25019, 25026, 25048, 25065, 25071,
25103, 25135, 25151, 25171, 25212,
25227, 25260, 25262, 25268, 25283,
25336, 25553, 25569, 25580, 25629,
25632, 25635, 25638, 25669, 25678,
25687, 25690, 25851, 25864, 25867,
25874, 25892, 25916, 25917, 25932,
25942, 25991, 25994, 26003, 26015,
26026, 26040, 26053, 26093, 26127,
26147, 26184, 26202, 26205, 26211,
26225, 26260, 26278, 26281, 26284,
26287, 26348, 26421, 26491, 26492,
26501, 26536, 26619, 26623, 26627,
26689, 26724, 26739, 27005, 27034,
27038, 27198, 27207, 27261, 27272,
27288, 27296, 27355, 27435, 27446,
27451, 27485, 27498, 27510, 27516,
27637, 27645, 27684, 27691, 27713,
27740, 27755, 27759, 27781, 27812,
27815, 27840, 27843, 27884, 27892,
27903, 27906, 28021, 28036, 28051,
28066, 28081, 28096, 28117, 28162,
28468, 28506, 28507, 28516, 28560,
28615, 28616, 28617, 28721, 28743,
28758, 28776, 28824, 28840, 29046,
29113, 29118, 29256, 29292, 29305,
29335, 29339, 29347, 29374, 29400,
29408, 29425, 29428, 30019, 30023,
30075, 30134, 30146, 30230, 30876,
30887, 30907, 31255, 31414, 31418,
31429, 31444, 31456, 31460, 31469,
31470, 31471, 31472, 31473, 31474,
31475, 31476, 31477, 31488, 31502,
31505, 31508, 31511, 31514, 31517,
31520, 31618, 31624, 31633, 31637,
32041, 32978, 32982, 32986, 32989,
32993, 33007, 33010, 33013, 33016,
33019, 33022, 33042, 33045, 33048,
33051, 33054, 33057, 33060, 33063,
33066, 33069, 33072, 33075, 33078,
33081, 33084, 33087, 33090, 33119,
33122, 33137, 33140, 33154, 33157,
33160, 33163, 33179, 33182, 33185,
34722, 34724, 34730, 39155, 39164,
39167, 39246, 39247, 39248, 39249,
39263, 39264, 39274, 39275, 39276,
39277, 39278, 39279, 39280, 39281,
39282, 40050, 40051, 40056, 40057
\em 34297
em 280
\emergencystretch 209
\emph 34270
\enablecjktoken 1202
\end 361, 210, 31662, 31672, 34313
\endcsname 663,
4, 8, 13, 17, 30, 53, 54, 61, 94, 211
\endgroup . 3, 7, 12, 16, 36, 67, 71, 77, 212
\endinput 78, 213
\endL 485
\endlinechar 93, 104, 214
\endlocalcontrol 815
\endR 486
\ensuremath 1296, 31667
\epTeXinputencoding 1141
\epTeXversion 1142
\eqno 215
\errhelp 68, 216
\errmessage 70, 217
\errorcontextlines 68, 218
\errorstopmode 219
\escapechar 220
escapehex 11895
\ETC 4260
\TeXglueshrinkorder 813
\TeXgluestretchorder 814
\TeXrevision 487

- | | | |
|--------------------------------------|-------------|-------------------------------------|
| <code>\TeXversion</code> | 488 | 25070, 25073, 25087, 25100, 25150, |
| <code>\etoksapp</code> | 816 | 25168, 25239, 25251, 25280, 25282, |
| <code>\etokspre</code> | 817 | 25286, 25288, 25346, 25356, 25366, |
| <code>\euc</code> | 1143 | 25378, 25560, 25577, 25587, 25753, |
| <code>\everycr</code> | 221 | 25754, 25755, 25936, 25939, 25947, |
| <code>\everydisplay</code> | 222 | 25957, 25965, 26978, 27509, 27531, |
| <code>\everyeof</code> | 489 | 27686, 27862, 28138, 28881, 28896, |
| <code>\everyhbox</code> | 223 | 28913, 28950, 28967, 29009, 29028, |
| <code>\everyjob</code> | 28, 29, 224 | 29041, 29073, 29088, 29099, 29195, |
| <code>\everymath</code> | 225 | 29242, 29282, 29318, 29518, 29520, |
| <code>\everypar</code> | 226 | 29523, 29528, 29540, 29586, 29697, |
| <code>\everyvbox</code> | 227 | 29699, 29875, 30040, 30140, 31717, |
| <code>ex</code> | 280 | 31756, 32026, 32101, 32180, 32194, |
| <code>\exceptionpenalty</code> | 818 | 32325, 34062, 39250, 39258, 39296 |
| <code>\exhyphenchar</code> | 819 | |
| <code>\exhyphenpenalty</code> | 228 | |
| <code>exp</code> | 275 | |
| exp commands: | | |
| <code>\exp:w</code> | 42, 43, | |
| 381, 388, 408, 409, 416, 573–576, | | |
| 594, 656, 722, 731, 745, 870, 1053, | | |
| 1055, 1056, 1059, 1060, 1078, 1083, | | |
| 1414, 1592, 1594, 2407, 2420, 2426, | | |
| 2468, 2472, 2477, 2483, 2489, 2501, | | |
| 2513, 2519, 2525, 2530, 2532, 2539, | | |
| 2546, 2584, 2589, 2596, 2605, 2607, | | |
| 2611, 2618, 2624, 2632, 2641, 2648, | | |
| 2663, 2676, 2680, 2685, 2687, 2975, | | |
| 7844, 7852, 7890, 7928, 7937, 7948, | | |
| 8394, 8541, 8543, 8545, 8547, 8564, | | |
| 8621, 8624, 10782, 12437, 12679, | | |
| 13085, 13166, 13324, 13330, 13347, | | |
| 13365, 13586, 13591, 13596, 13601, | | |
| 13621, 13626, 13631, 13636, 13801, | | |
| 13810, 13865, 17728, 17733, 17738, | | |
| 17743, 18392, 18400, 18461, 18763, | | |
| 18773, 19185, 19662, 19664, 19666, | | |
| 19668, 19670, 19672, 19674, 19676, | | |
| 19678, 19680, 19682, 19684, 19751, | | |
| 20856, 20891, 20896, 20901, 20906, | | |
| 23303, 23418, 23422, 23788, 23914, | | |
| 23915, 23916, 23917, 24036, 24054, | | |
| 24083, 24127, 24139, 24144, 24152, | | |
| 24160, 24181, 24187, 24259, 24272, | | |
| 24273, 24282, 24295, 24313, 24314, | | |
| 24334, 24347, 24351, 24373, 24401, | | |
| 24414, 24427, 24451, 24462, 24472, | | |
| 24491, 24504, 24517, 24520, 24532, | | |
| 24555, 24584, 24598, 24615, 24635, | | |
| 24646, 24652, 24662, 24704, 24711, | | |
| 24742, 24757, 24765, 24782, 24798, | | |
| 24802, 24811, 24848, 24857, 24866, | | |
| 24871, 24873, 24884, 24886, 24901, | | |
| 24904, 24911, 24922, 25006, 25052, | | |
| 25070, 25073, 25087, 25100, 25150, | | |
| 25168, 25239, 25251, 25280, 25282, | | |
| 25286, 25288, 25346, 25356, 25366, | | |
| 25378, 25560, 25577, 25587, 25753, | | |
| 25754, 25755, 25936, 25939, 25947, | | |
| 25957, 25965, 26978, 27509, 27531, | | |
| 27686, 27862, 28138, 28881, 28896, | | |
| 28913, 28950, 28967, 29009, 29028, | | |
| 29041, 29073, 29088, 29099, 29195, | | |
| 29242, 29282, 29318, 29518, 29520, | | |
| 29523, 29528, 29540, 29586, 29697, | | |
| 29699, 29875, 30040, 30140, 31717, | | |
| 31756, 32026, 32101, 32180, 32194, | | |
| 32325, 34062, 39250, 39258, 39296 | | |
| <code>\exp_after:wN</code> | | 39, 41, 42, 208, 381, 384, |
| | | 406, 409, 446, 463, 553, 572, 574, |
| 575, 623, 715, 728, 731, 857, 883, | | 895, 919, 1028, 1052, 1053, 1055, |
| 1056, 1123, 1124, 1188, 1411, 1411, | | 1429, 1431, 1436, 1438, 1592, 1594, |
| 1656, 1680, 1698, 1700, 1764, 1769, | | 1776, 1805, 1809, 1814, 1825, 1850, |
| 1866, 1894, 1910, 1930, 1941, 1946, | | 2079, 2099, 2101, 2140, 2217, 2326, |
| 2346, 2355, 2364, 2376, 2386, 2392, | | 2399, 2401, 2406, 2407, 2419, 2420, |
| 2425, 2426, 2431, 2436, 2438, 2441, | | 2450, 2452, 2455, 2456, 2457, 2460, |
| 2462, 2464, 2466, 2468, 2471, 2476, | | 2481, 2482, 2483, 2487, 2488, 2489, |
| 2493, 2494, 2499, 2500, 2501, 2505, | | 2506, 2507, 2511, 2512, 2513, 2517, |
| 2518, 2519, 2523, 2524, 2525, 2529, | | 2530, 2531, 2532, 2536, 2537, 2538, |
| 2539, 2543, 2544, 2545, 2546, 2550, | | 2551, 2552, 2557, 2558, 2559, 2564, |
| 2565, 2566, 2567, 2571, 2572, 2573, | | 2574, 2580, 2583, 2584, 2588, 2589, |
| 2595, 2596, 2603, 2605, 2607, 2609, | | 2611, 2613, 2616, 2617, 2622, 2623, |
| 2627, 2630, 2631, 2635, 2638, 2639, | | 2640, 2645, 2646, 2647, 2656, 2659, |
| 2660, 2661, 2662, 2667, 2669, 2671, | | 2672, 2676, 2679, 2684, 2722, 2726, |
| 2748, 2755, 2775, 2918, 2920, 2973, | | 2975, 2992, 3010, 3021, 3151, 3237, |
| 3238, 3281, 3300, 3301, 3316, 3317, | | 3318, 3561, 3567, 3569, 3580, 3664, |
| 3665, 3666, 3667, 3673, 3674, 3690, | | 3708, 3710, 3716, 3717, 3748, 3750, |
| 3752, 3782, 3797, 3799, 3800, 3802, | | 3831, 3841, 3849, 3859, 3869, 3871, |
| 3873, 3899, 3936, 3945, 3948, 3949, | | |

3951, 3952, 3960, 3961, 3975, 4013,
4053, 4057, 4058, 4059, 4061, 4063,
4073, 4076, 4093, 4095, 4097, 4118,
4122, 4154, 4156, 4161, 4162, 4163,
4197, 4234, 4279, 4305, 4309, 4317,
4374, 4381, 4388, 4392, 4399, 4405,
4452, 4574, 4577, 4618, 4636, 4641,
4643, 4644, 4651, 4654, 4655, 4661,
4673, 4685, 4704, 4712, 4803, 4824,
4842, 4853, 4873, 4968, 5165, 5208,
5272, 5284, 5416, 5419, 5429, 5431,
5613, 5620, 5628, 5713, 5807, 6223,
6513, 6519, 6525, 6635, 6659, 6690,
6721, 6747, 6785, 6835, 6836, 6847,
6964, 6993, 7055, 7056, 7059, 7060,
7068, 7070, 7071, 7094, 7097, 7174,
7544, 7545, 7558, 7581, 7582, 7593,
7632, 7745, 7746, 7844, 7848, 7849,
7850, 7890, 7928, 7945, 7946, 7947,
8392, 8411, 8416, 8420, 8565, 8894,
8895, 8896, 8997, 9132, 9399, 9595,
9596, 10142, 10143, 10229, 10413,
10431, 10472, 10712, 10715, 10765,
10774, 10777, 10780, 10781, 10783,
10823, 10889, 10920, 10941, 10953,
10981, 10989, 11080, 11081, 11082,
11120, 11471, 11596, 12102, 12167,
12168, 12175, 12176, 12192, 12196,
12208, 12213, 12218, 12225, 12232,
12233, 12239, 12244, 12249, 12256,
12263, 12264, 12280, 12284, 12289,
12295, 12303, 12304, 12308, 12312,
12317, 12323, 12331, 12332, 12358,
12382, 12383, 12384, 12385, 12386,
12445, 12446, 12447, 12510, 12520,
12525, 12570, 12601, 12615, 12663,
12665, 12762, 12815, 12820, 12880,
12888, 12891, 12938, 12948, 12972,
12982, 12983, 12984, 12987, 12991,
12992, 13016, 13083, 13162, 13164,
13165, 13166, 13171, 13172, 13174,
13246, 13264, 13265, 13324, 13330,
13345, 13348, 13363, 13365, 13366,
13383, 13391, 13396, 13398, 13401,
13452, 13457, 13462, 13467, 13653,
13654, 13666, 13731, 13754, 13788,
13789, 13800, 13801, 13809, 13817,
13819, 13826, 13831, 13849, 13850,
13851, 13863, 13864, 13891, 13893,
13899, 13905, 13919, 13939, 13950,
13966, 13974, 13982, 13989, 13996,
14008, 14209, 14225, 14244, 14253,
14274, 14275, 14280, 14281, 14306,
14307, 14322, 14323, 14371, 14376,
14443, 14772, 14808, 14810, 14825,
14831, 14995, 14997, 15064, 15083,
15084, 15098, 15099, 15126, 15127,
15242, 15264, 15277, 15278, 15305,
15400, 15421, 15430, 16363, 16369,
16371, 16395, 16446, 16447, 16526,
16560, 16615, 16623, 16634, 16789,
16791, 16803, 16811, 16919, 16929,
17015, 17049, 17061, 17074, 17075,
17076, 17098, 17099, 17144, 17179,
17180, 17181, 17281, 17282, 17284,
17285, 17293, 17294, 17298, 17301,
17343, 17344, 17370, 17371, 17374,
17427, 17467, 17481, 17486, 17489,
17490, 17497, 17498, 17514, 17515,
17536, 17537, 17546, 17659, 17664,
17669, 17692, 17694, 17830, 17831,
17832, 17857, 17858, 18043, 18071,
18076, 18104, 18117, 18127, 18154,
18156, 18157, 18165, 18182, 18226,
18302, 18335, 18337, 18343, 18346,
18391, 18399, 18461, 18539, 18554,
18576, 18590, 18645, 18653, 18659,
18740, 18762, 18772, 18890, 18891,
18894, 18895, 19185, 19186, 19222,
19265, 19266, 19268, 19269, 19270,
19432, 19451, 19499, 19608, 19637,
19638, 19640, 19646, 19649, 19731,
19734, 19750, 19756, 19759, 19762,
19769, 19770, 19772, 19808, 19810,
19820, 19821, 19822, 19824, 19830,
19831, 19833, 19840, 19841, 19843,
19870, 19875, 19877, 19883, 19927,
19932, 19939, 19963, 20008, 20019,
20027, 20028, 20058, 20083, 20087,
20090, 20107, 20127, 20203, 20204,
20211, 20219, 20253, 20259, 20260,
20262, 20276, 20277, 20280, 20298,
20306, 20310, 20317, 20319, 20328,
20334, 20453, 20458, 20465, 20467,
20475, 20505, 20524, 20526, 20527,
20532, 20624, 20647, 20652, 20659,
20726, 20736, 20811, 20815, 20818,
20819, 20826, 20827, 20851, 20856,
20867, 20870, 20977, 20978, 20979,
21034, 21054, 21074, 21097, 21105,
21106, 21127, 21132, 21140, 21146,
21161, 21239, 21651, 21666, 21694,
21839, 21867, 21875, 21887, 21990,
22343, 22512, 22513, 22592, 22615,
23048, 23049, 23050, 23068, 23076,
23100, 23101, 23143, 23150, 23151,
23162, 23256, 23258, 23259, 23277,
23278, 23279, 23289, 23291, 23299,

23301, 23308, 23309, 23310, 23311,
23312, 23313, 23318, 23319, 23320,
23321, 23322, 23323, 23324, 23367,
23380, 23383, 23394, 23396, 23411,
23415, 23416, 23417, 23420, 23421,
23485, 23487, 23514, 23518, 23543,
23547, 23564, 23571, 23573, 23643,
23651, 23668, 23677, 23723, 23788,
23857, 23858, 23859, 23919, 23929,
23948, 23954, 23973, 23975, 23977,
23988, 23989, 23992, 24003, 24007,
24014, 24015, 24026, 24027, 24036,
24043, 24045, 24046, 24054, 24083,
24101, 24102, 24105, 24106, 24108,
24109, 24113, 24114, 24116, 24117,
24126, 24127, 24132, 24138, 24144,
24152, 24160, 24179, 24180, 24183,
24184, 24186, 24193, 24194, 24196,
24214, 24215, 24243, 24246, 24251,
24252, 24257, 24258, 24260, 24269,
24270, 24271, 24272, 24275, 24276,
24277, 24280, 24295, 24312, 24313,
24323, 24324, 24334, 24346, 24350,
24363, 24365, 24373, 24383, 24385,
24391, 24396, 24398, 24400, 24406,
24407, 24411, 24413, 24425, 24426,
24448, 24450, 24456, 24459, 24461,
24465, 24470, 24475, 24476, 24486,
24487, 24489, 24490, 24493, 24497,
24502, 24516, 24519, 24531, 24540,
24547, 24548, 24549, 24550, 24552,
24554, 24565, 24566, 24567, 24568,
24570, 24572, 24574, 24575, 24576,
24582, 24583, 24593, 24597, 24598,
24600, 24601, 24602, 24607, 24613,
24614, 24625, 24627, 24634, 24635,
24637, 24638, 24645, 24651, 24661,
24725, 24738, 24739, 24740, 24741,
24755, 24756, 24758, 24763, 24764,
24779, 24781, 24798, 24802, 24811,
24845, 24846, 24847, 24853, 24854,
24855, 24856, 24862, 24863, 24864,
24865, 24872, 24885, 24893, 24899,
24900, 24902, 24903, 24909, 24910,
24912, 24938, 24951, 24971, 24972,
24974, 24975, 24982, 24983, 24985,
24988, 25000, 25001, 25002, 25004,
25005, 25006, 25013, 25014, 25016,
25017, 25024, 25025, 25027, 25030,
25045, 25046, 25047, 25050, 25051,
25052, 25062, 25063, 25064, 25067,
25068, 25069, 25072, 25076, 25085,
25086, 25097, 25098, 25099, 25102,
25104, 25105, 25106, 25133, 25134,
25136, 25137, 25138, 25148, 25149,
25150, 25152, 25153, 25154, 25166,
25167, 25170, 25172, 25173, 25174,
25194, 25195, 25196, 25197, 25198,
25199, 25200, 25210, 25211, 25213,
25214, 25215, 25221, 25232, 25233,
25234, 25235, 25236, 25237, 25238,
25239, 25244, 25245, 25246, 25247,
25248, 25249, 25250, 25266, 25267,
25269, 25270, 25277, 25278, 25279,
25284, 25285, 25287, 25303, 25318,
25327, 25337, 25343, 25344, 25345,
25350, 25363, 25364, 25365, 25371,
25559, 25576, 25586, 25622, 25623,
25670, 25752, 25850, 25852, 25891,
25893, 25896, 25931, 25933, 25935,
25938, 25945, 25946, 25949, 25950,
25955, 25956, 25963, 25964, 25999,
26000, 26001, 26003, 26014, 26039,
26041, 26047, 26048, 26052, 26055,
26077, 26079, 26092, 26094, 26100,
26102, 26105, 26111, 26113, 26115,
26116, 26117, 26119, 26124, 26126,
26128, 26132, 26135, 26141, 26142,
26146, 26148, 26149, 26150, 26158,
26160, 26161, 26168, 26174, 26181,
26182, 26187, 26188, 26189, 26190,
26209, 26210, 26211, 26217, 26218,
26219, 26224, 26226, 26234, 26236,
26238, 26239, 26241, 26252, 26254,
26256, 26257, 26262, 26313, 26314,
26321, 26322, 26324, 26326, 26328,
26331, 26334, 26336, 26338, 26347,
26349, 26355, 26357, 26359, 26360,
26361, 26367, 26369, 26371, 26372,
26373, 26394, 26395, 26398, 26406,
26408, 26412, 26413, 26414, 26415,
26420, 26422, 26429, 26432, 26435,
26438, 26447, 26450, 26453, 26456,
26463, 26465, 26471, 26479, 26481,
26483, 26500, 26502, 26509, 26511,
26514, 26520, 26522, 26524, 26525,
26526, 26528, 26542, 26543, 26546,
26564, 26566, 26568, 26580, 26583,
26586, 26589, 26592, 26595, 26598,
26601, 26605, 26617, 26621, 26625,
26628, 26643, 26649, 26651, 26653,
26663, 26687, 26690, 26702, 26704,
26708, 26709, 26710, 26712, 26713,
26715, 26722, 26730, 26731, 26737,
26738, 26744, 26747, 26748, 26749,
26750, 26758, 26801, 26806, 26808,
26815, 26818, 26821, 26824, 26827,
26830, 26838, 26839, 26851, 26859,

26861, 26871, 26873, 26880, 26889,
 26891, 26894, 26897, 26900, 26903,
 26916, 26918, 26926, 26928, 26936,
 26938, 26948, 26951, 26954, 26961,
 26976, 26977, 26994, 26996, 26997,
 27054, 27067, 27069, 27075, 27088,
 27090, 27092, 27116, 27130, 27132,
 27139, 27141, 27182, 27183, 27184,
 27186, 27187, 27188, 27190, 27191,
 27197, 27199, 27200, 27206, 27208,
 27209, 27210, 27211, 27223, 27229,
 27231, 27268, 27275, 27282, 27302,
 27303, 27305, 27307, 27309, 27322,
 27327, 27328, 27329, 27330, 27331,
 27335, 27340, 27342, 27348, 27354,
 27356, 27357, 27363, 27364, 27365,
 27366, 27367, 27368, 27369, 27370,
 27375, 27377, 27379, 27381, 27383,
 27388, 27390, 27392, 27394, 27396,
 27398, 27416, 27420, 27428, 27429,
 27434, 27436, 27445, 27448, 27449,
 27450, 27452, 27453, 27454, 27462,
 27468, 27480, 27483, 27484, 27486,
 27487, 27511, 27512, 27515, 27517,
 27533, 27537, 27538, 27539, 27555,
 27561, 27627, 27628, 27629, 27636,
 27638, 27639, 27644, 27646, 27647,
 27656, 27657, 27659, 27662, 27665,
 27681, 27685, 27686, 27690, 27692,
 27727, 27733, 27734, 27736, 27738,
 27739, 27741, 27742, 27752, 27753,
 27756, 27757, 27758, 27760, 27761,
 27762, 27779, 27780, 27782, 27783,
 27789, 27791, 27794, 27797, 27800,
 27803, 27811, 27814, 27816, 27819,
 27826, 27830, 27838, 27839, 27842,
 27844, 27846, 27851, 27852, 27858,
 27863, 27864, 27872, 27873, 27874,
 27875, 27918, 27940, 27941, 27944,
 27945, 27954, 27955, 27956, 27960,
 27967, 27968, 27969, 28110, 28111,
 28112, 28114, 28132, 28133, 28134,
 28135, 28136, 28137, 28144, 28153,
 28160, 28161, 28385, 28386, 28392,
 28393, 28396, 28401, 28404, 28407,
 28410, 28413, 28416, 28419, 28422,
 28438, 28439, 28449, 28458, 28466,
 28467, 28469, 28470, 28475, 28476,
 28485, 28492, 28501, 28502, 28515,
 28517, 28545, 28546, 28555, 28558,
 28583, 28589, 28590, 28632, 28633,
 28635, 28649, 28650, 28658, 28669,
 28703, 28706, 28716, 28717, 28720,
 28722, 28728, 28742, 28744, 28785,
 28788, 28808, 28881, 28891, 28895,
 28913, 28916, 28937, 28938, 28945,
 28949, 28967, 28970, 28999, 29000,
 29006, 29007, 29008, 29015, 29023,
 29027, 29041, 29044, 29060, 29061,
 29068, 29072, 29083, 29087, 29099,
 29104, 29105, 29106, 29112, 29114,
 29117, 29119, 29184, 29194, 29201,
 29206, 29207, 29217, 29244, 29255,
 29257, 29259, 29261, 29266, 29267,
 29269, 29280, 29281, 29301, 29307,
 29308, 29310, 29313, 29318, 29324,
 29325, 29355, 29357, 29360, 29363,
 29365, 29373, 29375, 29379, 29383,
 29388, 29393, 29404, 29438, 29458,
 29476, 29517, 29521, 29525, 29527,
 29538, 29539, 29545, 29565, 29585,
 29685, 29694, 29695, 29696, 29700,
 29701, 29869, 29870, 29871, 29872,
 29873, 29874, 29877, 29879, 29916,
 29917, 29918, 29922, 29923, 29936,
 29947, 29950, 29954, 29959, 29963,
 30010, 30011, 30035, 30036, 30037,
 30038, 30039, 30047, 30058, 30064,
 30090, 30091, 30092, 30099, 30100,
 30107, 30108, 30116, 30117, 30121,
 30122, 30123, 30128, 30133, 30139,
 30142, 30143, 30144, 30145, 30146,
 30285, 30574, 30575, 30789, 30828,
 30842, 30859, 31211, 31213, 31357,
 31417, 31419, 31426, 31427, 31430,
 31459, 31461, 31467, 31479, 31487,
 31489, 31610, 31615, 31616, 31619,
 31620, 31625, 31632, 31635, 31638,
 31715, 31754, 31771, 31772, 31816,
 31817, 31840, 31891, 31911, 31990,
 32016, 32025, 32040, 32042, 32099,
 32178, 32192, 32324, 33690, 34060,
 34108, 34109, 34170, 34171, 34172,
 34179, 34242, 36696, 36700, 36719,
 36720, 37186, 37239, 37334, 37361,
 37366, 37372, 37378, 37523, 37541,
 37705, 37982, 39152, 39187, 39188,
 39200, 39258, 39296, 39358, 40026
 \exp_args:cc 36, 1428, 1430, 2449
 \exp_args:Nc
 . . . 33, 36, 400, 1428, 1428, 1432,
 1440, 1704, 1717, 1725, 1733, 1988,
 2011, 2049, 2054, 2061, 2072, 2119,
 2131, 2216, 2261, 2262, 2263, 2264,
 2286, 2290, 2449, 2719, 3893, 5569,
 10147, 10153, 10399, 12489, 12715,
 13311, 13373, 13378, 13386, 13419,
 18366, 22674, 23937, 24176, 25056,

- 25080, 25110, 25112, 25114, 25116,
 25118, 25120, 25122, 25124, 25142,
 25158, 25159, 25178, 25180, 29562,
 30483, 30484, 30485, 30513, 31158,
 34310, 35778, 35809, 37383, 39156
 \exp_args:Ncc 37, 2051,
 2055, 2063, 2269, 2270, 2271, 2272,
 2449, 2451, 14391, 14573, 16535, 16571
 \exp_args:Nccc 37, 2449, 2453
 \exp_args:Ncco 37, 2534, 2569
 \exp_args:Nccx 38, 3083
 \exp_args:Nce 39950, 39974
 \exp_args:Ncf 37, 2479, 2521
 \exp_args:NcNc 37, 2534, 2555
 \exp_args:NcNo 37, 2534, 2562
 \exp_args:Ncno 38, 3083
 \exp_args:NcnV 38, 3083
 \exp_args:Ncnx 38, 3083
 \exp_args:Nco 37, 411, 2479, 2503
 \exp_args:Ncoo 38, 3083
 \exp_args:NcV 37, 2479, 2509
 \exp_args:Ncv 37, 2479, 2515
 \exp_args:NcVV 38, 3083
 \exp_args:Ncx 37, 3057
 \exp_args:Ne 36, 2077,
 2465, 2465, 4478, 5259, 5260, 5631,
 5940, 5942, 6362, 8362, 8363, 9372,
 10304, 10307, 10548, 10551, 10617,
 10973, 10975, 11099, 11113, 11142,
 11230, 11239, 11260, 11270, 11280,
 11293, 11481, 11507, 13245, 14109,
 14408, 15464, 15479, 18315, 18368,
 19253, 22038, 22074, 22104, 22136,
 22672, 25461, 29900, 30538, 30951,
 30967, 31000, 31314, 31337, 31852,
 31958, 31968, 32102, 32292, 32497,
 32543, 32621, 32637, 32686, 32889,
 32911, 33242, 33374, 33389, 33465,
 33707, 34063, 34209, 36783, 36814,
 37016, 37023, 37153, 37438, 37858,
 37961, 38081, 38227, 38232, 38470,
 38477, 38482, 38550, 38557, 38562,
 38634, 38640, 39322, 39346, 39794
 \exp_args:Nee . . . 37, 3057, 10151,
 11367, 11538, 37029, 37630, 38850
 \exp_args:Neee
 . 38, 3083, 11245, 36776, 37056, 37139
 \exp_args:Nf . 36, 2107, 2467, 2467,
 8387, 9425, 9426, 9754, 9756, 10815,
 10873, 12426, 13108, 13109, 13125,
 13143, 13151, 13155, 13179, 13185,
 13195, 13242, 13277, 13778, 13780,
 13839, 13841, 13857, 14288, 14293,
 14628, 14656, 15180, 15181, 15345,
 15356, 17147, 17148, 17164, 17729,
 17734, 17739, 17744, 17916, 17986,
 17988, 18006, 18015, 18026, 18035,
 18173, 18190, 18963, 18977, 18999,
 19010, 19067, 19137, 19143, 19149,
 19155, 19953, 19955, 20065, 20694,
 20892, 20897, 20902, 20907, 23119,
 25803, 28877, 29417, 29453, 29601,
 30292, 30419, 30592, 30647, 30865,
 31095, 31100, 31202, 31220, 32971,
 33000, 33034, 33112, 33130, 33147,
 33172, 36958, 37945, 39241, 39260
 \exp_args:Nff
 37, 3057, 13149, 16889, 23584
 \exp_args:Nffo 38, 3083
 \exp_args:Nfo 37, 3057
 \exp_args:NNc . 37, 376, 2050, 2053,
 2062, 2133, 2265, 2266, 2267, 2268,
 2303, 2306, 2449, 2449, 2975, 10229,
 10382, 10383, 10472, 16576, 17274,
 17872, 17883, 21008, 21015, 25813,
 25820, 29714, 29839, 29987, 30515
 \exp_args:Nnc 37, 3057, 34049
 \exp_args:NNcf 38, 3083
 \exp_args:NNe 37, 2311, 2479,
 2491, 5737, 11565, 11579, 11645,
 31123, 37773, 39806, 39867, 39915
 \exp_args:Nne
 . 37, 3057, 11106, 11157, 11192, 13035
 \exp_args:NNf 37, 923, 2479,
 2497, 6633, 10228, 10471, 10699,
 20097, 21001, 28104, 28105, 39148
 \exp_args:Nnf
 37, 3057, 12404, 22670, 31191
 \exp_args:Nnff 38, 3083
 \exp_args:Nnnc 38, 3083
 \exp_args:NNne
 37, 429, 2534, 2548, 4931, 5412
 \exp_args:Nnne 11487
 \exp_args:Nnnf 38, 3083
 \exp_args:NNno 37, 465, 2460,
 2463, 4132, 4923, 6034, 6097, 7448
 \exp_args:NNno 38, 3083
 \exp_args:Nnno 38, 3083
 \exp_args:NNNV 37,
 2534, 2534, 31701, 36870, 37423, 37513
 \exp_args:NNnv . 37, 2534, 2541, 10385
 \exp_args:NNnV 38, 3083
 \exp_args:NNNx 38, 3083
 \exp_args:NNnx 38, 3083
 \exp_args:Nnnx 38, 3083
 \exp_args:NNo
 . . 31, 37, 2460, 2461, 6914, 7837,
 12362, 20115, 20274, 22510, 29885

- \exp_args:Nno 37, 3057, 6934,
8902, 10356, 11089, 12389, 12450,
12572, 12581, 12886, 12979, 13013,
13721, 20859, 23628, 23636, 23645,
23662, 23670, 23698, 24202, 24206
- \exp_args:NNoo 38, 3083
- \exp_args:NNox 38, 3083
- \exp_args:Nnox 38, 3083
- \exp_args:NNV 37, 2479, 2479
- \exp_args:NNv 37, 2479, 2485
- \exp_args:NnV 37, 3057, 36937
- \exp_args:Nnv 37, 3057
- \exp_args:NNVV 38, 3083
- \exp_args:NNx 37, 3057
- \exp_args:Nnx 37, 3057
- \exp_args:No 33, 36, 113,
731, 2295, 2300, 2460, 2460, 2941,
2964, 2971, 3043, 3060, 3888, 3919,
3979, 3981, 4313, 5013, 5077, 5097,
5782, 5831, 6336, 6758, 6763, 6956,
6971, 7066, 7264, 7662, 7666, 7691,
7692, 7887, 8877, 8892, 9589, 10415,
10611, 10707, 11077, 11177, 12377,
12626, 12627, 12628, 12655, 12656,
12657, 12658, 12659, 12694, 12724,
12734, 12755, 12824, 12827, 12829,
12885, 12894, 13101, 13103, 13127,
13134, 13136, 13193, 13202, 13660,
13687, 13692, 13706, 13727, 13774,
13785, 13835, 13846, 13913, 13932,
13970, 13985, 14501, 14554, 14840,
16767, 17992, 17998, 18652, 18664,
18666, 18701, 18706, 18881, 18993,
18997, 19031, 21245, 22040, 22076,
22106, 22138, 22240, 22510, 22543,
25460, 29735, 29750, 29770, 29821,
29897, 29966, 30283, 31454, 32109,
34754, 39033, 39041, 39798, 40034
- \exp_args:Noc 37, 3057
- \exp_args:Nof 37, 3057, 12419
- \exp_args:Noo 37, 3057, 5119, 6349
- \exp_args:Noof 38, 3083
- \exp_args:Nooo 38, 3083, 22207
- \exp_args:Noooo 10153
- \exp_args:Noox 38, 3083
- \exp_args:Nox 37, 3057
- \exp_args:NV 36, 2467,
2474, 10849, 10925, 11064, 11625,
11630, 21881, 22036, 22072, 22102,
22134, 30528, 31700, 32223, 33852,
36828, 37388, 37677, 37690, 37788,
37812, 37854, 38199, 39221, 39223
- \exp_args:Nv
. 36, 2467, 2469, 31181, 32004,
33801, 33959, 34205, 36827, 38270
- \exp_args:NVo 37, 3057, 21873
- \exp_args:NVV 37, 2479, 2527, 10760
- \exp_args:Nx 36, 2576,
2576, 2984, 22042, 22078, 22108, 22140
- \exp_args:Nxo 37, 3057
- \exp_args:Nxx 37, 3057
- \exp_args_generate:n
. 34, 3041, 3041, 10148, 11498
- \exp_args:Nn 38140
- \exp_end: 42, 381, 384, 388, 408, 409,
416, 573–575, 596, 715, 722, 730,
731, 745, 1053, 1083, 1415, 1705,
1718, 1726, 1734, 2438, 2447, 2687,
2975, 7844, 7849, 7898, 7928, 7939,
7946, 8560, 8596, 8599, 8600, 8601,
8602, 8603, 8604, 8605, 8606, 8607,
8609, 12453, 13055, 13174, 13317,
13336, 13650, 13834, 17755, 18387,
19212, 19219, 19222, 19261, 19265,
19699, 20918, 23919, 24878, 27533,
29244, 29246, 29318, 29521, 29538,
29700, 31736, 32152, 34081, 39255
- \exp_end_continue_f:nw 43, 2687, 2695
- \exp_end_continue_f:w 42,
43, 408, 1055, 1056, 2407, 2468,
2501, 2525, 2596, 2611, 2624, 2648,
2663, 2676, 2687, 2689, 8394, 10071,
10782, 12679, 18461, 19751, 20856,
23303, 23418, 23422, 24036, 24054,
24075, 24139, 24144, 24152, 24160,
24181, 24259, 24295, 24303, 24334,
24704, 24711, 24757, 24804, 24811,
24848, 24892, 24898, 24901, 24911,
24922, 25087, 25280, 25282, 25286,
25288, 25346, 25356, 25366, 25378,
25560, 25577, 25587, 25753, 25754,
25755, 25936, 25947, 25957, 25965,
26978, 27686, 27862, 28138, 28881,
28896, 28913, 28950, 28967, 29009,
29028, 29041, 29073, 29088, 29099,
29195, 29282, 29523, 29528, 29586,
29875, 30040, 30140, 32026, 39296
- \exp_last_two_unbraced:Nnn . . . 39,
2666, 2666, 20707, 35852, 36376, 36380
- \exp_last_unbraced:cf
. 36740, 36746, 36752
- \exp_last_unbraced:Nco
. 39, 2603, 2626, 18783
- \exp_last_unbraced:NcV 39, 2603, 2628
- \exp_last_unbraced:Ne
. 39, 2603, 2608, 29152
- \exp_last_unbraced:Nf
. 39, 2603, 2610, 4629, 5960, 14724,

- 15009, 15198, 15370, 16464, 16485,
 16606, 16628, 18004, 18024, 20154,
 20572, 23133, 23148, 23579, 25551,
 26029, 29471, 29683, 36730, 39219
 \exp_last_unbraced:Nfo
 39, 2603, 2653, 29990
 \exp_last_unbraced:NNf 39, 2603, 2620
 \exp_last_unbraced:Nnf
 39, 2603, 2651, 20545, 20583
 \exp_last_unbraced:NNnf
 39, 2603, 2643, 8307
 \exp_last_unbraced:NNNnf
 39, 2603, 2657, 8312
 \exp_last_unbraced:NNNNo
 39, 2603, 2655,
 2735, 2739, 2902, 10991, 14028,
 14841, 21704, 23371, 23389, 31495
 \exp_last_unbraced:NNNo
 39, 2603, 2634, 20703, 20741
 \exp_last_unbraced:NnNo 39, 2603, 2654
 \exp_last_unbraced:NNNV 39, 2603, 2636
 \exp_last_unbraced:NNo
 39, 2603, 2612,
 10614, 13063, 31762, 34096, 36347
 \exp_last_unbraced:Nno
 39, 2603, 2650, 17232, 18812
 \exp_last_unbraced:NNV 39, 2603, 2614
 \exp_last_unbraced:No 39,
 2603, 2603, 36503, 36508, 36576, 36582
 \exp_last_unbraced:Noo 39, 2603, 2652
 \exp_last_unbraced:NV
 39, 2603, 2604, 7945, 38212
 \exp_last_unbraced:Nv
 39, 1238, 2603, 2606
 \exp_last_unbraced:Nx 39, 2603, 2665
 \exp_not:N
 . . 40, 98, 166, 167, 280, 409, 415,
 446, 456, 463, 544, 572, 718–720,
 895, 902, 912, 1061, 1411, 1412,
 1660, 1751, 1754, 2139, 2140, 2216,
 2217, 2326, 2392, 2431, 2577, 2671,
 2671, 2712, 2714, 2715, 2722, 2723,
 2724, 2725, 2726, 2727, 2733, 2759,
 2768, 2823, 2824, 2884, 2890, 2892,
 2898, 2945, 2962, 2964, 2992, 3652,
 3655, 3808, 3809, 3810, 3811, 3812,
 3813, 3814, 3815, 3816, 3817, 3818,
 3819, 3820, 3821, 3822, 3823, 4041,
 4050, 4051, 4053, 4059, 4070, 4074,
 4083, 4084, 4085, 4087, 4109, 4113,
 4197, 4199, 4201, 4207, 4209, 4253,
 4605, 4607, 4609, 4611, 4613, 4615,
 5001, 5003, 5201, 5203, 5214, 5218,
 5376, 6049, 6752, 7166, 7179, 7922,
 8339, 8345, 9470, 9472, 9474, 9476,
 9602, 9604, 9608, 9610, 9615, 9617,
 10253, 10254, 10258, 11074, 11077,
 11078, 11080, 11081, 11082, 11083,
 11086, 11087, 11184, 11186, 11442,
 11443, 11444, 11445, 11446, 11449,
 11450, 11485, 11487, 11489, 11492,
 11495, 12513, 12901, 12919, 12947,
 12954, 12965, 12966, 13038, 13041,
 13042, 13678, 13679, 14049, 14052,
 14054, 14055, 14056, 14059, 14618,
 14619, 14646, 14647, 14648, 15486,
 15487, 15488, 15490, 15491, 15493,
 16766, 16768, 17256, 17325, 17888,
 18153, 18715, 19173, 19227, 19229,
 19231, 19232, 19233, 19235, 19237,
 19239, 19240, 19242, 19244, 19246,
 19248, 19256, 19270, 19290, 19293,
 19296, 19299, 19302, 19305, 19308,
 19311, 19314, 19317, 19354, 19358,
 19363, 19368, 19373, 19380, 19387,
 19392, 19397, 19402, 19407, 19412,
 19419, 19424, 19429, 19432, 19433,
 19436, 19446, 19451, 19466, 19485,
 19490, 19491, 19492, 19493, 19494,
 19495, 19497, 19499, 19500, 19501,
 19505, 19506, 19509, 19510, 19602,
 19605, 19606, 19608, 19609, 19613,
 19616, 19617, 19619, 19622, 19742,
 19755, 19769, 19782, 19788, 19819,
 19822, 19829, 19830, 19839, 19840,
 19854, 19855, 19866, 19867, 19987,
 19989, 19993, 19994, 20080, 20329,
 20441, 20476, 20482, 20493, 20628,
 20666, 20671, 20672, 20688, 20691,
 20693, 20694, 20695, 20698, 21020,
 21059, 21718, 21720, 21774, 21775,
 21791, 21856, 21858, 21891, 21892,
 22003, 22004, 22235, 22237, 22687,
 22692, 22696, 22699, 22708, 22709,
 23367, 23368, 24100, 24101, 24196,
 24197, 24198, 24199, 24305, 24345,
 24349, 24371, 24464, 24496, 24581,
 24595, 24612, 24623, 24633, 24670,
 24672, 24775, 24776, 24778, 24779,
 24780, 24781, 24782, 24783, 24786,
 24788, 24790, 24969, 24970, 25011,
 25012, 25132, 25147, 25825, 25982,
 27991, 27992, 27993, 27997, 27998,
 27999, 29438, 29439, 29440, 29442,
 29447, 29577, 29578, 29846, 29847,
 29848, 29907, 29908, 29909, 29935,
 30579, 30802, 30834, 31080, 31082,
 31084, 31085, 31088, 31089, 31090,

- 31411, 31412, 31415, 31426, 31500,
 31503, 31506, 31509, 31512, 31515,
 31518, 31554, 31557, 31559, 31560,
 31561, 31564, 31607, 31610, 31611,
 31614, 31615, 31616, 31617, 31618,
 31619, 31620, 31621, 31622, 31624,
 31625, 31626, 31663, 31805, 31931,
 31932, 31937, 31938, 31968, 32040,
 32114, 32115, 32119, 32121, 32202,
 32273, 33736, 35579, 35580, 36809,
 36810, 36811, 36812, 36813, 36814,
 36815, 37037, 37203, 37204, 37205,
 37206, 37207, 37208, 37444, 37445,
 37449, 37451, 37555, 37556, 37658,
 37664, 37666, 37667, 37773, 37774,
 37775, 37776, 37931, 37932, 37933,
 38070, 38072, 38075, 38110, 38163,
 38592, 38605, 38613, 38764, 38772,
 38788, 38791, 38993, 38996, 38999,
 39002, 39005, 39008, 39325, 39329,
 39339, 39346, 39349, 39380, 39383,
 39796, 39810, 39812, 39870, 39872,
 39918, 39920, 39923, 39925, 39953,
 39955, 39959, 39961, 39977, 39979
- $\backslash\text{exp_not:n}$. 40, 41, 52, 98, 120–123,
 154, 155, 160, 161, 166, 167, 191–
 193, 208, 218, 253, 254, 294, 296,
 366, 440, 445, 446, 452, 456, 463–
 465, 472, 544, 551, 555, 566, 696,
 706, 725, 731–733, 827, 830, 871,
 872, 875, 878, 888, 920, 970, 1299,
 1300, 1303, 1466, 1411, 1413, 1661,
 1667, 1669, 1675, 1676, 1756, 2079,
 2338, 2339, 2392, 2403, 2414, 2592,
 2600, 2671, 2672, 2673, 2675, 2677,
 2682, 2828, 2843, 2858, 2933, 2966,
 2989, 3391, 3680, 3794, 3825, 4071,
 4073, 4102, 4114, 4116, 4122, 4135,
 4253, 4332, 5001, 5003, 5634, 6106,
 6270, 6488, 6676, 6741, 6753, 6831,
 6832, 7094, 7097, 7166, 7174, 7191,
 7206, 7247, 7439, 7567, 7575, 7603,
 7608, 7610, 7890, 8934, 8966, 9464,
 9639, 10570, 10589, 11766, 11769,
 11771, 12213, 12218, 12244, 12249,
 12284, 12289, 12312, 12317, 12514,
 12515, 12516, 13037, 13040, 13044,
 13124, 13317, 13318, 13394, 13497,
 13542, 13553, 13699, 16738, 16740,
 16802, 16803, 16810, 16811, 16827,
 16859, 16928, 16932, 16935, 16936,
 16947, 16950, 16953, 17054, 17086,
 17110, 17163, 17257, 17335, 17386,
 17396, 17889, 18427, 18470, 18471,
 18485, 18487, 18563, 18616, 18652,
 18660, 18680, 18716, 18907, 18912,
 18938, 18941, 18944, 18976, 19008,
 19028, 19257, 19504, 19723, 19743,
 19783, 19789, 19938, 20014, 20015,
 20081, 20090, 20108, 20285, 20292,
 20333, 20334, 20339, 20442, 20448,
 20449, 20452, 20480, 20484, 20491,
 20666, 20674, 20724, 20735, 21021,
 21485, 21494, 21777, 21791, 21806,
 21856, 21858, 21894, 22005, 22219,
 22229, 22231, 22524, 22534, 22560,
 22562, 24191, 25421, 25423, 25425,
 25523, 25826, 25983, 29163, 29928,
 29929, 29933, 29936, 29937, 31083,
 31430, 31459, 31662, 31664, 31694,
 31854, 31855, 31856, 31993, 32014,
 32058, 32073, 32294, 32295, 32554,
 32565, 34372, 34376, 34377, 35696,
 37665, 38008, 38771, 39811, 39871,
 39919, 39924, 39954, 39960, 39978
- $\backslash\text{exp_stop_f}$: 41, 42, 179, 408,
 440, 446, 656, 724, 826, 827, 842,
 1019, 1032, 1103, 1104, 1195, 1224,
 2404, 2410, 3150, 3410, 3594, 3603,
 3604, 3614, 3625, 3626, 3662, 3732,
 3765, 3766, 3770, 3846, 3862, 3868,
 3947, 4110, 4412, 4413, 4414, 4419,
 4700, 4721, 4722, 4726, 4730, 4731,
 4734, 4735, 4750, 4751, 4754, 4758,
 4759, 4762, 4823, 5174, 5179, 5193,
 5194, 5207, 5269, 5270, 5309, 6275,
 6328, 6842, 6846, 6994, 7018, 7053,
 7058, 7064, 7409, 8619, 8621, 8622,
 8624, 9031, 10229, 10472, 10770,
 10784, 10796, 11006, 13161, 13220,
 13226, 13816, 13832, 13872, 13878,
 13890, 13907, 14043, 14233, 14241,
 14450, 14451, 14452, 14457, 14458,
 14482, 14853, 14915, 14919, 14949,
 14952, 14968, 14972, 14993, 15073,
 15075, 15095, 15096, 15113, 15115,
 15169, 15172, 15173, 15292, 15297,
 15438, 16814, 17469, 17483, 17493,
 17501, 17686, 17691, 17853, 19063,
 19065, 19133, 19135, 19139, 19141,
 19145, 19147, 19151, 19153, 19192,
 19193, 19194, 19195, 19201, 19205,
 19223, 19332, 20340, 20865, 21036,
 23028, 23031, 23158, 23206, 23411,
 23526, 23541, 23788, 23814, 23863,
 23875, 23941, 24082, 24112, 24268,
 24311, 24362, 24382, 24409, 24423,
 24458, 24485, 24494, 24513, 24529,

- 24545, 24563, 24624, 24643, 24659,
24893, 24981, 25023, 25261, 25265,
25642, 25644, 25659, 25676, 25684,
25685, 25872, 25992, 25998, 26013,
26050, 26123, 26145, 26199, 26200,
26208, 26545, 26563, 26707, 26719,
26735, 26752, 27031, 27032, 27129,
27222, 27257, 27275, 27284, 27286,
27442, 27477, 27630, 27680, 27726,
27731, 27813, 27880, 27886, 27901,
27913, 27951, 27972, 28012, 28027,
28042, 28057, 28072, 28087, 28115,
28159, 28425, 28435, 28465, 28617,
28619, 28668, 28741, 28750, 28765,
28817, 28830, 28914, 28968, 29019,
29042, 29292, 29293, 29294, 29303,
29313, 29331, 29400, 29403, 29406,
29545, 29565, 29685, 29936, 29963,
30016, 30020, 30062, 30134, 30141,
30234, 30870, 30871, 30877, 31442,
31486, 31607, 31614, 31631, 31634,
32976, 32979, 32980, 32983, 32984,
33005, 33008, 33011, 33014, 33017,
33020, 33039, 33040, 33046, 33049,
33052, 33055, 33058, 33061, 33064,
33067, 33070, 33073, 33076, 33079,
33082, 33085, 33088, 33117, 33120,
33135, 33138, 33152, 33155, 33158,
33161, 33177, 33180, 33183, 39293
- exp internal commands:
- __exp_arg_last_unbraced:nn
.. 2578, 2578, 2580, 2583, 2588, 2595
 - __exp_arg_next:Nnn . 2392, 2393, 2399
 - __exp_arg_next:nnn
408, 2392, 2392, 2401, 2406, 2419, 2425
 - __exp_eval_error_msg:w
. 2429, 2433, 2442
 - __exp_eval_register:N 2420,
2426, 2429, 2429, 2440, 2441, 2472,
2477, 2483, 2489, 2513, 2519, 2531,
2532, 2539, 2546, 2584, 2589, 2605,
2607, 2618, 2632, 2641, 2680, 2685
 - \l__exp_internal_tl
. 378, 1491, 1495, 1496,
2392, 2392, 2413, 2415, 2600, 2601
 - __exp_last_two_unbraced:nnN
. 2666, 2667, 2668
- \expandafter 3, 4, 7,
8, 12, 13, 16, 17, 28, 29, 53, 54, 61, 229
- \expanded 821
- \expandglyphsinfont 935
- \ExplFileName 10, 11590, 11605, 11619, 11623
- \ExplFileDate 10, 11589, 11602
- \ExplFileDescription . . . 10, 11589, 11602
- \ExplFileExtension . . 11592, 11607, 11616
- \ExplFileName . . . 10, 11591, 11606, 11615
- \ExplFileVersion 10, 11593, 11608, 11617
- \explicitdiscretionary 822
- \explicitlyhyphenpenalty 820
- \ExplSyntaxOff 5,
9, 184, 336, 337, 366, 82, 110, 123
- \ExplSyntaxOn 5, 9,
184, 289, 336, 337, 366, 701, 892, 106
- F**
- fact 275
- false 280
- \fam 230
- \fi 6, 15, 20, 32, 33, 34, 54, 56, 57, 72, 80, 231
- fi commands:
- \fi: 28, 65, 72, 99, 179, 180, 208, 238,
314, 381, 383–385, 388, 389, 446,
467, 468, 565, 566, 569, 596, 665,
701, 706, 745, 747, 749, 848, 857,
895, 926, 927, 936, 937, 1032, 1060,
1075, 1109, 1392, 1396, 1439, 1657,
1665, 1673, 1681, 1701, 1706, 1719,
1727, 1735, 1737, 1738, 1739, 1740,
1765, 1770, 1777, 1804, 1809, 1831,
1836, 1847, 1850, 1857, 1863, 1865,
1866, 1873, 1878, 1886, 1891, 1893,
1894, 1901, 1907, 1909, 1910, 1926,
1929, 1930, 1937, 1940, 1941, 1947,
2071, 2092, 2102, 2116, 2174, 2259,
2377, 2387, 2434, 2437, 2444, 2445,
2694, 2733, 2749, 2756, 2765, 2779,
2780, 2785, 2786, 2787, 2805, 2806,
2807, 2808, 2809, 2810, 2811, 2812,
2813, 2821, 2840, 2842, 2872, 2873,
2874, 2921, 3009, 3020, 3030, 3152,
3163, 3164, 3208, 3239, 3282, 3293,
3302, 3311, 3366, 3376, 3386, 3498,
3500, 3571, 3575, 3579, 3606, 3617,
3629, 3630, 3641, 3659, 3660, 3661,
3668, 3684, 3690, 3693, 3701, 3711,
3726, 3734, 3742, 3753, 3769, 3789,
3803, 3825, 3847, 3855, 3857, 3860,
3867, 3872, 3953, 3954, 4014, 4050,
4051, 4052, 4055, 4064, 4087, 4098,
4133, 4134, 4144, 4157, 4200, 4201,
4208, 4209, 4214, 4215, 4240, 4244,
4318, 4375, 4382, 4383, 4389, 4393,
4400, 4401, 4406, 4407, 4416, 4417,
4421, 4422, 4436, 4444, 4453, 4454,
4481, 4637, 4645, 4656, 4662, 4674,
4713, 4714, 4724, 4727, 4728, 4732,
4736, 4737, 4738, 4739, 4744, 4755,
4756, 4760, 4763, 4764, 4765, 4770,
4804, 4805, 4825, 4826, 4835, 4843,

4844, 4854, 4855, 4864, 4874, 4875,
4886, 4887, 4900, 4919, 4920, 4928,
4929, 4994, 5004, 5037, 5051, 5055,
5118, 5163, 5166, 5167, 5183, 5186,
5209, 5267, 5268, 5273, 5300, 5301,
5312, 5316, 5350, 5355, 5363, 5398,
5405, 5410, 5420, 5432, 5458, 5521,
5550, 5589, 5596, 5607, 5695, 5714,
5720, 5725, 5748, 5760, 5761, 5764,
5776, 5810, 5840, 6224, 6263, 6279,
6303, 6323, 6332, 6389, 6396, 6416,
6434, 6445, 6447, 6477, 6480, 6514,
6520, 6526, 6623, 6660, 6691, 6692,
6722, 6748, 6793, 6845, 6891, 6892,
6904, 6955, 6957, 7010, 7022, 7032,
7041, 7061, 7072, 7098, 7114, 7122,
7172, 7174, 7189, 7191, 7212, 7391,
7412, 7490, 7507, 7508, 7528, 7567,
7569, 7575, 7577, 7582, 7612, 7635,
7651, 7737, 7739, 7740, 7746, 7925,
7941, 8334, 8385, 8404, 8426, 8446,
8462, 8472, 8488, 8498, 8611, 8613,
8615, 8617, 8621, 8624, 8890, 8898,
10331, 10334, 10337, 10414, 10432,
10722, 10763, 10772, 10793, 10803,
10807, 10814, 10822, 11017, 11021,
11024, 11074, 11087, 11114, 11123,
11382, 11391, 11402, 12107, 12371,
12378, 12524, 12525, 12554, 12564,
12579, 12588, 12607, 12621, 12634,
12638, 12651, 12669, 12684, 12875,
12880, 12905, 12923, 12943, 12951,
12961, 12971, 12977, 12984, 12995,
13000, 13002, 13006, 13011, 13015,
13016, 13163, 13175, 13224, 13230,
13231, 13335, 13342, 13347, 13384,
13391, 13397, 13401, 13544, 13549,
13554, 13561, 13566, 13667, 13745,
13749, 13750, 13768, 13821, 13834,
13876, 13882, 13883, 13894, 13907,
13908, 13929, 13967, 13993, 14104,
14210, 14218, 14226, 14237, 14254,
14256, 14402, 14454, 14455, 14460,
14463, 14464, 14486, 14539, 14639,
14855, 14888, 14924, 14925, 14956,
14957, 14977, 14978, 14996, 15081,
15085, 15095, 15105, 15121, 15125,
15128, 15133, 15135, 15178, 15182,
15183, 15274, 15279, 15290, 15296,
15308, 15311, 15313, 15317, 15431,
15445, 15446, 16364, 16372, 16396,
16410, 16418, 16429, 16439, 16459,
16489, 16490, 16562, 16588, 16599,
16618, 16621, 16855, 16858, 16927,
16963, 17016, 17033, 17043, 17097,
17102, 17476, 17477, 17486, 17509,
17526, 17527, 17529, 17546, 17547,
17591, 17644, 17652, 17679, 17687,
17693, 17716, 17724, 17762, 17770,
17855, 18072, 18105, 18153, 18158,
18274, 18300, 18327, 18336, 18540,
18555, 18578, 18592, 19192, 19193,
19194, 19195, 19200, 19201, 19209,
19210, 19211, 19240, 19246, 19264,
19273, 19275, 19321, 19322, 19323,
19324, 19325, 19326, 19327, 19328,
19329, 19330, 19359, 19364, 19369,
19374, 19381, 19388, 19393, 19398,
19403, 19408, 19413, 19420, 19425,
19447, 19458, 19459, 19509, 19510,
19632, 19641, 19650, 19658, 19735,
19764, 19765, 19773, 19811, 19825,
19834, 19844, 19866, 19867, 19868,
19878, 19885, 19887, 20212, 20219,
20263, 20311, 20320, 20468, 20507,
20516, 20547, 20548, 20549, 20550,
20559, 20560, 20561, 20562, 20585,
20586, 20587, 20588, 20597, 20598,
20599, 20600, 20815, 20838, 20847,
20864, 20868, 20882, 20885, 21047,
21048, 21091, 21106, 21107, 21613,
21668, 21680, 23033, 23034, 23069,
23077, 23141, 23160, 23174, 23260,
23276, 23280, 23292, 23302, 23397,
23450, 23453, 23454, 23459, 23473,
23511, 23512, 23513, 23514, 23515,
23516, 23517, 23518, 23519, 23520,
23521, 23522, 23535, 23537, 23548,
23551, 23565, 23570, 23574, 23703,
23794, 23795, 23804, 23805, 23816,
23817, 23818, 23829, 23830, 23831,
23838, 23849, 23850, 23851, 23861,
23862, 23866, 23867, 23875, 23878,
23879, 23887, 23898, 23918, 23941,
23976, 23993, 24012, 24013, 24022,
24028, 24048, 24049, 24077, 24086,
24103, 24110, 24118, 24119, 24220,
24221, 24222, 24225, 24228, 24267,
24283, 24309, 24310, 24317, 24325,
24354, 24355, 24358, 24360, 24361,
24366, 24376, 24379, 24381, 24386,
24417, 24430, 24435, 24441, 24444,
24445, 24479, 24480, 24507, 24508,
24521, 24524, 24535, 24558, 24577,
24587, 24603, 24612, 24618, 24624,
24628, 24633, 24639, 24654, 24665,
24682, 24690, 24692, 24698, 24719,
24747, 24770, 24803, 24805, 24928,

- 24976, 24980, 24990, 24991, 25007,
 25018, 25022, 25032, 25033, 25053,
 25074, 25077, 25107, 25139, 25155,
 25175, 25216, 25228, 25241, 25243,
 25263, 25264, 25271, 25289, 25328,
 25338, 25555, 25571, 25582, 25622,
 25623, 25624, 25631, 25633, 25634,
 25640, 25641, 25644, 25671, 25679,
 25680, 25688, 25689, 25691, 25692,
 25853, 25866, 25876, 25877, 25882,
 25883, 25884, 25885, 25886, 25887,
 25894, 25904, 25911, 25922, 25923,
 25934, 25951, 25996, 25997, 26004,
 26017, 26032, 26042, 26056, 26086,
 26095, 26129, 26151, 26169, 26186,
 26203, 26204, 26206, 26207, 26212,
 26227, 26260, 26289, 26290, 26291,
 26292, 26293, 26306, 26350, 26423,
 26490, 26492, 26493, 26503, 26532,
 26535, 26536, 26547, 26567, 26630,
 26631, 26632, 26644, 26683, 26684,
 26685, 26686, 26692, 26695, 26697,
 26707, 26725, 26740, 26752, 26759,
 27007, 27011, 27013, 27017, 27024,
 27025, 27035, 27036, 27039, 27131,
 27201, 27212, 27224, 27256, 27263,
 27274, 27290, 27297, 27358, 27415,
 27425, 27427, 27437, 27455, 27456,
 27488, 27491, 27500, 27502, 27504,
 27518, 27532, 27556, 27640, 27648,
 27679, 27687, 27693, 27704, 27707,
 27710, 27719, 27728, 27730, 27736,
 27743, 27746, 27755, 27763, 27784,
 27817, 27818, 27845, 27847, 27865,
 27866, 27885, 27896, 27905, 27908,
 27919, 27922, 27925, 27943, 27953,
 27964, 27966, 27975, 28022, 28037,
 28052, 28067, 28082, 28097, 28100,
 28102, 28119, 28164, 28471, 28507,
 28508, 28518, 28559, 28560, 28584,
 28611, 28612, 28615, 28617, 28618,
 28623, 28635, 28654, 28659, 28667,
 28670, 28702, 28712, 28713, 28723,
 28745, 28760, 28778, 28786, 28789,
 28817, 28825, 28841, 28913, 28931,
 28967, 28985, 29018, 29041, 29047,
 29120, 29121, 29226, 29227, 29236,
 29243, 29248, 29258, 29268, 29293,
 29296, 29309, 29341, 29349, 29350,
 29378, 29400, 29401, 29402, 29405,
 29410, 29430, 29431, 29948, 29951,
 30025, 30026, 30067, 30075, 30128,
 30134, 30147, 30232, 30905, 30906,
 30909, 31212, 31254, 31258, 31259,
 31274, 31416, 31420, 31431, 31446,
 31458, 31462, 31478, 31490, 31522,
 31523, 31524, 31525, 31526, 31527,
 31528, 31622, 31626, 31640, 31641,
 32043, 32988, 32991, 32992, 32995,
 32996, 33024, 33025, 33026, 33027,
 33028, 33029, 33044, 33092, 33093,
 33094, 33095, 33096, 33097, 33098,
 33099, 33100, 33101, 33102, 33103,
 33104, 33105, 33106, 33107, 33124,
 33125, 33142, 33143, 33165, 33166,
 33167, 33168, 33187, 33188, 33189,
 34722, 34724, 34730, 39159, 39170,
 39171, 39251, 39252, 39253, 39254,
 39267, 39268, 39284, 39285, 39286,
 39287, 39288, 39289, 39290, 39291,
 39292, 40051, 40052, 40057, 40058
- file commands:
- \file_compare_timestamp:nNn
 11364, 11372
 - \file_compare_timestamp:nNnTF
 102, 11364
 - \file_compare_timestamp_p:nNn
 102, 11364
 - \g_file_curr_dir_str
 99, 10928, 11457, 11463, 11476
 - \g_file_curr_ext_str
 99, 10928, 11459, 11465, 11478
 - \g_file_curr_name_str 99,
 9163, 9286, 10928, 11458, 11464, 11477
 - \file_full_name:n
 . 102, 11097, 11097, 11102, 11214,
 11231, 11239, 11246, 11293, 11368,
 11369, 11409, 11481, 38228, 38233
 - \file_get:nnN
 103, 11054, 11054, 11059, 11060, 11071
 - \file_get:nnNTF . . . 103, 11054, 11056
 - \file_get_full_name:nN
 102, 11205, 11205, 11210, 11211, 11219
 - \file_get_full_name:nNTF
 102, 10220, 11062,
 11205, 11207, 11416, 11422, 11435
 - \file_get_hex_dump:nN
 100, 11310, 11310, 11312, 11322, 11324
 - \file_get_hex_dump:nnnN
 100, 11346, 11346, 11351, 11352, 11361
 - \file_get_hex_dump:nnnNTF
 100, 11346, 11348
 - \file_get_hex_dump:nNTF
 100, 11310, 11311
 - \file_get_md5five_hash:nN
 101, 11310, 11313, 11315, 11326, 11328
 - \file_get_md5five_hash:nNTF
 101, 11310, 11314

- \file_get_size:nN
 101, [11310](#), [11316](#), [11318](#), [11330](#), [11332](#)
- \file_get_size:nNTF 101, [11310](#), [11317](#)
- \file_get_timestamp:nN
 101, [11310](#), [11319](#), [11321](#), [11334](#), [11336](#)
- \file_get_timestamp:nNTF
 101, [11310](#), [11320](#)
- \file_hex_dump:n
 100, [11243](#), [11292](#), [11294](#)
- \file_hex_dump:nnn
 100, [11243](#), [11243](#), [11250](#), [11356](#), [38210](#)
- \file_if_exist:n [11407](#), [11413](#)
- \file_if_exist:nTF 100, [102](#),
 103, [11407](#), [11739](#), [11741](#), [11745](#), [14421](#)
- \file_if_exist_input:n
 103, [11414](#), [11414](#), [11419](#)
- \file_if_exist_input:nTF
 103, [11414](#), [11420](#), [11426](#)
- \file_if_exist_p:n 100, [11407](#)
- \file_input:n
 103, [104](#), [11433](#), [11433](#), [11439](#), [14425](#)
- \file_input_raw:n
 103, [11480](#), [11480](#), [11482](#)
- \file_input_stop: .. 104, [11427](#), [11427](#)
- \file_log_list: ... 104, [11558](#), [11559](#)
- \file_md5five_hash:n
 101, [11222](#), [11238](#), [11240](#)
- \file_parse_full_name:n
 103, [675](#), [11499](#), [11499](#), [11504](#)
- \file_parse_full_name:nNNN
 102, [103](#), [11461](#), [11546](#), [11546](#), [11557](#)
- \file_parse_full_name_apply:nN ..
 103,
 675, [11499](#), [11501](#), [11505](#), [11510](#), [11548](#)
- \l_file_search_path_seq
 100, [101](#), [103](#), [10962](#), [11129](#)
- \file_show_list: ... 104, [11558](#), [11558](#)
- \file_size:n . 101, [11222](#), [11222](#), [11224](#)
- \file_timestamp:n
 74, [101](#), [11222](#), [11225](#), [11227](#)
- file internal commands:
- \l_file_base_name_tl [10957](#)
- __file_compare_timestamp:nnN ...
 11364, [11367](#), [11374](#)
- __file_const:nn [11753](#)
- __file_details:nn
 11222, [11223](#), [11226](#), [11228](#)
- __file_details_aux:nn
 11222, [11230](#), [11233](#), [11261](#)
- \l__file_dir_str . [10959](#), [11462](#), [11463](#)
- __file_ext_check:nn
 11138, [11164](#), [11171](#)
- __file_ext_check:nnn . [11186](#), [11191](#)
- __file_ext_check:nnnn . [11192](#), [11193](#)
- __file_ext_check:nnnw . [11177](#), [11182](#)
- __file_ext_check:nnw
 11172, [11173](#), [11180](#)
- \l_file_ext_str . [10959](#), [11462](#), [11465](#)
- _file_full_name:n
 11097, [11099](#), [11103](#)
- __file_full_name_assign:nnnNNN .
 11549, [11551](#)
- __file_full_name_aux:n
 11097, [11106](#), [11108](#), [11157](#), [11192](#)
- _file_full_name_aux:nN
 11097, [11142](#), [11156](#)
- __file_full_name_aux:Nnn
 11097, [11130](#), [11134](#), [11140](#)
- __file_full_name_aux:nnN
 11097, [11157](#), [11158](#)
- _file_full_name_auxi:nn
 11097, [11113](#), [11116](#)
- __file_full_name_auxii:nn
 11097, [11106](#), [11125](#)
- __file_full_name_slash:n
 11097, [11143](#), [11146](#)
- _file_full_name_slash:nw
 11148, [11150](#)
- __file_full_name_slash:w [11097](#)
- \l_file_full_name_tl
 10957, [11062](#), [11065](#), [11416](#),
 11417, [11422](#), [11423](#), [11435](#), [11436](#)
- _file_get_aux:nnN
 11054, [11064](#), [11072](#)
- __file_get_details:nnN .. [11310](#),
 11323, [11327](#), [11331](#), [11335](#), [11338](#)
- __file_get_do:Nw [11054](#), [11080](#), [11090](#)
- __file_get_full_name_search:nN .
 11205
- _file_hex_dump:n
 11243, [11293](#), [11297](#), [11304](#)
- __file_hex_dump_auxi:nnn
 11243, [11245](#), [11251](#)
- __file_hex_dump_auxii:nnnn
 11243, [11260](#), [11265](#)
- _file_hex_dump_auxiii:nnnn ...
 11243, [11268](#), [11270](#), [11275](#)
- __file_hex_dump_auxiiv:nnn .. [11243](#)
- __file_hex_dump_auxiv:nnn
 11278, [11280](#), [11285](#)
- __file_id_info_auxi:w
 11587, [11598](#), [11600](#)
- __file_id_info_auxii:w
 678, [11587](#), [11610](#), [11612](#)
- __file_id_info_auxiii:w
 11587, [11620](#), [11622](#)
- __file_if_recursion_tail_-
 break:NN [10969](#)

- __file_if_recursion_tail_stop:N
..... 10969
- __file_if_recursion_tail_stop_
do:Nn 10969
- __file_if_recursion_tail_stop_
do:nn 10970
- __file_input:n 11417,
11423, 11433, 11436, 11440, 11452
- __file_input_pop:
..... 11433, 11450, 11468, 11473
- __file_input_pop:nnn
..... 11433, 11471, 11474
- __file_input_push:n
..... 11433, 11445, 11453, 11467
- __file_input_raw:nn
..... 11480, 11481, 11483
- \g_file_internal_ior
..... 11221, 11645, 11656, 11658
- \l_file_internal_tl
..... 10927, 11470, 11471
- __file_kernel_dependency_
compare:nnn
..... 11624, 11630, 11633, 11635
- __file_list:N
..... 11558, 11558, 11559, 11560
- __file_list_aux:n 11558, 11571, 11574
- \c_file_marker_tl
..... 665, 11053, 11078, 11091
- _file_md5_hash:n
..... 11222, 11239, 11241
- __file_mismatched_dependency_
error:nn 11640, 11643, 11643
- __file_name_cleanup:w
..... 11097, 11165, 11169
- __file_name_end:
..... 11097, 11136, 11169, 11170
- __file_name_expand:n
..... 10971, 10976, 10979
- __file_name_expand_cleanup:Nw ..
..... 663, 10971, 10981, 10985
- _file_name_expand_cleanup:w ...
..... 663, 10971, 10989, 10992
- __file_name_expand_end:
. 663, 10971, 10983, 10985, 10988,
10993, 10997, 10999, 11000, 11002
- __file_name_expand_error:Nw ...
..... 663, 664, 10971, 10988, 10999
- __file_name_expand_error_aux:Nw
..... 664, 10971, 11000, 11001
- __file_name_ext_check:nn 11097
- __file_name_ext_check:nnn ... 11097
- __file_name_ext_check:nnnn .. 11097
- __file_name_ext_check:nnnw .. 11097
- __file_name_ext_check:nnw ... 11097
- __file_name_quote:nw
..... 11045, 11046, 11047
- \l_file_name_str 10959, 11462, 11464
- __file_name_strip_quotes:n
..... 10971, 10975, 11008
- __file_name_strip_quotes:nnn . 10971
- __file_name_strip_quotes:nnnw 10971
- __file_name_strip_quotes:nw ...
..... 11010, 11013, 11019, 11022
- __file_name_strip_quotes_
end:wwn 11016, 11021
- __file_name_trim_spaces:n
..... 10971, 10973, 11031
- __file_name_trim_spaces:nw
..... 10971, 11032, 11033
- _file_name_trim_spaces_aux:n ..
..... 10971, 11038, 11042
- __file_name_trim_spaces_aux:w ..
..... 10971, 11043, 11044
- __file_parse_full_name_area:nw .
.... 676, 11511, 11513, 11516, 11520
- __file_parse_full_name_auxi:nN .
..... 11507, 11511, 11511
- __file_parse_full_name_base:nw .
.... 676, 11519, 11522, 11522, 11534
- __file_parse_full_name_tidy:nnnN
676, 11529, 11530, 11532, 11536, 11536
- _file_parse_version:w
..... 11624, 11638, 11639, 11642
- __file_quark_if_nil:n 10966
- __file_quark_if_nil:nTF
.. 10966, 11035, 11049, 11175, 11184
- __file_quark_if_nil_p:n 10966
- \g_file_record_seq 674,
677, 10956, 11444, 11568, 11582, 11583
- _file_size:n .. 11096, 11096, 11113
- \g_file_stack_seq
..... 674, 10931, 11455, 11470
- __file_str_cmp:nn 11363, 11363, 11395
- __file_timestamp:n
..... 11364, 11396, 11397, 11406
- _file_tmp:w
.. 10933, 10937, 10941, 10947, 10953
- \l_file_tmp_seq ... 10963, 11562,
11565, 11568, 11569, 11571, 11579,
11584, 11655, 11657, 11676, 11680
- \filedump 771
- \filemdate 772
- \filesize 773
- \finalhyphenemerits 232
- \firstmark 233
- \firstmarks 490
- \firstvalidlanguage 823
- \fixupboxesmode 824

flag commands:

`\flag_clear:N` ... 182, 5625, 7513,
 7514, 14499, 14527, 14621, 14650,
 14719, 14767, 14768, 14820, 14821,
 15057, 15058, 15059, 15060, 15061,
 15162, 15257, 15258, 15259, 15260,
 15415, 15416, 15417, 18290, 18290,
 18295, 18306, 18349, 29781, 39639
`\flag_clear:n` 18348, 18349
`\flag_clear_new:N`
 .. 182, 778, 15002, 15003, 15004,
 15005, 15185, 15186, 15187, 15365,
 15366, 18305, 18305, 18307, 18350
`\flag_clear_new:n` 18348, 18350
`\flag_ensure_raised:N`
 ... 182, 5652, 5674, 18345, 18345,
 18347, 18361, 23633, 23642, 23650,
 23667, 23676, 23707, 29780, 39613
`\flag_ensure_raised:n` . 18348, 18361
`\flag_height:N` 182,
 7523, 7525, 14343, 18315, 18331,
 18331, 18341, 18343, 18359, 39614
`\flag_height:n` .. 18348, 18359, 18369
`\flag_if_exist:N` 18317, 18319
`\flag_if_exist:NTF` ... 182, 18306,
18317, 18352, 18353, 18354, 39113
`\flag_if_exist:nTF`
 18348, 18352, 18353, 18354
`\flag_if_exist_p:N`
 182, 364, 18317, 18351
`\flag_if_exist_p:n` 18348
`\flag_if_raised:N` 18321, 18329
`\flag_if_raised:NTF` ... 182, 5633,
 14336, 14341, 14343, 15029, 15035,
 15040, 15047, 15219, 15224, 15229,
 15380, 15387, 18321, 18356, 18357,
 18358, 29783, 39615, 39616, 39617
`\flag_if_raised:nTF`
 18348, 18356, 18357, 18358
`\flag_if_raised_p:N`
 182, 18321, 18355, 39618
`\flag_if_raised_p:n` 18348
`\flag_log:N` .. 182, 18308, 18310, 18311
`\flag_log:n` 18362, 18363
`\flag_new:N`
 . 181, 182, 778, 5615, 7373, 7374,
 14205, 14206, 18285, 18285, 18287,
 18288, 18289, 18306, 18348, 23599,
 23600, 23601, 23602, 29767, 39632
`\flag_new:n` 18348, 18348
`\flag_raise:N` 182,
 7566, 7574, 14485, 14535, 14635,
 14668, 14739, 14752, 14790, 14795,
 14876, 15078, 15079, 15102, 15103,

15116, 15117, 15136, 15137, 15143,
 15144, 15174, 15324, 15325, 15434,
 15435, 15439, 15440, 15454, 15455,
18342, 18342, 18344, 18360, 39619
`\flag_raise:n` 18348, 18360
`\flag_show:N` . 182, 18308, 18308, 18309
`\flag_show:n` 18362, 18362
`\l_tmpa_flag` .. 183, 364, 18288, 39117
`\l_tmpb_flag` 183, 18288
 flag internal commands:
`__flag_clear:wN`
 18290, 18292, 18296, 18302
`__flag_height_end:wN`
 18331, 18335, 18340
`__flag_height_loop:wN`
 18331, 18331, 18332, 18337
`__flag_show:NN`
 18308, 18308, 18310, 18312
`__flag_show:Nn`
 18362, 18362, 18363, 18364
`\floatingpenalty` 234
 floor 276
`\fmtname` 8687, 8690, 8691,
 8703, 8713, 31674, 31935, 31936,
 35576, 35577, 36464, 36465, 38445
`\font` 235
`\fontcharp` 491
`\fontcharht` 492
`\fontcharic` 493
`\fontcharwd` 494
`\fontdimen` 1004, 236
`\fontencoding` 34259
`\fontfamily` 34260
`\fontid` 826
`\fontname` 237
`\fontseries` 34261
`\fontshape` 34262
`\fontsize` 34265
`\footnotesize` 34301
`\forcecjktoken` 1203
`\formatname` 827
 fp commands:
`\c_e_fp` 269, 272, 25532
`\fp_abs:n` 274,
 280, 1218, 29142, 29142, 35135,
 35237, 35239, 35241, 36179, 36181
`\fp_add:Nn`
 260, 1217, 1218, 25443, 25443, 25449
`\fp_clear_function:n` 268, 29965, 29965
`\fp_clear_variable:n`
 268, 29733, 29733, 29904
`\fp_compare:n` 25557
`\fp_compare:nNn` 25573

- \fp_compare:nNnTF [263–265](#), [25573](#), [25725](#),
[25731](#), [25736](#), [25744](#), [25795](#), [25801](#),
[34992](#), [34994](#), [34999](#), [35269](#), [35284](#),
[35293](#), [35921](#), [36153](#), [36902](#), [37085](#),
[37089](#), [37097](#), [37104](#), [37111](#), [37118](#),
[37125](#), [37172](#), [37584](#), [37587](#), [38033](#)
- \fp_compare:nTF [263–265](#),
[274](#), [25557](#), [25697](#), [25703](#), [25708](#), [25716](#)
- \fp_compare_p:n [264](#), [25557](#)
- \fp_compare_p:nNn
[263](#), [25573](#), [36878](#), [36879](#), [36898](#), [36899](#)
- \fp_const:Nn ... [260](#), [25420](#), [25424](#),
[25428](#), [25532](#), [25533](#), [25534](#), [25535](#)
- \l_fp_division_by_zero_flag
..... [270](#), [23599](#), [23667](#), [23676](#)
- \fp_do_until:nn
..... [265](#), [25694](#), [25694](#), [25698](#)
- \fp_do_until:nNnn
..... [264](#), [25722](#), [25722](#), [25726](#)
- \fp_do_while:nn
..... [265](#), [25694](#), [25700](#), [25704](#)
- \fp_do_while:nNnn
..... [265](#), [25722](#), [25728](#), [25732](#)
- \fp_eval:n [261](#), [264](#), [268](#),
[273–280](#), [288](#), [1097](#), [1254](#), [29137](#),
[29139](#), [30292](#), [36769](#), [36771](#), [36777](#),
[36778](#), [36779](#), [36784](#), [36788](#), [36789](#),
[36790](#), [36794](#), [36797](#), [36798](#), [36799](#),
[36959](#), [36969](#), [36973](#), [36974](#), [36975](#),
[36980](#), [36981](#), [36982](#), [36983](#), [37011](#),
[37016](#), [37024](#), [37030](#), [37039](#), [37040](#),
[37041](#), [37057](#), [37058](#), [37059](#), [37067](#),
[37068](#), [37069](#), [37076](#), [37077](#), [37078](#),
[37140](#), [37145](#), [37176](#), [37736](#), [37737](#),
[37738](#), [37739](#), [37751](#), [37752](#), [37753](#),
[37764](#), [37889](#), [37890](#), [37952](#), [38111](#),
[38112](#), [38113](#), [38114](#), [38122](#), [38123](#),
[38124](#), [38125](#), [38141](#), [38147](#), [38164](#),
[38165](#), [38166](#), [38174](#), [38175](#), [38176](#)
- \fp_format:nn [281](#)
- \fp_gadd:Nn .. [260](#), [25443](#), [25444](#), [25450](#)
- .fp_gset:N [242](#), [22087](#)
- \fp_gset:Nn
[260](#), [25420](#), [25422](#), [25427](#), [25444](#), [25446](#)
- \fp_gset_eq:NN [260](#), [25429](#),
[25430](#), [25432](#), [25434](#), [39402](#), [39531](#)
- \fp_gsub:Nn .. [261](#), [25443](#), [25446](#), [25452](#)
- \fp_gzero:N
..... [260](#), [25433](#), [25434](#), [25436](#), [25440](#)
- \fp_gzero_new:N
..... [260](#), [25437](#), [25439](#), [25442](#)
- \fp_if_exist:N [25547](#), [25548](#)
- \fp_if_exist:NnTF
..... [262](#), [25438](#), [25440](#), [25547](#), [29681](#)
- \fp_if_exist_p:N [262](#), [25547](#)
- \fp_if_nan:n [25549](#)
- \fp_if_nan:nTF [264](#), [281](#), [25549](#)
- \fp_if_nan_p:n [264](#), [25549](#)
- \l_fp_invalid_operation_flag ...
..... [270](#), [23599](#), [23633](#), [23642](#), [23650](#)
- \fp_log:N [271](#), [25453](#), [25455](#), [25456](#)
- \fp_log:n [271](#), [25528](#), [25530](#)
- \fp_max:nn [280](#), [29144](#), [29144](#)
- \fp_min:nn [280](#), [29144](#), [29146](#)
- \fp_new:N [260](#), [25417](#),
[25417](#), [25419](#), [25438](#), [25440](#), [25536](#),
[25537](#), [25538](#), [25539](#), [29495](#), [34958](#),
[34959](#), [34960](#), [35086](#), [35087](#), [35446](#),
[35447](#), [35948](#), [35949](#), [36113](#), [36114](#)
- \fp_new_function:n . [268](#), [29820](#), [29820](#)
- \fp_new_variable:n
..... [266–268](#), [1240](#), [29748](#), [29748](#)
- \l_fp_overflow_flag [270](#), [23599](#)
- .fp_set:N [242](#), [22087](#)
- \fp_set:Nn
.. [260](#), [266](#), [25420](#), [25420](#), [25426](#),
[25443](#), [25445](#), [29778](#), [29782](#), [29925](#),
[34980](#), [34981](#), [34982](#), [35105](#), [35107](#),
[35148](#), [35168](#), [35188](#), [35205](#), [35207](#),
[35225](#), [35226](#), [35266](#), [35267](#), [35966](#),
[35967](#), [36133](#), [36135](#), [36173](#), [36174](#)
- \fp_set_eq:NN [260](#), [25429](#),
[25429](#), [25431](#), [25433](#), [35153](#), [35173](#),
[35190](#), [35270](#), [35271](#), [39401](#), [39450](#)
- \fp_set_function:nnn
..... [268](#), [1243](#), [29883](#), [29883](#)
- \fp_set_variable:nn
.. [266–268](#), [1238](#), [1240](#), [29767](#), [29768](#)
- \fp_show:N
.. [266](#), [267](#), [271](#), [25453](#), [25453](#), [25454](#)
- \fp_show:n
..... [266–268](#), [271](#), [1240](#), [25528](#), [25528](#)
- \fp_sign:n [261](#), [29140](#), [29140](#)
- \fp_step_function:nnnN
..... [266](#), [25750](#), [25750](#), [25757](#), [25832](#)
- \fp_step_inline:nnnn [266](#), [25810](#), [25810](#)
- \fp_step_variable:nnnN
..... [266](#), [25810](#), [25817](#)
- \fp_sub:Nn ... [261](#), [25443](#), [25445](#), [25451](#)
- \fp_to_decimal:N . [261](#), [262](#), [23590](#),
[28944](#), [28944](#), [28946](#), [28975](#), [29137](#)
- \fp_to_decimal:n
..... [261](#), [262](#), [28944](#), [28947](#),
[29139](#), [29141](#), [29143](#), [29145](#), [29147](#)
- \fp_to_dim:N
..... [261](#), [1216](#), [29067](#), [29067](#), [29069](#)

- \fp_to_dim:n
 .. [261](#), [270](#), [29067](#), [29070](#), [35024](#),
 [35035](#), [35135](#), [35876](#), [35898](#), [35926](#),
 [35940](#), [36050](#), [36058](#), [36189](#), [36191](#)
- \fp_to_int:N . [261](#), [29083](#), [29083](#), [29084](#)
- \fp_to_int:n . [261](#), [29083](#), [29085](#), [37588](#)
- \fp_to_scientific:N
 [262](#), [28890](#), [28890](#), [28892](#), [28921](#), [28928](#)
- \fp_to_scientific:n [262](#), [28890](#), [28893](#)
- \fp_to_tl:N [262](#), [284](#), [23591](#),
 [25461](#), [29023](#), [29023](#), [29024](#), [29788](#)
- \fp_to_tl:n [262](#), [23217](#),
 [23632](#), [23641](#), [23666](#), [23675](#), [23704](#),
 [25301](#), [25316](#), [25529](#), [25531](#), [25767](#),
 [25768](#), [25787](#), [25798](#), [29023](#), [29025](#)
- \fp_trap:nn [270](#), [1037](#), [23603](#),
 [23603](#), [23718](#), [23719](#), [23720](#), [23721](#)
- \l_fp_underflow_flag [270](#), [23599](#)
- \fp_until_do:nn
 [265](#), [25694](#), [25706](#), [25711](#)
- \fp_until_do:nNnn
 [265](#), [25722](#), [25734](#), [25739](#)
- \fp_use:N [262](#), [284](#), [29137](#), [29137](#), [29138](#)
- \fp_while_do:nn
 [265](#), [25694](#), [25714](#), [25719](#)
- \fp_while_do:nNnn
 [265](#), [25722](#), [25742](#), [25747](#)
- \fp_zero:N
 [260](#), [25433](#), [25433](#), [25435](#), [25438](#)
- \fp_zero_new:N [260](#), [25437](#), [25437](#), [25441](#)
- \c_inf_fp [269](#),
 [279](#), [23231](#), [24813](#), [26303](#), [26385](#),
 [26723](#), [27484](#), [27507](#), [27709](#), [27712](#),
 [27716](#), [27739](#), [27941](#), [28104](#), [30145](#)
- \c_minus_inf_fp
 [269](#), [279](#), [23231](#), [26304](#),
 [26388](#), [26721](#), [27259](#), [28105](#), [30146](#)
- \c_minus_zero_fp
 [269](#), [23231](#), [26300](#), [28824](#), [30144](#)
- \c_nan_fp
 [269](#), [279](#), [1040](#), [1065](#), [23231](#), [23643](#),
 [23651](#), [23723](#), [23929](#), [23948](#), [23954](#),
 [23977](#), [24144](#), [24152](#), [24160](#), [24238](#),
 [24295](#), [24334](#), [24725](#), [24802](#), [24814](#),
 [25303](#), [25318](#), [25791](#), [27683](#), [29201](#),
 [29780](#), [29959](#), [30058](#), [30117](#), [30143](#)
- \c_one_degree_fp [269](#), [279](#), [24816](#), [25534](#)
- \c_one_fp [269](#), [1093](#),
 [1202](#), [24817](#), [25246](#), [25267](#), [25532](#),
 [25891](#), [26744](#), [27478](#), [27678](#), [27729](#),
 [27914](#), [28028](#), [28058](#), [28607](#), [29217](#)
- \c_pi_fp . [269](#), [279](#), [1075](#), [24815](#), [25534](#)
- \g_tmpa_fp [269](#), [25536](#)
- \l_tmpa_fp [269](#), [25536](#)
- \g_tmpb_fp [269](#), [25536](#)
- \l_tmpb_fp [266](#), [267](#), [269](#), [25536](#)
- \c_zero_fp [269](#), [1097](#),
 [1114](#), [1246](#), [23231](#), [23285](#), [24818](#),
 [25258](#), [25270](#), [25418](#), [25433](#), [25434](#),
 [25893](#), [25896](#), [26132](#), [26299](#), [27487](#),
 [27508](#), [27706](#), [27742](#), [28822](#), [28928](#),
 [29112](#), [30142](#), [34992](#), [34994](#), [34999](#),
 [35284](#), [35293](#), [36153](#), [36902](#), [38033](#)
- fp internal commands:
 fp [25900](#), [25907](#), [25916](#), [25917](#)
 fp&o:ww [1100](#), [1109](#), [25897](#)
 fp&symbolic_o:ww [29549](#)
 fp&tuple_o:ww [25897](#)
 fp*_o:ww [26264](#)
 fp*_symbolic_o:ww [29549](#)
 fp*_tuple_o:ww [26771](#)
 _fp+_o:ww . [1112](#), [1113](#), [1142](#), [25985](#)
 _fp+_symbolic_o:ww [29549](#)
 _fp-_o:ww [1112](#), [1113](#), [25980](#)
 _fp-_symbolic_o:ww [29549](#)
 _fp/_o:ww . [1121](#), [1122](#), [1165](#), [26376](#)
 _fp/_symbolic_o:ww [29549](#)
 _fp{op}_o:w [1232](#)
 _fp^_o:ww [27674](#)
 _fp^_symbolic_o:ww [29549](#)
 _fp_acos_o:w [1206](#), [1209](#), [28763](#), [28763](#)
 _fp_acot_o:Nw
 [28003](#), [28005](#), [28595](#), [28601](#)
 _fp_acotii_o:Nww [28605](#), [28608](#), [28628](#)
 _fp_acotii_o:ww [1202](#)
 _fp_acsc_normal_o:NnwNnw
 ... [1208](#), [28821](#), [28836](#), [28844](#), [28844](#)
 _fp_acsc_o:w [28815](#), [28815](#)
 _fp_add:NNnn [25443](#),
 [25443](#), [25444](#), [25445](#), [25446](#), [25447](#)
 _fp_add_big_i:wNww [1115](#)
 _fp_add_big_i_o:wNww
 [1112](#), [1115](#), [26052](#), [26059](#), [26059](#)
 _fp_add_big_ii:wNww [1115](#)
 _fp_add_big_ii_o:wNww
 [26055](#), [26059](#), [26067](#)
 _fp_add_inf_o:Nww
 [26001](#), [26021](#), [26021](#)
 _fp_add_normal_o:Nww
 [1115](#), [26000](#), [26036](#), [26036](#)
 _fp_add_npos_o:NnwNnw
 [1115](#), [26039](#), [26045](#), [26045](#)
 _fp_add_return_ii_o:Nww
 [26003](#), [26009](#), [26009](#), [26014](#)
 _fp_add_significand_carry_
 o:wwwNN . [1117](#), [26092](#), [26107](#), [26107](#)
 _fp_add_significand_no_carry_
 o:wwwNN . [1116](#), [26094](#), [26097](#), [26097](#)

- __fp_add_significand_o:NnnwnnnN
1115, 1116, 26062, 26070, 26075, 26075
- __fp_add_significand_pack:NNNNNNN
26075, 26079, 26082
- __fp_add_significand_test_o:N ..
26075, 26077, 26089
- __fp_add_zeros_o:Nww ..
25999, 26011, 26011
- __fp_and_return:wNw ..
25897, 25903, 25910, 25922
- __fp_array_bounds:NNnTF ..
30014, 30014, 30045, 30115
- __fp_array_bounds_error:NNn ..
30014, 30017, 30021, 30028
- __fp_array_count:n 23334, 23334,
23913, 25651, 25652, 26784, 28863
- __fp_array_gset:NNNww ..
30033, 30036, 30043
- __fp_array_gset:w 30033, 30049, 30060
- __fp_array_gset_normal:w ..
30033, 30064, 30070
- __fp_array_gset_recover:Nw ..
30033, 30050, 30055
- __fp_array_gset_special:nnNNN ..
30033,
30063, 30065, 30066, 30078, 30090
- __fp_array_gzero:N .. 1246
- __fp_array_if_all_fp:nTF ..
23346, 23346, 23346, 25296
- __fp_array_if_all_fp_loop:w ..
23346, 23348, 23351, 23354
- \g__fp_array_int ..
29979, 29986, 29988, 30000
- __fp_array_item:N 30097, 30121, 30126
- __fp_array_item:NNnN ..
30097, 30116, 30119
- __fp_array_item:NwN ..
30097, 30099, 30107, 30113
- __fp_array_item:w 30097, 30129, 30131
- __fp_array_item_normal:w ..
30097, 30133, 30149
- __fp_array_item_special:w ..
30097, 30128, 30137
- \l__fp_array_loop_int ..
29980, 30086, 30089, 30092
- __fp_array_new:nNNN .. 29981
- __fp_array_new:nNNNN .. 29990, 29994
- __fp_array_to_clist:n ..
23981, 29148, 29148, 29241, 29651
- __fp_array_to_clist_loop:Nw ..
29148, 29154, 29159, 29164
- __fp_asec_o:w .. 28828, 28828
- __fp_asin_auxi_o:NnNww ..
1207, 1209, 28793, 28796, 28796, 28855
- __fp_asin_isqrt:wn ..
28796, 28799, 28806
- __fp_asin_normal_o:NnwNnnnw ..
28754, 28770, 28781, 28781
- __fp_asin_o:w .. 28748, 28748
- __fp_atan_auxi:ww ..
1204, 28673, 28687, 28687
- __fp_atan_auxii:w 28687, 28688, 28689
- __fp_atan_combine_aux:ww ..
28714, 28728, 28735
- __fp_atan_combine_o:NwwwwN ..
1203, 28632, 28649, 28714, 28714
- __fp_atan_default:w ..
1093, 1202, 28595, 28599, 28605, 28607
- __fp_atan_div:wnwww ..
1203, 28660, 28662, 28662
- __fp_atan_inf_o:NNNw ..
1202, 28620, 28621,
28622, 28630, 28630, 28766, 28839
- __fp_atan_near:wwn ..
28662, 28669, 28675
- __fp_atan_near_aux:wwn ..
28662, 28680, 28682
- __fp_atan_normal_o:NNnwNw ..
1202, 28624, 28640, 28640
- __fp_atan_o:Nw ..
28007, 28009, 28595, 28595
- __fp_atan_Taylor_break:w ..
28698, 28701, 28711
- __fp_atan_Taylor_loop:ww ..
1204, 28693, 28698, 28698, 28706
- __fp_atan_test_o:NwNwNw ..
1208, 28643, 28647, 28647, 28803
- __fp_atanii_o:Nww ..
28599, 28608, 28608, 28629
- __fp_basics_pack_high:NNNNNw ..
1116,
1133, 23444, 23446, 26100, 26252,
26355, 26367, 26509, 26702, 27229
- __fp_basics_pack_high_carry:w ..
1029, 23444, 23449, 23453
- __fp_basics_pack_low:NNNNNw ..
1123, 1133, 23444,
23444, 26102, 26254, 26357, 26369,
26511, 26651, 26653, 26704, 27231
- __fp_basics_pack_weird_high:NNNNNNNw ..
23455, 23463, 26111, 26520
- __fp_basics_pack_weird_low:NNNNw ..
23455, 23455, 26113, 26522
- __fp_bcmp:ww .. 25589, 25617
- \c__fp_big_leading_shift_int ..
23430, 26581, 26917, 26927, 26937
- \c__fp_big_middle_shift_int ..
23430, 26584, 26587, 26590,

- 26593, 26596, 26599, 26603, 26919,
26929, 26939, 26949, 26952, 26955
- \c__fp_big_trailing_shift_int ...
..... 23430, 26607, 26962
- \c__fp_Bigg_leading_shift_int ...
..... 23435, 26430, 26448
- \c__fp_Bigg_middle_shift_int ...
.. 23435, 26433, 26436, 26451, 26454
- \c__fp_Bigg_trailing_shift_int ..
..... 23435, 26439, 26457
- __fp_binary_rev_type_o:Nww
..... 24936, 24949, 26774, 26776
- __fp_binary_type_o:Nww
..... 24936, 24936, 26772, 26785
- \c__fp_block_int 23236, 27181
- __fp_case_return:nw 1032,
23512, 23512, 23542, 23545, 23550,
24042, 27443, 27938, 28620, 28621,
28622, 28915, 28969, 29043, 29045,
29046, 29112, 30063, 30065, 30066
- __fp_case_return_i_o:ww
..... 23519, 23519,
26002, 26016, 26025, 26297, 28611
- __fp_case_return_ii_o:ww 23519,
23521, 26298, 27727, 27745, 28612
- __fp_case_return_o:Nw
..... 1032, 1033, 23513,
23513, 26723, 27478, 27483, 27486,
27678, 27683, 27706, 27709, 27712,
27914, 28028, 28058, 28822, 28824
- __fp_case_return_o:Nww
.... 23517, 23517, 26299, 26300,
26303, 26304, 27729, 27738, 27741
- __fp_case_return_same_o:w
..... 1032, 1033,
23515, 23515, 26532, 26536, 26724,
26736, 26739, 27262, 27490, 27703,
27918, 27921, 28013, 28021, 28036,
28051, 28066, 28073, 28081, 28096,
28751, 28759, 28777, 28823, 28840
- __fp_case_use:nw ... 1032, 23511,
23511, 26027, 26295, 26296, 26301,
26302, 26384, 26387, 26534, 26720,
27255, 27258, 27714, 27924, 28014,
28019, 28029, 28034, 28044, 28049,
28059, 28064, 28074, 28079, 28089,
28094, 28753, 28756, 28766, 28768,
28774, 28818, 28820, 28831, 28834,
28839, 28918, 28925, 28972, 28979
- __fp_change_func_type:NNN
23374, 23374, 24729, 26767, 28900,
28954, 29031, 29077, 29092, 30047
- __fp_change_func_type_aux:w ...
..... 23374, 23383, 23390
- __fp_change_func_type_chk:NNN ..
..... 23374, 23380, 23391
- __fp_chk:w
... 1018–1020, 1075, 1113, 1115,
1117, 1123, 1126, 1233, 23218,
23219, 23220, 23231, 23232, 23233,
23234, 23235, 23245, 23250, 23252,
23253, 23281, 23284, 23286, 23296,
23309, 23328, 23523, 23539, 23699,
23704, 23931, 23985, 23994, 23996,
24827, 25467, 25485, 25502, 25507,
25508, 25618, 25619, 25771, 25787,
25791, 25855, 25856, 25859, 25870,
25871, 25879, 25880, 25888, 25900,
25903, 25907, 25910, 25986, 26006,
26007, 26009, 26010, 26011, 26019,
26022, 26033, 26034, 26036, 26045,
26121, 26273, 26307, 26308, 26311,
26392, 26530, 26538, 26540, 26717,
26726, 26728, 26733, 26741, 26743,
26745, 26749, 27252, 27264, 27266,
27475, 27492, 27494, 27675, 27694,
27696, 27697, 27700, 27717, 27720,
27723, 27747, 27748, 27750, 27766,
27855, 27868, 27870, 27874, 27878,
27911, 27927, 28010, 28023, 28025,
28038, 28040, 28053, 28055, 28068,
28070, 28083, 28085, 28098, 28108,
28609, 28625, 28626, 28630, 28641,
28748, 28761, 28763, 28779, 28782,
28792, 28815, 28826, 28828, 28842,
28844, 28849, 28911, 28932, 28935,
28965, 28986, 28989, 29039, 29055,
29058, 29133, 29134, 29218, 29220,
29252, 30060, 30068, 30071, 30150
- __fp_clear_function:n
..... 29965, 29966, 29967
- __fp_clear_variable:n
..... 29733, 29735, 29737
- __fp_clear_variable_aux:n
... 1239, 29733, 29741, 29743, 29900
- __fp_compare:wNNNNw 25186
- __fp_compare_aux:wn
..... 25573, 25576, 25584
- __fp_compare_back:ww 1224,
25589, 25589, 25605, 25869, 29236
- __fp_compare_back_any:ww
..... 1101–1103,
25261, 25586, 25589, 25600, 25668
- __fp_compare_back_tuple:ww
..... 25645, 25645
- __fp_compare_nan:w
... 1102, 25589, 25622, 25623, 25644
- __fp_compare_npos:nwnw

- [1100](#), [1102](#), [1104](#),
[25628](#), [25674](#), [25674](#), [26123](#), [27031](#)
- _fp_compare_return:w
- [25557](#), [25559](#), [25562](#)
- _fp_compare_significand:nnnnnnnn
..... [25674](#), [25677](#), [25682](#)
- _fp_cos_o:w [28025](#), [28025](#)
- _fp_cot_o:w [1187](#), [28085](#), [28085](#)
- _fp_cot_zero_o:Nnw
- [1186](#), [1187](#), [28043](#), [28085](#), [28088](#), [28100](#)
- _fp_csc_o:w [28040](#), [28040](#)
- _fp_decimate:nNnnnn
- .. [1030](#), [1033](#), [1181](#), [23465](#), [23465](#),
[23530](#), [23557](#), [23998](#), [26061](#), [26069](#),
[26148](#), [27521](#), [27525](#), [27893](#), [28995](#)
- _fp_decimate:Nnnnn . [23477](#), [23477](#)
- _fp_decimate_auxi:Nnnnn [1031](#), [23481](#)
- _fp_decimate_auxii:Nnnnn ... [23481](#)
- _fp_decimate_auxiii:Nnnnn ... [23481](#)
- _fp_decimate_auxiv:Nnnnn ... [23481](#)
- _fp_decimate_auxix:Nnnnn ... [23481](#)
- _fp_decimate_auxv:Nnnnn ... [23481](#)
- _fp_decimate_auxvi:Nnnnn ... [23481](#)
- _fp_decimate_auxvii:Nnnnn . [23481](#)
- _fp_decimate_auxviii:Nnnnn . [23481](#)
- _fp_decimate_auxx:Nnnnn [23481](#)
- _fp_decimate_auxxi:Nnnnn ... [23481](#)
- _fp_decimate_auxxii:Nnnnn ... [23481](#)
- _fp_decimate_auxxiii:Nnnnn ... [23481](#)
- _fp_decimate_auxxiv:Nnnnn ... [23481](#)
- _fp_decimate_auxxv:Nnnnn ... [23481](#)
- _fp_decimate_auxxvi:Nnnnn . [23481](#)
- _fp_decimate_pack:nnnnnnnnnw .
..... [1031](#), [23488](#), [23507](#), [23507](#)
- _fp_decimate_pack:nnnnnnw
..... [23508](#), [23509](#)
- _fp_decimate_tiny:Nnnnn
- [23477](#), [23479](#)
- _fp_div_npos_o:Nww
- [1125](#), [1126](#), [26381](#), [26391](#), [26391](#)
- _fp_div_significand_calc:wnnnnnnnn
..... [1129](#), [26408](#),
[26417](#), [26417](#), [26465](#), [27335](#), [27342](#)
- _fp_div_significand_calc_
i:wnnnnnnnn ... [26417](#), [26420](#), [26425](#)
- _fp_div_significand_calc_
ii:wnnnnnnnn .. [26417](#), [26422](#), [26443](#)
- _fp_div_significand_i_o:wnnw ..
.... [1126](#), [1129](#), [26398](#), [26404](#), [26404](#)
- _fp_div_significand_ii:wnn ...
..... [1131](#),
[26412](#), [26413](#), [26414](#), [26461](#), [26461](#)
- _fp_div_significand_iii:wnnnnnn
..... [1131](#), [26415](#), [26468](#), [26468](#)
- _fp_div_significand_iv:wnnnnnnnn
..... [1131](#), [26471](#), [26476](#), [26476](#)
- _fp_div_significand_large_
o:wnnnnnnwN
- [1133](#), [26502](#), [26516](#), [26516](#)
- _fp_div_significand_pack:NNN ..
..... [1133](#), [1167](#), [26463](#),
[26496](#), [26496](#), [27322](#), [27340](#), [27348](#)
- _fp_div_significand_small_
o:wnnnnnnwN
- [1133](#), [26500](#), [26506](#), [26506](#)
- _fp_div_significand_test_o:w ..
..... [1133](#), [26406](#), [26497](#), [26497](#)
- _fp_div_significand_v:NN
- [26481](#), [26483](#), [26486](#)
- _fp_div_significand_v:NNw .. [26476](#)
- _fp_div_significand_vi:Nw
..... [1132](#), [26476](#), [26479](#), [26487](#)
- _fp_division_by_zero_o:Nnw ...
..... [1037](#), [23663](#), [23711](#),
[23714](#), [26721](#), [27259](#), [28104](#), [28105](#)
- _fp_division_by_zero_o:NNww ...
..... [1037](#), [23671](#),
[23711](#), [23715](#), [26385](#), [26388](#), [27716](#)
- \c__fp_empty_tuple_fp
- [23329](#), [24138](#), [24788](#), [24798](#)
- _fp_ep_compare:www
- [27026](#), [27026](#), [28656](#)
- _fp_ep_compare_aux:www
- [27026](#), [27027](#), [27028](#)
- _fp_ep_div:www
- [1199](#), [27056](#), [27056](#), [27167](#),
[28585](#), [28672](#), [28676](#), [28685](#), [28852](#)
- _fp_ep_div_eps_pack:NNNNw ...
..... [27086](#), [27090](#), [27092](#), [27095](#)
- _fp_ep_div_epsilon:wNNNNn [1156](#)
- _fp_ep_div_epsilon:wNNNNNn
- [27083](#), [27086](#), [27086](#)
- _fp_ep_div_epsilonii:wnnNNNNn ...
..... [27086](#), [27088](#), [27097](#)
- _fp_ep_div_esti:www
- [1156](#), [27062](#), [27065](#), [27065](#)
- _fp_ep_div_estii:wnnwn
- [27065](#), [27067](#), [27073](#)
- _fp_ep_div_estiii:NNNNwww ..
..... [27065](#), [27075](#), [27080](#)
- _fp_ep_inv_to_float_o:wN ... [1188](#)
- _fp_ep_inv_to_float_o:wN [1198](#),
[27163](#), [27165](#), [27171](#), [28047](#), [28062](#)
- _fp_ep_isqrt:wnn [27109](#), [27109](#), [28813](#)
- _fp_ep_isqrt_aux:wnn
- [27109](#)
- _fp_ep_isqrt_auxi:wnn [27112](#), [27114](#)
- _fp_ep_isqrt_auxii:wnnnnwn ...
..... [27109](#), [27116](#), [27122](#)

- __fp_ep_isqrt_epsilon:wN
..... [1159](#), [27146](#), [27149](#), [27149](#)
- __fp_ep_isqrt_epsilon:wwN
.. [27149](#), [27152](#), [27153](#), [27154](#), [27156](#)
- __fp_ep_isqrt_esti:wwnnwn
..... [27124](#), [27127](#), [27127](#), [27132](#)
- __fp_ep_isqrt_estii:wwnnwn
..... [27127](#), [27130](#), [27137](#)
- __fp_ep_isqrt_estiii:NNNNwwnn .
..... [27127](#), [27139](#), [27143](#)
- __fp_ep_mul:wwwn
..... [1182](#), [27041](#), [27041](#), [27954](#),
[27967](#), [28542](#), [28572](#), [28800](#), [28811](#)
- __fp_ep_mul_raw:wwwN
.. [27041](#), [27047](#), [27051](#), [28126](#), [28492](#)
- __fp_ep_to_ep:wwN
..... [26992](#), [26992](#), [27043](#),
[27046](#), [27058](#), [27061](#), [27111](#), [28801](#)
- __fp_ep_to_ep_end:www
..... [26992](#), [27006](#), [27010](#)
- __fp_ep_to_ep_loop:N [1197](#), [26992](#),
[26997](#), [27001](#), [27008](#), [27011](#), [28493](#)
- __fp_ep_to_ep_zero:ww
..... [26992](#), [27016](#), [27024](#)
- __fp_ep_to_fixed:wwn [26974](#), [26974](#),
[28123](#), [28679](#), [28688](#), [28798](#), [29261](#)
- __fp_ep_to_fixed_auxi:www
..... [26974](#), [26976](#), [26981](#)
- __fp_ep_to_fixed_auxii:mnnnnnnwn
..... [26974](#), [26987](#), [26990](#)
- __fp_ep_to_float_o:wN [1188](#)
- __fp_ep_to_float_o:wwN
. [1185](#), [1198](#), [27163](#), [27163](#), [27168](#),
[27175](#), [27978](#), [28017](#), [28032](#), [28591](#)
- __fp_error:nmm
..... [23632](#), [23640](#), [23649](#),
[23666](#), [23674](#), [23702](#), [23725](#), [23725](#),
[23727](#), [23924](#), [23926](#), [23947](#), [23952](#),
[24724](#), [25299](#), [25314](#), [25767](#), [25786](#),
[25797](#), [28906](#), [28960](#), [29034](#), [30057](#)
- __fp_exp_after_?_f:nw
..... [1026](#), [1061](#), [24122](#)
- __fp_exp_after_any_f:Nnw
..... [23399](#), [23399](#), [23405](#)
- __fp_exp_after_any_f:nw .. [1027](#),
[23399](#), [23401](#), [23425](#), [24124](#), [24893](#)
- __fp_exp_after_array_f:w
..... [1027](#), [23410](#), [23419](#), [23424](#),
[23425](#), [24778](#), [25937](#), [25948](#), [25958](#),
[25966](#), [29524](#), [29683](#), [29876](#), [29962](#)
- __fp_exp_after_expr_mark_f:nw ..
..... [1061](#), [24122](#), [24130](#)
- __fp_exp_after_expr_stop_f:nw ..
..... [23399](#), [23409](#)
- __fp_exp_after_f:nw . [1023](#), [1061](#),
[23286](#), [23296](#), [23404](#), [24826](#), [24964](#)
- __fp_exp_after_normal:nNNw
..... [23289](#), [23299](#), [23316](#), [23316](#)
- __fp_exp_after_normal:Nwwww
..... [23318](#), [23326](#)
- __fp_exp_after_o:w
..... [1023](#), [23286](#), [23286](#),
[23516](#), [23520](#), [23522](#), [23992](#), [24036](#),
[24054](#), [25281](#), [25887](#), [25905](#), [25914](#),
[25923](#), [26010](#), [26747](#), [27867](#), [27872](#)
- __fp_exp_after_special:nNNw ...
... [1024](#), [23291](#), [23301](#), [23306](#), [23306](#)
- __fp_exp_after_symbolic_aux:w ..
..... [29514](#), [29517](#), [29530](#)
- __fp_exp_after_symbolic_f:nw ...
..... [1233](#), [29514](#), [29514](#),
[29545](#), [29565](#), [29587](#), [29710](#), [29935](#)
- __fp_exp_after_symbolic_loop:N .
..... [29514](#),
[29519](#), [29536](#), [29541](#), [29698](#), [29920](#)
- __fp_exp_after_tuple_f:nw
..... [23410](#), [23411](#), [23412](#), [25088](#)
- __fp_exp_after_tuple_o:w [23410](#),
[23410](#), [25912](#), [25915](#), [25918](#), [25920](#)
- \c__fp_exp_intarray
.. [27568](#), [27654](#), [27661](#), [27664](#), [27666](#)
- __fp_exp_intarray:w
..... [27625](#), [27638](#), [27651](#)
- __fp_exp_intarray_aux:w
.. [27625](#), [27659](#), [27662](#), [27665](#), [27668](#)
- __fp_exp_large:NwN [1174](#),
[27625](#), [27627](#), [27633](#), [27646](#), [27851](#)
- __fp_exp_large_after:wwn
..... [1174](#), [27625](#), [27644](#), [27669](#)
- __fp_exp_normal_o:w
..... [27480](#), [27494](#), [27494](#)
- __fp_exp_o:w ... [27238](#), [27475](#), [27475](#)
- __fp_exp_overflow:NN
..... [27494](#), [27507](#), [27508](#), [27535](#)
- __fp_exp_pos_large:NnnNwn
..... [27526](#), [27625](#), [27625](#)
- __fp_exp_pos_o:NNwnw
..... [27497](#), [27499](#), [27502](#)
- __fp_exp_pos_o:Nwnw [27494](#)
- __fp_exp_Taylor:Nnnwn
..... [27522](#), [27541](#), [27541](#), [27671](#)
- __fp_exp_Taylor_break:Nnw
..... [27541](#), [27555](#), [27566](#)
- __fp_exp_Taylor_ii:ww . [27547](#), [27550](#)
- __fp_exp_Taylor_loop:www
..... [27541](#), [27551](#), [27552](#), [27561](#)
- __fp_expand:n [1218](#)
- __fp_exponent:w [23253](#), [23253](#)

- __fp_facorial_int_o:n 1182
- __fp_fact_int_o:n 27932, 27935
- __fp_fact_int_o:w 27929
- __fp_fact_loop_o:w
 - 27947, 27949, 27949, 27960
- \c__fp_fact_max_arg_int 27910, 27937
- __fp_fact_o:w .. 27242, 27911, 27911
- __fp_fact_pos_o:w 27926, 27929, 27929
- __fp_fact_small_o:w .. 27952, 27964
- \c__fp_five_int 23786,
 - 23810, 23823, 23836, 23843, 23896
- __fp_fixed_(calculation):wnn . 1144
- __fp_fixed_add:nnNnnwn
 - 26867, 26875, 26877
- __fp_fixed_add:Nnnnnwn
 - 26867, 26867, 26868, 26869
- __fp_fixed_add:wnn .. 1144, 1147,
 - 26867, 26867, 27107, 27417, 27425,
 - 27436, 27454, 28684, 28744, 29276
- __fp_fixed_add_after:NNNNwn ...
 - 26867, 26871, 26885
- __fp_fixed_add_one:wN
 - 1145, 26799, 26799,
 - 27100, 27558, 27567, 28810, 29267
- __fp_fixed_add_pack:NNNNwn ...
 - 26867, 26873, 26880, 26883
- __fp_fixed_continue:wn
 - 26798, 26798, 27044,
 - 27049, 27059, 27636, 27826, 28161,
 - 28530, 28802, 28811, 29259, 29271
- __fp_fixed_div_int:wnN
 - 26836, 26841, 26849, 26861
- __fp_fixed_div_int:wwN ... 1146,
 - 26836, 26836, 27416, 27557, 28703
- __fp_fixed_div_int_after:Nw ...
 - 1147, 26836, 26838, 26866
- __fp_fixed_div_int_auxi:wnn ...
 - 26836, 26842,
 - 26843, 26844, 26845, 26846, 26856
- __fp_fixed_div_int_auxii:wnn ...
 - 1147, 26836, 26847, 26864
- __fp_fixed_div_int_pack:Nw
 - 1147, 26836, 26859, 26865
- __fp_fixed_div_myriad:wn
 - 26804, 26804, 27104
- __fp_fixed_inv_to_float_o:wN ...
 - 27170, 27170, 27499, 27762
- __fp_fixed_mul:nnnnnnw
 - 26887, 26907, 26909
- __fp_fixed_mul:wnn .. 1144, 1146,
 - 1148, 1196, 1198, 26887, 26887,
 - 27053, 27084, 27099, 27101, 27105,
 - 27158, 27161, 27174, 27418, 27428,
 - 27468, 27559, 27657, 27672, 27772,
 - 28499, 28553, 28691, 28724, 28726
- __fp_fixed_mul_add:nnnnwnnnn ...
 - 1151, 26956, 26958, 26958
- __fp_fixed_mul_add:nnnnwnnN ...
 - 1152, 26963, 26969, 26969
- __fp_fixed_mul_add:Nwnnnwnnn ...
 - 1151,
 - 26920, 26930, 26941, 26945, 26945
- __fp_fixed_mul_add:wnn
 - 1149, 26914, 26914, 29281
- __fp_fixed_mul_after:wnn
 - 1149, 26806, 26812, 26812, 26815,
 - 26889, 26916, 26926, 26936, 27789
- __fp_fixed_mul_one_minus_-
 - mul:wnn 26914
- __fp_fixed_mul_short:wnn
 - 1146, 26813, 26813,
 - 27082, 27103, 27145, 27147, 28737
- __fp_fixed_mul_sub_back:wwwn ...
 - 1149, 26914, 26924,
 - 27159, 28520, 28522, 28523, 28524,
 - 28525, 28526, 28527, 28528, 28529,
 - 28533, 28535, 28536, 28537, 28538,
 - 28539, 28540, 28541, 28566, 28568,
 - 28569, 28570, 28571, 28574, 28576,
 - 28577, 28578, 28579, 28704, 28712
- __fp_fixed_one_minus_mul:wnn ...
 - 1149-1151, 26934
- __fp_fixed_sub:wnn
 - 26867, 26868, 27151,
 - 27434, 27450, 27462, 28165, 28685,
 - 28742, 28808, 29269, 29278, 29310
- __fp_fixed_to_float_o:Nw
 - 27177, 27177, 27443
- __fp_fixed_to_float_o:wN
 - 1145, 1160, 1205, 27164,
 - 27177, 27178, 27179, 27463, 27473,
 - 27497, 27758, 28732, 29209, 29315
- __fp_fixed_to_float_pack:ww ...
 - 27210, 27220
- __fp_fixed_to_float_rad_o:wN ...
 - 27172, 27172, 28732
- __fp_fixed_to_float_round_-
 - up:wnnnnw 27223, 27227
- __fp_fixed_to_float_zero:w ...
 - 27206, 27215
- __fp_fixed_to_loop:N
 - 27183, 27193, 27197
- __fp_fixed_to_loop_end:w
 - 27199, 27203
- __fp_from_dim:wNNnnnnnn
 - 29102, 29125, 29128
- __fp_from_dim:wnnnnwNn 29129, 29130

__fp_from_dim:wnnnwNw [29102](#)
 __fp_from_dim:wNw [29102](#), [29114](#), [29123](#)
 __fp_from_dim_test:ww [1217](#), [24214](#),
 [24251](#), [24845](#), [29102](#), [29104](#), [29109](#)
 __fp_func_to_name:N
 [23577](#), [23577](#), [24724](#), [24733](#)
 __fp_func_to_name_aux:w
 [23577](#), [23580](#), [23583](#)
 __fp_function_arg_few:w
 [29944](#), [29947](#), [29959](#)
 __fp_function_arg_get:w
 [29944](#), [29950](#), [29960](#)
 \l__fp_function_arg_int
 [29882](#), [29896](#), [29899](#), [29902](#), [29910](#)
 __fp_function_arg_o:w [29909](#),
 [29914](#), [29918](#), [29944](#), [29944](#), [29954](#)
 __fp_function_o:w
 [29624](#), [29847](#), [29864](#), [29864](#), [29871](#)
 __fp_function_set_parsing:Nn
 [29834](#), [29837](#), [29837](#), [29894](#), [29973](#)
 __fp_function_set_parsing_
 aux:NMn [29837](#), [29839](#), [29842](#)
 \c__fp_half_prec_int
 [23236](#), [24455](#), [24487](#)
 __fp_id_if_invalid:n [29655](#)
 __fp_id_if_invalid:nTF
 [1239](#), [29654](#), [29739](#),
 [29754](#), [29774](#), [29824](#), [29890](#), [29969](#)
 __fp_id_if_invalid_aux:N
 [29654](#), [29663](#), [29669](#), [29677](#)
 __fp_if_has_symbolic:nTF
 [29505](#), [29505](#), [29532](#)
 __fp_if_has_symbolic_aux:w
 [29505](#), [29507](#), [29512](#)
 __fp_if_type_fp:NTwFw
 [1025](#), [1093](#), [23266](#),
 [23345](#), [23345](#), [23353](#), [23360](#), [23376](#),
 [23403](#), [25308](#), [25322](#), [25565](#), [25602](#),
 [25603](#), [25760](#), [25761](#), [25762](#), [25928](#)
 __fp_inf_fp:N [23249](#), [23251](#), [23687](#)
 __fp_int:w [23523](#)
 __fp_int:wTF [23523](#), [29220](#)
 __fp_int_eval:w [1028](#),
 [1043](#), [1045](#), [1060](#), [1075](#), [1115](#), [1123](#),
 [1127](#), [1131](#), [1160](#), [23203](#), [23203](#),
 [23263](#), [23338](#), [23469](#), [23472](#), [23860](#),
 [23864](#), [23876](#), [23877](#), [23913](#), [24004](#),
 [24008](#), [24047](#), [24261](#), [24266](#), [24308](#),
 [24397](#), [24408](#), [24457](#), [24488](#), [24494](#),
 [24495](#), [24541](#), [24551](#), [24553](#), [24569](#),
 [24571](#), [24594](#), [24596](#), [24759](#), [24979](#),
 [25021](#), [25221](#), [25578](#), [26049](#), [26057](#),
 [26078](#), [26080](#), [26101](#), [26103](#), [26112](#),
 [26114](#), [26143](#), [26149](#), [26159](#), [26161](#),
 [26235](#), [26237](#), [26253](#), [26255](#), [26259](#),
 [26275](#), [26315](#), [26323](#), [26325](#), [26327](#),
 [26329](#), [26332](#), [26335](#), [26337](#), [26356](#),
 [26358](#), [26368](#), [26370](#), [26396](#), [26399](#),
 [26407](#), [26409](#), [26430](#), [26433](#), [26436](#),
 [26439](#), [26448](#), [26451](#), [26454](#), [26457](#),
 [26464](#), [26466](#), [26472](#), [26480](#), [26482](#),
 [26484](#), [26490](#), [26510](#), [26512](#), [26521](#),
 [26523](#), [26544](#), [26565](#), [26569](#), [26581](#),
 [26584](#), [26587](#), [26590](#), [26593](#), [26596](#),
 [26599](#), [26602](#), [26606](#), [26618](#), [26622](#),
 [26626](#), [26629](#), [26650](#), [26652](#), [26654](#),
 [26664](#), [26703](#), [26705](#), [26714](#), [26802](#),
 [26807](#), [26809](#), [26816](#), [26819](#), [26822](#),
 [26825](#), [26828](#), [26831](#), [26840](#), [26852](#),
 [26860](#), [26862](#), [26872](#), [26874](#), [26881](#),
 [26890](#), [26892](#), [26895](#), [26898](#), [26901](#),
 [26904](#), [26917](#), [26919](#), [26927](#), [26929](#),
 [26937](#), [26939](#), [26949](#), [26952](#), [26955](#),
 [26962](#), [26977](#), [26995](#), [26998](#), [27054](#),
 [27068](#), [27070](#), [27076](#), [27089](#), [27091](#),
 [27093](#), [27117](#), [27133](#), [27140](#), [27141](#),
 [27164](#), [27181](#), [27185](#), [27230](#), [27232](#),
 [27276](#), [27287](#), [27306](#), [27308](#), [27310](#),
 [27323](#), [27336](#), [27341](#), [27343](#), [27349](#),
 [27366](#), [27367](#), [27368](#), [27369](#), [27370](#),
 [27371](#), [27376](#), [27378](#), [27380](#), [27382](#),
 [27384](#), [27389](#), [27391](#), [27393](#), [27395](#),
 [27397](#), [27399](#), [27421](#), [27429](#), [27513](#),
 [27562](#), [27639](#), [27647](#), [27655](#), [27661](#),
 [27664](#), [27769](#), [27790](#), [27792](#), [27795](#),
 [27798](#), [27801](#), [27804](#), [27820](#), [27846](#),
 [27860](#), [27876](#), [27946](#), [27956](#), [27961](#),
 [28113](#), [28145](#), [28154](#), [28386](#), [28400](#),
 [28403](#), [28406](#), [28409](#), [28412](#), [28415](#),
 [28418](#), [28421](#), [28424](#), [28440](#), [28450](#),
 [28459](#), [28477](#), [28486](#), [28493](#), [28504](#),
 [28514](#), [28547](#), [28557](#), [28582](#), [28591](#),
 [28634](#), [28651](#), [28653](#), [28665](#), [28666](#),
 [28707](#), [28718](#), [28729](#), [28787](#), [28939](#),
 [29062](#), [29115](#), [29185](#), [29208](#), [29262](#),
 [29314](#), [29336](#), [29338](#), [29340](#), [29345](#),
 [29364](#), [29376](#), [29384](#), [29389](#), [29394](#)
 __fp_int_eval_end: [23203](#),
 [23204](#), [23263](#), [23341](#), [23460](#), [23913](#),
 [24018](#), [24022](#), [25222](#), [25578](#), [26259](#),
 [26294](#), [26486](#), [26862](#), [26998](#), [27820](#),
 [27876](#), [28146](#), [28155](#), [28504](#), [28514](#),
 [28557](#), [28582](#), [28666](#), [29343](#), [29345](#)
 __fp_int_p:w [23523](#)
 __fp_int_to_roman:w [23203](#), [23205](#),
 [23472](#), [24469](#), [24501](#), [27303](#), [29988](#)
 __fp_invalid_operation:nw
 [1036](#), [1037](#), [23629](#),

- [23711](#), [23711](#), [23723](#), [28920](#), [28927](#),
[28974](#), [28981](#), [29081](#), [29096](#), [29596](#)
__fp_invalid_operation_o:nw ...
..... [1037](#), [23722](#), [23722](#),
[23724](#), [24733](#), [26534](#), [26760](#), [27255](#),
[27924](#), [27933](#), [28020](#), [28035](#), [28050](#),
[28065](#), [28080](#), [28095](#), [28757](#), [28775](#),
[28791](#), [28819](#), [28832](#), [28848](#), [29466](#)
__fp_invalid_operation_o:Nww ...
..... [1037](#), [23637](#), [23711](#), [23712](#), [24934](#),
[26029](#), [26301](#), [26302](#), [27861](#), [29489](#)
__fp_invalid_operation_o:nww . [26786](#)
__fp_invalid_operation_tl_o:nn .
..... [1037](#),
[23646](#), [23711](#), [23713](#), [23979](#), [29240](#)
__fp_kind:w [23264](#), [23264](#), [23972](#), [25551](#)
\c__fp_leading_shift_int
..... [23426](#), [26807](#),
[26816](#), [26890](#), [27790](#), [28440](#), [28477](#)
__fp_ln_c:NwNw
..... [1168](#), [1169](#), [27400](#), [27431](#), [27431](#)
__fp_ln_div_after:Nw
..... [1167](#), [27302](#), [27351](#)
__fp_ln_div_i:w [27324](#), [27333](#)
__fp_ln_div_ii:wn
.. [27327](#), [27328](#), [27329](#), [27330](#), [27338](#)
__fp_ln_div_vi:wn ... [27331](#), [27346](#)
__fp_ln_exponent:wn
..... [1170](#), [27278](#), [27440](#), [27440](#)
__fp_ln_exponent_one:ww [27445](#), [27459](#)
__fp_ln_exponent_small:NNww ...
..... [27448](#), [27452](#), [27465](#)
\c__fp_ln_i_fixed_tl [27243](#)
\c__fp_ln_ii_fixed_tl [27243](#)
\c__fp_ln_iii_fixed_tl [27243](#)
\c__fp_ln_iv_fixed_tl [27243](#)
\c__fp_ln_ix_fixed_tl [27243](#)
__fp_ln_npos_o:w
..... [1162](#), [1163](#), [27264](#), [27266](#), [27266](#)
__fp_ln_o:w
..... [1162](#), [1178](#), [27240](#), [27252](#), [27252](#)
__fp_ln_significand:NNNNnnN ...
... [1164](#), [27277](#), [27280](#), [27280](#), [27770](#)
__fp_ln_square_t_after:w
..... [27375](#), [27407](#)
__fp_ln_square_t_pack:NNNNw ...
.. [27377](#), [27379](#), [27381](#), [27383](#), [27405](#)
__fp_ln_t_large:NNw
..... [1167](#), [27356](#), [27363](#), [27373](#)
__fp_ln_t_small:Nw ... [27354](#), [27361](#)
__fp_ln_t_small:w [1167](#)
__fp_ln_Taylor:wwNw
..... [1168](#), [27408](#), [27409](#), [27409](#)
__fp_ln_Taylor_break:w [27414](#), [27425](#)
__fp_ln_Taylor_loop:ww
..... [27410](#), [27411](#), [27420](#)
__fp_ln_twice_t_after:w [27388](#), [27404](#)
__fp_ln_twice_t_pack:Nw . [27390](#),
[27392](#), [27394](#), [27396](#), [27398](#), [27403](#)
\c__fp_ln_vi_fixed_tl [27243](#)
\c__fp_ln_vii_fixed_tl [27243](#)
\c__fp_ln_viii_fixed_tl [27243](#)
\c__fp_ln_x_fixed_tl
..... [27243](#), [27462](#), [27469](#)
__fp_ln_x_ii:wnnnn
..... [27282](#), [27300](#), [27300](#)
__fp_ln_x_iii:NNNNNw . [27309](#), [27313](#)
__fp_ln_x_iii_var:NNNNw
..... [27307](#), [27315](#)
__fp_ln_x_iv:wnnnnnnn
..... [1166](#), [27305](#), [27320](#)
__fp_logb_aux_o:w [26717](#), [26722](#), [26728](#)
__fp_logb_o:w .. [25975](#), [26717](#), [26717](#)
\c__fp_max_exp_exponent_int
..... [23242](#), [27505](#)
\c__fp_max_exponent_int .. [23240](#),
[23246](#), [23274](#), [27015](#), [27217](#), [27825](#)
\c__fp_middle_shift_int
..... [23426](#), [26819](#),
[26822](#), [26825](#), [26828](#), [26892](#), [26895](#),
[26898](#), [26901](#), [27792](#), [27795](#), [27798](#),
[27801](#), [28443](#), [28450](#), [28480](#), [28486](#)
__fp_minmax_aux_o:Nw
..... [25841](#), [25845](#), [25847](#)
__fp_minmax_auxi:ww
..... [25863](#), [25875](#), [25882](#), [25882](#)
__fp_minmax_auxii:ww
..... [25865](#), [25873](#), [25882](#), [25884](#)
__fp_minmax_break_o:w
..... [25856](#), [25886](#), [25886](#)
__fp_minmax_loop:Nw ... [1108](#),
[25850](#), [25852](#), [25858](#), [25858](#), [25878](#)
__fp_minmax_o:Nw
... [1100](#), [25544](#), [25546](#), [25841](#), [25841](#)
\c__fp_minus_min_exponent_int ...
..... [23240](#), [23275](#)
__fp_misused:n
..... [23216](#), [23216](#), [23220](#), [23331](#)
__fp_mul_cases_o:NnNnw
..... [1125](#), [26266](#), [26272](#), [26378](#)
__fp_mul_cases_o:nNnw [26272](#)
__fp_mul_npos_o:Nw
..... [1122](#), [1123](#), [1125](#),
[1216](#), [1217](#), [26269](#), [26310](#), [26310](#), [29132](#)
__fp_mul_significand_drop:NNNNw
..... [1123](#), [26319](#),
[26328](#), [26331](#), [26334](#), [26336](#), [26340](#)

- __fp_mul_significand_keep:NNNNw
..... 26319, 26324, 26326, 26342
- __fp_mul_significand_large_
f:NwwNNNN 26349, 26353, 26353
- __fp_mul_significand_o:nnnnNnnnn
..... 1123, 26317, 26319, 26319
- __fp_mul_significand_small_
f:NNwwwN 26347, 26364, 26364
- __fp_mul_significand_test_f:NNN
..... 1124, 26321, 26344, 26344
- \c__fp_myriad_int 23239,
26802, 26833, 26834, 26911, 26972
- __fp_neg_sign:N
... 1113, 23262, 23262, 25983, 26136
- __fp_new_function:n
..... 29820, 29821, 29822
- __fp_new_variable:n
..... 29748, 29750, 29752
- __fp_not_o:w 1100, 24752, 25888, 25888
- \c__fp_one_fixed_tl 26796,
27416, 27629, 27826, 27853, 28636,
28703, 28808, 29259, 29269, 29310
- __fp_overflow:w 1023, 1037, 1039,
23277, 23711, 23716, 27507, 27940
- \c__fp_overflowing_fp
..... 23243, 28921, 28975
- __fp_pack:NNNNw
..... 23426, 23429, 26808,
26818, 26821, 26824, 26827, 26830,
26891, 26894, 26897, 26900, 26903,
27791, 27794, 27797, 27800, 27803
- __fp_pack_big:NNNNNw
..... 23430, 23433,
26583, 26586, 26589, 26592, 26595,
26598, 26601, 26605, 26918, 26928,
26938, 26948, 26951, 26954, 26961
- __fp_pack_Bigg:NNNNNw
..... 23435, 23438, 26432,
26435, 26438, 26450, 26453, 26456
- __fp_pack_eight:wNNNNNNNN
..... 1029, 1119, 23442, 23442,
26245, 26554, 26983, 28132, 28133
- __fp_pack_twice_four:wNNNNNNNN .
1029, 23440, 23440, 24029, 24030,
26187, 26188, 26984, 26985, 26986,
27018, 27019, 27020, 27208, 27209,
27544, 27545, 27546, 28134, 28135,
28429, 28430, 28431, 28432, 29125
- __fp_parse:n
..... 1051, 1063, 1075, 1083,
1096, 1097, 1105, 1217, 1218, 1246,
24060, 24211, 24869, 24869, 25421,
25423, 25425, 25448, 25551, 25560,
25577, 25587, 25755, 25805, 26730,
28896, 28950, 29028, 29073, 29088,
29141, 29143, 29145, 29147, 30040
- __fp_parse_after:ww
..... 24869, 24872, 24880, 24885
- __fp_parse_apply_binary:NwNwN . .
..... 1055,
1056, 1059, 1087, 24907, 24907, 25098
- __fp_parse_apply_binary_chk:NN .
.. 24907, 24912, 24924, 24938, 24951
- __fp_parse_apply_binary_
error:NNN 24907, 24927, 24931
- __fp_parse_apply_comma:NwNwN . . .
..... 1087, 25057, 25068, 25083
- __fp_parse_apply_compare:NwNNNNNwN
..... 25245, 25254
- __fp_parse_apply_compare_
aux:NNwN 25266, 25269, 25274
- __fp_parse_apply_function:NNNwN
..... 1078, 24701, 24701, 24862
- __fp_parse_apply_unary:NNNwN . . .
..... 24706, 24706, 24738, 24853
- __fp_parse_apply_unary_chk:nNNNNw
..... 24717, 24718, 24721
- __fp_parse_apply_unary_chk:nNNNw
..... 24706
- __fp_parse_apply_unary_chk:NwNw
..... 24706, 24708, 24713
- __fp_parse_apply_unary_error:NNw
..... 24706, 24729, 24732, 26768
- __fp_parse_apply_unary_type:NNN
..... 24706, 24709, 24727
- __fp_parse_caseless_inf:N
..... 24819, 24819
- __fp_parse_caseless_infinity:N .
..... 24819, 24820
- __fp_parse_caseless_nan:N
..... 24819, 24821
- __fp_parse_compare:NNNNNNN
..... 25186, 25187, 25189,
25191, 25194, 25207, 25215, 25276
- __fp_parse_compare_auxi:NNNNNNN
..... 25186, 25210, 25218, 25232
- __fp_parse_compare_auxii:NNNNN .
..... 25186,
25223, 25224, 25225, 25226, 25230
- __fp_parse_compare_end:NNNw . . .
..... 25186, 25227, 25241
- __fp_parse_continue:NwN
..... 1055, 1056,
1083, 24896, 24899, 24906, 24909,
25085, 25284, 25945, 25955, 25963
- __fp_parse_continue_compare:NNwNN
..... 25277, 25292

- __fp_parse_digits_:N
..... [24078](#), [24096](#), [24097](#)
- __fp_parse_digits_i:N . [24078](#), [24095](#)
- __fp_parse_digits_ii:N [24078](#), [24094](#)
- __fp_parse_digits_iii:N [24078](#), [24093](#)
- __fp_parse_digits_iv:N [24078](#), [24092](#)
- __fp_parse_digits_v:N . [24078](#), [24091](#)
- __fp_parse_digits_vi:N
..... [24078](#), [24090](#), [24413](#), [24461](#)
- __fp_parse_digits_vii:N
..... [1068](#), [24078](#), [24400](#), [24450](#)
- __fp_parse_excl_error:
..... [25186](#), [25202](#), [25211](#)
- __fp_parse_expand:w
 . [1059](#), [1060](#), [24075](#), [24075](#), [24077](#),
 [24087](#), [24127](#), [24187](#), [24231](#), [24240](#),
 [24243](#), [24247](#), [24284](#), [24318](#), [24356](#),
 [24358](#), [24377](#), [24379](#), [24401](#), [24418](#),
 [24431](#), [24451](#), [24481](#), [24509](#), [24525](#),
 [24536](#), [24559](#), [24588](#), [24598](#), [24605](#),
 [24619](#), [24635](#), [24655](#), [24666](#), [24748](#),
 [24771](#), [24783](#), [24858](#), [24867](#), [24875](#),
 [24888](#), [25006](#), [25052](#), [25076](#), [25102](#),
 [25150](#), [25170](#), [25239](#), [25252](#), [25941](#)
- __fp_parse_exponent:N [1073](#), [24186](#),
 [24392](#), [24541](#), [24608](#), [24610](#), [24610](#)
- __fp_parse_exponent:Nw
..... [24416](#), [24429](#), [24478](#),
 [24506](#), [24557](#), [24586](#), [24605](#), [24605](#)
- __fp_parse_exponent_aux:NN
..... [24610](#), [24613](#), [24621](#)
- __fp_parse_exponent_body:N
..... [24637](#), [24641](#), [24641](#)
- __fp_parse_exponent_digits:N ...
..... [24645](#), [24657](#), [24657](#), [24661](#)
- __fp_parse_exponent_keep:N .. [24668](#)
- __fp_parse_exponent_keep:NTF ...
..... [24648](#), [24668](#)
- __fp_parse_exponent_sign:N
..... [24627](#), [24631](#), [24631](#), [24634](#)
- __fp_parse_function:NNN
 [23772](#), [23774](#), [23776](#), [23779](#), [24851](#),
 [24860](#), [25544](#), [25546](#), [28003](#), [28005](#),
 [28007](#), [28009](#), [29170](#), [29172](#), [29846](#)
- __fp_parse_function_all_fp_
 o:nnw ... [23906](#), [25294](#), [25294](#), [25843](#)
- __fp_parse_function_one_two:nnw
 [1202](#),
 [25306](#), [25306](#), [28597](#), [28603](#), [29213](#)
- __fp_parse_function_one_two_
 aux:nnw [25306](#), [25310](#), [25320](#)
- __fp_parse_function_one_two_
 auxii:nnw [25306](#), [25332](#), [25334](#)
- __fp_parse_function_one_two_
 error_o:w
 .. [25306](#), [25309](#), [25312](#), [25329](#), [25337](#)
- __fp_parse_infix:NN
 [1061](#), [1065](#), [1081](#), [1086](#),
 [24126](#), [24296](#), [24335](#), [24811](#), [24826](#),
 [24848](#), [24964](#), [24967](#), [25050](#), [29710](#)
- __fp_parse_infix_!:N [25186](#)
- __fp_parse_infix_&:Nw [25143](#)
- __fp_parse_infix(:N [25126](#)
- __fp_parse_infix):N [25040](#)
- __fp_parse_infix*:N [25128](#)
- __fp_parse_infix+:N
 [1059](#), [24075](#), [25092](#)
- __fp_parse_infix_,:N [25057](#)
- __fp_parse_infix -:N [25092](#)
- __fp_parse_infix/:N [25092](#)
- __fp_parse_infix::N . [25160](#), [25926](#)
- __fp_parse_infix<:N [25186](#)
- __fp_parse_infix=:N [25186](#)
- __fp_parse_infix>:N [25186](#)
- __fp_parse_infix?:N [25160](#)
- __fp_parse_infix(operation):N [1059](#)
- __fp_parse_infix^:N [25092](#)
- __fp_parse_infix_after_operand:NwN
 [1065](#),
 [24179](#), [24257](#), [24755](#), [24962](#), [24962](#)
- __fp_parse_infix_after_paren:NN
 [24780](#), [24806](#), [25009](#), [25009](#)
- __fp_parse_infix_and:N [25092](#), [25159](#)
- __fp_parse_infix_check:NNN
 [24985](#), [24995](#), [25027](#)
- __fp_parse_infix_comma:w
 [1087](#), [25057](#), [25072](#), [25081](#)
- __fp_parse_infix_end:N
 [1083](#), [1087](#), [24876](#),
 [24881](#), [24889](#), [25038](#), [25038](#), [25039](#)
- __fp_parse_infix_juxt:N
 [1086](#), [24975](#), [24983](#), [25092](#)
- __fp_parse_infix_mark:NNN
 [24972](#), [25014](#), [25037](#), [25037](#)
- __fp_parse_infix_mul:N
 [1086](#), [1089](#), [25000](#),
 [25017](#), [25025](#), [25092](#), [25127](#), [25136](#)
- __fp_parse_infix_or:N . [25092](#), [25158](#)
- __fp_parse_infix_|:Nw [25143](#)
- __fp_parse_large:N
 [1067](#), [24363](#), [24446](#), [24446](#)
- __fp_parse_large_leading:wwNN ..
 [1071](#), [24448](#), [24453](#), [24453](#)
- __fp_parse_large_round:NN
 [1071](#), [24489](#), [24561](#), [24561](#)
- __fp_parse_large_round_aux:wNN .
 [24561](#), [24570](#), [24590](#)

- __fp_parse_large_round_test:NN .
..... [24561](#), [24574](#), [24579](#)
- __fp_parse_large_trailing:wwNN .
..... [1071](#), [24459](#), [24483](#), [24483](#)
- __fp_parse_letters:N
... [1065](#), [24272](#), [24286](#), [24301](#), [24313](#)
- __fp_parse_lparen_after:NwN ...
..... [24761](#), [24763](#), [24773](#)
- __fp_parse_o:n
... [1051](#), [24869](#), [24882](#), [25753](#), [25754](#)
- __fp_parse_one:Nw [1054](#)–
[1059](#), [1066](#), [1081](#), [1083](#), [24075](#),
[24098](#), [24098](#), [24340](#), [24700](#), [24902](#)
- __fp_parse_one_digit:NN
..... [1079](#), [24114](#), [24255](#), [24255](#)
- __fp_parse_one_fp:NN
..... [1061](#), [24106](#), [24122](#), [24122](#)
- __fp_parse_one_other:NN
..... [24117](#), [24263](#), [24263](#)
- __fp_parse_one_register:NN
..... [24109](#), [24177](#), [24177](#)
- __fp_parse_one_register_aux:Nw .
..... [24177](#), [24183](#), [24189](#)
- __fp_parse_one_register_–
auxii:wwNw ... [24177](#), [24194](#), [24203](#)
- __fp_parse_one_register_dim:ww .
..... [24177](#), [24197](#), [24209](#), [24212](#)
- __fp_parse_one_register_int:www
..... [24177](#), [24199](#), [24210](#)
- __fp_parse_one_register_–
math:NNw [24218](#), [24224](#), [24227](#), [24230](#)
- __fp_parse_one_register_mu:www .
..... [24177](#), [24198](#), [24207](#)
- __fp_parse_one_register_–
special:N [24182](#), [24218](#), [24218](#)
- __fp_parse_one_register_wd:Nw ..
..... [24218](#), [24246](#), [24249](#)
- __fp_parse_one_register_wd:w ...
.. [24218](#), [24220](#), [24221](#), [24222](#), [24242](#)
- __fp_parse_operand:Nw [1054](#)–
[1057](#), [1059](#), [1083](#), [1087](#), [24075](#),
[24744](#), [24746](#), [24767](#), [24769](#), [24858](#),
[24867](#), [24874](#), [24887](#), [24896](#), [24896](#),
[25075](#), [25101](#), [25169](#), [25252](#), [25940](#)
- __fp_parse_pack_carry:w
..... [1070](#), [24433](#), [24441](#), [24444](#)
- __fp_parse_pack_leading:NNNNww
..... [24396](#), [24433](#), [24438](#), [24456](#)
- __fp_parse_pack_trailing:NNNNww
..... [24406](#),
[24433](#), [24433](#), [24475](#), [24486](#), [24493](#)
- __fp_parse_prefix:NNN
..... [24275](#), [24320](#), [24320](#)
- __fp_parse_prefix_!:Nw [24734](#)
- __fp_parse_prefix_(Nw [24761](#)
- __fp_parse_prefix_):Nw [24793](#)
- __fp_parse_prefix_+:Nw [24700](#)
- __fp_parse_prefix_–:Nw [24734](#)
- __fp_parse_prefix_.:Nw [24753](#)
- __fp_parse_prefix_unknown:NNN ..
..... [24320](#), [24323](#), [24328](#)
- __fp_parse_return_semicolon:w ..
.... [24076](#), [24076](#), [24085](#), [24316](#),
[24523](#), [24534](#), [24617](#), [24649](#), [24664](#)
- __fp_parse_round:Nw .. [23777](#), [23783](#)
- __fp_parse_round_after:wN [1073](#),
[24538](#), [24538](#), [24543](#), [24552](#), [24593](#)
- __fp_parse_round_loop:N
..... [1073](#), [1074](#), [24511](#),
[24511](#), [24516](#), [24554](#), [24572](#), [24597](#)
- __fp_parse_round_up:N
..... [24511](#), [24519](#), [24527](#), [24531](#)
- __fp_parse_small:N
..... [1068](#), [24383](#), [24394](#), [24394](#)
- __fp_parse_small_leading:wwNN ..
... [1069](#), [24398](#), [24403](#), [24403](#), [24465](#)
- __fp_parse_small_round:NN
..... [24425](#), [24543](#), [24543](#), [24582](#)
- __fp_parse_small_trailing:wwNN .
... [1069](#), [24411](#), [24420](#), [24420](#), [24497](#)
- __fp_parse_strim_end:w
..... [24369](#), [24375](#), [24379](#)
- __fp_parse_strim_zeros:N
..... [1067](#), [1079](#),
[24350](#), [24369](#), [24369](#), [24373](#), [24759](#)
- __fp_parse_trim_end:w
..... [24343](#), [24353](#), [24358](#)
- __fp_parse_trim_zeros:N
..... [24261](#), [24343](#), [24343](#), [24346](#)
- __fp_parse_unary_function:NNN ..
..... [24851](#),
[24851](#), [25973](#), [25975](#), [25977](#), [25979](#),
[27238](#), [27240](#), [27242](#), [27991](#), [27997](#)
- __fp_parse_word:Nw
..... [1065](#), [24269](#), [24286](#), [24286](#)
- __fp_parse_word_abs:N . [25972](#), [25972](#)
- __fp_parse_word_acos:N [27983](#)
- __fp_parse_word_acosd:N [27983](#)
- __fp_parse_word_acot:N [28002](#), [28002](#)
- __fp_parse_word_acotd:N [28002](#), [28004](#)
- __fp_parse_word_acsc:N [27983](#)
- __fp_parse_word_acscd:N [27983](#)
- __fp_parse_word_asec:N [27983](#)
- __fp_parse_word_asecd:N [27983](#)
- __fp_parse_word_asin:N [27983](#)
- __fp_parse_word_asind:N [27983](#)
- __fp_parse_word_atan:N [28002](#), [28006](#)
- __fp_parse_word_atand:N [28002](#), [28008](#)

- __fp_parse_word_bp:N [24822](#)
- __fp_parse_word_cc:N [24822](#)
- __fp_parse_word_ceil:N [23771](#), [23775](#)
- __fp_parse_word_cm:N [24822](#)
- __fp_parse_word_cos:N [27983](#)
- __fp_parse_word_cosd:N [27983](#)
- __fp_parse_word_cot:N [27983](#)
- __fp_parse_word_cotd:N [27983](#)
- __fp_parse_word_csc:N [27983](#)
- __fp_parse_word_cscd:N [27983](#)
- __fp_parse_word_dd:N [24822](#)
- __fp_parse_word_deg:N [24808](#)
- __fp_parse_word_em:N [24841](#)
- __fp_parse_word_ex:N [24841](#)
- __fp_parse_word_exp:N . [27237](#), [27237](#)
- __fp_parse_word_fact:N [27237](#), [27241](#)
- __fp_parse_word_false:N [24808](#)
- __fp_parse_word_floor:N [23771](#), [23773](#)
- __fp_parse_word_in:N [24822](#)
- __fp_parse_word_inf:N
..... [24808](#), [24819](#), [24820](#)
- __fp_parse_word_ln:N . [27237](#), [27239](#)
- __fp_parse_word_logb:N [25972](#), [25974](#)
- __fp_parse_word_max:N . [25543](#), [25543](#)
- __fp_parse_word_min:N . [25543](#), [25545](#)
- __fp_parse_word_mm:N [24822](#)
- __fp_parse_word_nan:N . [24808](#), [24821](#)
- __fp_parse_word_nc:N [24822](#)
- __fp_parse_word_nd:N [24822](#)
- __fp_parse_word_pc:N [24822](#)
- __fp_parse_word_pi:N [24808](#)
- __fp_parse_word_pt:N [24822](#)
- __fp_parse_word_rand:N [29169](#), [29169](#)
- __fp_parse_word_randint:N
..... [29169](#), [29171](#)
- __fp_parse_word_round:N [23777](#), [23777](#)
- __fp_parse_word_sec:N [27983](#)
- __fp_parse_word_sec_d:N [27983](#)
- __fp_parse_word_sign:N [25972](#), [25976](#)
- __fp_parse_word_sin:N [27983](#)
- __fp_parse_word_sind:N [27983](#)
- __fp_parse_word_sp:N [24822](#)
- __fp_parse_word_sqrt:N [25972](#), [25978](#)
- __fp_parse_word_tan:N [27983](#)
- __fp_parse_word_tand:N [27983](#)
- __fp_parse_word_true:N [24808](#)
- __fp_parse_word_trunc:N [23771](#), [23771](#)
- __fp_parse_zero:
... [1067](#), [24365](#), [24385](#), [24389](#), [24389](#)
- __fp_pow_B:wwN [27773](#), [27808](#)
- __fp_pow_C_neg:w [27811](#), [27828](#)
- __fp_pow_C_overflow:w
..... [27816](#), [27823](#), [27844](#)
- __fp_pow_C_pack:w [27830](#), [27838](#), [27849](#)
- __fp_pow_C_pos:w [27814](#), [27833](#)
- __fp_pow_C_pos_loop:wN
..... [27834](#), [27835](#), [27842](#)
- __fp_pow_exponent:Nwnnnnw
..... [27779](#), [27782](#), [27787](#)
- __fp_pow_exponent:wnN . [27771](#), [27776](#)
- __fp_pow_neg:www
..... [1180](#), [27685](#), [27855](#), [27855](#)
- __fp_pow_neg_aux:wNN
..... [1180](#), [27855](#), [27858](#), [27870](#)
- __fp_pow_neg_case:w
..... [27857](#), [27878](#), [27878](#)
- __fp_pow_neg_case_aux:nnnnn ...
..... [27878](#), [27882](#), [27888](#)
- __fp_pow_neg_case_aux:Nnnw
..... [1181](#), [27878](#), [27894](#), [27898](#)
- __fp_pow_normal_o:ww
..... [1176](#), [27690](#), [27722](#), [27722](#)
- __fp_pow_npos_aux:NNnww
..... [27756](#), [27760](#), [27766](#), [27766](#)
- __fp_pow_npos_o:Nww
..... [1177](#), [27733](#), [27750](#), [27750](#)
- __fp_pow_zero_or_inf:ww
..... [1176](#), [27692](#), [27699](#), [27699](#)
- \c__fp_prec_and_int ... [24060](#), [25123](#)
- \c__fp_prec_colon_int
..... [24060](#), [25181](#), [25940](#)
- \c__fp_prec_comma_int
..... [1080](#), [24060](#), [24134](#),
[24767](#), [24795](#), [25061](#), [25066](#), [25075](#)
- \c__fp_prec_comp_int
..... [24060](#), [25209](#), [25252](#)
- \c__fp_prec_end_int .. [1083](#), [1087](#),
[24060](#), [24136](#), [24874](#), [24887](#), [25044](#)
- \c__fp_prec_func_int
... [1080](#), [24060](#), [24766](#), [24858](#), [24867](#)
- \c__fp_prec_hat_int ... [24060](#), [25111](#)
- \c__fp_prec_hatii_int . [24060](#), [25111](#)
- \c__fp_prec_int
[23236](#), [23469](#), [23530](#), [23557](#), [23998](#),
[27525](#), [27890](#), [27893](#), [28993](#), [28995](#),
[29001](#), [29052](#), [29224](#), [29263](#), [29314](#)
- \c__fp_prec_juxt_int .. [24060](#), [25113](#)
- \c__fp_prec_not_int
..... [1079](#), [24060](#), [24751](#), [24752](#)
- \c__fp_prec_or_int [24060](#), [25125](#)
- \c__fp_prec_plus_int
..... [1054](#), [24060](#), [25119](#), [25121](#)
- \c__fp_prec_quest_int
..... [24060](#), [25164](#), [25179](#)
- \c__fp_prec_times_int
..... [24060](#), [25115](#), [25117](#)
- \c__fp_prec_tuple_int
... [1080](#), [24060](#), [24135](#), [24769](#), [24797](#)

__fp_rand_myriads:n
 1223, 1224, 29179, 29179, 29196, 29282
 __fp_rand_myriads_get:w
 29179, 29184, 29189
 __fp_rand_myriads_loop:w
 29179, 29180, 29181, 29187
 __fp_rand_o:Nw . 29170, 29190, 29190
 __fp_rand_o:w .. 29190, 29194, 29204
 __fp_randinat_wide_aux:w 29352
 __fp_randinat_wide_auxii:w .. 29352
 __fp_randint:n . 29414, 29417, 29419
 __fp_randint:ww
 .. 29320, 29324, 29329, 29334, 29424
 __fp_randint_auxi_o:ww
 29211, 29238, 29246
 __fp_randint_auxii:wn
 29211, 29249, 29250, 29252
 __fp_randint_auxiii_o:ww
 29211, 29250, 29274
 __fp_randint_auxiv_o:ww
 29211, 29285, 29289
 __fp_randint_auxv_o:w
 29211, 29287, 29297, 29299
 __fp_randint_badarg:w
 .. 1224, 29211, 29218, 29234, 29235
 __fp_randint_default:w
 29211, 29215, 29217
 __fp_randint_o:Nw 29172, 29211, 29211
 __fp_randint_o:w 29211, 29215, 29231
 __fp_randint_split_aux:w
 29352, 29375, 29381
 __fp_randint_split_o:Nw
 1227, 29352,
 29357, 29360, 29363, 29365, 29370
 __fp_randint_wide_aux:w
 1227, 29355, 29386
 __fp_randint_wide_auxii:w
 29388, 29397
 __fp_reverse_args:Nww
 1208, 1209, 23212, 23212,
 28583, 28658, 28771, 28837, 29308
 __fp_round:NNN
 1043, 1045, 1124, 1140,
 23787, 23854, 23857, 26104, 26115,
 26359, 26371, 26513, 26524, 26708
 __fp_round:Nwn
 .. 23915, 23968, 23970, 23985, 29100
 __fp_round:Nww
 23916, 23937, 23968, 23968
 __fp_round:Nwww . 23917, 23931, 23931
 __fp_round_aux_o:Nw
 23904, 23908, 23910
 __fp_round_digit:Nw
 1031, 1045, 1123, 1124,
 1140, 23487, 23871, 23871, 26118,
 26261, 26362, 26374, 26527, 26713
 __fp_round_name_from_cs:N 23907,
 23927, 23953, 23957, 23957, 23980
 __fp_round_neg:NNN
 1043, 1046, 1120,
 23882, 23903, 26223, 26238, 26256
 __fp_round_no_arg_o:Nw
 23914, 23921, 23921
 __fp_round_normal:NnnwNnn
 23968, 23999, 24001
 __fp_round_normal:NNwNnn
 23968, 24003, 24023
 __fp_round_normal:NwNNnw
 23968, 23988, 23996
 __fp_round_normal_end:wwNnn ...
 23968, 24031, 24034
 __fp_round_o:Nw 23772,
 23774, 23776, 23780, 23904, 23904
 __fp_round_pack:Nw
 23968, 24007, 24021
 __fp_round_return_one:
 1043, 23787, 23793,
 23803, 23811, 23815, 23824, 23828,
 23837, 23844, 23848, 23886, 23897
 __fp_round_s:NNNw 1043,
 1045, 1073, 23855, 23855, 24547, 24565
 __fp_round_special:NwwNnn
 23968, 24026, 24039
 __fp_round_special_aux:Nw
 23968, 24045, 24052
 __fp_round_to_nearest:NNN 1046,
 1047, 23780, 23783, 23787, 23808,
 23854, 23891, 23923, 23933, 29100
 __fp_round_to_nearest_neg:NNN ..
 23882, 23891, 23903
 __fp_round_to_nearest_ninf:NNN .
 1047, 23787, 23821, 23902
 __fp_round_to_nearest_ninf_-
 neg:NNN 23882, 23892
 __fp_round_to_nearest_pinf:NNN .
 1047, 23787, 23841, 23893
 __fp_round_to_nearest_pinf_-
 neg:NNN 23882, 23901
 __fp_round_to_nearest_zero:NNN .
 1047, 23787, 23834
 __fp_round_to_nearest_zero_-
 neg:NNN 23882, 23894
 __fp_round_to_ninf:NNN
 .. 23774, 23787, 23789, 23890, 23961
 __fp_round_to_ninf_neg:NNN
 23882, 23882
 __fp_round_to_pinf:NNN
 .. 23776, 23787, 23799, 23882, 23963

- _fp_round_to_pinf_neg:NNN [23882](#), [23890](#)
- _fp_round_to_zero:NNN [23772](#), [23787](#), [23798](#), [23959](#)
- _fp_round_to_zero_neg:NNN [23882](#), [23883](#)
- _fp_rroot:www . . . [23213](#), [23213](#), [28704](#)
- _fp_sanitize:Nw [1115](#), [1118](#), [1123](#), [1126](#), [1134](#), [1182](#), [1198](#), [1205](#), [1224](#), [23271](#), [23271](#), [23283](#), [24037](#), [24055](#), [26047](#), [26141](#), [26313](#), [26394](#), [26542](#), [27268](#), [27511](#), [27752](#), [27944](#), [28545](#), [28589](#), [28716](#), [29206](#), [29301](#)
- _fp_sanitize:wN [1064](#), [1068](#), [23271](#), [23283](#), [24260](#), [24758](#)
- _fp_sanitize_zero:w [23271](#), [23279](#), [23284](#)
- _fp_sec_o:w [28055](#), [28055](#)
- _fp_set_function:Nnnn [1239](#), [29883](#), [29885](#), [29888](#)
- _fp_set_sign_o:w [24751](#), [25973](#), [26744](#), [26745](#), [26745](#), [26767](#)
- _fp_set_variable:nn [29767](#), [29770](#), [29772](#)
- _fp_show:NN [25453](#), [25453](#), [25455](#), [25457](#)
- _fp_show_validate:n [25460](#), [25463](#), [25463](#)
- _fp_show_validate:nn [25463](#), [25465](#), [25474](#), [25476](#)
- _fp_show_validate:w [25463](#), [25484](#), [25501](#)
- _fp_show_validate_aux:n [25463](#), [25472](#), [25509](#), [25516](#), [25524](#), [25525](#)
- _fp_sign_aux_o:w [26733](#), [26737](#), [26738](#), [26743](#)
- _fp_sign_o:w . . . [25977](#), [26733](#), [26733](#)
- _fp_sin_o:w [1035](#), [1078](#), [1207](#), [28010](#), [28010](#)
- _fp_sin_series_aux_o:NNwww [28497](#), [28501](#), [28512](#)
- _fp_sin_series_o:NNwww [1185](#), [1199](#), [28016](#), [28031](#), [28046](#), [28061](#), [28497](#), [28497](#)
- _fp_small_int:wTF [1181](#), [23539](#), [23539](#), [23970](#), [27931](#)
- _fp_small_int_normal:NnwTF [23539](#), [23543](#), [23555](#)
- _fp_small_int_test:NnnwNwTF . [23539](#)
- _fp_small_int_test:NnnwNw [23558](#), [23561](#)
- _fp_small_int_true:wTF [23539](#), [23542](#), [23547](#), [23554](#), [23564](#)
- _fp_sqrt_auxi_o:NNNNwnnN [26564](#), [26572](#), [26572](#)
- _fp_sqrt_auxii_o:NnnnnnnnN [1136](#), [1138](#), [26574](#), [26578](#), [26578](#), [26658](#), [26670](#)
- _fp_sqrt_auxiii_o:wnnnnnnnn [26575](#), [26613](#), [26613](#), [26659](#)
- _fp_sqrt_auxiv_o:NNNNNw [26613](#), [26617](#), [26634](#)
- _fp_sqrt_auxix_o:wwnw [26647](#), [26649](#), [26656](#)
- _fp_sqrt_auxv_o:NNNNNw [26613](#), [26621](#), [26636](#)
- _fp_sqrt_auxvi_o:NNNNNw [26613](#), [26625](#), [26638](#)
- _fp_sqrt_auxvii_o:NNNNNw [26613](#), [26628](#), [26640](#)
- _fp_sqrt_auxviii_o:nnnnnnn [26635](#), [26637](#), [26639](#), [26645](#), [26647](#), [26647](#)
- _fp_sqrt_auxx_o:Nnnnnnnn [26643](#), [26661](#), [26661](#)
- _fp_sqrt_auxxi_o:wwnnN [26661](#), [26663](#), [26668](#)
- _fp_sqrt_auxxii_o:nnnnnnnnw [26671](#), [26675](#), [26675](#)
- _fp_sqrt_auxxiii_o:w [26675](#), [26682](#), [26695](#)
- _fp_sqrt_auxxiv_o:wnnnnnnnN [26687](#), [26690](#), [26698](#), [26700](#), [26700](#)
- _fp_sqrt_Newton_o:wwn . . . [1135](#), [26549](#), [26560](#), [26561](#), [26561](#), [26568](#)
- _fp_sqrt_npos_auxi_o:wwnnN [26540](#), [26546](#), [26551](#)
- _fp_sqrt_npos_auxii_o:wNNNNNNNN [26540](#), [26555](#), [26559](#)
- _fp_sqrt_npos_o:w [26537](#), [26540](#), [26540](#)
- _fp_sqrt_o:w . . . [25979](#), [26530](#), [26530](#)
- _fp_step:NNnnnn [25810](#), [25813](#), [25820](#), [25829](#)
- _fp_step:NnnnnN . . . [1106](#), [25750](#), [25776](#), [25777](#), [25793](#), [25804](#), [25809](#)
- _fp_step:wwnN . [25750](#), [25752](#), [25758](#)
- _fp_step_fp:wwnN [25750](#), [25763](#), [25771](#)
- _fp_str_if_eq:nn . [23576](#), [23576](#), [24672](#), [24684](#), [24970](#), [25012](#), [27725](#)
- _fp_sub_back_far_o:NnnwnnnnN [1119](#), [26150](#), [26196](#), [26196](#)
- _fp_sub_back_near_after:wNNNNw [26156](#), [26158](#), [26165](#), [26234](#)
- _fp_sub_back_near_o:nnnnnnnnN [1118](#), [26146](#), [26156](#), [26156](#)

- __fp_sub_back_near_pack:NNNNNNw
..... [26156](#), [26160](#), [26163](#), [26236](#)
- __fp_sub_back_not_far_o:wwwNN .
..... [26211](#), [26231](#), [26231](#)
- __fp_sub_back_quite_far_ii:NN ..
..... [26215](#), [26217](#), [26221](#)
- __fp_sub_back_quite_far_o:wwNN .
..... [26209](#), [26215](#), [26215](#)
- __fp_sub_back_shift:wnnnn
..... [1119](#), [26168](#), [26172](#), [26172](#)
- __fp_sub_back_shift_ii:ww
..... [26172](#), [26174](#), [26177](#)
- __fp_sub_back_shift_iii:NNNNNNNNw
..... [26172](#), [26182](#), [26185](#), [26194](#)
- __fp_sub_back_shift_iv:nnnnw ...
..... [26172](#), [26189](#), [26195](#)
- __fp_sub_back_very_far_ii_-
o:nnNwwNN [26243](#), [26246](#), [26250](#)
- __fp_sub_back_very_far_o:wwwNN
..... [26210](#), [26243](#), [26243](#)
- __fp_sub_eq_o:Nnwnw
..... [26121](#), [26124](#), [26132](#)
- __fp_sub_npos_i_o:Nnwnw
... [1117](#), [26126](#), [26135](#), [26139](#), [26139](#)
- __fp_sub_npos_ii_o:Nnwnw
..... [26121](#), [26128](#), [26133](#)
- __fp_sub_npos_o:NnwNw
..... [1117](#), [26041](#), [26121](#), [26121](#)
- __fp_symbolic_&o:ww [29549](#)
- __fp_symbolic_&symbolic_o:ww [29549](#)
- __fp_symbolic*_o:ww [29549](#)
- __fp_symbolic*_symbolic_o:ww [29549](#)
- __fp_symbolic+_o:ww [29549](#)
- __fp_symbolic+_symbolic_o:ww [29549](#)
- __fp_symbolic-_o:ww [29549](#)
- __fp_symbolic-_symbolic_o:ww [29549](#)
- __fp_symbolic/_o:ww [29549](#)
- __fp_symbolic/_symbolic_o:ww [29549](#)
- __fp_symbolic^_o:ww [29549](#)
- __fp_symbolic^symbolic_o:ww [29549](#)
- __fp_symbolic_acos_o:w [29569](#)
- __fp_symbolic_acsc_o:w [29569](#)
- __fp_symbolic_asec_o:w [29569](#)
- __fp_symbolic_asin_o:w [29569](#)
- __fp_symbolic_binary_o:Nww
..... [29543](#), [29543](#), [29553](#)
- __fp_symbolic_binary_to_tl:Nww .
..... [29617](#), [29623](#), [29637](#)
- __fp_symbolic_chk:w
..... [1232](#), [25469](#), [25495](#), [25519](#),
[25523](#), [29497](#), [29497](#), [29502](#), [29515](#),
[29533](#), [29546](#), [29566](#), [29618](#), [29690](#),
[29695](#), [29711](#), [29870](#), [29908](#), [29917](#)
- __fp_symbolic_convert:wnnN
..... [29581](#), [29585](#), [29593](#)
- __fp_symbolic_cos_o:w [29569](#)
- __fp_symbolic_cot_o:w [29569](#)
- __fp_symbolic_cs_arg_to_fn:NN ..
..... [29599](#), [29599](#), [29633](#)
- __fp_symbolic_csc_o:w [29569](#)
- __fp_symbolic_exp_o:w [29569](#)
- \l__fp_symbolic_flag [29767](#)
- \l__fp_symbolic_fp .. [1240](#), [29495](#),
[29778](#), [29782](#), [29788](#), [29925](#), [29929](#)
- __fp_symbolic_function_to_tl:Nw
..... [29617](#), [29624](#), [29646](#)
- __fp_symbolic_ln_o:w [29569](#)
- __fp_symbolic_not_o:w [29569](#)
- __fp_symbolic_op_arg_to_fn:nN ..
..... [29599](#), [29601](#), [29604](#)
- __fp_symbolic_sec_o:w [29569](#)
- __fp_symbolic_set_sign_o:w .. [29569](#)
- __fp_symbolic_show_validate:w ..
..... [25463](#), [25494](#), [25518](#)
- __fp_symbolic_sin_o:w [29569](#)
- __fp_symbolic_tan_o:w [29569](#)
- __fp_symbolic_to_decimal:w .. [29581](#)
- __fp_symbolic_to_int:w [29581](#)
- __fp_symbolic_to_scientific:w [29581](#)
- __fp_symbolic_to_tl:w . [29617](#), [29617](#)
- __fp_symbolic_unary_o:NNw
..... [29563](#), [29563](#), [29577](#)
- __fp_symbolic_unary_to_tl:NNw ..
..... [29617](#), [29622](#), [29629](#)
- __fp_symbolic_|_o:ww [29549](#)
- __fp_symbolic_|_symbolic_o:ww [29549](#)
- __fp_tan_o:w [28070](#), [28070](#)
- __fp_tan_series_aux_o:Nnwww ...
..... [28551](#), [28555](#), [28564](#)
- __fp_tan_series_o:NNwww
... [1187](#), [28077](#), [28092](#), [28551](#), [28551](#)
- __fp_ternary:NwN
..... [1100](#), [25179](#), [25924](#), [25924](#)
- __fp_ternary_auxi:NwN
... [1100](#), [1110](#), [25924](#), [25933](#), [25953](#)
- __fp_ternary_auxii:NwN
[1100](#), [1110](#), [25181](#), [25924](#), [25931](#), [25961](#)
- __fp_tmp:w
. [1031](#), [1088](#), [23481](#), [23491](#), [23492](#),
[23493](#), [23494](#), [23495](#), [23496](#), [23497](#),
[23498](#), [23499](#), [23500](#), [23501](#), [23502](#),
[23503](#), [23504](#), [23505](#), [23506](#), [23582](#),
[23584](#), [24078](#), [24090](#), [24091](#), [24092](#),
[24093](#), [24094](#), [24095](#), [24096](#), [24154](#),
[24176](#), [24734](#), [24751](#), [24752](#), [24808](#),
[24813](#), [24814](#), [24815](#), [24816](#), [24817](#),
[24818](#), [24822](#), [24830](#), [24831](#), [24832](#),

- 24833, 24834, 24835, 24836, 24837,
- 24838, 24839, 24840, 25040, 25056,
- 25057, 25080, 25092, 25110, 25112,
- 25114, 25116, 25118, 25120, 25122,
- 25124, 25128, 25142, 25143, 25158,
- 25159, 25160, 25178, 25180, 26777,
- 26791, 26792, 29549, 29562, 29581,
- 29590, 29591, 29592, 29708, 29719,
- 29725, 29729, 29844, 29850, 29856,
- 29860, 29928, 29933, 29937, 29941
- _fp_to_decimal:w 28955,
- 28965, 28965, 29082, 29099, 30102
- _fp_to_decimal_dispatch:w
- . 1212, 1215, 1216, 25803, 28945,
- 28949, 28952, 28952, 28964, 29590
- _fp_to_decimal_huge:wnnnn 28965, 29000, 29022
- _fp_to_decimal_large:Nnnw
- 28965, 28996, 29013
- _fp_to_decimal_normal:wnnnnn ..
- 28965, 28970, 28988, 29053
- _fp_to_decimal_recover:w 28952, 28955, 28958
- _fp_to_dim:w .. 29067, 29077, 29082
- _fp_to_dim_dispatch:w 1215, 29067, 29068, 29072, 29075
- _fp_to_dim_recover:w 29067, 29077, 29080
- _fp_to_int:w ... 1216, 29092, 29097
- _fp_to_int_dispatch:w 29083, 29083, 29087, 29090, 29591
- _fp_to_int_recover:w 29083, 29092, 29095
- _fp_to_scientific:w 1213, 28901, 28911, 28911
- _fp_to_scientific_dispatch:w .. 1211, 1215, 28891, 28895, 28898, 28898, 28910, 29592
- _fp_to_scientific_normal:wnnnnn 28911, 28916, 28934
- _fp_to_scientific_normal:wNw .. 28911, 28937, 28942
- _fp_to_scientific_recover:w ... 28898, 28901, 28904
- _fp_to_tl:w 29031, 29039, 29039, 30110
- _fp_to_tl_dispatch:w 1210, 1214, 29023, 29027, 29030, 29030, 29038, 29163, 29501, 29634, 29641, 29643
- _fp_to_tl_normal:nnnnn 29039, 29044, 29049
- _fp_to_tl_recover:w 29030, 29031, 29032
- _fp_to_tl_scientific:wnnnnn ... 29039, 29054, 29057
- _fp_to_tl_scientific:wNw 29039, 29060, 29065
- \c_fp_trailing_shift_int 23426, 26809, 26831, 26904, 27804, 28443, 28480
- _fp_trap_division_by_zero_set:N 23654, 23655, 23657, 23659, 23660
- _fp_trap_division_by_zero_set_error: 23654, 23654
- _fp_trap_division_by_zero_set_flag: 23654, 23656
- _fp_trap_division_by_zero_set_none: 23654, 23658
- _fp_trap_invalid_operation_set:N 23620, 23621, 23623, 23625, 23626
- _fp_trap_invalid_operation_set_error: 23620, 23620
- _fp_trap_invalid_operation_set_flag: 23620, 23622
- _fp_trap_invalid_operation_set_none: 23620, 23624
- _fp_trap_overflow_set:N 23680, 23681, 23683, 23685, 23686
- _fp_trap_overflow_set:NnNn ... 23680, 23687, 23695, 23696
- _fp_trap_overflow_set_error: .. 23680, 23680
- _fp_trap_overflow_set_flag: ... 23680, 23682
- _fp_trap_overflow_set_none: ... 23680, 23684
- _fp_trap_underflow_set:N 23680, 23689, 23691, 23693, 23694
- _fp_trap_underflow_set_error: . 23680, 23688
- _fp_trap_underflow_set_flag: .. 23680, 23690
- _fp_trap_underflow_set_none: .. 23680, 23692
- _fp_trig:NNNNwn 28016, 28031, 28046, 28061, 28076, 28091, 28108, 28108
- \c_fp_trig_intarray 1195, 28169, 28399, 28402, 28405, 28408, 28411, 28414, 28417, 28420, 28423
- _fp_trig_large:ww 28116, 28383, 28383
- _fp_trig_large_auxi:w 28383, 28385, 28390

__fp_trig_large_auxii:w
 [1195](#), [28383](#), [28393](#), [28427](#)
 __fp_trig_large_auxiii:w . [1195](#),
 [28383](#), [28401](#), [28404](#), [28407](#), [28410](#),
 [28413](#), [28416](#), [28419](#), [28422](#), [28435](#)
 __fp_trig_large_auxix:Nw
 [28456](#), [28466](#), [28469](#), [28473](#)
 __fp_trig_large_auxv:www
 [28433](#), [28436](#), [28436](#)
 __fp_trig_large_auxvi:wnnnnnnn
 [28436](#), [28442](#), [28447](#)
 __fp_trig_large_auxvii:w
 [28439](#), [28456](#), [28456](#)
 __fp_trig_large_auxviii:w ... [28456](#)
 __fp_trig_large_auxviii:ww
 [28458](#), [28462](#)
 __fp_trig_large_auxx:wNNNNN ...
 [28456](#), [28479](#), [28483](#)
 __fp_trig_large_auxxi:w
 [28456](#), [28476](#), [28490](#)
 __fp_trig_large_pack:NNNNw ...
 [28436](#), [28449](#), [28454](#), [28485](#)
 __fp_trig_small:ww .. [1189](#), [1197](#),
 [28118](#), [28122](#), [28122](#), [28128](#), [28495](#)
 __fp_trigd_large:ww
 [28116](#), [28130](#), [28130](#)
 __fp_trigd_large_auxi:nnnwNNNN
 [28130](#), [28136](#), [28142](#)
 __fp_trigd_large_auxii:wNw
 [28130](#), [28144](#), [28150](#)
 __fp_trigd_large_auxiii:www ...
 [28130](#), [28153](#), [28157](#)
 __fp_trigd_small:ww
 ... [1189](#), [28118](#), [28124](#), [28124](#), [28167](#)
 __fp_trim_zeros:w
 ... [28882](#), [28882](#), [29006](#), [29015](#), [29066](#)
 __fp_trim_zeros_dot:w
 [28882](#), [28885](#), [28888](#)
 __fp_trim_zeros_end:w
 [28882](#), [28888](#), [28889](#)
 __fp_trim_zeros_loop:w
 [28882](#), [28884](#), [28885](#), [28887](#)
 __fp_tuple_ [25914](#), [25915](#), [25918](#), [25919](#)
 __fp_tuple_&o:ww [25897](#)
 __fp_tuple_&tuple_o:ww [25897](#)
 __fp_tuple_*_o:ww [26771](#)
 __fp_tuple+_tuple_o:ww [26777](#)
 __fp_tuple-_tuple_o:ww [26777](#)
 __fp_tuple/_o:ww [26771](#)
 __fp_tuple_chk:w
 [1024](#), [23329](#), [23330](#),
 [23331](#), [23333](#), [23335](#), [23336](#), [23413](#),
 [23416](#), [25089](#), [25301](#), [25316](#), [25341](#),
 [25344](#), [25360](#), [25361](#), [25364](#), [25468](#),
 [25490](#), [25513](#), [25516](#), [25648](#), [25649](#),
 [26780](#), [26781](#), [26787](#), [26788](#), [28861](#)
 __fp_tuple_compare_back:ww
 [25645](#), [25646](#)
 __fp_tuple_compare_back_loop:w .
 [25645](#), [25655](#), [25663](#), [25672](#)
 __fp_tuple_compare_back_-
 tuple:ww [25645](#), [25647](#)
 __fp_tuple_convert:Nw
 .. [28861](#), [28861](#), [28910](#), [28964](#), [29038](#)
 __fp_tuple_convert_end:w
 [28861](#), [28866](#), [28870](#), [28880](#)
 __fp_tuple_convert_loop:nNw ...
 [28861](#), [28869](#), [28874](#), [28877](#)
 __fp_tuple_count:w
 [23334](#), [23335](#), [23336](#)
 __fp_tuple_count_loop:Nw
 [23334](#), [23339](#), [23343](#), [23344](#)
 __fp_tuple_map_loop_o:nw
 [25341](#), [25347](#), [25352](#), [25357](#)
 __fp_tuple_map_o:nw [25341](#),
 [25341](#), [26764](#), [26772](#), [26774](#), [26776](#)
 __fp_tuple_maphread_loop_o:nw .
 [25359](#), [25367](#), [25373](#), [25379](#)
 __fp_tuple_maphread_o:nww
 [25359](#), [25359](#), [26785](#)
 __fp_tuple_not_o:w ... [25888](#), [25896](#)
 __fp_tuple_set_sign_aux_o:Nnw ..
 [26755](#), [26758](#), [26763](#)
 __fp_tuple_set_sign_aux_o:w ...
 [26755](#), [26764](#), [26765](#)
 __fp_tuple_set_sign_o:w [26755](#), [26755](#)
 __fp_tuple_show_validate:w
 [25463](#), [25489](#), [25512](#)
 __fp_tuple_to_decimal:w [28952](#), [28963](#)
 __fp_tuple_to_scientific:w
 [28898](#), [28909](#)
 __fp_tuple_to_tl:w ... [29030](#), [29037](#)
 __fp_tuple|_o:ww [25897](#)
 __fp_tuple|_tuple_o:ww [25897](#)
 __fp_type_from_scan:N ... [1025](#),
 [23358](#), [23358](#), [24915](#), [24917](#), [24941](#),
 [24943](#), [24954](#), [24956](#), [25593](#), [25595](#),
 [25609](#), [25611](#), [29459](#), [29479](#), [29481](#)
 __fp_type_from_scan:w
 [23358](#), [23367](#), [23372](#)
 __fp_type_from_scan_other:N ...
 .. [23358](#), [23362](#), [23365](#), [23382](#), [23400](#)
 __fp_types_binary:Nww
 [1230](#), [1232](#), [29469](#), [29469](#), [29547](#), [29623](#)
 __fp_types_binary_auxi:Nww
 [29469](#), [29471](#), [29474](#)
 __fp_types_binary_auxii:NNww ...
 [29469](#), [29476](#), [29486](#)

- __fp_types_cs_to_op:N 29436, 29436,
29454, 29472, 29602, 29642, 29650
- __fp_types_cs_to_op_auxi:wwn 29436, 29438, 29447
- __fp_types_unary:NNw 1230, 1232, 29451, 29451, 29567, 29622
- __fp_types_unary_auxi:nNw 29451, 29453, 29456
- __fp_types_unary_auxii:NnNw 29451, 29458, 29463
- __fp_underflow:w 1023,
1037, 1039, 23278, 23711, 23717, 27508
- __fp_use_i:ww 1153, 1207, 23214, 23214, 27021, 28790
- __fp_use_i:www 23214, 23215
- __fp_use_i_delimit_by_s_stop:nw 23225,
23225, 25566, 25929, 29440, 29442
- __fp_use_i_until_s:nw 1197, 23209, 23210, 23258,
23268, 23531, 28160, 28438, 28444,
28475, 29224, 29295, 29952, 30048
- __fp_use_ii_until_s:nnw 23209, 23211, 23256, 23267
- __fp_use_none_stop_f:n 23206, 23206, 27186, 27187, 27188
- __fp_use_none_until_s:w 23209,
23209, 26566, 27864, 28785, 28788
- __fp_use_s:n 23207, 23207
- __fp_use_s:nn 23207, 23208
- __fp_variable_o:w 1232,
29679, 29679, 29691, 29696, 29712
- __fp_variable_set_parsing:Nn 29706, 29706, 29746, 29764, 29777
- __fp_variable_set_parsing_-
aux:NNn 29706, 29714, 29717
- __fp_zero_fp:N 23249, 23249, 23695, 24043
- __fp_l_o:ww 1100, 25897
- __fp_l_symbolic_o:ww 29549
- __fp_l_tuple_o:ww 25897
- fpararray commands:
 - \fpararray_count:N 283, 284, 30008,
30008, 30013, 30020, 30031, 30087
 - \fpararray_gset:Nnn 283, 1247, 30033, 30033, 30042
 - \fpararray_gzero:N 283, 30084, 30084, 30096
 - \fpararray_if_exist:N 30151, 30153
 - \fpararray_if_exist:NTF 284, 30151
 - \fpararray_if_exist_p:N 284, 30151
 - \fpararray_item:Nn 284, 1247, 30097, 30097, 30104
 - \fpararray_item_to_tl:Nn 284, 30097, 30105, 30112
 - \fpararray_new:Nn 283, 29981, 29981, 29993
 - \futurelet 238
- G**
- \gdef 239
- get commands:
 - get_luaadata 12083
 - \GetIdInfo 10, 11587
 - \gleaders 833
 - \glet 834
 - \global 105, 140, 240
 - \globaldefs 241
 - \glueexpr 495
 - \glueshrink 496
 - \glueshrinkorder 497
 - \gluestretch 498
 - \gluestretchorder 499
 - \gluetomu 500
 - \glyphdimensionsmode 835
- group commands:
 - \group_align_safe_begin/end: 449, 596
 - \group_align_safe_begin: 73, 589, 705,
710, 3584, 4019, 8391, 8618, 8620,
12508, 13072, 19287, 19724, 19745,
19777, 31723, 32138, 34069, 37286
 - \group_align_safe_end: 73, 705, 710, 3587, 4031,
8393, 8618, 8623, 12529, 13055,
19331, 19733, 19742, 19782, 19788,
31735, 32151, 34080, 37289, 37293
 - \group_begin: 13, 701, 1413,
1430, 1421, 1422, 2303, 2306, 2309,
2687, 2882, 3059, 3224, 3261, 3540,
3583, 3590, 3610, 3687, 3852, 4038,
4103, 4472, 4564, 4891, 5400, 5734,
5975, 6076, 6503, 6878, 7158, 7416,
7442, 7454, 7464, 7473, 7659, 7688,
7810, 8618, 8665, 8889, 8985, 9138,
9150, 9349, 9373, 9389, 9454, 10379,
10624, 10670, 10932, 11075, 11594,
12184, 12370, 12599, 12612, 13413,
13734, 13757, 14258, 14368, 14423,
14718, 14766, 14812, 14819, 15148,
15330, 16978, 17009, 19162, 19168,
19215, 19277, 19334, 19352, 19376,
19461, 19480, 19860, 21069, 21348,
21477, 21519, 22610, 25897, 29895,
30447, 30656, 30681, 30983, 31206,
31243, 31346, 31407, 31681, 33634,
33667, 34320, 34382, 34766, 36689,

- 36865, 37419, 37508, 37550, 39297,
39300, 39362, 39710, 39783, 39786
- `\c_group_begin_token` *114, 200, 208, 718, 898, 3652, 4195, 12916, 12956, 19311, 19334, 19358, 31469, 34809, 34815, 34829, 34835, 34913, 34919, 34934, 34940, 37345, 37346, 37353*
- `\group_end:`
13, 14, 462, 571, 828, 1260, 1264, 1413, 1421, 1423, 2303, 2306, 2312, 2696, 2885, 3062, 3228, 3270, 3479, 3553, 3588, 3609, 3633, 3694, 3876, 4025, 4125, 4484, 4578, 4924, 4932, 5413, 5738, 6035, 6083, 6090, 6098, 6507, 6508, 6915, 7222, 7421, 7449, 7537, 7682, 7729, 7811, 7812, 8625, 8684, 8906, 9013, 9142, 9154, 9368, 9381, 9400, 9600, 10385, 10628, 10699, 10955, 11093, 11597, 12187, 12392, 12442, 12602, 12616, 13431, 13739, 13762, 14268, 14383, 14426, 14731, 14778, 14837, 14899, 15329, 15461, 16990, 17019, 17024, 19170, 19177, 19276, 19281, 19351, 19355, 19383, 19479, 19528, 19884, 21085, 21474, 21500, 21559, 22624, 25921, 29940, 30451, 30660, 30714, 31216, 31217, 31311, 31381, 31423, 31701, 33660, 33693, 34324, 34627, 34772, 36690, 36870, 37423, 37513, 37565, 39316, 39373, 39781, 39800, 40020
- `\c_group_end_token` *200, 898, 3655, 19314, 19334, 19363, 31470, 34823, 34928, 37349, 37357*
- `\group_insert_after:N`
..... *14, 1427, 1427, 4480, 36697, 37349, 37350, 37383, 37667*
- `\group_log_list:` *14, 2315, 2317*
- `\group_show_list:` *14, 2315, 2315*
- groups commands:
- `.groups:n` *243, 22095*
- `\gtoksapp` *836*
- `\gtokspre` *837*
- H**
- `\H` *65, 32057, 34380, 34400, 34547, 34548, 34575, 34576*
- `\halign` *242*
- `\hangafter` *243*
- `\hangindent` *244*
- `\hbadness` *245*
- `\hbox` *246*
- hbox commands:
- `\hbox:n` *302, 306, 34780, 34780, 35007, 35304, 36473*
- `\hbox_gset:Nn` *306, 34782, 34787, 34793, 34974, 35097, 35141, 35161, 35181, 35198, 35219, 35248, 35259, 35316, 35527, 35962, 39539*
- `\hbox_gset:Nw`
307, 34806, 34812, 34819, 35600, 39541
- `\hbox_gset_end:`
..... *307, 34806, 34825, 35603*
- `\hbox_gset_to_wd:Nnn`
..... *307, 34794, 34799, 34805, 39540*
- `\hbox_gset_to_wd:Nnw`
..... *307, 34826, 34832, 34839, 39542*
- `\hbox_overlap_center:n`
..... *307, 34850, 34850*
- `\hbox_overlap_left:n` *307, 34850, 34852*
- `\hbox_overlap_right:n`
..... *307, 34850, 34854*
- `\hbox_set:Nn` *302, 306, 307, 321, 34782, 34782, 34792, 34971, 35003, 35004, 35091, 35138, 35158, 35178, 35195, 35216, 35245, 35253, 35277, 35313, 35326, 35334, 35342, 35351, 35360, 35377, 35385, 35393, 35399, 35412, 35514, 35959, 35982, 36239, 36326, 36605, 39458*
- `\hbox_set:Nw`
307, 34806, 34806, 34818, 35587, 39460
- `\hbox_set_end:`
..... *307, 34806, 34820, 34825, 35590*
- `\hbox_set_to_wd:Nnn`
..... *307, 34794, 34794, 34804, 39459*
- `\hbox_set_to_wd:Nnw`
..... *307, 34826, 34826, 34838, 39461*
- `\hbox_to_wd:nn` *306, 34840, 34840, 35295*
- `\hbox_to_zero:n` *306, 34840, 34845, 34851, 34853, 34855, 38700*
- `\hbox_unpack:N`
..... *307, 34856, 34856, 34858, 36243*
- `\hbox_unpack_drop:N`
..... *310, 34856, 34857, 34859*
- hcoffin commands:
- `\hcoffin_gset:Nn`
..... *316, 35510, 35523, 35535*
- `\hcoffin_gset:Nw`
..... *316, 35583, 35596, 35608*
- `\hcoffin_gset_end:`
..... *316, 35583, 35601, 35610*
- `\hcoffin_set:Nn`
..... *316, 317, 35510, 35510, 35522, 36477, 36484, 36522, 36557*
- `\hcoffin_set:Nw`
..... *316, 35583, 35583, 35595*

- \hcoffin_set_end:
..... 316, 35583, 35588, 35609
 - \hfi 1144
 - \hfil 247
 - \hfill 248
 - \hfilneg 249
 - \hfuzz 250
 - \hjcode 828
 - \hoffset 251
 - \holdinginserts 252
 - hook commands:
 - \hook_gput_code:nnm ... 30583, 30585
 - \hpack 829
 - \hrule 253
 - \hsize 254
 - \hskip 255
 - \hss 256
 - \ht 257
 - \Huge 34298
 - \huge 34302
 - \hyphenation 258
 - \hyphenationbounds 830
 - \hyphenationmin 831
 - \hyphenchar 259
 - \hyphenpenalty 260
 - \hyphenpenaltymode 832
- I
- \i 33658,
34355, 34456, 34458, 34460, 34462,
34513, 34516, 34519, 34522, 34593
 - \if 261
 - if commands:
 - \if:w ... 28, 29, 195, 387, 388, 420,
488, 692, 707, 708, 711, 720, 721,
738, 1392, 1398, 1804, 2173, 2174,
2774, 2777, 2778, 2779, 2780, 2795,
2796, 2797, 2798, 2799, 2800, 2801,
2802, 2803, 2867, 2868, 2870, 4083,
4719, 4748, 4802, 11118, 12560,
12570, 12663, 12970, 12990, 13005,
13553, 13560, 13565, 18153, 19645,
21091, 21106, 21107, 23972, 24345,
24349, 24371, 24464, 24496, 24515,
24581, 24595, 24612, 24633, 24672,
24684, 24970, 25012, 25132, 25147,
25551, 27725, 27755, 29236, 31247,
31256, 31272, 31453, 39151, 39163,
39165, 39246, 39247, 39248, 39249,
39263, 39264, 39274, 39275, 39276,
39277, 39278, 39279, 39280, 39281,
39282, 40049, 40051, 40055, 40057
 - \if_bool:N .. 72, 584, 1402, 8279, 8330
 - \if_box_empty:N
..... 314, 34718, 34720, 34730
 - \if_case:w 179, 745,
747, 786, 857, 1032, 1125, 1180,
1224, 2080, 3663, 3862, 4049, 4431,
4703, 5517, 5546, 5603, 6275, 6328,
6947, 6994, 7092, 7409, 7920, 7931,
10784, 13832, 13906, 14244, 15275,
17450, 17454, 18044, 18077, 19267,
23273, 23526, 23541, 23912, 23941,
25220, 25261, 25988, 26123, 26198,
26223, 26275, 26719, 26735, 26752,
27030, 27257, 27284, 27442, 27477,
27635, 27680, 27731, 27857, 27880,
27913, 27972, 28012, 28027, 28042,
28057, 28072, 28087, 28613, 28666,
28750, 28765, 28817, 28830, 28914,
28968, 29042, 29233, 30062, 30141
 - \if_catcode:w 29, 718, 719,
897, 912, 1392, 1400, 2914, 3652,
3655, 3810, 3812, 3814, 3816, 3818,
3820, 3822, 4050, 4051, 4195, 12911,
12954, 19290, 19293, 19296, 19299,
19302, 19305, 19308, 19311, 19314,
19317, 19358, 19363, 19368, 19373,
19380, 19387, 19392, 19397, 19402,
19407, 19412, 19419, 19446, 19755,
19814, 19819, 19866, 19867, 23172,
24100, 24305, 24623, 24670, 24969,
25011, 31411, 31412, 31454, 31469,
31470, 31471, 31472, 31473, 31474,
31475, 31476, 31477, 31500, 31503,
31506, 31509, 31512, 31515, 31518
 - \if_charcode:w
..... 29, 195, 455, 718, 719, 749,
912, 1392, 1399, 3722, 3746, 3795,
4091, 4128, 4130, 4640, 4650, 5161,
5716, 6884, 6887, 11378, 11387,
12897, 12947, 13990, 14224, 14888,
19424, 19816, 23529, 25564, 25927
 - \if_cs_exist:N ... 29, 1407, 1407,
1832, 1870, 2690, 19218, 19454, 19654
 - \if_cs_exist:w ... 29, 1407, 1408,
1435, 1842, 1855, 1884, 1899, 1924,
1935, 2068, 4236, 11110, 18298,
18323, 18334, 20258, 20305, 21610
 - \if_dim:w 238, 20746,
20746, 20834, 20846, 20869, 21040
 - \if_eof:w 99,
644, 10322, 10322, 10327, 10412, 10430
 - \if_false:
.. 28, 65, 208, 446, 467, 553, 565,
566, 569, 596, 665, 701, 706, 710,
717, 831, 848, 895, 936, 1392, 1393,

1859, 1865, 3575, 3641, 3690, 3693,
 4199, 4200, 4207, 4208, 4900, 4919,
 4920, 4929, 4994, 5037, 5051, 5055,
 5267, 5300, 5312, 5316, 5350, 5355,
 5363, 5398, 5405, 5410, 5458, 5695,
 5714, 5725, 5748, 5760, 5761, 5764,
 7174, 7191, 7528, 7567, 7575, 7582,
 7612, 7737, 7739, 7740, 7746, 8621,
 8624, 8890, 8898, 10763, 10803,
 10807, 10814, 10822, 11074, 11087,
 12112, 12116, 12371, 12378, 12524,
 12525, 12634, 12638, 12678, 12875,
 12880, 12971, 12984, 13002, 13006,
 13016, 13335, 13347, 13391, 13401,
 16855, 16858, 17097, 17102, 17664,
 19240, 19246, 19264, 20559, 20560,
 20561, 20562, 20597, 20598, 20599,
 20600, 20856, 21668, 21680, 30226
 \if_hbox:N ... 314, 34718, 34718, 34722
 \if_int_compare:w
 28, 179, 738, 848, 849, 1425, 1425,
 3149, 3206, 3235, 3277, 3288, 3291,
 3309, 3364, 3374, 3384, 3603, 3625,
 3699, 3732, 3740, 3763, 3787, 3843,
 3858, 3944, 3947, 4142, 4316, 4373,
 4379, 4380, 4387, 4391, 4397, 4398,
 4403, 4404, 4412, 4413, 4414, 4419,
 4450, 4451, 4700, 4721, 4722, 4723,
 4726, 4730, 4731, 4734, 4735, 4750,
 4751, 4754, 4758, 4759, 4762, 4823,
 4841, 4851, 4860, 4868, 4870, 4880,
 4883, 4911, 4998, 5110, 5174, 5179,
 5207, 5265, 5298, 5409, 5426, 5772,
 5805, 5836, 6222, 6293, 6319, 6380,
 6393, 6404, 6420, 6471, 6512, 6518,
 6524, 6688, 6689, 6716, 6743, 6842,
 6899, 7018, 7027, 7038, 7053, 7109,
 7118, 7170, 7187, 7210, 7476, 7505,
 7563, 7571, 7640, 10325, 10326,
 10770, 11394, 13161, 13170, 13219,
 13220, 13226, 13541, 13548, 13816,
 13871, 13872, 13878, 13890, 13906,
 14040, 14233, 14241, 14450, 14451,
 14452, 14457, 14458, 14482, 14534,
 14634, 14853, 14915, 14919, 14949,
 14952, 14968, 14972, 14993, 15073,
 15075, 15094, 15095, 15113, 15115,
 15169, 15172, 15173, 15291, 15292,
 15433, 15438, 17450, 17505, 17546,
 17547, 17644, 17697, 17699, 17701,
 17703, 17705, 17707, 17709, 17712,
 17720, 17853, 19192, 19193, 19194,
 19195, 19200, 19201, 19205, 19636,
 20885, 23028, 23031, 23075, 23139,
 23158, 23274, 23275, 23469, 23566,
 23792, 23802, 23810, 23823, 23836,
 23843, 23864, 23876, 23885, 23896,
 24005, 24010, 24082, 24112, 24265,
 24267, 24304, 24309, 24362, 24382,
 24409, 24423, 24458, 24485, 24513,
 24529, 24545, 24563, 24623, 24643,
 24659, 24743, 24766, 24795, 24797,
 24978, 24980, 25020, 25022, 25044,
 25061, 25066, 25096, 25164, 25209,
 25575, 25633, 25636, 25667, 25676,
 25679, 25684, 25685, 25688, 25691,
 25868, 25992, 26013, 26050, 26145,
 26199, 26200, 26203, 26206, 26276,
 26285, 26490, 26563, 26616, 26620,
 26624, 26642, 26677, 26678, 26679,
 26680, 26681, 26707, 27032, 27035,
 27129, 27222, 27270, 27286, 27413,
 27447, 27505, 27514, 27554, 27726,
 27737, 27755, 27778, 27810, 27813,
 27860, 27890, 27937, 27951, 28115,
 28159, 28617, 28655, 28664, 28700,
 28784, 28787, 29016, 29223, 29291,
 29292, 29293, 29303, 29331, 29336,
 29337, 29400, 29401, 29402, 29406,
 29421, 29426, 29949, 30016, 30020,
 30228, 30870, 30871, 30877, 31248,
 31442, 31486, 31607, 31614, 31631,
 31634, 32976, 32979, 32980, 32983,
 32984, 33005, 33008, 33011, 33014,
 33017, 33020, 33039, 33040, 33046,
 33049, 33052, 33055, 33058, 33061,
 33064, 33067, 33070, 33073, 33076,
 33079, 33082, 33085, 33088, 33117,
 33120, 33135, 33138, 33152, 33155,
 33158, 33161, 33177, 33180, 33183
 \if_int_odd:w 180,
 1200, 3868, 4440, 4831, 4839, 4849,
 5271, 5586, 17450, 17453, 17580,
 17758, 17766, 18268, 19191, 19199,
 19865, 23814, 23861, 23873, 25257,
 26259, 26545, 27901, 28465, 28504,
 28514, 28557, 28581, 28741, 29399
 \if_meaning:w 29, 463, 719, 812, 826,
 1109, 1287, 1392, 1401, 1653, 1679,
 1697, 1761, 1766, 1775, 1828, 1850,
 1866, 1874, 1894, 1910, 1945, 2098,
 2112, 2258, 2375, 2431, 2432, 2722,
 2745, 2754, 3005, 3017, 3018, 3159,
 3160, 3615, 3627, 3649, 3679, 3707,
 3808, 4012, 4052, 4053, 4153, 4192,
 4234, 4479, 4635, 4660, 4672, 4801,
 4822, 5158, 5160, 5593, 6257, 6428,
 6439, 6454, 6615, 6658, 6793, 7065,

- 7387, 7504, 7617, 8404, 8426, 10720,
11015, 12550, 12603, 12617, 12938,
13340, 13382, 13395, 13665, 13743,
13766, 13927, 13965, 14396, 15123,
15272, 15287, 15314, 15429, 16362,
16368, 16394, 16406, 16414, 16446,
16453, 16477, 16481, 16559, 16595,
16613, 16927, 16959, 17014, 17029,
17037, 17473, 17476, 17486, 17521,
17526, 17527, 17679, 18536, 18551,
18573, 18587, 19451, 19490, 19493,
19628, 19730, 19758, 19769, 19807,
19868, 20219, 20316, 20464, 20502,
20512, 20815, 20862, 21043, 23255,
23276, 23288, 23298, 23393, 23448,
23457, 23548, 23563, 23565, 23703,
23791, 23801, 23813, 23826, 23827,
23846, 23847, 23861, 23862, 23873,
23874, 23940, 23987, 24022, 24025,
24041, 24048, 24101, 24104, 24220,
24221, 24222, 24223, 24226, 24322,
24435, 24441, 24671, 24715, 24926,
24997, 25258, 25276, 25326, 25336,
25622, 25623, 25624, 25625, 25626,
25627, 25849, 25861, 25862, 25890,
25902, 25909, 25926, 25989, 26024,
26038, 26084, 26091, 26167, 26179,
26279, 26282, 26293, 26346, 26419,
26489, 26492, 26499, 26532, 26533,
26536, 26757, 27003, 27014, 27195,
27205, 27254, 27353, 27433, 27482,
27496, 27643, 27677, 27689, 27702,
27705, 27708, 27711, 27736, 27837,
27841, 27900, 27917, 27923, 28507,
28560, 28611, 28612, 28614, 28615,
28635, 28652, 28719, 28817, 28913,
28967, 29041, 29111, 29116, 29222,
29254, 29265, 29372, 29946, 30075,
30128, 30134, 31210, 31426, 32040
- `\if_mode_horizontal`:
..... 29, 1403, 1404, 8613
- `\if_mode_inner`: . 29, 1403, 1406, 8615
- `\if_mode_math`: .. 29, 1403, 1403, 8617
- `\if_mode_vertical`:
..... 29, 1403, 1405, 2385, 8611
- `\if_predicate:w`
..... 63, 65, 72, 8279, 8279,
8381, 8442, 8457, 8468, 8483, 8494
- `\if_true`:
.. 28, 66, 1392, 1392, 1887, 1893,
1903, 1909, 12110, 12114, 12996, 13002
- `\if_vbox:N` ... 314, 34718, 34719, 34724
- `\ifabsdim` 936
- `\ifabsnum` 937
- `\ifcase` 262
- `\ifcat` 263
- `\ifcondition` 838
- `\ifcsname` 366, 663, 501
- `\ifdbox` 1145
- `\ifddir` 1146
- `\ifdefined` 502
- `\ifdim` 264
- `\IfDocumentMetadataTF` 38711, 38712
- `\ifeof` 265
- `\iffalse` 266
- `\IfFileExists` 666
- `\iffontchar` 503
- `\ifhbox` 267
- `\ifhmode` 268
- `\ifincsname` 672
- `\ifinner` 269
- `\ifjfont` 1147
- `\ifmbox` 1148
- `\ifmdir` 1149
- `\ifmmode` 270
- `\ifnum` 10, 22, 52, 56, 271
- `\ifodd` 272
- `\ifpdfabsdim` 624
- `\ifpdfabsnum` 625
- `\ifpdfprimitive` 626
- `\ifprimitive` 775
- `\iftbox` 1150
- `\iftdir` 1152
- `\iftfont` 1151
- `\iftrue` 273
- `\ifvbox` 274
- `\ifvmode` 275
- `\ifvoid` 276
- `\ifx` 4, 8, 13, 17, 53, 54, 61, 277
- `\ifybox` 1153
- `\ifydir` 1154
- `\ignoreligaturesinfont` 938
- `\ignorespaces` 278
- `\IJ` 32065, 33649, 34345
- `\ij` 32065, 33649, 34357
- `\immediate` 68, 279
- `\immediateassigned` 839
- `\immediateassignment` 840
- `in` 280
- `\indent` 280
- `inf` 279
- `\infty` 24223, 24224
- inherit commands:
 `.inherit:n` 243, 22097
- `\inhibitglue` 1155
- `\inhibitxspcode` 1156
- `\initcatcodetable` 841

- initial commands:
- `.initial:n` [243](#), [22099](#)
 - `\input` [14](#), [281](#)
 - `\inputlineno` [282](#)
 - `\insert` [283](#)
 - `\insertht` [939](#)
 - `\insertpenalties` [284](#)
- int commands:
- `\int_abs:n`
[168](#), [842](#), [17479](#), [17479](#), [23075](#), [39895](#)
 - `\int_add:Nn` [169](#), [4420](#),
[5588](#), [6410](#), [6411](#), [6668](#), [6740](#), [10879](#),
[17610](#), [17610](#), [17618](#), [39453](#), [39834](#)
 - `\int_case:nn` [172](#), [857](#), [17726](#), [17741](#),
[17906](#), [17912](#), [30756](#), [32891](#), [32913](#),
[32918](#), [32930](#), [33376](#), [33391](#), [33467](#)
 - `\int_case:nnTF` [172](#),
[4170](#), [8245](#), [17367](#), [17726](#), [17726](#),
[17731](#), [17736](#), [18887](#), [24132](#), [28863](#)
 - `\int_compare:n` [17657](#)
 - `\int_compare:nNn` [17710](#)
 - `\int_compare:nNnTF` [170–173](#),
[263](#), [3136](#), [3991](#), [4458](#), [4470](#), [4623](#),
[4953](#), [4955](#), [5818](#), [6618](#), [6970](#), [7129](#),
[7529](#), [7722](#), [8262](#), [8694](#), [8700](#), [9178](#),
[10575](#), [10692](#), [11267](#), [11277](#), [11637](#),
[11676](#), [12373](#), [12401](#), [12416](#), [12424](#),
[13116](#), [13123](#), [13190](#), [13795](#), [13797](#),
[13806](#), [14049](#), [14054](#), [14064](#), [14067](#),
[14121](#), [14590](#), [14666](#), [15473](#), [16894](#),
[16895](#), [16897](#), [16899](#), [16972](#), [17155](#),
[17162](#), [17561](#), [17567](#), [17710](#), [17750](#),
[17802](#), [17810](#), [17819](#), [17825](#), [17837](#),
[17840](#), [17902](#), [17991](#), [17997](#), [18003](#),
[18023](#), [18177](#), [18196](#), [18198](#), [18240](#),
[18959](#), [18961](#), [18966](#), [18975](#), [18996](#),
[19013](#), [19030](#), [19951](#), [20094](#), [20112](#),
[21063](#), [21112](#), [21115](#), [22786](#), [23013](#),
[23018](#), [23025](#), [23131](#), [25651](#), [26783](#),
[28846](#), [28991](#), [28993](#), [29996](#), [30199](#),
[30201](#), [30282](#), [30423](#), [30601](#), [30634](#),
[30780](#), [30786](#), [30797](#), [30823](#), [30826](#),
[30960](#), [30973](#), [31054](#), [31062](#), [31074](#),
[31115](#), [31118](#), [31142](#), [31554](#), [31559](#),
[31569](#), [31572](#), [31592](#), [31601](#), [32553](#),
[32594](#), [37666](#), [37938](#), [37944](#), [38538](#)
 - `\int_compare:nTF` [170](#), [171](#), [173](#), [264](#),
[946](#), [6041](#), [6081](#), [7981](#), [8210](#), [8211](#),
[8216](#), [8218](#), [10000](#), [10002](#), [10284](#),
[10528](#), [17657](#), [17774](#), [17782](#), [17791](#),
[17797](#), [29051](#), [29672](#), [29674](#), [30640](#)
 - `\int_compare_p:n` [171](#), [6088](#), [17657](#)
 - `\int_compare_p:nNn`
[28](#), [170](#), [5663](#), [5664](#), [8708](#), [8996](#),
[9082](#), [9084](#), [9086](#), [10443](#), [11199](#),
[11200](#), [11256](#), [11257](#), [17710](#), [30556](#),
[32486](#), [33523](#), [33524](#), [33545](#), [33546](#),
[33792](#), [38646](#), [38647](#), [38654](#), [38657](#),
[38658](#), [38666](#), [38669](#), [38670](#), [38713](#)
 - `\int_const:Nn` [168](#), [4351](#),
[4352](#), [4353](#), [4354](#), [4774](#), [4775](#), [4776](#),
[4777](#), [4778](#), [4779](#), [4783](#), [4784](#), [4785](#),
[4786](#), [4787](#), [4788](#), [4789](#), [4790](#), [4791](#),
[4792](#), [4793](#), [4794](#), [4795](#), [8910](#), [9006](#),
[9008](#), [9010](#), [9011](#), [9012](#), [9069](#), [10193](#),
[10372](#), [10438](#), [10439](#), [14173](#), [14174](#),
[17556](#), [17556](#), [17558](#), [18206](#), [18207](#),
[18208](#), [18209](#), [18210](#), [18211](#), [18212](#),
[18213](#), [18214](#), [18215](#), [18216](#), [18217](#),
[18218](#), [18219](#), [18264](#), [18265](#), [18266](#),
[19942](#), [23236](#), [23237](#), [23238](#), [23239](#),
[23240](#), [23241](#), [23242](#), [23426](#), [23427](#),
[23428](#), [23430](#), [23431](#), [23432](#), [23435](#),
[23436](#), [23437](#), [23786](#), [24060](#), [24061](#),
[24062](#), [24063](#), [24064](#), [24065](#), [24066](#),
[24067](#), [24068](#), [24069](#), [24070](#), [24071](#),
[24072](#), [24073](#), [24074](#), [27910](#), [29173](#),
[30981](#), [38523](#), [38628](#), [39600](#), [39838](#)
 - `\int_decr:N` [169](#),
[3297](#), [3298](#), [3299](#), [3362](#), [3363](#), [3372](#),
[3373](#), [3382](#), [3383](#), [3642](#), [7111](#), [7188](#),
[7411](#), [7506](#), [17622](#), [17624](#), [17631](#), [39456](#)
 - `\int_div_round:nn` .. [168](#), [17511](#), [17532](#)
 - `\int_div_truncate:nn`
..... [168](#), [8723](#), [8744](#),
[9009](#), [14289](#), [14294](#), [14942](#), [14943](#),
[14998](#), [15180](#), [15346](#), [15357](#), [17511](#),
[17511](#), [17917](#), [18016](#), [18036](#), [19954](#),
[30883](#), [30896](#), [30901](#), [30913](#), [30991](#),
[31119](#), [31127](#), [31228](#), [38618](#), [39942](#)
 - `\int_do_until:nn`
..... [173](#), [17772](#), [17794](#), [17798](#)
 - `\int_do_until:nNnn`
..... [172](#), [17800](#), [17822](#), [17826](#)
 - `\int_do_while:nn`
..... [173](#), [17772](#), [17788](#), [17792](#)
 - `\int_do_while:nNnn`
..... [173](#), [17800](#), [17816](#), [17820](#)
 - `\int_eval:n` [19](#), [34](#), [166–](#)
[172](#), [179](#), [365](#), [368](#), [396](#), [471](#), [628](#),
[715](#), [844](#), [861](#), [1007](#), [1008](#), [1012](#),
[1013](#), [1018](#), [1052](#), [1102](#), [1127](#), [1129](#),
[2080](#), [2109](#), [2125](#), [3238](#), [3503](#), [3504](#),
[3765](#), [3846](#), [3850](#), [3873](#), [5832](#), [6040](#),
[7031](#), [7985](#), [8030](#), [8031](#), [8251](#), [8566](#),
[9031](#), [10007](#), [10246](#), [10497](#), [10815](#),
[10873](#), [11247](#), [11248](#), [11271](#), [11281](#),
[11288](#), [11289](#), [12427](#), [12775](#), [12780](#),

- 12788, 13109, 13117, 13125, 13152,
 13156, 13165, 13172, 13207, 13217,
 13789, 13802, 13827, 13851, 13852,
 13864, 13869, 13900, 13917, 13954,
 14045, 14074, 14078, 14085, 14094,
 14244, 14264, 14282, 14559, 15084,
 15099, 15127, 15276, 15281, 15299,
 15443, 16890, 17148, 17156, 17164,
 17341, 17462, 17462, 17557, 17729,
 17734, 17739, 17744, 17898, 17986,
 17988, 18118, 18128, 18163, 18174,
 18180, 18191, 18222, 18259, 18263,
 18855, 18867, 18953, 18963, 18977,
 18984, 19000, 19063, 19065, 19133,
 19135, 19139, 19141, 19145, 19147,
 19151, 19153, 19186, 19187, 20610,
 21090, 21128, 21133, 21141, 21147,
 21156, 22785, 22871, 22919, 22937,
 22953, 23012, 23050, 23051, 23102,
 23119, 23246, 29322, 29325, 29326,
 29416, 29417, 29955, 29991, 30039,
 30101, 30109, 30304, 30419, 30592,
 30866, 30918, 30921, 30926, 30932,
 30951, 31000, 31050, 31069, 31096,
 31152, 31157, 31192, 31203, 31222,
 31249, 31332, 32539, 32972, 33001,
 33035, 33113, 33131, 33148, 33173,
 34748, 34758, 37946, 37989, 38035,
 38546, 38551, 38558, 38563, 38616,
 38624, 39240, 39242, 39739, 39894
 \int_eval:w ... 167, 366, 370, 3568,
 3832, 3842, 10766, 10775, 10800,
 10812, 13820, 14276, 17295, 17462,
 17464, 18303, 18338, 21162, 22967,
 23144, 23151, 23152, 23163, 26731
 \int_from_alpha:n ... 176, 18161, 18161
 \int_from_base:nn
177, 18178, 18178, 18201, 18203, 18205
 \int_from_bin:n
176, 288, 1254, 18200, 18200, 30283
 \int_from_hex:n
177, 18200, 18202, 37067, 37068, 37069
 \int_from_oct:n ... 177, 18200, 18204
 \int_from_roman:n .. 177, 18220, 18220
 \int_gadd:Nn
169, 17610, 17614, 17619, 39534, 39835
 \int_gdecr:N 169, 3903, 4030,
 10407, 12721, 13684, 17221, 17277,
17622, 17628, 17633, 17896, 18789,
 20575, 21028, 25833, 34052, 39537
 \int_gincr:N 169,
 3892, 4020, 6186, 10398, 12712,
 13673, 17213, 17271, 17622, 17626,
 17632, 17871, 17882, 18780, 19947,
 20570, 21007, 21014, 22779, 23003,
 25812, 25819, 29986, 30512, 34046,
 37653, 38246, 38251, 38256, 39536
 .int_gset:N 243, 22109
 \int_gset:Nn 169, 845,
2328, 6203, 9406, 17634, 17636,
 17639, 34757, 34759, 39538, 39833
 \int_gset_eq:NN
 169, 17602, 17604, 17605, 39533
 \int_gsub:Nn 170, 17610,
 17616, 17621, 30000, 39535, 39837
 \int_gzero:N 169, 2318, 6166, 6183,
17592, 17593, 17595, 17599, 39532
 \int_gzero_new:N
 169, 17596, 17598, 17601
 \int_if_even:n 17764
 \int_if_even:nTF 172, 17756
 \int_if_even_p:n 172, 17756
 \int_if_exist:N 17606, 17608
 \int_if_exist:NTF
 169, 5512, 5567, 17597,
 17599, 17606, 18234, 18238, 38528
 \int_if_exist_p:N 169, 17606
 \int_if_odd:n 17756
 \int_if_odd:nTF 172,
 7328, 7351, 7425, 11025, 17756, 27118
 \int_if_odd_p:n 172, 6114, 17756
 \int_if_zero:n 17718
 \int_if_zero:nTF 172, 17718
 \int_if_zero_p:n 172, 17718
 \int_incr:N . 169, 3210, 3307, 3308,
 3683, 3725, 3738, 3756, 4285, 4286,
 5403, 6046, 6210, 6250, 6339, 6669,
 6765, 7099, 7171, 7405, 7410, 7445,
 7503, 7588, 7589, 7625, 7652, 7752,
 7753, 7915, 16997, 17622, 17622,
 17630, 21770, 22936, 23091, 23125,
 23176, 29899, 30089, 37979, 39455
 \int_log:N ... 178, 18260, 18260, 18261
 \int_log:n 178, 18262, 18262
 \int_max:nn . 168, 1218, 5891, 5892,
 5899, 5900, 6197, 6363, 7718, 7720,
17479, 17487, 26979, 28139, 39940
 \int_min:nn
168, 1222, 17479, 17495, 31102, 39941
 \int_mod:nn
 ... 168, 8725, 8746, 9007, 14595,
 14659, 14943, 14944, 15181, 17511,
 17534, 17907, 18007, 18027, 19956,
 30915, 31130, 31241, 38625, 39943
 \int_new:N
 . 168, 169, 3115, 3116, 3117, 3118,
 3119, 3120, 3121, 3122, 3123, 3124,
 3125, 3554, 3555, 3556, 3557, 4009,

- 4338, 4339, 4340, 4350, 4772, 4773,
4780, 4781, 4798, 6131, 6133, 6134,
6135, 6138, 6161, 6162, 6543, 6544,
6545, 6546, 6547, 6548, 6549, 6551,
6552, 6553, 6554, 6557, 6558, 6559,
6819, 7372, 7375, 7376, 7377, 7383,
7384, 8266, 8626, 10601, 10604,
10606, 10619, 14641, 17550, 17550,
17555, 17563, 17569, 17597, 17599,
18276, 18277, 18278, 18279, 18280,
18281, 19941, 21568, 22763, 22766,
22767, 22999, 29882, 29979, 29980,
30220, 30370, 36705, 37594, 38462
- `\int_rand:n`
.... 177, 22928, 23112, 29414, 29414
- `\int_rand:nn`
77, 177, 1221, 1228, 13132, 17170,
18264, 19014, 19019, 29320, 29320
- `\int_range:nn` 1222
- `.int_set:N` 243, 22109
- `\int_set:Nn` .. 169, 365, 2310, 2324,
2325, 2330, 2332, 3130, 3132, 3134,
3156, 3157, 3172, 3180, 3181, 3193,
3194, 3212, 3215, 3637, 3700, 4044,
4140, 4143, 4273, 5594, 6132, 6196,
6199, 6237, 6239, 6308, 6359, 6360,
6370, 6381, 6405, 6423, 6472, 6604,
6606, 6630, 6673, 6674, 6715, 6750,
7450, 7522, 7524, 7668, 7694, 7717,
7719, 10358, 10360, 10581, 10583,
10602, 10612, 10625, 10672, 10678,
10690, 10695, 12374, 12409, 14720,
14769, 14822, 16998, 17634, 17634,
17638, 21775, 23167, 30480, 30481,
30496, 30667, 30682, 30718, 34767,
34768, 34769, 34770, 39457, 39832
- `\int_set_eq:NN` 169, 3173,
3203, 4411, 4881, 4885, 4894, 4896,
4939, 5006, 5302, 5402, 5415, 5514,
6152, 6172, 6189, 6194, 6214, 6248,
6249, 6299, 6402, 6403, 6455, 6504,
6582, 6605, 6609, 6610, 6624, 6628,
6631, 6670, 6680, 6811, 6812, 7401,
7618, 7911, 8891, 11076, 12372,
12375, 17602, 17602, 17603, 39452
- `\int_show:N` .. 177, 18256, 18256, 18257
- `\int_show:n` 178, 628, 863, 18258, 18258
- `\int_sign:n`
168, 951, 17465, 17465, 30563, 39896
- `\int_step_function:nN`
.... 174, 17828, 17861, 30686, 30687
- `\int_step_function:nnN` 174,
6681, 7518, 17828, 17863, 19260,
30688, 30689, 30690, 30691, 30712
- `\int_step_function:nnnN`
..... 73, 174, 853, 1105, 7697,
7705, 17828, 17828, 17862, 17864,
17895, 40004, 40008, 40012, 40016
- `\int_step_inline:nn` 174,
1013, 16985, 17865, 17865, 23006,
30400, 30434, 30491, 31136, 31179
- `\int_step_inline:nnn`
174, 3264, 6597, 8269, 9120, 9122,
9124, 10197, 10450, 14579, 14588,
17865, 17867, 30407, 30410, 30668
- `\int_step_inline:nnnn`
174, 1107, 17865, 17866, 17868, 17869
- `\int_step_variable:nNn`
..... 174, 17865, 17876
- `\int_step_variable:nnNn`
..... 174, 17865, 17878
- `\int_step_variable:nnnNn`
.... 174, 17865, 17877, 17879, 17880
- `\int_sub:Nn` 170, 4415,
5117, 6458, 6466, 6475, 9350, 10887,
17610, 17612, 17620, 39454, 39836
- `\int_to_Alph:n` 175, 176, 17921, 17953
- `\int_to_alph:n` 175, 176, 17921, 17921
- `\int_to_arabic:n`
..... 175, 17898, 17898, 17899
- `\int_to_Base:n` 176
- `\int_to_base:n` 176
- `\int_to_Base:mn`
..... 176, 177, 17985, 17987, 18112
- `\int_to_base:mn` 176,
177, 17985, 17985, 18108, 18110, 18114
- `\int_to_bin:n` 176, 18107, 18107
- `\int_to_Hex:n` 176,
177, 4626, 18107, 18111, 31267, 37588
- `\int_to_hex:n` . 176, 177, 18107, 18109
- `\int_to_oct:n` . 176, 177, 18107, 18113
- `\int_to_Roman:n` 176, 177, 18115, 18125
- `\int_to_roman:n` 176, 177, 18115, 18115
- `\int_to_symbols:nnn`
175, 17900, 17900, 17916, 17923, 17955
- `\int_until_do:nn`
..... 173, 17772, 17780, 17785
- `\int_until_do:nNnn`
..... 173, 17800, 17808, 17813
- `\int_use:N` 165, 167, 170, 1045, 1051,
2329, 2331, 2333, 3894, 4022, 4042,
4913, 5000, 5078, 5089, 5098, 5102,
5113, 5114, 5120, 5121, 5127, 5128,
5285, 6114, 6204, 6209, 6230, 6232,
6337, 6350, 6351, 6751, 6803, 6901,
6912, 7067, 7450, 7534, 7535, 7726,
7727, 8263, 8734, 8736, 8739, 8755,
8757, 8761, 8764, 8769, 9310, 10009,

- 10361, 10400, 10577, 12714, 12716,
 13675, 13679, 15080, 15104, 15118,
 15138, 15145, 15327, 15436, 15441,
 15457, 17214, 17220, 17273, 17275,
 17640, 17640, 17641, 17874, 17885,
 18782, 18784, 20569, 20577, 21010,
 21017, 21776, 22780, 22874, 22922,
 25815, 25822, 29910, 30514, 30516,
 30547, 30596, 34048, 34050, 37659,
 38530, 39736, 39737, 39738, 39773
- `\int_value:w` 166, 179,
 370, 446, 590, 842, 848, 946, 1007,
 1008, 1012, 1013, 1022, 1028, 1032,
 1045, 1053, 1060, 1063, 1068, 1075,
 1103, 1104, 1113, 1121, 1129, 1195,
 1199, 1213, 1809, 3238, 3568, 3580,
 3783, 3830, 3832, 3842, 3850, 3869,
 3871, 3879, 4084, 4119, 4183, 4203,
 4212, 4619, 5146, 5152, 5182, 5184,
 5193, 5194, 5309, 5794, 5809, 6836,
 6837, 6848, 7559, 8417, 8420, 8566,
 9053, 9058, 10766, 10775, 13165,
 13172, 13789, 13790, 13802, 13820,
 13827, 13850, 13851, 13852, 13864,
 13900, 14512, 14636, 14998, 15076,
 15084, 15099, 15127, 16920, 16930,
 17283, 17295, 17450, 17450, 17467,
 17468, 17481, 17482, 17489, 17490,
 17491, 17497, 17498, 17499, 17513,
 17515, 17516, 17533, 17536, 17537,
 17538, 17545, 17660, 17664, 17694,
 17831, 17832, 17833, 17859, 18071,
 18104, 18303, 18338, 19186, 19187,
 19287, 20843, 21034, 21068, 21075,
 21098, 21106, 21107, 21162, 23022,
 23025, 23050, 23051, 23097, 23102,
 23144, 23151, 23152, 23163, 23320,
 23321, 23322, 23323, 23324, 23338,
 23486, 23547, 23565, 23860, 23990,
 24004, 24006, 24008, 24011, 24047,
 24185, 24215, 24216, 24253, 24261,
 24392, 24397, 24399, 24408, 24412,
 24449, 24457, 24460, 24466, 24477,
 24488, 24494, 24495, 24498, 24541,
 24551, 24553, 24569, 24571, 24594,
 24608, 24684, 24685, 24759, 24847,
 25621, 25654, 25999, 26000, 26001,
 26003, 26049, 26052, 26055, 26078,
 26080, 26101, 26103, 26112, 26114,
 26118, 26136, 26143, 26149, 26159,
 26161, 26175, 26183, 26191, 26235,
 26237, 26253, 26255, 26258, 26261,
 26315, 26323, 26325, 26327, 26329,
 26332, 26335, 26337, 26356, 26358,
 26362, 26368, 26370, 26374, 26396,
 26399, 26407, 26409, 26412, 26413,
 26414, 26415, 26430, 26433, 26436,
 26439, 26448, 26451, 26454, 26457,
 26464, 26466, 26472, 26480, 26482,
 26484, 26510, 26512, 26521, 26523,
 26527, 26544, 26565, 26569, 26581,
 26584, 26587, 26590, 26593, 26596,
 26599, 26602, 26606, 26618, 26622,
 26626, 26629, 26650, 26652, 26654,
 26664, 26688, 26691, 26703, 26705,
 26711, 26714, 26731, 26751, 26802,
 26807, 26809, 26816, 26819, 26822,
 26825, 26828, 26831, 26840, 26852,
 26860, 26862, 26872, 26874, 26881,
 26890, 26892, 26895, 26898, 26901,
 26904, 26917, 26919, 26927, 26929,
 26937, 26939, 26949, 26952, 26955,
 26962, 26977, 26995, 26998, 27054,
 27068, 27070, 27076, 27089, 27091,
 27093, 27117, 27133, 27140, 27141,
 27185, 27187, 27188, 27189, 27230,
 27232, 27269, 27276, 27283, 27304,
 27306, 27308, 27310, 27323, 27327,
 27328, 27329, 27330, 27331, 27336,
 27341, 27343, 27349, 27366, 27367,
 27368, 27369, 27370, 27371, 27376,
 27378, 27380, 27382, 27384, 27389,
 27391, 27393, 27395, 27397, 27399,
 27421, 27429, 27445, 27450, 27454,
 27513, 27562, 27630, 27639, 27647,
 27658, 27660, 27663, 27666, 27754,
 27790, 27792, 27795, 27798, 27801,
 27804, 27811, 27814, 27816, 27820,
 27842, 27844, 27876, 27946, 27956,
 27961, 27971, 28113, 28145, 28154,
 28386, 28387, 28398, 28401, 28404,
 28407, 28410, 28413, 28416, 28419,
 28422, 28440, 28450, 28459, 28477,
 28486, 28493, 28503, 28547, 28556,
 28591, 28634, 28651, 28707, 28718,
 28729, 28939, 29015, 29062, 29107,
 29115, 29117, 29119, 29185, 29208,
 29262, 29302, 29314, 29325, 29326,
 29356, 29359, 29362, 29364, 29366,
 29373, 29376, 29384, 29389, 29394,
 29955, 30039, 30101, 30109, 30122,
 30123, 30124, 30134, 31468, 39201
- `\int_while_do:nn`
 173, 17772, 17772, 17777
- `\int_while_do:nNnn`
 173, 17800, 17800, 17805
- `\int_zero:N`
 169, 3638, 3639, 3640, 3739,

- 4893, 5115, 5551, 6078, 6151, 6182,
6603, 6880, 7395, 7396, 7444, 7631,
7906, 10732, 16979, 17592, 17592,
17594, 17597, 21767, 22933, 23088,
23117, 29896, 30086, 37976, 39451
- `\int_zero_new:N`
..... 169, 17596, 17596, 17600
- `\c_max_char_int`
..... 178, 4623, 18266, 19201, 19942
- `\c_max_int` ... 178, 255, 535, 1221,
1222, 4050, 4051, 4052, 18265,
29367, 34745, 34751, 36646, 36649
- `\c_max_register_int`
..... 178, 431, 1445, 3132,
3157, 3194, 10007, 10009, 16972, 17450
- `\c_one_int` 178,
3740, 4054, 4700, 4823, 5836, 5892,
5900, 5943, 6088, 6194, 6309, 6382,
6393, 6404, 6407, 6424, 6464, 6473,
6598, 6601, 6609, 6630, 6683, 6700,
6701, 6711, 6773, 6774, 6848, 7018,
7031, 7053, 7519, 7700, 7708, 8271,
17623, 17625, 17627, 17629, 18264,
18303, 18338, 19869, 23144, 23163,
26616, 26620, 26624, 26678, 27270,
27413, 27554, 27860, 28700, 28784,
29225, 29229, 29236, 29421, 29949
- `\g_tmpa_int` 178, 18276
- `\l_tmpa_int` 4, 53, 178, 18276
- `\g_tmpb_int` 178, 18276
- `\l_tmpb_int` 4, 178, 18276
- `\c_zero_int` 178, 376, 388, 715, 1444,
1807, 1809, 3763, 3787, 3843, 3944,
4056, 4316, 4458, 4470, 4911, 5409,
5664, 5805, 5891, 5899, 6081, 6180,
6202, 6293, 6319, 6380, 6420, 6471,
6533, 6864, 6871, 6899, 6970, 7038,
7109, 7118, 7129, 7161, 7170, 7187,
7210, 7480, 7496, 7530, 7563, 7571,
7622, 7624, 7696, 7718, 7720, 7723,
8891, 9061, 11076, 11398, 12372,
13170, 13219, 13543, 13548, 13871,
13906, 14634, 15433, 17546, 17547,
17561, 17592, 17593, 17644, 17652,
17720, 17837, 17840, 18264, 19200,
19866, 19867, 19868, 20885, 21063,
23007, 23131, 23566, 23792, 23796,
23798, 23802, 23806, 23819, 23832,
23839, 23852, 23864, 23876, 23885,
23888, 23899, 24005, 24010, 25636,
25668, 26642, 26677, 26679, 26680,
26681, 27447, 27514, 27737, 27755,
27778, 27810, 28656, 29016, 29208,
29237, 29337, 29401, 29922, 30228
- int internal commands:
- `__int_abs:N` 17479, 17481, 17485
- `__int_case:nnTF` 17726,
17729, 17734, 17739, 17744, 17746
- `__int_case:nw`
..... 17726, 17747, 17748, 17752
- `__int_case_end:nw` 17726, 17751, 17754
- `__int_compare:nnN`
. 849, 17657, 17689, 17697, 17699,
17701, 17703, 17705, 17707, 17709
- `__int_compare:NNw`
..... 848, 849, 17657, 17669, 17673
- `__int_compare:Nw`
848, 849, 17657, 17665, 17667, 17694
- `__int_compare:w`
..... 848, 17657, 17659, 17662
- `__int_compare_!=:NNw` 17657
- `__int_compare_<:NNw` 17657
- `__int_compare_<=:NNw` 17657
- `__int_compare_=:NNw` 17657
- `__int_compare_>:NNw` 17657
- `__int_compare_>=:NNw` 17657
- `__int_compare_end_=:NNw` . 849, 17657
- `__int_compare_error:` 847,
848, 17642, 17642, 17646, 17660, 17662
- `__int_compare_error:Nw`
..... 847-849, 17642, 17648, 17682
- `__int_const:nN`
..... 17556, 17557, 17559, 17579
- `__int_constdef:Nw` . 17556, 17574,
17585, 17586, 17587, 17589, 17590
- `__int_div_truncate:NwNw`
..... 17511, 17514, 17519, 17542
- `__int_eval:w` 365,
842, 843, 848, 17450, 17451, 17463,
17464, 17468, 17482, 17490, 17491,
17498, 17499, 17513, 17515, 17516,
17533, 17536, 17537, 17538, 17545,
17577, 17611, 17613, 17615, 17617,
17635, 17637, 17660, 17694, 17712,
17720, 17758, 17766, 17831, 17832,
17833, 17859, 18044, 18071, 18077,
18104, 39840, 39898, 39945, 39969,
39985, 39986, 40007, 40011, 40015
- `__int_eval_end:`
..... 17450, 17452, 17463,
17468, 17482, 17517, 17533, 17539,
17548, 17577, 17611, 17613, 17615,
17617, 17635, 17637, 17712, 17758,
17766, 18044, 18071, 18077, 18104
- `__int_from_alpha:N`
..... 860, 18161, 18174, 18176

- __int_from_alpha:nN
 [860](#), [18161](#), [18166](#), [18170](#), [18173](#)
- __int_from_base:N
 [861](#), [18178](#), [18191](#), [18194](#)
- __int_from_base:nnN
 [861](#), [18178](#), [18183](#), [18187](#), [18190](#)
- __int_from_roman:NN
 .. [18220](#), [18226](#), [18231](#), [18247](#), [18251](#)
- \c__int_from_roman_C_int [18206](#)
- \c__int_from_roman_c_int [18206](#)
- \c__int_from_roman_D_int [18206](#)
- \c__int_from_roman_d_int [18206](#)
- __int_from_roman_error:w
 [18220](#), [18235](#), [18239](#), [18254](#)
- \c__int_from_roman_I_int [18206](#)
- \c__int_from_roman_i_int [18206](#)
- \c__int_from_roman_L_int [18206](#)
- \c__int_from_roman_l_int [18206](#)
- \c__int_from_roman_M_int [18206](#)
- \c__int_from_roman_m_int [18206](#)
- \c__int_from_roman_V_int [18206](#)
- \c__int_from_roman_v_int [18206](#)
- \c__int_from_roman_X_int [18206](#)
- \c__int_from_roman_x_int [18206](#)
- __int_if_recursion_tail_stop:N .
 [17460](#), [17461](#), [18233](#)
- __int_if_recursion_tail_stop-
 do:Nn
 .. [17460](#), [17460](#), [18172](#), [18189](#), [18236](#)
- \l__int_internal_a_int [18280](#)
- \l__int_internal_b_int [18280](#)
- \c__int_max_constdef_int [17556](#)
- __int_maxmin:wwN
 [17479](#), [17489](#), [17497](#), [17503](#)
- __int_mod:ww ... [17511](#), [17536](#), [17541](#)
- __int_pass_signs:wn
 [860](#), [18151](#), [18151](#), [18154](#), [18165](#), [18182](#)
- __int_pass_signs_end:wn
 [18151](#), [18156](#), [18160](#)
- __int_show:nN [18256](#)
- __int_sign:Nw .. [17465](#), [17467](#), [17471](#)
- __int_step:NNnnn
 [17865](#), [17872](#), [17883](#), [17892](#)
- __int_step:NwnnN
 .. [17828](#), [17838](#), [17846](#), [17851](#), [17857](#)
- __int_step:wwnN . [17828](#), [17830](#), [17835](#)
- __int_to_Base:nn [17985](#), [17988](#), [17995](#)
- __int_to_base:nn [17985](#), [17986](#), [17989](#)
- __int_to_Base:nnN
 .. [17985](#), [17998](#), [17999](#), [18021](#), [18035](#)
- __int_to_base:nnN
 .. [17985](#), [17992](#), [17993](#), [18001](#), [18015](#)
- __int_to_Base:nnnN
 [17985](#), [18026](#), [18033](#)
- __int_to_base:nnnN
 [17985](#), [18006](#), [18013](#)
- __int_to_Letter:n
 [17985](#), [18024](#), [18027](#), [18074](#)
- __int_to_letter:n
 [17985](#), [18004](#), [18007](#), [18041](#)
- __int_to_roman:N
 [18115](#), [18117](#), [18120](#), [18123](#)
- __int_to_roman:w [848](#), [859](#),
 [1425](#), [1426](#), [17450](#), [17670](#), [18118](#), [18128](#)
- __int_to_Roman_aux:N
 [18127](#), [18130](#), [18133](#)
- __int_to_Roman_c:w ... [18115](#), [18147](#)
- __int_to_roman_c:w ... [18115](#), [18139](#)
- __int_to_Roman_d:w ... [18115](#), [18148](#)
- __int_to_roman_d:w ... [18115](#), [18140](#)
- __int_to_Roman_i:w ... [18115](#), [18143](#)
- __int_to_roman_i:w ... [18115](#), [18135](#)
- __int_to_Roman_l:w ... [18115](#), [18146](#)
- __int_to_roman_l:w ... [18115](#), [18138](#)
- __int_to_Roman_m:w ... [18115](#), [18149](#)
- __int_to_roman_m:w ... [18115](#), [18141](#)
- __int_to_Roman_Q:w ... [18115](#), [18150](#)
- __int_to_roman_Q:w ... [18115](#), [18142](#)
- __int_to_Roman_v:w ... [18115](#), [18144](#)
- __int_to_roman_v:w ... [18115](#), [18136](#)
- __int_to_Roman_x:w ... [18115](#), [18145](#)
- __int_to_roman_x:w ... [18115](#), [18137](#)
- __int_to_symbols:nnnn
 [17900](#), [17904](#), [17914](#), [17920](#)
- __int_use_none_delimit_by_s-
 stop:w [17457](#), [17457](#), [17692](#)
- intarray commands:
- \intarray_const_from_clist:Nn ...
 . [255](#), [22930](#), [22930](#), [22939](#), [23114](#),
 [23114](#), [23122](#), [27568](#), [28169](#), [39601](#)
- \intarray_count:N
 .. [256](#), [367](#), [14596](#), [14659](#), [22786](#),
 [22789](#), [22830](#), [22844](#), [22874](#), [22922](#),
 [22928](#), [23013](#), [23016](#), [23018](#), [23019](#),
 [23022](#), [23022](#), [23023](#), [23031](#), [23041](#),
 [23089](#), [23112](#), [23131](#), [23194](#), [30011](#)
- \intarray_gset:Nnn [256](#), [367](#),
 [1012](#), [1014](#), [14580](#), [14592](#), [14605](#),
 [14611](#), [22866](#), [22869](#), [22877](#), [23044](#),
 [23046](#), [23053](#), [30395](#), [31199](#), [38599](#)
- \intarray_gzero:N
 [255](#), [22879](#), [22888](#), [23086](#), [23086](#), [23095](#)
- \intarray_if_exist:N .. [23180](#), [23182](#)
- \intarray_if_exist:NTF
 [256](#), [23180](#), [38597](#)
- \intarray_if_exist_p:N ... [256](#), [23180](#)
- \intarray_item:Nn [256](#), [367](#),
 [1008](#), [1012](#), [1014](#), [14590](#), [14595](#),

- 14629, 14658, 14666, [22890](#), 22917,
22926, 22928, [23096](#), 23098, 23104,
23112, 30602, 31226, 31240, 38610
- `\intarray_log:N`
..... [256](#), [23184](#), 23186, 23187
- `\intarray_new:Nn`
[255](#), [1005](#), [1011](#), [1014](#), 6560, 6561,
7378, 7379, 7380, 7381, 7382, 14578,
14587, [22775](#), [22782](#), [22792](#), [23000](#),
23009, 23021, 30003, 30004, 30005,
30391, 30990, 31177, 38598, 39643
- `\intarray_rand_item:N` [256](#), [22927](#),
[22927](#), [22929](#), [23111](#), 23111, 23113
- `\intarray_show:N`
[256](#), [1009](#), [1014](#), [23184](#), 23184, 23185
- intarray internal commands:
- `__intarray:w` [22769](#), [22780](#)
- `\l_intarray_bad_index_int`
..... [22766](#), [22874](#), [22922](#)
- `__intarray_bounds:NNnTF`
..... [23026](#), [23026](#), [23056](#), [23107](#)
- `__intarray_bounds_error:NNnw` ...
..... [23026](#), [23029](#), [23032](#), [23037](#)
- `__intarray_const_from_clist:nN` .
..... [23114](#), 23119, [23123](#)
- `__intarray_count:w` [22996](#),
[22997](#), 23012, 23022, 23120, 23139
- `__intarray_entry:w`
.. [22996](#), [22996](#), 23045, 23092, 23097
- `\g__intarray_font_int`
..... [22999](#), 23003, 23005
- `__intarray_gset:Nnn` [23044](#)
- `__intarray_gset:Nww` .. [23048](#), [23054](#)
- `__intarray_gset:w` [22846](#), [22868](#)
- `__intarray_gset:wTF` .. [22846](#), [22871](#)
- `__intarray_gset_count:Nw`
..... [1004](#), [22764](#), [22785](#), [22830](#)
- `__intarray_gset_overflow:Nnn` . [23044](#)
- `__intarray_gset_overflow:NNnn` ..
..... [23068](#), [23076](#), [23080](#)
- `__intarray_gset_overflow_-
test:nw` [1010](#), [1014](#), [22990](#),
[22991](#), 23058, 23065, 23073, 23126
- `__intarray_gset_range:nNw` ... [22964](#)
- `__intarray_gset_range:Nw`
..... [23165](#), [23168](#), 23170, [23177](#)
- `__intarray_gset_range:w` [22967](#)
- `__intarray_item:Nw`
..... [23096](#), 23100, 23105
- `__intarray_item:w` [22890](#), [22916](#)
- `__intarray_item:wTF` .. [22890](#), [22919](#)
- `\l_intarray_loop_int`
..... [22763](#), [22933](#), [22936](#), [22937](#),
23088, 23091, 23092, 23117, 23120,
23125, 23127, 23167, 23175, 23176
- `__intarray_new:N`
..... [22775](#), [22776](#), [22784](#),
[22932](#), [23000](#), 23000, 23011, 23116
- `__intarray_range_to_clist:w` ...
..... [22949](#), [22952](#)
- `__intarray_range_to_clist:ww` ...
..... [23146](#), 23150, 23156, 23162
- `__intarray_show:NN`
..... 23184, 23186, 23188
- `__intarray_signed_max_dim:n` ...
..... [23024](#), [23024](#), 23083, 23084
- `\c__intarray_sp_dim`
..... [22998](#), 23005, 23045
- `__intarray_table` [22804](#)
- `\g__intarray_table_int`
..... [22766](#), [22779](#), [22780](#)
- `__intarray_to_clist:Nn`
... [1009](#), [22940](#), [23129](#), 23129, 23195
- `__intarray_to_clist:w`
.. [22940](#), [23129](#), 23134, 23137, 23143
- `\interactionmode` 504
- `\interlinepenalties` 505
- `\interlinepenalty` 285
- ior commands:
- `\ior_close:N`
.... [91](#), [92](#), 10241, [10282](#), 10282,
10293, 11658, 31276, 31310, 31380
- `\ior_get:NN`
..... [92-94](#), [96](#), [10339](#), 10339, 10343, 10419
- `\ior_get:NNTF` [93](#), [10339](#), 10340
- `\ior_get_term:nN` [96](#), [10373](#), 10373
- `\ior_if_eof:N` [644](#), 10323
- `\ior_if_eof:NTF`
..... [95](#), [10323](#), 10345, 10365, 10405, 10424
- `\ior_if_eof_p:N` [95](#), [10323](#)
- `\ior_log:N` ... [92](#), [10294](#), [10296](#), [10297](#)
- `\ior_log_list:` [92](#), [10310](#), 10311
- `\ior_map_break:` [95](#),
[10388](#), 10388, 10389, 10391, 10406,
10413, 10425, 10431, 31211, 31306
- `\ior_map_break:n` [95](#), [10388](#), 10390
- `\ior_map_inline:Nn`
..... [94](#), [10392](#), [10392](#), 11656
- `\ior_map_variable:NNn`
..... [94](#), [10418](#), 10418, 31208
- `\ior_new:N` [91](#), [10210](#), 10210, 10211,
10212, 10213, 11221, 30982, 31345
- `\ior_open:Nn` [91](#),
[674](#), [10214](#), 10214, 10216, 10218,
10227, 31205, 31244, 31277, 31347
- `\ior_open:NnTF` [91](#), 10215, [10218](#)

- \ior_shell_open:Nn
..... [91](#), [367](#), [10260](#), [10260](#), [11645](#)
- \ior_show:N .. [92](#), [10294](#), [10294](#), [10295](#)
- \ior_show_list: [92](#), [10310](#), [10310](#)
- \ior_str_get:NN
..... [92](#), [93](#), [96](#), [10352](#), [10352](#), [10363](#), [10421](#)
- \ior_str_get:NNTF ... [93](#), [10352](#), [10353](#)
- \ior_str_get_term:nN [96](#), [10373](#), [10375](#)
- \ior_str_map_inline:Nn
..... [94](#), [10392](#), [10394](#), [31270](#), [31299](#), [31372](#)
- \ior_str_map_variable:NNn
..... [94](#), [10418](#), [10420](#)
- \g_tmpa_ior [99](#), [10212](#)
- \g_tmpb_ior [99](#), [10212](#)
- ior internal commands:
- \l_ior_file_name_tl
..... [10217](#), [10220](#), [10222](#)
- __ior_get:NN
.. [10339](#), [10341](#), [10348](#), [10374](#), [10393](#)
- __ior_get_term:NnN
..... [10373](#), [10374](#), [10376](#), [10377](#)
- \l_ior_internal_tl
.. [10192](#), [10302](#), [10305](#), [10411](#), [10415](#)
- __ior_list:N
..... [10310](#), [10310](#), [10311](#), [10312](#)
- __ior_map_inline:NNn
..... [10392](#), [10393](#), [10395](#), [10396](#)
- __ior_map_inline:NNNn
..... [10392](#), [10399](#), [10402](#)
- __ior_map_inline_loop:NNN
..... [10392](#), [10405](#), [10409](#), [10416](#)
- __ior_map_variable:NNNn
..... [10418](#), [10419](#), [10421](#), [10422](#)
- __ior_map_variable_loop:NNNn ...
..... [10418](#), [10424](#), [10427](#), [10434](#)
- __ior_new:N
..... [640](#), [10228](#), [10228](#), [10232](#), [10233](#), [10245](#)
- __ior_new_aux:N [10232](#), [10236](#)
- __ior_open_stream:Nn
..... [10239](#), [10243](#), [10247](#), [10251](#)
- __ior_shell_open:nN
..... [10260](#), [10263](#), [10266](#), [10275](#)
- __ior_show:NN
..... [10294](#), [10294](#), [10296](#), [10298](#)
- __ior_str_get:NN
.. [10352](#), [10354](#), [10368](#), [10376](#), [10395](#)
- \l_ior_stream_tl
..... [10195](#), [10242](#), [10246](#), [10253](#)
- \g__ior_streams_prop
..... [641](#), [10196](#), [10254](#), [10287](#), [10302](#), [10317](#)
- \g__ior_streams_seq
..... [10194](#), [10242](#), [10288](#), [10289](#)
- \c_ior_term_ior [10193](#),
..... [10210](#), [10284](#), [10290](#), [10326](#), [10383](#)
- \c__ior_term_noprompt_ior
..... [10372](#), [10382](#)
- ior commands:
- \ior_char:N [83](#), [97](#), [3491](#),
..... [3494](#), [3495](#), [3519](#), [3520](#), [3527](#), [3528](#),
..... [4597](#), [4598](#), [4605](#), [4607](#), [4609](#), [4611](#),
..... [4613](#), [4615](#), [5259](#), [5260](#), [5994](#), [6001](#),
..... [6002](#), [6003](#), [6127](#), [7953](#), [7956](#), [7957](#),
..... [7962](#), [7996](#), [8005](#), [8009](#), [8014](#), [8034](#),
..... [8036](#), [8037](#), [8039](#), [8042](#), [8044](#), [8049](#),
..... [8051](#), [8053](#), [8058](#), [8062](#), [8065](#), [8066](#),
..... [8069](#), [8071](#), [8075](#), [8077](#), [8083](#), [8085](#),
..... [8089](#), [8091](#), [8095](#), [8100](#), [8102](#), [8144](#),
..... [8146](#), [8151](#), [8153](#), [8159](#), [8164](#), [8169](#),
..... [8173](#), [8183](#), [8186](#), [8190](#), [8191](#), [8195](#),
..... [8203](#), [8274](#), [9855](#), [9858](#), [9859](#), [9891](#),
..... [9919](#), [10077](#), [10600](#), [10600](#), [11715](#),
..... [11717](#), [11718](#), [11719](#), [14698](#), [27674](#),
..... [30735](#), [30737](#), [30738](#), [30741](#), [30743](#),
..... [30744](#), [30747](#), [30749](#), [30750](#), [30751](#),
..... [30755](#), [30762](#), [39169](#), [40039](#), [40040](#)
- \ior_close:N
.. [91](#), [92](#), [10492](#), [10526](#), [10526](#), [10537](#)
- \ior_indent:n [98](#),
..... [653](#), [654](#), [8230](#), [9812](#), [9953](#), [10024](#),
..... [10032](#), [10048](#), [10650](#), [10650](#), [10653](#),
..... [10665](#), [10682](#), [10687](#), [14696](#), [15021](#),
..... [15209](#), [23735](#), [23747](#), [38288](#), [38317](#),
..... [38339](#), [38362](#), [38371](#), [38380](#), [38401](#)
- \l_ior_line_count_int
..... [98](#), [99](#), [461](#), [654](#), [3993](#),
..... [3997](#), [9350](#), [10601](#), [10691](#), [10696](#), [10734](#)
- \ior_log:N ... [92](#), [10538](#), [10540](#), [10541](#)
- \ior_log:n [96](#),
..... [365](#), [1975](#), [1975](#), [9550](#), [9557](#), [10593](#),
..... [10593](#), [10594](#), [10595](#), [13271](#), [39177](#)
- \ior_log_list: [92](#), [10554](#), [10555](#)
- \ior_new:N
..... [91](#), [10467](#), [10467](#), [10468](#), [10469](#), [10470](#)
- \ior_newline: [83](#), [96-98](#), [368](#), [618](#),
..... [650](#), [728](#), [3943](#), [6049](#), [9372](#), [10599](#),
..... [10599](#), [10679](#), [10688](#), [10694](#), [11574](#),
..... [25467](#), [25468](#), [36633](#), [36634](#), [36635](#)
- \ior_now:Nn
..... [96](#), [97](#), [8938](#), [10586](#), [10586](#), [10591](#),
..... [10592](#), [10593](#), [10594](#), [10596](#), [10597](#)
- \ior_open:Nn . [91](#), [10483](#), [10483](#), [10489](#)
- \ior_shell_open:Nn
..... [91](#), [367](#), [10510](#), [10510](#)
- \ior_shipout:Nn [96](#), [97](#),
..... [650](#), [8970](#), [10569](#), [10569](#), [10571](#), [10572](#)
- \ior_shipout_e:Nn [96](#),
..... [97](#), [10566](#), [10566](#), [10568](#), [38805](#), [38806](#)

- \iow_shipout_x:Nn 650, 38805, 38806, 38807
- \iow_show:N .. 92, 10538, 10538, 10539
- \iow_show_list: 92, 10554, 10554
- \iow_term:n 96, 1975, 1977, 8264, 9384, 9540, 9545, 9563, 9589, 10593, 10596, 10597, 10598
- \iow_wrap:nnnN 96–99, 628, 654, 728, 9348, 9351, 9363, 9521, 9555, 9561, 9568, 10642, 10648, 10653, 10665, 10668, 10668, 10702, 13255, 13271
- \iow_wrap_allow_break: . 98, 10639, 10639, 10642, 10648, 10681, 10686
- \iow_wrap_allow_break:n 653
- \c_log_iow 99, 645, 10438, 10528, 10593, 10594
- \c_term_iow .. 99, 645, 646, 10438, 10467, 10528, 10534, 10596, 10597
- \g_tmpa_iow 99, 10469
- \g_tmpb_iow 99, 10469
- iow internal commands:
 - \l_iow_file_name_tl 10482, 10485, 10487
 - __iow_indent:n 653, 10650, 10656, 10682
 - __iow_indent_error:n 653, 10650, 10662, 10687
 - \l__iow_indent_int 10618, 10732, 10750, 10862, 10879, 10887
 - \l__iow_indent_tl .. 10618, 10733, 10749, 10861, 10880, 10888, 10889
 - \l__iow_internal_tl 10437, 10546, 10549
 - \l__iow_line_break_bool 10622, 10728, 10856, 10870, 10878, 10886, 10894, 10896, 10901, 10903
 - \l__iow_line_part_tl 656, 657, 659, 10620, 10730, 10742, 10763, 10821, 10824, 10855, 10869, 10871, 10877, 10885, 10908
 - \l__iow_line_target_int 659, 10604, 10690, 10692, 10695, 10857, 10862, 10897
 - \l__iow_line_tl 10620, 10729, 10746, 10836, 10852, 10868, 10869, 10877, 10885, 10907, 10908, 10913, 10915
 - __iow_list:N 10554, 10554, 10555, 10556
 - __iow_new:N 10471, 10471, 10475, 10476, 10496
 - __iow_new_aux:N 10475, 10479
 - \l__iow_newline_tl 10603, 10688, 10689, 10691, 10694, 10912
 - \l__iow_one_indent_int 10605, 10879, 10887
 - \l__iow_one_indent_tl 652, 10605, 10880
 - __iow_open_stream:Nn 10483, 10494, 10498, 10502, 10509
 - __iow_set_indent:n 651, 10605, 10608, 10617
 - __iow_shell_open:nN 10510, 10513, 10516, 10525
 - __iow_show:NN 10538, 10538, 10540, 10542
 - \l__iow_stream_tl 10448, 10493, 10497, 10504
 - \g__iow_streams_prop 649, 10449, 10505, 10531, 10546, 10561
 - \g__iow_streams_seq 10447, 10493, 10532, 10533
 - __iow_tmp:w 657, 10736, 10760, 10817, 10849, 10917, 10925
 - __iow_unindent:w 651, 10605, 10607, 10615, 10889
 - __iow_use_i_delimit_by_s-stop:nw 10465, 10465, 10721
 - __iow_with:nNnn 10573, 10577, 10579, 10585
 - __iow_wrap_allow_break: 653, 10639, 10644, 10681
 - __iow_wrap_allow_break:n 10866, 10866
 - __iow_wrap_allow_break_error: .. 653, 10639, 10645, 10686
 - \c__iow_wrap_allow_break_marker_tl 10624, 10644
 - __iow_wrap_break:w 10803, 10817, 10819
 - __iow_wrap_break_end:w 657, 10817, 10826, 10846
 - __iow_wrap_break_first:w 10817, 10823, 10829
 - __iow_wrap_break_loop:w 10817, 10832, 10840, 10844
 - __iow_wrap_break_none:w 10817, 10831, 10834
 - __iow_wrap_chunk:nw 10734, 10736, 10738, 10872, 10873, 10881, 10890, 10897
 - __iow_wrap_do: . 10698, 10703, 10703
 - __iow_wrap_end:n 10892, 10899
 - __iow_wrap_end_chunk:w 655, 10754, 10761, 10811, 10853
 - \c__iow_wrap_end_marker_tl 10624, 10708

- __iow_wrap_fix_newline:w 34480, 34497, 34498, 34520, 34521,
..... 10703, 10712, 10717, 10724 34522, 34577, 34578, 34603, 34604
- __iow_wrap_indent:n .. 10875, 10875 \kanjiskip 1161
- \c__iow_wrap_indent_marker_tl ... \kansuji 1162
- 10624, 10658 \kansujichar 1163
- __iow_wrap_line:nw 655, \kcatcode 1164
- 658, 10748, 10752, 10761, 10761, 10860 \kchar 1204
- __iow_wrap_line_aux:Nw \kchardef 1205
- 10761, 10771, 10777 \kern 287
- __iow_wrap_line_end:NnnnnnnN .. kernel internal commands:
- 10761, 10780, 10797 __kernel_backend_align_begin: . 372
- __iow_wrap_line_end:nw 657, 10761, __kernel_backend_align_end: . 372
- 10802, 10805, 10837, 10838, 10847 \g__kernel_backend_header_bool . 372
- __iow_wrap_line_loop:w __kernel_backend_literal:n ... 372
- 10761, 10765, 10768, 10774 __kernel_backend_literal_pdf:n 372
- __iow_wrap_line_seven:nnnnnn .. __kernel_backend_literal_-
- 10761, 10792, 10796 postscript:n 372
- \c__iow_wrap_marker_tl __kernel_backend_literal_svg:n 372
- 652, 655, 10624, 10760 __kernel_backend_matrix:n 372
- __iow_wrap_newline:n . 10892, 10892 __kernel_backend_postscript:n . 372
- \c__iow_wrap_newline_marker_tl .. __kernel_backend_scope_begin: . 372
- 654, 10624, 10723 __kernel_backend_scope_end: . 372
- __iow_wrap_next:nw __kernel_chk_cs_exist:N
.. 10736, 10743, 10757, 10815, 10857 364, 1477,
- __iow_wrap_next_line:w 39083, 39084, 39085, 39101, 39141,
..... 10809, 10850, 10850 39622, 39691, 39695, 39699, 39703
- __iow_wrap_start:w __kernel_chk_defined:NTF
..... 10703, 10715, 10726 365, 2274, 2274, 2293,
- __iow_wrap_store_do:n 8358, 10300, 10544, 13240, 13275,
.. 10808, 10895, 10902, 10905, 10905 18314, 18366, 23190, 30322, 30339
- \l__iow_wrap_tl __kernel_chk_expr:nNnN
..... 654, 659, 660, 10623, .. 365, 1479, 39182, 39184, 39193,
10685, 10700, 10705, 10707, 10710, 39194, 39810, 39870, 39918, 39923,
10712, 10715, 10731, 10909, 10911 39953, 39959, 39977, 39991, 39995,
10869, 10895, 10902, 10917, 10919 39999, 40007, 40011, 40015, 40028
- __iow_wrap_trim:N ... 660, 10838, __kernel_chk_flag_exist:NN
10869, 10895, 10902, 10917, 10919 364, 39083,
__iow_wrap_trim:w 10917, 10920, 10921 39086, 39110, 39142, 39610, 39636
- __iow_wrap_trim_aux:w __kernel_chk_if_free_cs:N
..... 10917, 10922, 10923 615, 898,
__iow_wrap_unindent:n . 10875, 10883 1979, 1979, 1987, 1988, 1994, 2058,
\c__iow_wrap_unindent_marker_tl . 8285, 12134, 12140, 13446, 16349,
..... 10624, 10660 16677, 17552, 17573, 19335, 19337,
19347, 19972, 19978, 20754, 21183,
21274, 22778, 23002, 30165, 30171,
30378, 30398, 30654, 34638, 39658
- \itshape 34293 __kernel_chk_tl_type:NnnTF
..... 365, 839, 888, 939, 941, 7263,
13273, 13273, 14156, 14163, 17424,
19028, 20642, 20722, 20732, 25459
- J**
- \j 33659, 34356, 34526, 34605 __kernel_chk_var_exist:N .. 364,
1477, 39083, 39083, 39092, 39128,
39134, 39140, 39394, 39414, 39415
- \jcharwidowpenalty 1157
- \jfam 1158
- \jfont 1159
- \jis 1160
- \jobname 286
- K**
- \k 32057, 34380, 34404, 34479,

- __kernel_chk_var_global:N
 364, 1477,
 39083, 39088, 39131, 39144, 39513
- __kernel_chk_var_local:N
 364, 1477,
 39083, 39087, 39125, 39143, 39433
- __kernel_chk_var_scope:NN
 364, 1477, 39083,
 39089, 39120, 39145, 39594, 39626,
 39630, 39635, 39641, 39645, 39649
- __kernel_codepoint_case:nn
 371, 14110, 31312, 31312, 32544
- __kernel_codepoint_data:nn
 371, 30946, 31218, 31218, 31249, 31332
- __kernel_codepoint_to_bytes:n
 365, 15480,
 30803, 30835, 30863, 30863, 38944
- \l__kernel_color_stack_int 373
- __kernel_cs_parm_from_arg_-
 count:nnTF
 365, 1641, 2075, 2075, 2122
- __kernel_debug_log:n
 365, 1479, 39174, 39176, 39180,
 39181, 39655, 39664, 39677, 39685
- __kernel_dependency_version_-
 check:Nn 366, 11624, 11624
- __kernel_dependency_version_-
 check:nn 366, 11624, 11625, 11626
- __kernel_deprecation_code:nn
 366, 1465, 1480,
 1583, 1585, 38748, 38774, 38781, 38782
- __kernel_deprecation_error:Nnn
 1465, 38751, 38784, 38784
- __kernel_exp_not:w 366, 415,
 446, 463, 715, 735, 938, 2670, 2670,
 2672, 2674, 2676, 2679, 2684, 4073,
 12141, 12167, 12168, 12175, 12176,
 12190, 12192, 12194, 12196, 12208,
 12213, 12218, 12224, 12225, 12232,
 12233, 12239, 12244, 12249, 12255,
 12256, 12263, 12264, 12280, 12284,
 12289, 12295, 12296, 12303, 12304,
 12308, 12312, 12317, 12323, 12324,
 12331, 12332, 12510, 12815, 12820,
 12874, 12879, 12888, 13083, 13246,
 13324, 13330, 13345, 13363, 13401,
 13452, 13457, 13462, 13467, 17675,
 20620, 20635, 21353, 30789, 30828,
 30842, 30859, 31715, 32099, 34060
- \l__kernel_expl_bool
 105, 108, 122, 135, 1391
- \c__kernel_expl_date_tl
 680, 1391, 11628, 11631, 11667, 11671
- __kernel_file_input_pop:
 366, 11433, 11473
- __kernel_file_input_push:n
 366, 11433, 11467
- __kernel_file_missing:n
 366, 10215, 11428, 11428, 11437
- __kernel_file_name_quote:n
 641, 10258, 10507,
 11045, 11045, 11086, 11449, 11495
- __kernel_file_name_sanitize:n
 366, 675, 10486, 10971,
 10971, 11100, 11431, 11489, 11508
- __kernel_group_show:NN
 2315, 2316, 2318, 2319
- __kernel_if_debug:TF
 1568, 1568, 38762, 40073, 40073
- __kernel_int_add:nnn
 366, 17543, 17543, 29367
- __kernel_intarray_gset:Nnn
 367, 1007, 1009,
 1012, 6600, 6706, 6709, 7407, 7479,
 7481, 7487, 7495, 7497, 7500, 7621,
 7623, 7627, 7629, 7641, 7644, 22866,
 22867, 22937, 23007, 23019, 23044,
 23044, 23059, 23127, 23175, 30073,
 30074, 30076, 30080, 30081, 30082,
 30401, 30402, 30436, 30439, 31158
- __kernel_intarray_gset_range_-
 from_clist:Nnn 367,
 6769, 22964, 22965, 23165, 23165
- __kernel_intarray_item:Nn 367,
 1008, 1013, 1195, 4319, 6717, 6744,
 6828, 6829, 6853, 6854, 6861, 6868,
 6925, 6929, 6948, 7484, 7675, 7894,
 22890, 22915, 23096, 23096, 23108,
 23142, 23161, 27654, 27660, 27663,
 27666, 28399, 28402, 28405, 28408,
 28411, 28414, 28417, 28420, 28423,
 30122, 30123, 30124, 30494, 30497
- __kernel_intarray_range_to_-
 clist:Nnn 367,
 6698, 22949, 22950, 23146, 23146
- __kernel_ior_open:Nn 367,
 641, 10222, 10239, 10239, 10250, 10273
- __kernel_iow_open:Nn
 367, 10483, 10487, 10490, 10501, 10523
- __kernel_iow_with:Nnn 368, 618,
 650, 728, 9385, 9387, 9591, 9593,
 10573, 10573, 10588, 13260, 13262
- __kernel_kern:n 368, 1391, 34634,
 34634, 35006, 35297, 35306, 35328,
 35330, 35379, 35381, 35984, 36242,
 36247, 36329, 36330, 36608, 36609

- \l__kernel_keyval_allow_blank_
 keys_bool 924,
 20165, 20171, 20173, 21347, 21508
- __kernel_msg_error:nnn .. 9762, 9768
- __kernel_msg_error:nnnn . 9762, 9770
- __kernel_msg_error:nnnnn 9762, 9772
- __kernel_msg_expandable_
 error:nnn 9774, 9774, 9776
- __kernel_msg_expandable_
 error:nnnn 9774, 9778
- __kernel_msg_info:nnnn .. 9762, 9762
- __kernel_msg_log_eval:Nn
 368, 8351, 9753, 9755,
 18263, 21172, 21265, 21333, 25531
- __kernel_msg_new:nnn 9758, 9760, 9986
- __kernel_msg_new:nnnn ... 9758, 9758
- __kernel_msg_show_eval:Nn
 368, 8349, 9753, 9753,
 18259, 21168, 21261, 21329, 25529
- __kernel_msg_warning:nnn 9762, 9764
- __kernel_msg_warning:nnnn 9762, 9766
- __kernel_patch:Nn
 39784, 39806, 39867, 39915,
 39950, 39974, 39988, 40004, 40019
- __kernel_patch:nnn
 1482, 39297, 39298, 39393, 39412,
 39432, 39512, 39593, 39609, 39621,
 39625, 39629, 39633, 39640, 39644,
 39648, 39652, 39674, 39682, 39707,
 39717, 39724, 39731, 39744, 39748,
 39752, 39759, 39766, 39770, 39777
- __kernel_patch_aux:Nn . 39788, 39790
- __kernel_patch_aux:nnn
 39297, 39302, 39304
- __kernel_patch_cond:nn .. 39946,
 39967, 39969, 39970, 39985, 39986
- __kernel_patch_deprecation:nnNNpn
 1465, 38744, 38744, 38802, 38805,
 38825, 38828, 38831, 38835, 38837,
 38841, 38847, 38857, 38859, 38861,
 38863, 38866, 38868, 38870, 38872,
 38874, 38876, 38878, 38880, 38882,
 38886, 38888, 38890, 38892, 38894,
 38897, 38900, 38903, 38907, 38910,
 38913, 38916, 38919, 38922, 38925,
 38927, 38929, 38931, 38936, 38938,
 38940, 38943, 38945, 38947, 38949,
 38951, 38953, 38955, 38957, 38959,
 38961, 38963, 38965, 38967, 38969,
 38971, 38973, 38975, 38979, 38981,
 38992, 38998, 39004, 39012, 39014
- __kernel_patch_eval:nn
 39802, 39818, 39830, 39841,
 39852, 39863, 39878, 39882, 39892,
 39899, 39906, 39911, 39931, 39938
- __kernel_patch_weird:nnn
 1483, 39297, 39360,
 39660, 39690, 39694, 39698, 39702
- __kernel_patch_weird_aux:nnn ...
 39297, 39364, 39366
- __kernel_pdf_object_id:n
 368, 38463, 38480
- __kernel_pdf_object_id_indexed:nn
 368, 38542, 38560
- __kernel_prefix_arg_replacement:wN
 2336, 2338, 2346, 2355, 2364
- \g__kernel_prg_map_int
 368, 458, 712,
 853, 950, 1391, 3892, 3894, 3903,
 4020, 4022, 4030, 4042, 8626, 10398,
 10400, 10407, 12712, 12714, 12716,
 12721, 13673, 13675, 13679, 13684,
 17213, 17214, 17220, 17221, 17271,
 17273, 17275, 17277, 17871, 17874,
 17882, 17885, 17896, 18780, 18782,
 18784, 18789, 20569, 20570, 20575,
 20577, 21007, 21010, 21014, 21017,
 21028, 25812, 25815, 25819, 25822,
 25833, 34046, 34048, 34050, 34052
- __kernel_primitive:NN
 338, 143, 143, 148, 149, 150, 151,
 152, 153, 154, 155, 156, 157, 158,
 159, 160, 161, 162, 163, 164, 165,
 166, 167, 168, 169, 170, 171, 172,
 173, 174, 175, 176, 177, 178, 179,
 180, 181, 182, 183, 184, 185, 186,
 187, 188, 189, 190, 191, 192, 193,
 194, 195, 196, 197, 198, 199, 200,
 201, 202, 203, 204, 205, 206, 207,
 208, 209, 210, 211, 212, 213, 214,
 215, 216, 217, 218, 219, 220, 221,
 222, 223, 224, 225, 226, 227, 228,
 229, 230, 231, 232, 233, 234, 235,
 236, 237, 238, 239, 240, 241, 242,
 243, 244, 245, 246, 247, 248, 249,
 250, 251, 252, 253, 254, 255, 256,
 257, 258, 259, 260, 261, 262, 263,
 264, 265, 266, 267, 268, 269, 270,
 271, 272, 273, 274, 275, 276, 277,
 278, 279, 280, 281, 282, 283, 284,
 285, 286, 287, 288, 289, 290, 291,
 292, 293, 294, 295, 296, 297, 298,
 299, 300, 301, 302, 303, 304, 305,
 306, 307, 308, 309, 310, 311, 312,
 313, 314, 315, 316, 317, 318, 319,
 320, 321, 322, 323, 324, 325, 326,
 327, 328, 329, 330, 331, 332, 333,

334, 335, 336, 337, 338, 339, 340,
341, 342, 343, 344, 345, 346, 347,
348, 349, 350, 351, 352, 353, 354,
355, 356, 357, 358, 359, 360, 361,
362, 363, 364, 365, 366, 367, 368,
369, 370, 371, 372, 373, 374, 375,
376, 377, 378, 379, 380, 381, 382,
383, 384, 385, 386, 387, 388, 389,
390, 391, 392, 393, 394, 395, 396,
397, 398, 399, 400, 401, 402, 403,
404, 405, 406, 407, 408, 409, 410,
411, 412, 413, 414, 415, 416, 417,
418, 419, 420, 421, 422, 423, 424,
425, 426, 427, 428, 429, 430, 431,
432, 433, 434, 435, 436, 437, 438,
439, 440, 441, 442, 443, 444, 445,
446, 447, 448, 449, 450, 451, 452,
453, 454, 455, 456, 457, 458, 459,
460, 461, 462, 463, 464, 465, 466,
467, 468, 469, 470, 471, 472, 473,
474, 475, 476, 477, 478, 479, 480,
481, 482, 483, 484, 485, 486, 487,
488, 489, 490, 491, 492, 493, 494,
495, 496, 497, 498, 499, 500, 501,
502, 503, 504, 505, 506, 507, 508,
509, 510, 511, 512, 513, 514, 515,
516, 517, 518, 519, 520, 521, 522,
523, 524, 525, 526, 527, 528, 529,
530, 531, 532, 533, 534, 535, 536,
537, 538, 539, 540, 541, 542, 543,
544, 545, 546, 547, 548, 549, 550,
551, 552, 553, 554, 555, 556, 557,
558, 559, 560, 561, 562, 563, 565,
566, 567, 568, 569, 570, 571, 572,
573, 574, 576, 577, 578, 579, 580,
581, 582, 583, 584, 585, 586, 587,
588, 589, 590, 591, 592, 593, 594,
595, 596, 597, 598, 599, 600, 601,
602, 603, 604, 605, 606, 607, 608,
610, 612, 614, 615, 616, 617, 618,
619, 620, 621, 622, 623, 624, 625,
626, 627, 629, 630, 631, 632, 633,
634, 635, 636, 637, 638, 639, 640,
641, 642, 643, 644, 645, 646, 647,
648, 649, 650, 651, 652, 653, 654,
655, 656, 657, 658, 659, 660, 661,
662, 663, 664, 665, 666, 667, 668,
669, 670, 671, 672, 673, 674, 675,
676, 677, 678, 679, 680, 681, 682,
683, 684, 685, 690, 699, 700, 701,
702, 703, 704, 706, 707, 708, 709,
710, 711, 712, 713, 714, 715, 716,
718, 720, 722, 723, 724, 726, 727,
728, 729, 730, 731, 733, 735, 736,
738, 740, 741, 742, 743, 744, 745,
746, 747, 748, 749, 750, 751, 752,
753, 754, 755, 756, 757, 758, 759,
760, 761, 762, 763, 764, 765, 767,
769, 770, 771, 772, 773, 774, 775,
776, 777, 778, 779, 780, 781, 782,
783, 784, 786, 787, 789, 790, 791,
792, 793, 794, 795, 796, 797, 798,
800, 801, 803, 804, 805, 806, 807,
809, 810, 811, 812, 813, 814, 815,
816, 817, 818, 819, 820, 821, 822,
823, 824, 826, 827, 828, 829, 830,
831, 832, 833, 834, 835, 836, 837,
838, 839, 840, 841, 842, 843, 844,
845, 846, 847, 848, 849, 850, 851,
852, 853, 854, 855, 856, 857, 858,
859, 860, 861, 862, 863, 864, 865,
866, 867, 868, 869, 870, 871, 872,
873, 874, 875, 876, 878, 880, 881,
882, 883, 884, 885, 886, 887, 888,
889, 890, 891, 892, 893, 894, 895,
896, 897, 898, 899, 900, 901, 902,
903, 904, 905, 906, 907, 908, 909,
910, 911, 912, 913, 914, 915, 916,
917, 918, 919, 920, 922, 923, 924,
925, 926, 927, 928, 929, 930, 931,
932, 933, 934, 935, 936, 937, 938,
939, 940, 942, 944, 946, 947, 948,
949, 950, 951, 952, 953, 954, 955,
956, 957, 958, 959, 960, 961, 962,
963, 964, 965, 966, 967, 968, 969,
970, 971, 972, 973, 974, 975, 976,
977, 978, 979, 980, 981, 982, 983,
984, 985, 986, 987, 988, 989, 990,
992, 994, 995, 996, 997, 999, 1000,
1001, 1002, 1004, 1005, 1007, 1009,
1010, 1011, 1012, 1013, 1015, 1017,
1018, 1019, 1020, 1022, 1023, 1024,
1025, 1026, 1027, 1028, 1029, 1030,
1031, 1032, 1033, 1034, 1035, 1036,
1037, 1038, 1039, 1040, 1041, 1042,
1043, 1044, 1045, 1046, 1047, 1048,
1049, 1050, 1051, 1052, 1053, 1054,
1055, 1056, 1057, 1058, 1059, 1061,
1063, 1064, 1066, 1068, 1069, 1070,
1071, 1073, 1074, 1075, 1077, 1079,
1081, 1082, 1083, 1084, 1085, 1086,
1087, 1088, 1089, 1090, 1091, 1092,
1094, 1096, 1097, 1098, 1099, 1100,
1101, 1102, 1103, 1104, 1105, 1106,
1107, 1108, 1109, 1110, 1111, 1112,
1114, 1116, 1117, 1118, 1119, 1120,
1121, 1122, 1123, 1124, 1125, 1126,
1127, 1128, 1129, 1130, 1131, 1132,

- 1133, 1134, 1135, 1136, 1137, 1138,
- 1139, 1140, 1141, 1142, 1143, 1144,
- 1145, 1146, 1147, 1148, 1149, 1150,
- 1151, 1152, 1153, 1154, 1155, 1156,
- 1157, 1158, 1159, 1160, 1161, 1162,
- 1163, 1164, 1165, 1166, 1167, 1168,
- 1169, 1170, 1171, 1172, 1173, 1174,
- 1175, 1176, 1177, 1178, 1179, 1180,
- 1181, 1183, 1185, 1186, 1187, 1188,
- 1189, 1191, 1192, 1193, 1194, 1195,
- 1196, 1197, 1198, 1199, 1200, 1201,
- 1202, 1203, 1204, 1205, 1206, 1207,
- 1208, 1209, 1210, 1211, 1212, 1213,
- 1214, 1215, 1216, 1217, 1218, 1219
- _kernel_quark_new_conditional:Nn
 - 370, 4362, 10966, 12353,
 - 16449, 16469, 19061, 21600, 31388
- _kernel_quark_new_test:N
 - 369, 811, 812, 814, 815,
 - 8327, 10969, 10970, 12352, 13411,
 - 13412, 16449, 16449, 17460, 17461,
 - 19911, 31393, 31394, 34057, 39030
- _kernel_randint:n
 - 370, 1222, 1226,
 - 29174, 29174, 29186, 29344, 29429
- _kernel_randint:nn
 - 370, 29348, 29352, 29352, 29427
- \c_kernel_randint_max_int
 - 1226, 1391, 29173, 29342, 29426
- _kernel_register_log:N
 - 370, 2283,
 - 2287, 2289, 2290, 18260, 21169,
 - 21170, 21262, 21263, 21330, 21331
- _kernel_register_show:N
 - 370, 727, 2283, 2283,
 - 2285, 2286, 18256, 21165, 21258, 21326
- _kernel_register_show_aux:NN ..
 - 2283, 2284, 2288, 2291
- _kernel_register_show_aux:nNN .
 - 2283, 2295, 2299
- _kernel_show:NN
 - 2301, 2301, 2304, 2307
- _kernel_str_to_other:n 371, 741,
- 743, 748, 13729, 13729, 13781, 13842
- _kernel_str_to_other_fast:n ...
 - ... 371, 4571, 5782, 10611, 10707,
 - 13680, 13700, 13752, 13752, 14370
- _kernel_str_to_other_fast_
loop:w 13752
- _kernel_sys_configuration_
load:n 603, 8787, 8853
- _kernel_sys_everyjob:
 - 370, 8973, 8973, 9167
- _kernel_tl_gset:Nn
 - ... 371, 565, 566, 694, 735, 3226,
 - 3774, 4570, 5780, 7516, 7527, 7591,
 - 7735, 9191, 12130, 12131, 12173,
 - 12194, 12196, 12238, 12243, 12248,
 - 12253, 12261, 12308, 12311, 12316,
 - 12321, 12329, 12457, 12461, 12829,
 - 13103, 13377, 13443, 13456, 13466,
 - 13479, 13483, 14304, 14320, 14370,
 - 14381, 14500, 14552, 14563, 14721,
 - 14770, 14823, 14829, 15062, 15262,
 - 15419, 16713, 16718, 16736, 16740,
 - 16781, 16808, 16848, 16877, 16883,
 - 16942, 17091, 17134, 17322, 17332,
 - 18453, 18480, 18499, 18548, 18584,
 - 18627, 18666, 25423, 39431, 39570
- _kernel_tl_set:Nn 371, 4460, 5350,
5355, 5626, 5695, 7670, 7703, 10246,
10485, 10497, 10610, 10685, 10688,
10689, 10705, 10710, 10868, 10888,
10907, 10909, 11213, 11340, 11355,
12130, 12130, 12165, 12190, 12192,
12207, 12212, 12217, 12222, 12230,
12280, 12283, 12288, 12293, 12301,
12455, 12459, 12827, 13101, 13372,
13389, 13441, 13451, 13461, 13477,
13481, 14259, 16703, 16708, 16734,
16738, 16760, 16779, 16800, 16846,
16875, 16881, 16940, 17047, 17072,
17089, 17103, 17131, 17320, 17330,
18451, 18478, 18497, 18546, 18582,
18625, 18664, 22229, 22230, 22231,
22335, 22542, 25421, 39075, 39218,
39220, 39222, 39430, 39489, 39711
- _kernel_tl_to_str:w
 - 371, 711, 735,
 - 1418, 1420, 12570, 12664, 12762,
 - 13441, 13443, 13447, 13452, 13457,
 - 13462, 13467, 13655, 13723, 16447
- keys commands:
- \l_keys_choice_int . 241, 244, 246,
248, 21568, 21767, 21770, 21775, 21776
- \l_keys_choice_tl
 - 241, 244, 246, 248, 21568, 21774
- \keys_define:nn
 - 240, 9951, 21616, 21616, 21618
- \keys_if_choice_exist:nmn 22651
- \keys_if_choice_exist:nmnTF
 - 251, 22651
- \keys_if_choice_exist_p:nmn
 - 251, 22651
- \keys_if_exist:nn 22643, 22650
- \keys_if_exist:nmnTF
 - 251, 1001, 22643, 22668

- \keys_if_exist_p:nn [251](#), [22643](#)
- \l_keys_key_str [249](#), [21572](#), 21841,
21842, 22344, 22345, 22441, 22445,
22470, 22473, 22474, 22522, 22579
- \l_keys_key_tl
. [21573](#), 21841, 21842, 22345
- \keys_log:nn [251](#), [22659](#), 22661
- \l_keys_path_str
. [249](#), [976](#), [21577](#), 21645,
21663, 21682, 21699, 21717, 21719,
21721, 21724, 21736, 21739, 21743,
21751, 21753, 21754, 21757, 21772,
21788, 21801, 21805, 21816, 21819,
21827, 21833, 21837, 21840, 21844,
21855, 21857, 21859, 21862, 21873,
21882, 21887, 21889, 21904, 21914,
21920, 21924, 21939, 21948, 21990,
22001, 22044, 22335, 22343, 22381,
22384, 22420, 22424, 22429, 22438,
22452, 22454, 22455, 22459, 22467,
22502, 22532, 22555, 22567, 22576
- \l_keys_path_tl . [21578](#), 21682, 21743
- \keys_precompile:nnN [251](#), [22315](#), 22315
- \keys_set:nn [240](#), [242](#), [243](#),
[248–251](#), [22247](#), 22247, 22257, 22319
- \keys_set_exclude_groups:nnn . . .
. [250](#), [22275](#), 22289, 22294, 38825, 38826
- \keys_set_exclude_groups:nnnN . . .
. [250](#), [22275](#), 22286, 22288, 38828, 38829
- \keys_set_exclude_groups:nnnnN . .
. [250](#), [22275](#), 22275,
22285, 22287, 22291, 38831, 38832
- \keys_set_filter:nnn
. [38825](#), 38826, 38827
- \keys_set_filter:nnnN
. [38825](#), 38829, 38830
- \keys_set_filter:nnnnN
. [38825](#), 38832, 38833
- \keys_set_groups:nnn
. [251](#), [22275](#), 22309, 22314
- \keys_set_groups:nnnN
. [251](#), [22275](#), 22306, 22308
- \keys_set_groups:nnnnN
. [251](#), [22275](#), 22295, 22305, 22307, 22311
- \keys_set_known:nn
. [249](#), [22258](#), 22272, 22274
- \keys_set_known:nnN
. [249](#), [22258](#), 22269, 22271
- \keys_set_known:nnnnN
. [249](#), [22258](#), 22258, 22268, 22270, 22273
- \keys_show:nn [251](#), [22659](#), 22659
- \l_keys_usage_load_prop
. [248](#), [21594](#), 21958, 21965, 21972
- \l_keys_usage_preamble_prop
. [248](#), [21594](#), 21960, 21967, 21974
- \l_keys_value_tl [249](#),
[21588](#), 21843, 21844, 21939, 22423,
22427, 22433, 22444, 22455, 22474,
22494, 22498, 22524, 22534, 22562
- keys internal commands:
 - __keys_bool_set:Nn
. [21707](#), 21707, 21709,
21728, 22016, 22018, 22020, 22022
 - __keys_bool_set:Nnnn
. [21707](#), 21708, 21711, 21713
 - __keys_bool_set_inverse:Nn
. [21707](#), 21710,
21712, 22024, 22026, 22028, 22030
 - __keys_check_forbidden: [21907](#), 21934
 - __keys_check_groups: 22385, 22393
 - __keys_check_required: [21907](#), 21943
 - \c_keys_check_root_str [21561](#),
21914, 21920, 21924, 22454, 22473
 - __keys_choice_find:n
. [21730](#), [22573](#), 22573, 22589
 - __keys_choice_find:nn
. [22573](#), 22576, 22578, 22582
 - __keys_choice_make: 21716,
[21729](#), 21729, 21761, 21854, 22032
 - __keys_choice_make:N
. [21729](#), 21730, 21732, 21733
 - __keys_choice_make_aux:N
. [21729](#), 21745, 21747, 21749
 - __keys_choices_make:nn
. [21760](#), 21760,
22034, 22036, 22038, 22040, 22042
 - __keys_choices_make:Nnn
. [21760](#), 21761, 21763, 21764
 - __keys_cmd_set:nn 21717, 21719,
[21771](#), [21781](#), 21781, 21783, 21855,
21857, 21859, 21889, 22001, 22044
 - __keys_cmd_set_direct:nn
. [21721](#), 21753, 21754, [21781](#),
21782, 21784, 21873, 21881, 39680
 - \c_keys_code_root_str
. [997](#), [21561](#), 21785, 21788,
21837, 22452, 22470, 22486, 22512,
22584, 22646, 22655, 22676, 39676
 - __keys_cs_set:NNpn [21786](#),
21786, 21795, 22054, 22056, 22058,
22060, 22062, 22064, 22066, 22068
 - __keys_cs_undefine:N [21608](#),
21608, 21800, 21815, 21903, 21923
 - __keys_default_inherit:
. [22416](#), 22430, 22435
 - \c_keys_default_root_str
. [21561](#), 21801, 21805, 22420,

- 22424, 22441, 22445, 22491, 22495
- __keys_default_set:n
 - 21726, 21796, 21796, 21864,
 - 22070, 22072, 22074, 22076, 22078
- __keys_define:n . 21622, 21626, 21626
- __keys_define:nn 21622, 21626, 21631
- __keys_define:nnn
 - 21616, 21617, 21619, 21625
- __keys_define_aux:nn
 - 21626, 21629, 21634, 21636
- __keys_define_code:n
 - 21640, 21690, 21690
- __keys_define_code:w
 - 21690, 21694, 21705
- \l__keys_exclude_bool
 - 21583, 22225, 22253, 22264,
 - 22281, 22301, 22388, 22406, 22411
- __keys_execute: 22349,
 - 22389, 22408, 22412, 22450, 22450
- __keys_execute:nn
 - 21844, 22450, 22455, 22474, 22498,
 - 22506, 22507, 22508, 22585, 22586
- __keys_execute_inherit:
 - 21834, 22450, 22460, 22464
- __keys_execute_unknown:
 - 996, 22450, 22461, 22478, 22480
- __keys_find_key_module:wNN
 - .. 21839, 21887, 22323, 22343, 22353
- __keys_find_key_module_auxi:Nw .
 - .. 22323, 22355, 22358, 22366, 22371
- __keys_find_key_module_auxii:Nw
 - 22323, 22355, 22362, 22363
- __keys_find_key_module_auxiii:Nn
 - 22323
- __keys_find_key_module_auxiii:Nw
 - 22366, 22368
- __keys_find_key_module_auxiv:Nw
 - 22323, 22356, 22373, 22375
- \l__keys_groups_clist 995, 21570,
 - 21812, 21813, 21820, 22383, 22398
- \c__keys_groups_root_str
 - .. 21561, 21816, 21819, 22381, 22384
- __keys_groups_set:n
 - 21810, 21810, 22096
- __keys_inherit:n 21823, 21823, 22098
- \l__keys_inherit_clist
 - 21571, 21826, 21828
- \c__keys_inherit_root_str
 - 21561, 21827,
 - 21833, 22429, 22438, 22459, 22467
- \l__keys_inherit_str 21579,
 - 21836, 22342, 22472, 22575, 22579
- __keys_initialise:n 21830, 21830,
 - 22100, 22102, 22104, 22106, 22108
- __keys_legacy_if_inverse:nn . 21848
- __keys_legacy_if_inverse:nnnn 21848
- __keys_legacy_if_set:nn
 - 21848, 21848, 22118, 22120
- __keys_legacy_if_set:nnnn
 - 21849, 21851, 21852
- __keys_legacy_if_set_inverse:nn
 - 21850, 22122, 22124
- __keys_meta_make:n
 - 21871, 21871, 22126
- __keys_meta_make:nn
 - 21871, 21879, 22128
- \l__keys_module_str 21574,
 - 21617, 21621, 21623, 21665, 21876,
 - 21987, 21994, 22240, 22243, 22245,
 - 22326, 22331, 22341, 22344, 22350,
 - 22486, 22491, 22495, 22498, 22502
- __keys_multichoice_find:n
 - 21732, 22573, 22588
- __keys_multichoice_make:
 - 21729, 21731, 21763, 22130
- __keys_multichoices_make:nn ...
 - 21760, 21762,
 - 22132, 22134, 22136, 22138, 22140
- \l__keys_no_value_bool
 - 21575, 21628, 21633,
 - 21692, 21936, 21945, 22325, 22330,
 - 22418, 22488, 22523, 22533, 22561
- \l__keys_only_known_bool
 - 21576, 22224,
 - 22252, 22263, 22280, 22300, 22482
- __keys_parent:n
 - 21736, 21739, 21743, 21833, 22429,
 - 22438, 22459, 22467, 22590, 22590
- __keys_parent_auxi:w
 - .. 22590, 22592, 22595, 22601, 22605
- __keys_parent_auxii:w
 - 22590, 22592, 22599
- __keys_parent_auxiii:n
 - 22590, 22601, 22603
- __keys_parent_auxiv:w
 - 22590, 22593, 22607
- __keys_precompile:n
 - 21601, 21601, 21782,
 - 21790, 22689, 22691, 22697, 22698
- \l__keys_precompile_bool
 - 21592, 21603, 22317, 22320
- \l__keys_precompile_tl
 - 21592, 21604, 22318, 22321
- __keys_prop_put:Nn 21884, 21884,
 - 21897, 22150, 22152, 22154, 22156
- __keys_property_find:n
 - 21638, 21649, 21649

- __keys_property_find_auxi:w ...
..... [21649](#), [21651](#), [21655](#), [21669](#)
- __keys_property_find_auxii:w ...
..... [21649](#), [21652](#), [21659](#), [21660](#)
- __keys_property_find_auxiii:w ...
..... [21649](#), [21669](#), [21672](#), [21677](#)
- __keys_property_find_auxiv:w ...
.... [976](#), [21649](#), [21670](#), [21676](#), [21678](#)
- __keys_property_find_err:w
.. [21649](#), [21653](#), [21661](#), [21684](#), [21685](#)
- \l__keys_property_str
[21582](#), [21639](#), [21642](#), [21645](#), [21681](#),
[21687](#), [21695](#), [21696](#), [21699](#), [21702](#)
- \c__keys_props_root_str
..... [21567](#), [21639](#),
[21696](#), [21702](#), [22015](#), [22017](#), [22019](#),
[22021](#), [22023](#), [22025](#), [22027](#), [22029](#),
[22031](#), [22033](#), [22035](#), [22037](#), [22039](#),
[22041](#), [22043](#), [22045](#), [22047](#), [22049](#),
[22051](#), [22053](#), [22055](#), [22057](#), [22059](#),
[22061](#), [22063](#), [22065](#), [22067](#), [22069](#),
[22071](#), [22073](#), [22075](#), [22077](#), [22079](#),
[22081](#), [22083](#), [22085](#), [22087](#), [22089](#),
[22091](#), [22093](#), [22095](#), [22097](#), [22099](#),
[22101](#), [22103](#), [22105](#), [22107](#), [22109](#),
[22111](#), [22113](#), [22115](#), [22117](#), [22119](#),
[22121](#), [22123](#), [22125](#), [22127](#), [22129](#),
[22131](#), [22133](#), [22135](#), [22137](#), [22139](#),
[22141](#), [22143](#), [22145](#), [22147](#), [22149](#),
[22151](#), [22153](#), [22155](#), [22157](#), [22159](#),
[22161](#), [22163](#), [22165](#), [22167](#), [22169](#),
[22171](#), [22173](#), [22175](#), [22177](#), [22179](#),
[22181](#), [22183](#), [22185](#), [22187](#), [22189](#),
[22191](#), [22193](#), [22195](#), [22197](#), [22199](#),
[22201](#), [22203](#), [38809](#), [38811](#), [38813](#),
[38815](#), [38817](#), [38819](#), [38821](#), [38823](#)
- __keys_quark_if_no_value:N .. [21600](#)
- __keys_quark_if_no_value:NTF ...
..... [21600](#), [22518](#)
- __keys_quark_if_no_value_p:N . [21600](#)
- \l__keys_relative_tl [21580](#), [22210](#),
[22216](#), [22231](#), [22518](#), [22528](#), [22542](#),
[22543](#), [22547](#), [22548](#), [22556](#), [22568](#)
- __keys_reset_bool:N
.. [22205](#), [22224](#), [22225](#), [22226](#), [22233](#)
- __keys_reset_var:N [22205](#)
- \l__keys_selective_bool
..... [21583](#), [22226](#),
[22254](#), [22265](#), [22282](#), [22302](#), [22347](#)
- \l__keys_selective_clist
[995](#), [21585](#), [22209](#), [22215](#), [22230](#), [22396](#)
- __keys_set:nn
.. [21875](#), [21882](#), [22205](#), [22222](#), [22239](#)
- __keys_set:nnn . [22205](#), [22240](#), [22241](#)
- __keys_set:nnnnNn [22205](#),
[22205](#), [22249](#), [22260](#), [22277](#), [22297](#)
- __keys_set:nnnnnnNn
..... [22205](#), [22207](#), [22212](#)
- __keys_set_keyval:n
..... [22244](#), [22323](#), [22323](#)
- __keys_set_keyval:nn
..... [22244](#), [22323](#), [22328](#)
- __keys_set_keyval:nnn
.. [22323](#), [22326](#), [22331](#), [22333](#), [22352](#)
- __keys_set_selective:
..... [22323](#), [22348](#), [22379](#)
- __keys_show:n .. [22659](#), [22672](#), [22685](#)
- __keys_show:Nnn
..... [22659](#), [22660](#), [22662](#), [22663](#)
- __keys_show:Nw . [22659](#), [22704](#), [22708](#)
- __keys_show:w .. [22659](#), [22687](#), [22696](#)
- __keys_store_unused: ... [22390](#),
[22407](#), [22413](#), [22450](#), [22483](#), [22516](#)
- __keys_store_unused:w
..... [22546](#), [22567](#), [22572](#)
- __keys_store_unused_aux:
..... [22450](#), [22537](#), [22540](#)
- __keys_tmp:w [22611](#), [22623](#)
- \l__keys_tmp_bool
..... [21589](#), [22395](#), [22400](#), [22404](#)
- \l__keys_tmp_clist
.. [21586](#), [22250](#), [22273](#), [22292](#), [22312](#)
- \l__keys_tmpa_tl ... [21589](#), [21888](#),
[21987](#), [21988](#), [21992](#), [21993](#), [21995](#)
- \l__keys_tmpt_tl [21589](#),
[21888](#), [21893](#), [21899](#), [21992](#), [21993](#)
- __keys_trim_spaces:n
. [976](#), [21621](#), [21666](#), [21772](#), [22243](#),
[22339](#), [22543](#), [22584](#), [22585](#), [22610](#),
[22613](#), [22646](#), [22655](#), [22666](#), [22677](#)
- __keys_trim_spaces_auxi:w
.. [22610](#), [22615](#), [22616](#), [22625](#), [22635](#)
- __keys_trim_spaces_auxii:w [22610](#),
[22617](#), [22619](#), [22629](#), [22636](#), [22638](#)
- __keys_trim_spaces_auxiii:w ...
..... [22610](#), [22620](#), [22633](#), [22639](#)
- \c__keys_type_root_str
..... [21561](#), [21736](#), [21739](#), [21751](#)
- __keys_undefine:
..... [21825](#), [21898](#), [21898](#), [22198](#)
- \l__keys_unused_clist
..... [21587](#), [22208](#), [22214](#),
[22228](#), [22229](#), [22520](#), [22530](#), [22558](#)
- __keys_usage:n . [21952](#), [21952](#), [22200](#)
- __keys_usage:NN
..... [21952](#), [21958](#), [21960](#),
[21965](#), [21967](#), [21972](#), [21974](#), [21985](#)
- __keys_usage:w . [21952](#), [21990](#), [21997](#)

- __keys_value_or_default:n 22346, 22416, 22416
- __keys_value_requirement:nn 21807, 21907, 21907, 22012, 22202, 22204
- __keys_variable_set:NnnN 21998, 21998, 22008, 22011, 22046, 22048, 22050, 22052, 22166, 22168, 22170, 22172, 22174, 22176, 22178, 22180, 22182, 22184, 22186, 22188, 22190, 22192, 22194, 22196, 38810, 38812, 38814, 38816, 38818, 38820, 38822, 38824
- __keys_variable_set_required:NnnN 21998, 22009, 22014, 22080, 22082, 22084, 22086, 22088, 22090, 22092, 22094, 22110, 22112, 22114, 22116, 22142, 22144, 22146, 22148, 22158, 22160, 22162, 22164
- keyval commands:
 - \keyval_parse:NNn 254, 971, 21351, 21361, 21475, 21622, 22244
 - \keyval_parse:nnn 253, 254, 924, 965, 970, 20172, 21351, 21351, 21361, 21476
- keyval internal commands:
 - __keyval_blank_key_error:w 21482, 21491, 21504, 21506
 - __keyval_blank_true:w 21436, 21504, 21504
 - __keyval_clean_up_active:w 968, 21378, 21391, 21412, 21412, 21444, 21464
 - __keyval_clean_up_other:w 968, 21417, 21422, 21433, 21433
 - __keyval_end_loop_active:w 21365, 21458, 21466
 - __keyval_end_loop_other:w 969, 21375, 21458, 21458
 - __keyval_if_blank:w 21436, 21482, 21491, 21501, 21502
 - __keyval_if_empty:w 21501, 21501, 21502
 - __keyval_if_recursion_tail:w 21364, 21374, 21501, 21503
 - __keyval_key:nn 968, 21438, 21477, 21489, 21504
 - __keyval_loop_active:nnw 21356, 21362, 21362, 21465
 - __keyval_loop_other:nnw 966, 21366, 21372, 21372, 21448, 21456, 21468, 21487, 21496, 21505, 21506, 21511
 - __keyval_misplaced_equal_after_active_error:w 967, 21385, 21389, 21440, 21440
 - __keyval_misplaced_equal_in_split_error:w 21396, 21401, 21405, 21409, 21425, 21430, 21440, 21450
 - __keyval_pair:nnnn 967, 968, 21411, 21432, 21477, 21480
 - __keyval_split_active:w 966, 967, 21368, 21370, 21376, 21395, 21460
 - __keyval_split_active_auxi:w 21377, 21382, 21382, 21413, 21463
 - __keyval_split_active_auxii:w 967, 21382, 21386, 21388, 21442
 - __keyval_split_active_auxiii:w 967, 21382, 21392, 21393
 - __keyval_split_active_auxiv:w 967, 21382, 21397, 21400
 - __keyval_split_active_auxv:w 21382, 21406, 21408
 - __keyval_split_other:w 21368, 21368, 21384, 21404, 21415, 21424
 - __keyval_split_other_auxi:w 968, 21416, 21419, 21419, 21434
 - __keyval_split_other_auxii:w 21419, 21420, 21421
 - __keyval_split_other_auxiii:w 968, 21419, 21426, 21429
 - __keyval_tmp:w 969, 21349, 21473, 21478, 21499, 21520, 21558
 - __keyval_trim:nN 21392, 21411, 21420, 21432, 21438, 21504, 21519, 21522
 - __keyval_trim_auxi:w 21519, 21524, 21533, 21536, 21541
 - __keyval_trim_auxii:w 21519, 21528, 21541
 - __keyval_trim_auxiii:w 21519, 21529, 21543, 21546, 21550, 21554
 - __keyval_trim_auxiv:w 21519, 21531, 21552
 - \knaccode 673
 - \knbccode 674
 - \knbscode 675
 - \kuten 1165, 1206
- L
- \L 32066, 33650, 34346
- \l 32066, 33650, 34358
- \label 31662, 31672, 34319
- \language 288
- \LARGE 34299
- \Large 34300

- \large 34303
- \lastallocatedtoks 3187
- \lastbox 289
- \lastkern 290
- \lastlinefit 506
- \lastnamedcs 842
- \lastnodechar 1166
- \lastnodefont 1167
- \lastnodesubtype 1168
- \lastnodetype 507
- \lastpenalty 291
- \lastsavedboxresourceindex 940
- \lastsavedimageresourceindex 942
- \lastsavedimageresourcepages 944
- \lastskip 292
- \lastxpos 946
- \lastypos 947
- \latelua 843
- \lateluafunction 844
- \lccode 64, 65, 293
- \leaders 294
- \left 295
- \leftghost 845
- \lefthyphenmin 296
- \leftmarginkern 676
- \leftskip 297
- legacy commands:
 - \legacy_if:n 12100
 - \legacy_if:nTF 108, 12100
 - .legacy_if_gset:n 243, 22117
 - \legacy_if_gset:nn . 108, 12117, 12122
 - \legacy_if_gset_false:n
 - 108, 12109, 12115, 12124
 - .legacy_if_gset_inverse:n 243, 22117
 - \legacy_if_gset_true:n
 - 108, 12109, 12113, 12124
 - \legacy_if_p:n 108, 12100
 - .legacy_if_set:n 243, 22117
 - \legacy_if_set:nn .. 108, 12117, 12117
 - \legacy_if_set_false:n
 - 108, 12109, 12111, 12119
 - .legacy_if_set_inverse:n . 243, 22117
 - \legacy_if_set_true:n
 - 108, 12109, 12109, 12119
- \leqno 298
- \let 5, 140, 141, 299
- \latcharcode 846
- \letterspacefont 677
- \limits 1429, 300
- \LineBreak 41,
 - 42, 43, 44, 45, 46, 47, 48, 49, 50, 60, 62
- \linedir 847
- \linedirection 848
- \linepenalty 301
- \lineskip 302
- \lineskiplimit 303
- \linewidth 35579
- \ln 27779, 27782
- ln 275
- \localbrokenpenalty 849
- \localinterlinepenalty 850
- \lcalleftbox 855
- \lcalrightbox 856
- \loccount 10204, 10457
- \loctoks 3159, 3160, 3186
- logb 275
- \long 143, 304, 19517, 19521
- \LongText 38, 76
- \looseness 305
- \lower 306
- \lowercase 67, 307
- \lpcode 678
- ltx.pdf.object commands:
 - ltx.pdf.object_id 38486
 - ltx.utils 106, 11826
 - ltx.utils.filedump 106, 11899
 - ltx.utils.filemd5sum 106, 11920
 - ltx.utils.filemoddate 106, 11929
 - ltx.utils.filesize 107, 11982
- lua commands:
 - \lua_escape:n
 - 106, 11763, 11765, 11770, 11771, 11785
 - \lua_load_module:n
 - 106, 11772, 11773, 11796
 - \lua_now:n
 - 105, 106, 8666, 8675, 11764, 11765, 11765, 11766, 11786, 30596
 - \lua_shipout:n 105, 11765, 11768, 11796
 - \lua_shipout_e:n
 - 105, 11765, 11767, 11769, 11796
- lua internal commands:
 - \l__lua_err_msg_str ... 11772, 11778
 - __lua_escape:n . 11760, 11760, 11770
 - __lua_load_module_p:n . 11775, 12042
 - __lua_now:n 11760, 11761, 11765
 - __lua_shipout:n . 11760, 11762, 11767
- \luabytecode 851
- \luabytecodecall 852
- \luacopyinputnodes 853
- \luadef 854
- luadef 11992
- \luaescapestring 857
- \luafunction 858
- \luafunctioncall 859
- \luatexbanner 860
- \luatexrevision 861
- \luatexversion 10, 56, 862

M	
<code>\mag</code>	308
<code>\mark</code>	309
<code>\marks</code>	508
<code>\mathaccent</code>	310
<code>\mathbin</code>	311
<code>\mathchar</code>	312, 19516
<code>\mathchardef</code>	313
<code>\mathchoice</code>	314
<code>\mathclose</code>	315
<code>\mathcode</code>	316
<code>\mathcolor</code>	1428
<code>\mathdefaultsmode</code>	863
<code>\mathdelimitersmode</code>	864
<code>\mathdir</code>	865
<code>\mathdirection</code>	866
<code>\mathdisplayskipmode</code>	867
<code>\matheqdirmode</code>	868
<code>\matheqnogapstep</code>	869
<code>\mathflattenmode</code>	870
<code>\mathinner</code>	317
<code>\mathitalicsmode</code>	871
<code>\mathnolimitsmode</code>	872
<code>\mathop</code>	318
<code>\mathopen</code>	319
<code>\mathoption</code>	873
<code>\mathord</code>	320
<code>\mathpenaltiesmode</code>	874
<code>\mathpunct</code>	321
<code>\mathrel</code>	322
<code>\mathrulesfam</code>	875
<code>\mathrulesmode</code>	876
<code>\mathrulethicknessmode</code>	878
<code>\mathscriptboxmode</code>	881
<code>\mathscriptcharmode</code>	882
<code>\mathscriptsmode</code>	880
<code>\mathstyle</code>	883
<code>\mathsurround</code>	323
<code>\mathsurroundmode</code>	884
<code>\mathsurroundskip</code>	885
<code>max</code>	275
<code>\maxdeadcycles</code>	324
<code>\maxdepth</code>	325
<code>md5.HEX</code>	<u>11912</u>
<code>\mdfivesum</code>	774
<code>\mdseries</code>	34292
<code>\meaning</code>	326
<code>\medmuskip</code>	327
<code>\message</code>	328
<code>\MessageBreak</code>	60
meta commands:	
<code>.meta:n</code>	<u>243</u> , <u>22125</u>
<code>.meta:nn</code>	<u>244</u> , <u>22127</u>
<code>\middle</code>	509
<code>min</code>	275
<code>\mkern</code>	329
<code>mm</code>	280
mode commands:	
<code>\mode_if_horizontal:</code>	8612
<code>\mode_if_horizontal:TF</code>	71, <u>8612</u>
<code>\mode_if_horizontal_p:</code>	71, <u>8612</u>
<code>\mode_if_inner:</code>	8614
<code>\mode_if_inner:TF</code>	72, <u>8614</u>
<code>\mode_if_inner_p:</code>	72, <u>8614</u>
<code>\mode_if_math:</code>	8616
<code>\mode_if_math:TF</code>	72, <u>8616</u>
<code>\mode_if_math_p:</code>	72, <u>8616</u>
<code>\mode_if_vertical:</code>	8610
<code>\mode_if_vertical:TF</code>	72, <u>8610</u>
<code>\mode_if_vertical_p:</code>	72, <u>8610</u>
<code>\mode_leave_vertical:</code>	
.....	30, <u>2383</u> , <u>2383</u> , <u>36411</u> , <u>36472</u>
<code>\month</code>	330, 1296, 9011
<code>\moveleft</code>	331
<code>\moveright</code>	332
msg commands:	
<code>\msg_critical:nn</code>	84, <u>104</u> , <u>9498</u>
<code>\msg_critical:nnn</code>	84, <u>9498</u>
<code>\msg_critical:nnnn</code>	84, <u>9498</u>
<code>\msg_critical:nnnnn</code>	84, <u>9498</u>
<code>\msg_critical:nnnnnn</code>	84, <u>9498</u>
<code>\msg_critical_text:n</code>	
.....	82, <u>9407</u> , <u>9412</u> , <u>9501</u>
<code>\msg_error:nn</code>	84, <u>1956</u> , <u>1971</u> , 4872, 4906, 4954, 4957, 5428, 5699, 7127, 7211, 8780, 8863, <u>9506</u> , 9508, 10264, 10514, 30542, 30588, 35857
<code>\msg_error:nnn</code>	84, 1654, 1709, 1762, 1767, <u>1956</u> , 1969, 2167, 2279, 2746, 3006, 3485, 4912, 5135, 5527, 5540, 5579, 5612, 5726, 6900, 6907, 7119, 7225, 8884, <u>9506</u> , 9507, 9622, 9769, 10270, 10520, 11430, 11800, 12482, 13492, 14337, 14398, 16454, 16478, 16482, 16640, 16974, 19919, 20176, 21688, 21723, 21861, 21929, 21947, 22788, 23015, 23217, 23615, 29499, 29740, 29755, 29759, 29775, 29825, 29829, 29891, 29970, 29998, 30561, 30619, 30625, 34775, 35475, 36859, 36928, 37250, 37492, 37528, 37638, 37647, 37681, 37694, 37709, 37714, 37784, 37803, 37816, 37833, 37843, 37851, 38203, 38216, 38239, 39036, 39044, 39097, 39106
<code>\msg_error:nnnn</code>	84, 1645, 1685, 1781, <u>1956</u> , 1956, 1970, 1972, 1983, 2124, 2831, 3026, 3049, 3348,

- 3355, 5112, 5175, 5390, 7131, 7147,
- 8802, 8821, 8837, 9253, 9506, 9506,
- 9648, 9771, 10641, 11659, 11777,
- 14430, 16497, 19039, 20100, 20118,
- 21644, 21698, 21742, 21756, 21938,
- 21979, 22501, 22554, 23611, 35695
- \msg_error:nnnnn .. 84, 2160, 3501,
- 7532, 7725, 8366, 9506, 9773, 10652,
- 13249, 13290, 16909, 22873, 23056,
- 29785, 30045, 36900, 38791, 39168
- \msg_error:nnnnnn
- 84, 87, 2846, 2860, 7330,
- 7353, 7427, 9506, 13284, 20714, 23082
- \msg_error_text:n 82,
- 9407, 9410, 9415, 9417, 9512, 10160
- \msg_expandable_error:nn
- 88, 2693, 4600, 8597, 10148, 10168,
- 10995, 16668, 19196, 19202, 19206,
- 21103, 21446, 21454, 21510, 24143
- \msg_expandable_error:nnn ... 88,
- 2446, 4695, 5717, 9059, 9775, 9777,
- 10148, 10166, 10173, 11027, 11487,
- 11791, 12768, 17381, 17653, 17842,
- 18902, 20989, 24150, 24165, 24170,
- 24236, 24293, 24332, 24338, 24674,
- 24679, 24688, 24695, 24786, 24800,
- 24998, 25049, 25782, 39116, 39265
- \msg_expandable_error:nnnn
- 88, 4625, 7030, 9779, 10148,
- 10164, 10172, 10647, 11004, 25183,
- 25204, 25943, 29332, 29422, 39204
- \msg_expandable_error:nnnnn . 88,
- 10148, 10162, 10171, 10664, 22921,
- 23107, 23726, 29199, 30115, 38788
- \msg_expandable_error:nnnnnn .. .
- 88, 10148, 10149,
- 10163, 10165, 10167, 10169, 10170
- \msg_fatal:nn 84, 9485
- \msg_fatal:nnn 84, 9485
- \msg_fatal:nnnn 84, 9485
- \msg_fatal:nnnnn 84, 9485
- \msg_fatal:nnnnnn 84, 9485
- \msg_fatal_text:n 82, 9407, 9407, 9488
- \msg_gset:nn 38835, 38838
- \msg_gset:nnnn 38835, 38836
- \msg_if_exist:nn 9244
- \msg_if_exist:nnTF 81, 9244, 9251, 9632
- \msg_if_exist_p:nn 81, 9244
- \msg_info:nn 85, 9516
- \msg_info:nnn 85, 9516
- \msg_info:nnnn 85, 9516, 9763
- \msg_info:nnnnn 85, 9516
- \msg_info:nnnnnn 85, 86, 9516
- \msg_info_text:n
- 83, 9407, 9421, 9545, 9550
- \msg_line_context:
- 82, 616, 1973, 1973, 9310,
- 9311, 21514, 21516, 30735, 39656,
- 39667, 39677, 39686, 40037, 40063
- \msg_line_number: 82, 9310, 9310, 9315
- \msg_log:nn 86, 9553
- \msg_log:nnn 86, 9553
- \msg_log:nnnn 86, 9553
- \msg_log:nnnnn 86, 9553
- \msg_log:nnnnnn
- . 86, 940, 3914, 3928, 7250, 7260,
- 9553, 10311, 10555, 11559, 17420,
- 19024, 19045, 20638, 22662, 23186,
- 30318, 30335, 36624, 36655, 38262
- \msg_module_name:n 81,
- 83, 9320, 9426, 9443, 9443, 9451, 9519
- \g_msg_module_name_prop
- 81, 3531, 8238,
- 9434, 9445, 9446, 10174, 10182,
- 10185, 10187, 11822, 15053, 21517,
- 22755, 23594, 30362, 30769, 38429
- \msg_module_type:n
- 81-83, 9425, 9437, 9437
- \g_msg_module_type_prop
- 81, 3532, 8239,
- 9434, 9439, 9440, 10175, 10183,
- 10186, 10188, 11823, 15054, 21518,
- 22756, 23595, 30363, 30770, 38430
- \msg_new:nnn
- 81, 4261, 7952, 7954, 7959,
- 8220, 8232, 9257, 9266, 9268, 9761,
- 9890, 9923, 9934, 9936, 9938, 9940,
- 9942, 10070, 10072, 10074, 10076,
- 10078, 10080, 10082, 10084, 10091,
- 10093, 10101, 10108, 11702, 14670,
- 14672, 14681, 21513, 21515, 22748,
- 22761, 23754, 23756, 23758, 23760,
- 23762, 23764, 23766, 25381, 25383,
- 25385, 25387, 25389, 25391, 25393,
- 25395, 25397, 25399, 25401, 25403,
- 25405, 25407, 25411, 25835, 25837,
- 25839, 30347, 30353, 30360, 36681,
- 38384, 38431, 38796, 39270, 40044
- \msg_new:nnnn
- 81, 615, 3490, 3507, 3514,
- 3523, 7965, 7972, 7978, 7988, 7994,
- 8018, 8025, 8033, 8041, 8048, 8055,
- 8061, 8068, 8074, 8082, 8088, 8094,
- 8104, 8111, 8120, 8123, 8131, 8137,
- 8143, 8150, 8157, 8167, 8178, 8188,
- 8198, 8207, 8213, 8222, 8225, 9257,
- 9257, 9265, 9267, 9759, 9780, 9788,

- 9796, 9803, 9814, 9822, 9831, 9838,
- 9847, 9851, 9861, 9870, 9877, 9883,
- 9892, 9899, 9907, 9915, 9944, 9947,
- 9956, 9962, 9969, 9976, 9988, 9995,
- 10004, 10012, 10019, 10035, 10043,
- 10052, 10062, 10119, 10125, 10131,
- 10276, 11663, 11696, 11708, 11714,
- 11721, 11726, 11805, 11812, 14541,
- 14674, 14689, 14703, 14709, 14756,
- 14801, 14891, 15006, 15188, 15195,
- 15367, 22712, 22715, 22718, 22724,
- 22730, 22736, 22742, 23586, 23728,
- 23743, 29792, 29798, 29804, 29810,
- 30728, 30734, 30740, 30746, 30753,
- 36665, 36672, 36675, 38283, 38293,
- 38299, 38306, 38312, 38321, 38327,
- 38335, 38344, 38350, 38357, 38366,
- 38375, 38390, 38396, 38405, 38411,
- 38417, 38423, 40036, 40045, 40062
- \msg_none:nn 86, 9565
- \msg_none:nnn 86, 9565
- \msg_none:nnnn 86, 9565
- \msg_none:nnnnn 86, 9565
- \msg_note:nn 85, 9516
- \msg_note:nnn 85, 9516
- \msg_note:nnnn 85, 9516
- \msg_note:nnnnn 85, 9516
- \msg_redirect_class:nn 89, 9705, 9705
- \msg_redirect_module:nnn
..... 89, 9705, 9707
- \msg_redirect_name:nnn 89, 9696, 9696
- \msg_see_documentation_text:n ...
..... 83, 9443, 9449
- \msg_set:nnn
..... 81, 9257, 9276, 38837, 38838
- \msg_set:nnnn
.. 81, 9257, 9269, 9277, 38835, 38836
- \msg_show:nn 87, 9566
- \msg_show:nnn 87, 9566
- \msg_show:nnnn 87, 9566
- \msg_show:nnnnn
.. 87, 940, 1411, 3912, 3926, 7249,
7259, 9566, 10310, 10554, 11558,
17418, 19022, 19044, 20636, 22660,
23184, 30316, 30332, 36621, 38260
- \msg_show_item:n
.. 87, 9601, 9601, 17433, 19035, 19049
- \msg_show_item:nn
..... 87, 9601, 9605, 20680, 30343
- \msg_show_item_unbraced:n
..... 87, 9601, 9603
- \msg_show_item_unbraced:nn
..... 87, 642, 9601, 9612, 10318,
10562, 22670, 36640, 38273, 38281
- \msg_term:nn 86, 9553
- \msg_term:nnn 86, 9553
- \msg_term:nnnn 86, 9553
- \msg_term:nnnnn 86, 9553
- \msg_term:nnnnnn 86, 1411, 9553, 36652
- \msg_warning:nn 85, 5418, 9516
- \msg_warning:nnn .. 85, 5334, 5338,
5380, 5442, 5480, 5499, 9516, 9765
- \msg_warning:nnnn 85, 5042,
5189, 9516, 9767, 29723, 29854, 30260
- \msg_warning:nnnnn .. 85, 9516, 38764
- \msg_warning:nnnnnn .. 85, 9516, 9736
- \msg_warning_text:n
..... 82, 9407, 9419, 9540
- msg internal commands:
- _msg_chk_free:nn . 9249, 9259, 39688
- _msg_chk_if_free:nn 9249
- _msg_class_chk_exist:nTF
 .. 9619, 9619, 9634, 9701, 9711, 9716
- \l_msg_class_loop_seq . 627, 9628,
 9720, 9728, 9738, 9739, 9742, 9744
- _msg_class_new:nn 624,
 9454, 9455, 9485, 9498, 9509, 9538,
 9543, 9548, 9553, 9559, 9565, 9566
- \l_msg_class_tl 625,
 627, 9624, 9641, 9654, 9675, 9679,
 9682, 9690, 9729, 9731, 9733, 9747
- \c_msg_coding_error_text_tl ...
 9278, 9783, 9791, 9817, 9825, 9834,
 9841, 9854, 9864, 9886, 9895, 9902,
 9910, 9918, 9950, 9959, 9965, 9972,
 9979, 9991, 10015, 10022, 10038,
 10046, 10055, 10065, 40048, 40065
- \c_msg_continue_text_tl . 9278, 9327
- \c_msg_critical_text_tl . 9278, 9503
- \l_msg_current_class_tl
 627, 9624, 9636,
 9674, 9679, 9682, 9690, 9719, 9733
- _msg_expandable_error:n 637
- _msg_expandable_error:nn
 10137, 10140, 10151
- _msg_fatal_exit: .. 9485, 9491, 9493
- \c_msg_fatal_text_tl ... 9278, 9490
- \c_msg_help_text_tl ... 9278, 9337
- \l_msg_hierarchy_seq
 626, 9627, 9657, 9667, 9672
- _msg_info_aux:NNnnnnn
 9516, 9516, 9540, 9545, 9550
- \l_msg_internal_tl
 .. 9236, 9363, 9369, 9496, 9590, 9596
- _msg_interrupt:n .. 9364, 9373, 9382

- _msg_interrupt:Nnnn [9317](#)
- _msg_interrupt:NnnnN
..... [9317](#), [9487](#), [9500](#), [9511](#)
- _msg_interrupt_more_text:n ...
..... [617](#), [9346](#), [9348](#), [9371](#)
- _msg_interrupt_text:n
..... [9346](#), [9362](#), [9366](#)
- _msg_interrupt_wrap:nnn
..... [9325](#), [9335](#), [9346](#), [9346](#)
- \c_msg_more_text_prefix_tl
..... [9242](#), [9262](#), [9273](#), [9322](#), [9339](#)
- \l_msg_name_str [9237](#),
[9320](#), [9353](#), [9357](#), [9519](#), [9527](#), [9531](#)
- \c_msg_no_info_text_tl .. [9278](#), [9329](#)
- _msg_no_more_text:nnnn
..... [9317](#), [9323](#), [9345](#)
- \c_msg_on_line_text_tl .. [9278](#), [9313](#)
- _msg_redirect:nnn
..... [9705](#), [9706](#), [9708](#), [9709](#)
- _msg_redirect_loop_chk:nnn ...
..... [9705](#), [9721](#), [9726](#), [9747](#), [9751](#)
- _msg_redirect_loop_list:n
..... [9705](#), [9743](#), [9752](#)
- \l_msg_redirect_prop
..... [9626](#), [9654](#), [9699](#), [9702](#)
- \c_msg_return_text_tl
..... [9278](#), [9786](#), [9794](#), [9801](#)
- _msg_show:n .. [623](#), [9566](#), [9570](#), [9572](#)
- _msg_show:nn
..... [9566](#), [9580](#), [9583](#), [9585](#), [9586](#)
- _msg_show:w [9566](#), [9577](#), [9584](#)
- _msg_show_dot:w ... [9566](#), [9577](#), [9582](#)
- _msg_show_eval:nnN
..... [9753](#), [9754](#), [9756](#), [9757](#)
- _msg_text:n . [9407](#), [9425](#), [9426](#), [9429](#)
- _msg_text:nn
..... [9407](#), [9418](#), [9420](#), [9422](#), [9423](#)
- \c_msg_text_prefix_tl
[637](#), [9242](#), [9246](#), [9260](#), [9271](#), [9326](#),
[9336](#), [9524](#), [9556](#), [9562](#), [9569](#), [10154](#)
- \l_msg_text_str [9237](#),
[9319](#), [9351](#), [9356](#), [9518](#), [9523](#), [9530](#)
- _msg_tmp:w [10137](#), [10147](#)
- \c_msg_trouble_text_tl [9278](#)
- _msg_use:nnnnnn .. [9464](#), [9629](#), [9629](#)
- _msg_use_code:
[625](#), [9629](#), [9637](#), [9651](#), [9655](#), [9680](#), [9691](#)
- _msg_use_hierarchy:nwwN
..... [9629](#), [9658](#), [9659](#), [9665](#)
- _msg_use_none_delimit_by_s_
stop:w [9241](#), [9241](#), [9660](#), [10143](#)
- _msg_use_redirect_module:n ...
..... [626](#), [9629](#), [9662](#), [9670](#), [9683](#)
- _msg_use_redirect_name:n
..... [9629](#), [9645](#), [9652](#)
- \mskip [333](#)
- \muexpr [510](#)
- multichoice commands:
.multichoice: [244](#), [22129](#)
- multichoices commands:
.multichoices:nn [244](#), [22129](#)
- \multiply [334](#)
- \muskip [335](#), [19525](#)
- muskip commands:
\c_max_muskip [238](#), [21334](#)
- \muskip_add:Nn
[236](#), [21310](#), [21310](#), [21314](#), [39464](#), [39845](#)
- \muskip_const:Nn [236](#), [21278](#), [21278](#),
[21283](#), [21334](#), [21335](#), [39602](#), [39849](#)
- \muskip_eval:n [237](#), [21281](#),
[21322](#), [21322](#), [21329](#), [21333](#), [39908](#)
- \muskip_gadd:Nn
[236](#), [21310](#), [21312](#), [21315](#), [39545](#), [39846](#)
- .muskip_gset:N [244](#), [22141](#)
- \muskip_gset:Nn
[237](#), [21300](#), [21302](#), [21305](#), [39544](#), [39844](#)
- \muskip_gset_eq:NN
.... [237](#), [21306](#), [21308](#), [21309](#), [39547](#)
- \muskip_gsub:Nn
[237](#), [21310](#), [21318](#), [21321](#), [39546](#), [39848](#)
- \muskip_gzero:N
[236](#), [21284](#), [21286](#), [21289](#), [21293](#), [39543](#)
- \muskip_gzero_new:N
..... [236](#), [21290](#), [21292](#), [21295](#)
- \muskip_if_exist:N [21296](#), [21298](#)
- \muskip_if_exist:NTF
..... [236](#), [21291](#), [21293](#), [21296](#)
- \muskip_if_exist_p:N [236](#), [21296](#)
- \muskip_log:N [238](#), [21330](#), [21330](#), [21331](#)
- \muskip_log:n [238](#), [21330](#), [21332](#)
- \muskip_new:N [236](#),
[21272](#), [21272](#), [21277](#), [21280](#), [21291](#),
[21293](#), [21336](#), [21337](#), [21338](#), [21339](#)
- .muskip_set:N [244](#), [22141](#)
- \muskip_set:Nn
[237](#), [21300](#), [21300](#), [21304](#), [39463](#), [39843](#)
- \muskip_set_eq:NN
.... [237](#), [21306](#), [21306](#), [21307](#), [39466](#)
- \muskip_show:N [237](#), [21326](#), [21326](#), [21327](#)
- \muskip_show:n [238](#), [964](#), [21328](#), [21328](#)
- \muskip_sub:Nn
[237](#), [21310](#), [21316](#), [21320](#), [39465](#), [39847](#)
- \muskip_use:N
.... [237](#), [21323](#), [21324](#), [21324](#), [21325](#)
- \muskip_zero:N
[236](#), [21284](#), [21284](#), [21288](#), [21291](#), [39462](#)

- 23279, 23528, 23543, 23544, 23915,
23916, 23941, 25224, 25225, 25226,
25262, 26000, 26001, 26002, 26125,
26210, 26296, 26297, 26298, 26299,
26300, 26301, 26302, 26303, 26304,
26383, 26386, 26722, 26723, 26737,
26738, 26752, 27037, 27260, 27285,
27291, 27292, 27293, 27294, 27295,
27444, 27479, 27481, 27489, 27682,
27732, 27735, 27744, 27859, 27882,
27883, 27915, 27916, 27920, 27973,
27974, 28014, 28019, 28029, 28034,
28044, 28049, 28059, 28064, 28074,
28079, 28089, 28094, 28621, 28622,
28667, 28752, 28755, 28767, 28773,
28820, 28822, 28823, 28833, 28839,
28916, 28917, 28924, 28970, 28971,
28978, 29044, 29045, 29239, 30064,
30065, 30066, 30143, 30144, 30145
- \oradical 1215
\orieveryjob 1308, 1309
\oripdfoutput 1311, 1312
\outer 844, 352
\output 353
\outputbox 891
\outputmode 949
\outputpenalty 354
\over 355
\overfullrule 356
\overline 357
\overwithdelims 358
- P**
- \PackageError 67, 75
\pagebottomoffset 892
\pagedepth 359
\pagedir 893
\pagedirection 894
\pagediscards 513
\pagefillstretch 360
\pagefillstretch 361
\pagefilstretch 362
\pagefistretch 1171
\pagegoal 363
\pageheight 950
\pageleftoffset 895
\pagerightoffset 896
\pageshrink 364
\pagestretch 365
\pagetopoffset 897
\pagetotal 366
\pagewidth 951
\paperheight 38729, 38733
\paperwidth 38730, 38733
- \par .. 15–19, 93, 395, 1365, 367, 34861,
34863, 34867, 34872, 34877, 34882,
34889, 34894, 34901, 34906, 34926
- \pardir 898
\pardirection 899
\parfillskip 368
\parindent 369
\parshape 370
\parshapedimen 514
\parshapeindent 515
\parshapelength 516
\parskip 371
\partokencontext 1216
\partokenname 1217
\patterns 372
\pausing 373
pc 280
- pdf commands:
- \pdf_destination:nn 332, 38696, 38696
\pdf_destination:nmnn
..... 332, 38698, 38698
\pdf_object_id:n 368
\pdf_object_id_indexed:mn 368
\pdf_object_if_exist:n 38536
\pdf_object_if_exist:nTF . 329, 38536
\pdf_object_if_exist_p:n . 329, 38536
\pdf_object_new:n
..... 329, 38463, 38463, 38841, 38845
\pdf_object_new:nn 38841, 38842
\pdf_object_new_indexed:nn
..... 330, 38542, 38542
\pdf_object_ref:n .. 329, 38463, 38475
\pdf_object_ref_indexed:nn
..... 330, 38542, 38555
\pdf_object_ref_last:
..... 331, 38638, 38638
\pdf_object_unnamed_write:nn ...
..... 330, 38632, 38632, 38637
\pdf_object_write:n 38847
\pdf_object_write:nn
..... 38841, 38848, 38855
\pdf_object_write:nnn
..... 329, 38463, 38468, 38474
\pdf_object_write_indexed:nmnn ..
..... 330, 38542, 38548, 38554
\pdf_pageobject_ref:n
..... 331, 38639, 38639
\pdf_pagesize_gset:nn
..... 331, 38694, 38694
\pdf_uncompress: ... 331, 38454, 38454
\pdf_version: 331, 38690, 38690
\pdf_version_compare:Nn .. 331, 38641
\pdf_version_compare:NnTF
..... 331, 38641, 38679

- \pdf_version_compare_p:Nn [331](#), [38641](#)
- \pdf_version_gset:n [331](#), [38675](#), [38675](#)
- \pdf_version_major: [331](#), [38690](#), [38692](#)
- \pdf_version_min_gset:n
..... [331](#), [38675](#), [38677](#)
- \pdf_version_minor: [331](#), [38690](#), [38693](#)
- pdf internal commands:
- __pdf_backend_compress_objects:n
..... [38459](#)
- __pdf_backend_compresslevel:n [38458](#)
- __pdf_backend_destination:nn . [38697](#)
- __pdf_backend_destination:mnmn .
..... [38701](#)
- __pdf_backend_object_id:n
..... [38482](#), [38562](#)
- \g__pdf_backend_object_int
..... [38462](#), [38466](#), [38546](#)
- __pdf_backend_object_last: . . [38638](#)
- __pdf_backend_object_new:
..... [38465](#), [38544](#)
- __pdf_backend_object_now:nn . [38634](#)
- __pdf_backend_object_ref:n
..... [38477](#), [38557](#)
- __pdf_backend_object_write:nnm .
..... [38470](#), [38550](#), [38850](#)
- __pdf_backend_pageobject_ref:n .
..... [38640](#)
- __pdf_backend_pagesize_gset:nn .
..... [38695](#), [38720](#), [38732](#)
- __pdf_backend_version_major: ...
..... [38646](#), [38654](#),
[38657](#), [38666](#), [38669](#), [38691](#), [38692](#)
- __pdf_backend_version_major_-
gset:n [38686](#)
- __pdf_backend_version_minor: ...
.. [38647](#), [38658](#), [38670](#), [38691](#), [38693](#)
- __pdf_backend_version_minor_-
gset:n [38687](#)
- \g__pdf_init_bool .. [38443](#), [38456](#),
[38472](#), [38552](#), [38635](#), [38684](#), [38853](#)
- \c__pdf_object_block_size_int ...
..... [38598](#), [38619](#), [38625](#), [38628](#)
- __pdf_object_index_split:nn ...
..... [38591](#), [38606](#), [38611](#)
- \g__pdf_object_prop
..... [38840](#), [38844](#), [38852](#)
- __pdf_object_record:nN
..... [38466](#), [38486](#), [38521](#)
- __pdf_object_record:NnN
..... [38566](#), [38590](#), [38595](#)
- __pdf_object_record:nnN
.. [38545](#), [38566](#), [38586](#), [38630](#), [38630](#)
- __pdf_object_retrieve:n
..... [38471](#), [38478](#),
[38483](#), [38486](#), [38526](#), [38538](#), [38851](#)
- __pdf_object_retrieve:Nn
..... [38566](#), [38605](#), [38609](#)
- __pdf_object_retrieve:nn
..... [38551](#), [38558](#),
[38563](#), [38566](#), [38601](#), [38630](#), [38631](#)
- __pdf_version_compare_<:w [38641](#)
- __pdf_version_compare_=:w [38641](#)
- __pdf_version_compare_>:w [38641](#)
- __pdf_version_gset:w
..... [38675](#), [38676](#), [38680](#), [38682](#)
- \pdfadjustinterwordglue [627](#)
- \pdfadjustspacing [629](#)
- \pdfannot [539](#)
- \pdfappendkern [630](#)
- \pdfcatalog [540](#)
- \pdfcolorstack [542](#)
- \pdfcolorstackinit [543](#)
- \pdfcompresslevel [541](#)
- \pdfcopyfont [631](#)
- \pdfcreationdate [632](#)
- \pdfdecimaldigits [544](#)
- \pdfdest [545](#)
- \pdfdestmargin [546](#)
- \pdfdraftmode [633](#)
- \pdfeachlinedepth [634](#)
- \pdfeachlineheight [635](#)
- \pdfelapsedtime [636](#)
- \pdfendlink [547](#)
- \pdfendthread [548](#)
- \pdfescapehex [637](#)
- \pdfescapename [638](#)
- \pdfescapestring [639](#)
- \pdfextension [900](#)
- \pdffakepace [549](#)
- \pdffeedback [901](#)
- \pdffiledump [703](#)
- \pdffilemoddate [702](#)
- \pdffilesize [700](#)
- \pdffirstlineheight [640](#)
- \pdffontattr [550](#)
- \pdffontexpand [641](#)
- \pdffontname [551](#)
- \pdffontobjnum [552](#)
- \pdffontsize [642](#)
- \pdfgamma [553](#)
- \pdfgentounicode [554](#)
- \pdfglyphtounicode [555](#)
- \pdfhorigin [556](#)
- \pdfignoreddimen [643](#)
- \pdfimageapplygamma [557](#)
- \pdfimagegamma [558](#)

<code>\pdfimagehicolor</code>	559	<code>\pdfrefximage</code>	598
<code>\pdfimageresolution</code>	560	<code>\pdfresettimer</code>	661
<code>\pdfincludechars</code>	561	<code>\pdfrestore</code>	599
<code>\pdfinclusioncopyfonts</code>	562	<code>\pdfretval</code>	600
<code>\pdfinclusionerrorlevel</code>	563	<code>\pdfrunninglinkoff</code>	601
<code>\pdfinfo</code>	565	<code>\pdfrunninglinkon</code>	602
<code>\pdfinfoomitdate</code>	566	<code>\pdfsave</code>	603
<code>\pdfinsertht</code>	644	<code>\pdfsavepos</code>	662
<code>\pdfinterwordspaceoff</code>	567	<code>\pdfsetmatrix</code>	604
<code>\pdfinterwordspaceon</code>	568	<code>\pdfsetrandomseed</code>	663
<code>\pdflastannot</code>	569	<code>\pdfshellescape</code>	664
<code>\pdflastlinedepth</code>	645	<code>\pdfstartlink</code>	605
<code>\pdflastlink</code>	570	<code>\pdfstartthread</code>	606
<code>\pdflastmatch</code>	646	<code>\pdfstrcmp</code>	133, 5, 699
<code>\pdflastobj</code>	571	<code>\pdfsuppressptexinfo</code>	607
<code>\pdflastxform</code>	572	<code>\pdfsuppresswarningdupdest</code>	608
<code>\pdflastximage</code>	573	<code>\pdfsuppresswarningdupmap</code>	610
<code>\pdflastximagecolordepth</code>	574	<code>\pdfsuppresswarningpagegroup</code>	612
<code>\pdflastximagepages</code>	576	<code>\pdftebanner</code>	668
<code>\pdflastxpos</code>	647	<code>\pdftexrevision</code>	669
<code>\pdflastypos</code>	648	<code>\pdftexversion</code>	670
<code>\pdflinkmargin</code>	577	<code>\pdfthread</code>	614
<code>\pdfliteral</code>	578	<code>\pdfthreadmargin</code>	615
<code>\pdfmajorversion</code>	581	<code>\pdftracingfonts</code>	665, 1266, 1267
<code>\pdfmapfile</code>	579	<code>\pdftrailer</code>	616
<code>\pdfmapline</code>	580	<code>\pdftrailerid</code>	617
<code>\pdfmatch</code>	649	<code>\pdfunescapehex</code>	666
<code>\pdfmdfivesum</code>	701	<code>\pdfuniformdeviate</code>	667
<code>\pdfminorversion</code>	582	<code>\pdfuniqeresname</code>	618
<code>\pdfnames</code>	583	<code>\pdfvariable</code>	902
<code>\pdfnobuiltintounicode</code>	584	<code>\pdfvorigin</code>	619
<code>\pdfnoligatures</code>	650	<code>\pdfxform</code>	620
<code>\pdfnormaldeviate</code>	651	<code>\pdfxformname</code>	621
<code>\pdfobj</code>	585	<code>\pdfximage</code>	622
<code>\pdfobjcompresslevel</code>	586	<code>\pdfximagebbox</code>	623
<code>\pdfomitcharset</code>	587	peek commands:	
<code>\pdfoutline</code>	588	<code>\peek_after:Nw</code>	73, 205, 4045, 19712, 19712, 19725, 19750, 19791
<code>\pdfoutput</code>	589	<code>\peek_analysis_map_break:</code>
<code>\pdfpageattr</code>	590	...	208, 4005, 4005, 4006, 4008, 4028
<code>\pdfpagebox</code>	592	<code>\peek_analysis_map_break:n</code>
<code>\pdfpageheight</code>	652	208, 4005, 4007, 7825
<code>\pdfpageref</code>	593	<code>\peek_analysis_map_inline:n</code>	46, 205, 208, 446, 571, 4017, 4017, 7818
<code>\pdfpageresources</code>	594	<code>\peek_catcode:NTF</code>	.. 206, 19846, 37287
<code>\pdfpagesattr</code>	591, 595	<code>\peek_catcode_ignore_spaces:NTF</code> 38983
<code>\pdfpagewidth</code>	653	<code>\peek_catcode_remove:NTF</code>
<code>\pdfpkmode</code>	654	206, 19846, 37353
<code>\pdfpkresolution</code>	655	<code>\peek_catcode_remove_ignore_</code>	-
<code>\pdfprependkern</code>	657	spaces:NTF 38983
<code>\pdfprimitive</code>	656	<code>\peek_charcode:NTF</code>	206, 209, 210, 19846
<code>\pdfprotrudechars</code>	658	<code>\peek_charcode_ignore_spaces:NTF</code> 38983
<code>\pdfpxdimen</code>	659		
<code>\pdfrandomseed</code>	660		
<code>\pdfrefobj</code>	596		
<code>\pdfrefxform</code>	597		

- \peek_charcode_remove:NTF 206, 209, 19846
- \peek_charcode_remove_ignore_spaces:NTF 38983
- \peek_gafter:Nw . . . 205, 19712, 19714
- \peek_meaning:NTF . . 206, 19846, 19917
- \peek_meaning_ignore_spaces:NTF 38983
- \peek_meaning_remove:NTF . 206, 19846
- \peek_meaning_remove_ignore_spaces:NTF 38983
- \peek_N_type:TF 207, 19860, 19892, 19897, 19899
- \peek_regex:NTF 209, 7762, 7771, 7777, 7778, 7779
- \peek_regex:nTF . . . 209, 522, 570, 572, 573, 7762, 7762, 7768, 7769, 7770
- \peek_regex_remove_once:NTF 209, 7762, 7790, 7796, 7797, 7798, 7799
- \peek_regex_remove_once:nTF 209, 572, 7762, 7780, 7786, 7787, 7788, 7789
- \peek_regex_replace_once:Nn 210, 7854, 7868
- \peek_regex_replace_once:nn 210, 7854, 7860
- \peek_regex_replace_once:NnTF 210, 7854, 7862, 7864, 7865, 7866, 7867, 7869
- \peek_regex_replace_once:nnTF . . . 210, 544, 548, 570, 575, 7854, 7854, 7856, 7857, 7858, 7859, 7861
- \peek_remove_filler:n 207, 19737, 19737, 37284, 37351
- \peek_remove_spaces:n 205, 206, 19721, 19721, 38992, 38995, 38998, 39001, 39004, 39007
- \g_peek_token 205, 19701, 19715
- \l_peek_token 205, 208, 463, 466, 467, 910, 912, 913, 1429, 4050, 4051, 4052, 4053, 4139, 4192, 4195, 4242, 19701, 19713, 19730, 19755, 19758, 19769, 19807, 19819, 19839, 19866, 19867, 19868, 19871, 37300, 37307, 37321
- peek internal commands:
 - __peek_execute_branches_-
 - catcode: 913, 19813, 19813
 - __peek_execute_branches_-
 - catcode_aux: 19813, 19814, 19816, 19817
 - __peek_execute_branches_-
 - catcode_auxii:N 19813, 19821, 19827
 - __peek_execute_branches_-
 - catcode_auxiii: 19813, 19824, 19837
 - __peek_execute_branches_-
 - charcode: 913, 19813, 19815
 - __peek_execute_branches_-
 - meaning: 913, 19805, 19805
 - __peek_execute_branches_N_type:
 - . . 19860, 19863, 19895, 19898, 19900
 - __peek_false:w 913, 914, 19705, 19707, 19723, 19734, 19740, 19770, 19786, 19810, 19833, 19843, 19877, 19890
 - __peek_N_type:w . 19860, 19870, 19880
 - __peek_N_type_aux:nnw 19860, 19872, 19885
 - __peek_remove_filler: 19737, 19750, 19753
 - __peek_remove_filler:w 19737, 19739, 19746, 19748, 19772
 - __peek_remove_filler_expand:w 19737, 19763, 19767
 - __peek_remove_spaces: 19721, 19725, 19728
 - \l_peek_search_tl 909, 912, 19704, 19779, 19830, 19840
 - \l_peek_search_token 909, 19703, 19778, 19807
 - __peek_tmp:w 19705, 19708, 19719, 19861, 19883
 - __peek_token_generic:NNTF . 913, 914, 19793, 19793, 19795, 19796, 19797, 19798, 19894, 19898, 19900
 - __peek_token_generic_aux:NNNTF 19775, 19775, 19794, 19800
 - __peek_token_remove_generic:NNTF 913, 19793, 19799, 19801, 19802, 19803, 19804
 - __peek_true:w 913, 914, 19705, 19705, 19785, 19808, 19831, 19841, 19875, 19889, 19890
 - __peek_true_aux:w 910, 911, 19705, 19706, 19718, 19725, 19726, 19739, 19780, 19794
 - __peek_true_remove:w 910, 911, 19716, 19716, 19731, 19756, 19760, 19800
 - __peek_use_none_delimit_by_s_-
 - stop:w . . . 913, 19711, 19711, 19873
- \penalty 374
- \pi 24226, 24227
- pi 279
- \postbreakpenalty 1172
- \postdisplaypenalty 375
- \postexhyphenchar 903
- \posthyphenchar 904
- \prebinoppenalty 905

- `\prebreakpenalty` 1173
- `\predisplaydirection` 517
- `\predisplaygapfactor` 906
- `\predisplaypenalty` 376
- `\predisplaysize` 377
- `\preexhyphenchar` 907
- `\prehyphenchar` 908
- `\prerelpenalty` 909
- `\pretolerance` 378
- `\prevdepth` 379
- `\prevgraf` 380
- prg commands:
 - `\prg_break:` [73](#), [543](#), [780](#),
[832](#), [833](#), [2380](#), 2381, 3404, 3479,
3859, 3937, 3967, 3968, 3969, 3970,
3971, 3972, 4331, 4597, 4601, 5860,
5870, 5875, 5884, 5908, 5953, 6816,
7639, [8627](#), 13122, 14308, 14324,
14444, 14476, 14582, 14585, 14726,
14773, 14826, 14832, 15065, 15146,
15318, 15459, 17150, 17183, 17234,
17287, 17302, 17308, 17844, 18393,
18871, 23339, 23348, 25348, 25368,
25369, 25656, 25657, 25670, 25760,
25761, 25762, 29154, 29180, 29404
 - `\prg_break:n`
..... [73](#), [2380](#), 2382, 6353, 6844,
[8627](#), 13124, 14210, 14218, 14230,
17024, 17163, 17854, 18299, 23140,
23159, 23173, 23355, 29664, 29675
 - `\prg_break_point:` [73](#), [433](#),
[440](#), [822](#), [2380](#), 2380, 2381, 2382,
3231, 3273, 3397, 3404, 3777, 3938,
3974, 4327, 4575, 5856, 5905, 6354,
6686, 6838, 7633, 7639, [8627](#), 13112,
14211, 14219, 14309, 14325, 14445,
14477, 14583, 14586, 14727, 14774,
14827, 14833, 15066, 15266, 15423,
17021, 17151, 17185, 17236, 17287,
17303, 17354, 17361, 17849, 18293,
18393, 18871, 23134, 23153, 23168,
23340, 23349, 25349, 25370, 25658,
25764, 29155, 29180, 29412, 29665
 - `\prg_break_point:Nn`
..... [72](#), [73](#), [149](#), [404](#), [462](#), [473](#),
[833](#), [853](#), [950](#), [2371](#), 2371, 2372,
3902, 4028, 6578, 6592, 6637, 7824,
[8627](#), 10406, 10425, 12691, 12720,
12731, 13657, 13683, 13703, 13725,
17186, 17227, 17237, 17260, 17267,
17276, 17896, 18743, 18765, 18788,
18815, 18837, 20551, 20573, 20589,
21028, 25833, 33712, 33731, 34051
- `\prg_do_nothing:`
.. [13](#), [73](#), [492](#), [546](#), [566](#), [675](#), [700](#),
[761](#), [815](#), [821](#), [870](#), [887](#), [1039](#), [1217](#),
[2369](#), 2369, 2380, 2789, 2816, 2915,
[2916](#), 2917, 3319, 3477, 3478, 3748,
3797, 4077, 4093, 4423, 4908, 4951,
4952, 4959, 4960, 6898, 7126, 7549,
7553, 7605, 8896, 10785, 11082,
11502, 11541, 11543, 12386, 13006,
13477, 13479, 14374, 15316, 16577,
16613, 16614, 16751, 16758, 17116,
17118, 18386, 18392, 18400, 18562,
18763, 18773, 18836, 18847, 18923,
18935, 18990, 18994, 19001, 22507,
23621, 23655, 23681, 23689, 25233,
29135, 29534, 29688, 30548, 32167
- `\prg_generate_conditional_`-
variant:Nnn [33](#),
[65](#), [2995](#), 2995, 7277, 7283, 7313,
7315, 8336, 10227, 11071, 11219,
11324, 11328, 11332, 11336, 11361,
11372, 11413, 12556, 12566, 12590,
12593, 12609, 12623, 12629, 12640,
12660, 12907, 12925, 12934, 13556,
13568, 13576, 13607, 13642, 16420,
16442, 16884, 16885, 16965, 17025,
17119, 17121, 17135, 17137, 17139,
17141, 18329, 18580, 18594, 18595,
18732, 18734, 20242, 20244, 20246,
20383, 20385, 20518, 20541, 22650,
34725, 34727, 34731, 35468, 38934
- `\prg_gset_conditional:Nnn`
..... [63](#), [1618](#), 1620
- `\prg_gset_conditional:Npnn`
..... [63](#), [1597](#), 1599,
1826, 1840, 1853, 1868, 1882, 1897
- `\prg_gset_eq_conditional:Nnn` ...
..... [65](#), [1741](#), 1743
- `\prg_gset_protected_conditional:Nnn`
..... [64](#), [1618](#), 1626
- `\prg_gset_protected_conditional:Npnn`
..... [64](#), [1597](#), 1605
- `\prg_map_break:Nn`
.. [72](#), [73](#), [404](#), [462](#), [713](#), [883](#), [938](#),
[2371](#), 2372, 2378, 4006, 4008, 4322,
[8627](#), 10389, 10391, 12758, 12760,
13716, 13718, 17174, 17176, 18850,
18852, 20605, 20607, 34041, 34043
- `\prg_new_conditional:Nnn`
..... [63](#), [1618](#), 1622, [8280](#)
- `\prg_new_conditional:Npnn`
.. [63-65](#), [370](#), [719](#), [899](#), [913](#), [1597](#),
1601, 2256, 4717, 4746, 4829, 4858,
[8280](#), 8328, 8379, 8440, 8455, 8466,

- 8481, 8491, 8610, 8612, 8614, 8616,
9244, 10323, 11364, 11407, 12100,
12548, 12558, 12573, 12582, 12644,
12661, 12672, 12895, 12909, 12927,
12968, 12988, 13003, 13539, 13546,
13551, 13558, 13563, 14207, 14216,
14231, 14239, 14913, 14947, 14966,
16404, 16412, 16422, 16432, 16582,
16593, 16957, 17657, 17710, 17718,
17756, 17764, 18321, 18402, 18685,
19356, 19361, 19366, 19371, 19378,
19384, 19390, 19395, 19400, 19405,
19410, 19417, 19422, 19429, 19444,
19449, 19485, 19593, 19602, 20500,
20520, 20844, 20849, 21229, 21237,
22643, 22651, 23523, 24668, 25549,
25557, 25573, 31530, 31589, 31598,
32969, 32998, 33031, 33109, 33127,
33145, 33170, 34721, 34723, 34729,
35458, 36709, 38536, 38641, 39259
- `\prg_new_eq_conditional:NNn`
. 65, 1741, 1745, 8280, 8375, 8377,
12181, 12182, 12592, 13528, 13530,
13532, 13534, 13536, 16794, 16796,
17412, 17413, 17414, 17415, 17416,
17417, 17606, 17608, 18317, 18319,
18492, 18494, 18681, 18683, 19415,
20496, 20498, 20775, 20777, 21203,
21205, 21296, 21298, 23180, 23182,
25547, 25548, 30151, 30153, 30185,
30187, 30607, 30609, 34666, 34668
- `\prg_new_protected_conditional:Nnn`
. 64, 1618, 1628, 8280
- `\prg_new_protected_conditional:Npnn`
. 64,
1597, 1607, 4438, 7272, 7278, 7308,
7310, 8280, 8874, 10218, 10343,
10363, 11060, 11211, 11322, 11326,
11330, 11334, 11352, 12597, 12610,
12631, 13570, 13578, 14348, 14357,
16880, 16882, 17006, 17115, 17117,
17123, 17126, 17129, 17132, 18571,
18581, 18583, 18699, 18703, 20237,
20371, 20377, 29654, 30221, 30611
- `\prg_replicate:nn`
71, 118, 158, 569, 594, 1013, 4282,
4914, 5667, 6284, 6310, 6456, 6464,
6627, 6792, 6903, 7564, 7572, 7619,
7737, 7739, 8562, 8562, 9354, 9528,
10617, 23089, 26979, 27831, 28139,
28395, 28441, 28478, 29001, 29009,
29984, 30087, 30206, 31113, 31124,
31129, 31154, 37865, 37873, 37941,
37975, 37987, 37990, 38070, 38071
- `\prg_return_false:` 64, 65,
383, 561, 828, 848, 879, 931, 935,
1591, 1593, 1665, 1673, 1835, 1846,
1862, 1877, 1890, 1906, 2259, 4443,
4743, 4769, 4834, 4863, 7390, 8280,
8333, 8384, 8445, 8461, 8471, 8487,
8497, 8611, 8613, 8615, 8617, 8878,
8886, 9247, 10225, 10330, 10346,
10366, 11069, 11216, 11343, 11358,
11381, 11390, 11401, 11410, 12104,
12553, 12563, 12578, 12587, 12606,
12620, 12637, 12650, 12668, 12683,
12904, 12922, 12942, 12950, 12960,
12976, 12999, 13010, 13544, 13549,
13554, 13561, 13566, 13574, 13582,
14212, 14220, 14236, 14252, 14355,
14364, 14917, 14920, 14923, 14950,
14953, 14970, 14973, 14976, 16409,
16417, 16428, 16438, 16587, 16598,
16912, 16962, 17020, 17039, 17655,
17687, 17692, 17715, 17723, 17761,
17769, 18326, 18418, 18421, 18574,
18588, 18688, 18723, 18729, 19359,
19364, 19369, 19374, 19381, 19388,
19393, 19398, 19403, 19408, 19413,
19420, 19425, 19442, 19447, 19452,
19457, 19491, 19494, 19506, 19606,
19631, 19648, 19657, 20240, 20375,
20381, 20515, 20529, 20533, 20847,
20866, 20881, 20882, 21233, 21240,
22648, 22657, 23534, 23536, 24681,
24691, 25554, 25568, 25581, 29664,
30231, 30621, 30627, 31540, 31543,
31593, 31603, 32977, 32981, 32987,
33023, 33043, 33091, 33123, 33141,
33164, 33186, 34722, 34724, 34730,
35464, 35466, 36714, 36717, 38539,
38649, 38661, 38673, 39264, 40030
- `\prg_return_true:` . . . 64, 65, 383,
557, 561, 665, 708, 719, 828, 931,
935, 1591, 1591, 1665, 1673, 1833,
1844, 1860, 1875, 1888, 1904, 2259,
4441, 4741, 4767, 4832, 4861, 7388,
8280, 8331, 8382, 8443, 8459, 8469,
8485, 8495, 8611, 8613, 8615, 8617,
8899, 9247, 10223, 10328, 10333,
10336, 10349, 10369, 11067, 11217,
11344, 11359, 11379, 11388, 11399,
11411, 12106, 12551, 12561, 12576,
12585, 12604, 12618, 12637, 12648,
12666, 12681, 12902, 12920, 12940,
12958, 12974, 12997, 13008, 13544,
13549, 13554, 13561, 13566, 13574,
13582, 14230, 14234, 14242, 14255,

- 14355, 14364, 14917, 14923, 14955,
 14970, 14976, 16407, 16415, 16426,
 16436, 16585, 16596, 16923, 16960,
 17024, 17042, 17687, 17713, 17721,
 17759, 17767, 18324, 18414, 18417,
 18423, 18577, 18591, 18689, 18719,
 18729, 19359, 19364, 19369, 19374,
 19381, 19388, 19393, 19398, 19403,
 19408, 19413, 19420, 19425, 19441,
 19447, 19455, 19505, 19629, 19655,
 20240, 20375, 20381, 20503, 20513,
 20533, 20539, 20847, 20882, 21232,
 21241, 22647, 22656, 23527, 23532,
 24676, 24697, 25552, 25570, 25579,
 29658, 29661, 29675, 30229, 30617,
 31541, 31593, 31603, 32985, 32990,
 32994, 33006, 33009, 33012, 33015,
 33018, 33021, 33041, 33047, 33050,
 33053, 33056, 33059, 33062, 33065,
 33068, 33071, 33074, 33077, 33080,
 33083, 33086, 33089, 33118, 33121,
 33136, 33139, 33153, 33156, 33159,
 33162, 33178, 33181, 33184, 34722,
 34724, 34730, 35463, 36715, 38540,
 38648, 38660, 38672, 39263, 40031
 \prg_set_conditional:Nnn
 [63](#), [1618](#), [1618](#), [8280](#)
 \prg_set_conditional:Npnn
 . [63-65](#), [389](#), [1597](#), [1597](#), [8280](#), [40024](#)
 \prg_set_eq_conditional:NNn
 [65](#), [1741](#), [1741](#), [8280](#)
 \prg_set_protected_conditional:Nnn
 [64](#), [1618](#), [1624](#), [8280](#)
 \prg_set_protected_conditional:Npnn
 [64](#), [1597](#), [1603](#), [8280](#)
 prg internal commands:
 __prg_break_point:Nn [404](#)
 __prg_F_true:w [1694](#), [1727](#), [1739](#)
 __prg_generate_conditional:nnNNNnnn
 [1613](#), [1642](#), [1651](#), [1651](#)
 __prg_generate_conditional:NNnnnnNw
 [1651](#), [1660](#), [1677](#), [1692](#)
 __prg_generate_conditional_
 count:NNNnn [1618](#), [1619](#),
[1621](#), [1623](#), [1625](#), [1627](#), [1629](#), [1630](#)
 __prg_generate_conditional_
 count:nnNNNnn ... [1618](#), [1634](#), [1639](#)
 __prg_generate_conditional_
 fast:nw . [383](#), [384](#), [1651](#), [1664](#), [1675](#)
 __prg_generate_conditional_
 parm:NNNpnn [1597](#), [1598](#),
[1600](#), [1602](#), [1604](#), [1606](#), [1608](#), [1609](#)
 __prg_generate_conditional_
 test:w [1651](#), [1662](#), [1672](#)
 __prg_generate_F_form:wNNnnnnN .
 [1694](#), [1721](#)
 __prg_generate_p_form:wNNnnnnN .
 [383](#), [1694](#), [1694](#)
 __prg_generate_T_form:wNNnnnnN .
 [1694](#), [1713](#)
 __prg_generate_TF_form:wNNnnnnN
 [1694](#), [1729](#)
 __prg_p_true:w [1694](#), [1706](#), [1737](#)
 __prg_replicate:N
 [8562](#), [8569](#), [8570](#), [8572](#)
 __prg_replicate_ [8562](#)
 __prg_replicate_0:n [8562](#)
 __prg_replicate_1:n [8562](#)
 __prg_replicate_2:n [8562](#)
 __prg_replicate_3:n [8562](#)
 __prg_replicate_4:n [8562](#)
 __prg_replicate_5:n [8562](#)
 __prg_replicate_6:n [8562](#)
 __prg_replicate_7:n [8562](#)
 __prg_replicate_8:n [8562](#)
 __prg_replicate_9:n [8562](#)
 __prg_replicate_first:N
 [8562](#), [8565](#), [8571](#)
 __prg_replicate_first--:n ... [8562](#)
 __prg_replicate_first_0:n ... [8562](#)
 __prg_replicate_first_1:n ... [8562](#)
 __prg_replicate_first_2:n ... [8562](#)
 __prg_replicate_first_3:n ... [8562](#)
 __prg_replicate_first_4:n ... [8562](#)
 __prg_replicate_first_5:n ... [8562](#)
 __prg_replicate_first_6:n ... [8562](#)
 __prg_replicate_first_7:n ... [8562](#)
 __prg_replicate_first_8:n ... [8562](#)
 __prg_replicate_first_9:n ... [8562](#)
 __prg_set_eq_conditional:NNNn ..
 [1741](#), [1742](#), [1744](#), [1746](#), [1747](#)
 __prg_set_eq_conditional:nnNnnNNw
 [1751](#), [1759](#), [1759](#)
 __prg_set_eq_conditional_F_
 form:nnn [1759](#)
 __prg_set_eq_conditional_F_
 form:wNnnnn [1796](#), [39705](#)
 __prg_set_eq_conditional_
 loop:nnnnNw . [1759](#), [1771](#), [1773](#), [1788](#)
 __prg_set_eq_conditional_p_
 form:nnn [1759](#)
 __prg_set_eq_conditional_p_
 form:wNnnnn [1790](#), [39693](#)
 __prg_set_eq_conditional_T_
 form:nnn [1759](#)
 __prg_set_eq_conditional_T_
 form:wNnnnn [1794](#), [39701](#)

- _prg_set_eq_conditional_TF_-
 form:nnn [1759](#)
- _prg_set_eq_conditional_TF_-
 form:wNnnnn [1792](#), [39697](#)
- _prg_T_true:w [1694](#), [1719](#), [1738](#)
- _prg_TF_true:w [384](#), [1694](#), [1735](#), [1740](#)
- _prg_use_none_delimit_by_q_-
 recursion_stop:w
 .. [1595](#), [1595](#), [1680](#), [1764](#), [1769](#), [1776](#)
- \primitive [776](#)
- prop commands:
- \c_empty_prop [222](#), [919](#), [920](#),
 [935](#), [19969](#), [19973](#), [20007](#), [20226](#), [20502](#)
- \prop_clear:N
 [213](#), [214](#), [925](#), [19998](#), [19998](#), [20000](#),
 [20031](#), [20037](#), [36250](#), [37205](#), [39467](#)
- \prop_clear_new:N [214](#), [20030](#),
 [20030](#), [20032](#), [37405](#), [37437](#), [37478](#)
- \prop_clear_new_linked:N
 [214](#), [20030](#), [20036](#), [20038](#)
- \prop_concat:NNN [213](#), [216](#),
 [217](#), [20134](#), [20134](#), [20136](#), [39421](#), [39468](#)
- \prop_const_from_keyval:Nn
 [215](#), [20189](#), [20189](#),
 [20194](#), [35427](#), [35434](#), [38183](#), [39603](#)
- \prop_const_linked_from_keyval:Nn
 [215](#), [20189](#), [20195](#), [20200](#), [39604](#)
- \prop_count:N [218](#), [20608](#), [20608](#), [20617](#)
- \prop_gc_clear:N [214](#), [19998](#),
 [20001](#), [20003](#), [20034](#), [20040](#), [39548](#)
- \prop_gc_clear_new:N [214](#), [1383](#),
 [20030](#), [20033](#), [20035](#), [35501](#), [35502](#)
- \prop_gc_clear_new_linked:N
 [214](#), [20030](#), [20039](#), [20041](#)
- \prop_gconcat:NNN
 [216](#), [20134](#), [20137](#), [20139](#), [39422](#), [39549](#)
- \prop_get:NnN [146](#), [147](#),
 [213](#), [217](#), [218](#), [927](#), [20229](#), [20229](#),
 [20234](#), [20235](#), [20236](#), [20237](#), [20242](#),
 [20244](#), [20246](#), [36490](#), [36494](#), [36563](#),
 [36567](#), [36724](#), [36839](#), [36939](#), [37980](#)
- \prop_get:NnNTF .. [217](#), [219](#), [9654](#),
 [9674](#), [9729](#), [10302](#), [10546](#), [14417](#),
 [20229](#), [21987](#), [30479](#), [35692](#), [36833](#),
 [36944](#), [37415](#), [37674](#), [37687](#), [37702](#),
 [37781](#), [37809](#), [37825](#), [38196](#), [38209](#)
- \prop_gpop:NnN . [217](#), [20355](#), [20361](#),
 [20369](#), [20370](#), [20377](#), [20385](#), [39550](#)
- \prop_gpop:NnNTF
 [217](#), [220](#), [20355](#), [39551](#), [39552](#), [39553](#)
- .prop_gput:N [244](#), [22149](#)
- \prop_gput:Nnn [216](#), [925](#), [3531](#), [3532](#),
 [8238](#), [8239](#), [9436](#), [10174](#), [10175](#),
 [10182](#), [10183](#), [10185](#), [10186](#), [10187](#),
 [10188](#), [10208](#), [10254](#), [10461](#), [10505](#),
 [11822](#), [11823](#), [14178](#), [14179](#), [14180](#),
 [14181](#), [14182](#), [14183](#), [14184](#), [14185](#),
 [14186](#), [14187](#), [14188](#), [14189](#), [14190](#),
 [14191](#), [14192](#), [14199](#), [14202](#), [15053](#),
 [15054](#), [20387](#), [20389](#), [20409](#), [20414](#),
 [20416](#), [20421](#), [21517](#), [21518](#), [22755](#),
 [22756](#), [23594](#), [23595](#), [30257](#), [30362](#),
 [30363](#), [30426](#), [30769](#), [30770](#), [35718](#),
 [35736](#), [35757](#), [35775](#), [35806](#), [37599](#),
 [37600](#), [37601](#), [37602](#), [37603](#), [37604](#),
 [37605](#), [37606](#), [37617](#), [37619](#), [37621](#),
 [37622](#), [37623](#), [37624](#), [37625](#), [37626](#),
 [37627](#), [37727](#), [37728](#), [37742](#), [37756](#),
 [37766](#), [38429](#), [38430](#), [38844](#), [39554](#)
- \prop_gput_from_keyval:Nn
 [217](#), [925](#), [20157](#),
 [20160](#), [20162](#), [20186](#), [30182](#), [39556](#)
- \prop_gput_if_new:Nnn
 [39012](#), [39015](#), [39018](#)
- \prop_gput_if_not_in:Nnn
 [216](#), [20387](#),
 [20393](#), [20430](#), [39014](#), [39015](#), [39555](#)
- \prop_gremove:Nn [218](#), [10287](#), [10531](#),
 [20341](#), [20347](#), [20354](#), [30424](#), [39557](#)
- \prop_gset_eq:NN
 . [214](#), [20042](#), [20045](#), [20047](#), [35503](#),
 [35505](#), [35670](#), [35672](#), [35709](#), [35711](#),
 [35962](#), [36128](#), [36169](#), [39404](#), [39558](#)
- \prop_gset_from_keyval:Nn
 [215](#), [20177](#), [20183](#), [20188](#), [30174](#), [39559](#)
- \prop_if_empty:N [20500](#), [20518](#)
- \prop_if_empty:NTF
 [219](#), [20500](#), [20621](#), [36713](#)
- \prop_if_empty_p:N [219](#), [20500](#)
- \prop_if_exist:N [20496](#), [20498](#)
- \prop_if_exist:NTF
 [218](#), [19962](#), [20031](#), [20034](#),
 [20037](#), [20040](#), [20496](#), [21886](#), [36711](#)
- \prop_if_exist_p:N [218](#), [20496](#)
- \prop_if_in:Nn [213](#), [20520](#), [20541](#)
- \prop_if_in:NnTF [219](#),
 [9439](#), [9445](#), [20520](#), [30246](#), [30302](#), [37417](#)
- \prop_if_in_p:Nn [219](#), [20520](#)
- \prop_item:Nn . [213](#), [218](#), [220](#), [928](#),
 [9440](#), [9446](#), [20270](#), [20270](#), [20287](#),
 [30250](#), [30307](#), [38043](#), [38082](#), [38852](#)
- \prop_log:N .. [222](#), [20636](#), [20638](#), [20639](#)
- \prop_make_flat:N
 [213](#), [215](#), [20092](#), [20092](#), [20101](#), [20104](#)
- \prop_make_linked:N [213](#),
 [215](#), [919](#), [20110](#), [20110](#), [20119](#), [20122](#)
- \prop_map_break: ... [221](#), [936](#), [937](#),
 [20547](#), [20548](#), [20549](#), [20550](#), [20551](#),

- 20573, 20585, 20586, 20587, 20588,
20589, 20604, 20604, 20605, 20607
- `\prop_map_break:n`
..... 221, 20285, 20539, 20604, 20606
- `\prop_map_function:NN`
..... 87, 220, 937, 10317,
10561, 20543, 20543, 20565, 20613,
20629, 20680, 30343, 36638, 38271
- `\prop_map_inline:Nn`
..... 220, 20566, 20566, 20580,
35972, 35974, 35977, 35995, 35997,
36071, 36088, 36149, 36151, 36155,
36157, 36337, 36356, 36537, 36546
- `\prop_map_tokens:Nn` 220, 834, 929,
935, 20274, 20524, 20581, 20581, 20603
- `\prop_new:N`
. 213–215, 9434, 9435, 9457, 9626,
10196, 10449, 14177, 19970, 19970,
19975, 20031, 20034, 20130, 20131,
20132, 20133, 20191, 21594, 21595,
21886, 30167, 30173, 30373, 35950,
35951, 35952, 36420, 36461, 37503,
37593, 37598, 37615, 37620, 38840
- `\prop_new_linked:N` 213–215, 19976,
19976, 19997, 20037, 20040, 20197
- `\prop_pop:NnN` 213, 217, 20355, 20355,
20367, 20368, 20371, 20383, 39469
- `\prop_pop:NnNTF` 217,
219, 926, 20355, 39470, 39471, 39472
- `.prop_put:N` 244, 22149
- `\prop_put:Nnn`
213, 216, 217, 418, 916, 924, 932,
9702, 9718, 9735, 20387, 20387,
20395, 20400, 20402, 20407, 21994,
35715, 35733, 35752, 35773, 35804,
36006, 36008, 36014, 36016, 36025,
36031, 36039, 36098, 36106, 36196,
36202, 36210, 36217, 36361, 36421,
36423, 36425, 36427, 36429, 36431,
36433, 36435, 36437, 36439, 36441,
36443, 36445, 36447, 36449, 36451,
36453, 36455, 36811, 37206, 37406,
37425, 37468, 37483, 37664, 39473
- `\prop_put_from_keyval:Nn` ... 217,
925, 20157, 20157, 20159, 20180, 39475
- `\prop_put_if_new:Nnn`
..... 39012, 39013, 39016
- `\prop_put_if_not_in:Nnn`
..... 216, 924, 20387,
20391, 20423, 39012, 39013, 39474
- `\prop_remove:Nn`
213, 218, 9699, 9714, 20341, 20341,
20353, 36532, 36535, 36539, 39476
- `\prop_set_eq:NN` 213, 214, 920, 923,
924, 20042, 20042, 20044, 35658,
35660, 35702, 35704, 35959, 35968,
35970, 36121, 36145, 36147, 36166,
36294, 36527, 37488, 39403, 39477
- `\prop_set_from_keyval:Nn` ... 215,
217, 20177, 20177, 20182, 37643, 39478
- `\prop_show:N` 221,
926, 939, 20225, 20636, 20636, 20637
- `\prop_to_keyval:N` .. 218, 20618, 20618
- `\g_tmpa_prop` 222, 20130
- `\l_tmpa_prop` 222, 20130
- `\g_tmpb_prop` 222, 20130
- `\l_tmpb_prop` 222, 20130
- prop internal commands:
- `\c__prop_basis_int`
..... 918, 19941, 19951, 19954, 19956
- `__prop_chk:w` 915, 917, 919,
922, 926, 936, 19914, 19914, 19931,
19969, 20219, 20292, 20448, 20646
- `__prop_chk_get:nw` 19914, 19918, 19921
- `__prop_chk_loop:nw`
..... 19914, 19914, 19915, 19922
- `__prop_clear:NNN` 19998,
19999, 20002, 20004, 20179, 20185
- `__prop_clear:wNNN`
920, 922, 19998, 20008, 20010, 20107
- `__prop_clear_entries:NN`
..... 920, 20013, 20017, 20064
- `__prop_clear_loop:Nw`
..... 920, 19998, 20019, 20022, 20027
- `__prop_concat:nNNN`
..... 20134, 20143, 20146, 20149
- `__prop_concat:NNNNN`
..... 20134, 20135, 20138, 20140
- `__prop_count:nn` . 20608, 20613, 20616
- `__prop_flatten:N`
..... 918, 19926, 19926, 20056
- `__prop_flatten:w`
..... 916, 919, 939, 940, 19924,
19924, 19965, 19968, 19986, 20011,
20014, 20062, 20072, 20256, 20303,
20456, 20506, 20510, 20671, 20684
- `__prop_flatten_aux:N`
..... 918, 19925, 19926, 19928, 19929
- `__prop_flatten_aux:w`
..... 19926, 19927, 19928
- `__prop_flatten_loop:w`
..... 918, 19926, 19932, 19935, 19939
- `__prop_from_keyval:nn` ... 20157,
20158, 20161, 20163, 20192, 20198
- `__prop_from_keyval:Nnn`
..... 20157, 20166, 20167, 20169

- __prop_get:NnnTF 20349, 20357, 20363, 20373, 20379
- 927, 20229, 20231, 20239, 20248
- __prop_get_linked:w 928,
- 935, 20229, 20253, 20255, 20280, 20532
- __prop_get_linked_aux:w 928, 20229, 20259, 20262, 20268
- __prop_if_empty:w 935
- __prop_if_empty_return:w 20500, 20505, 20509
- __prop_if_flat:NTF .. 939, 19960,
- 19960, 20006, 20052, 20054, 20096,
- 20114, 20272, 20522, 20644, 20655
- __prop_if_flat_aux:w 19960, 19963, 19967
- __prop_if_in_flat:nnn 20520, 20526, 20536
- __prop_if_recursion_tail_stop:n 19911, 19911, 19912, 20665
- \l__prop_internal_tl 918, 919, 922, 923,
- 933, 934, 940, 941, 19906, 19945,
- 19987, 19988, 19989, 19991, 20080,
- 20081, 20083, 20090, 20126, 20128,
- 20439, 20449, 20452, 20481, 20492,
- 20658, 20715, 20723, 20727, 20733
- __prop_item:nnn 929, 20270, 20276, 20282
- \prop_make_flat:N__prop_make_-
- flat:Nn 20092
- __prop_make_flat:Nn .. 20097, 20105
- __prop_make_linked:Nn 20110, 20115, 20123
- __prop_map_function:Nw 936, 20543, 20546, 20553, 20563
- __prop_map_tokens:nw 20581, 20584, 20591, 20601
- __prop_missing_eq:n 924, 20157, 20172, 20175
- __prop_new_linked:N 923, 19976, 19979, 19981, 20125
- __prop_next_prefix: 19943, 19943, 19983
- __prop_pair:wn 915–917, 926, 937, 941, 942, 19917,
- 19921, 19923, 19923, 19933, 19935,
- 19938, 20020, 20023, 20075, 20078,
- 20084, 20151, 20152, 20155, 20209,
- 20212, 20441, 20547, 20548, 20549,
- 20550, 20554, 20555, 20556, 20557,
- 20569, 20571, 20576, 20585, 20586,
- 20587, 20588, 20592, 20593, 20594,
- 20595, 20649, 20663, 20666, 20737
- __prop_pop:NnNnTF 930, 20288, 20288, 20343,
- 20349, 20357, 20363, 20373, 20379
- __prop_pop_linked:NNNn 20288, 20306, 20314
- __prop_pop_linked:w 930, 20288, 20319, 20322
- __prop_pop_linked:wnNnTF 929, 930, 20288, 20298, 20302
- __prop_pop_linked_next:w 20288, 20334, 20340
- __prop_pop_linked_prev:w 20288, 20328, 20338, 20475
- \g__prop_prefix_int 918, 19941, 19946, 19947
- __prop_put:NNNn 20387
- __prop_put:nNNn 20153, 20158, 20161, 20192, 20198,
- 20388, 20390, 20392, 20394, 20437
- __prop_put_linked:NNNN 20387
- __prop_put_linked:NNnN 20458, 20462
- __prop_put_linked:wnNN .. 933, 20387
- __prop_put_linked:wnnN 20453, 20455
- __prop_put_linked_new:w 933, 934, 20387, 20465, 20470
- __prop_put_linked_old:w 933, 934, 20387, 20467, 20486
- __prop_set_eq:NNNN 20042, 20043, 20046, 20048, 20145
- __prop_set_eq:nNnNN 922, 20042, 20065, 20067
- __prop_set_eq:wNNNN 923, 20042, 20058, 20061, 20127
- __prop_set_eq_end:w 922, 20042, 20082, 20086
- __prop_set_eq_loop:NNnw 20042, 20071, 20077, 20083, 20087
- __prop_show:NN 20636, 20636, 20638, 20640
- __prop_show_bad_name:NNN 940, 941, 20636, 20700, 20712
- __prop_show_end:NNN 940, 20636, 20699, 20720
- __prop_show_finally:NNn 940, 20636, 20656, 20676, 20726
- __prop_show_flat:w 939,
- 941, 20636, 20647, 20663, 20667, 20736
- __prop_show_linked:w 939, 20636, 20652, 20669
- __prop_show_loop:NNw 941, 20636, 20703, 20705, 20741
- __prop_show_loop_key:wNNN 940, 941, 20636, 20694, 20730
- __prop_show_prepare:w 940, 20636, 20659, 20683

- `__prop_split:NnTFn` 926, 927, 933,
 935, 20201, 20201, 20250, 20290, 20444
`__prop_split_aux:nNTFn`
 20201, 20203, 20206
`__prop_split_flat:w`
 926, 927, 20201, 20208, 20216, 20219
`__prop_split_linked:w`
 926, 20201, 20208, 20220, 20222
`__prop_split_test:wn`
 926, 20201, 20211, 20217
`__prop_split_wrong:Nw`
 926, 927, 20201, 20212, 20223
`__prop_tmp:w` ... 940, 941, 19905,
 19905, 20151, 20155, 20688, 20707
`__prop_to_keyval:nn`
 20618, 20629, 20634
`__prop_to_keyval:nnw` 20618
`__prop_to_keyval_exp_after:wN` 20618
`__prop_to_prefix:n`
 .. 19943, 19946, 19949, 19953, 19955
 prop *(prefix)* internal commands:
`__prop <prefix>` 915, 916
`\protect`
 1298, 10684, 24161, 31694, 31931,
 31946, 31955, 31969, 31971, 34225
`\protected` . 82, 84, 106, 518, 19519, 19521
`\protrudechars` 952
`\protrusionboundary` 910
`\ProvidesExplClass` 9
`\ProvidesExplFile` 9, 39023
`\ProvidesExplPackage` 9
`pt` 280
`\ptexfontname` 1174
`\ptexlineendmode` 1175
`\ptexminorversion` 1176
`\ptexrevision` 1177
`\ptextracingfonts` 1178
`\ptexversion` 1179
`\pdxdimen` 953
- Q**
- quark commands:
`\q_mark` 147, 446, 16352,
 33255, 33257, 33264, 33267, 33277
`\q_nil` 26, 27, 126, 147, 379,
 808, 810, 812, 1551, 1554, 11148,
 11150, 16352, 16406, 16425, 16431,
 16446, 16447, 16453, 16477, 16481,
 22592, 22593, 22595, 22597, 22599,
 22601, 22607, 37926, 37932, 37933
`\q_no_value`
 .. 77, 93, 100–103, 146, 147, 153,
 154, 161, 192, 217, 808, 810, 829,
 830, 875, 931, 8872, 10340, 10353,
 11057, 11208, 11311, 11314, 11317,
 11320, 11349, 16352, 16414, 16435,
 16441, 17030, 17038, 17050, 17076,
 18537, 18552, 20232, 20359, 20365
`\quark_if_nil:N` 16404
`\quark_if_nil:n` 810, 811, 16422, 16442
`\quark_if_nil:NTF` 147, 370, 812, 16404
`\quark_if_nil:nTF` .. 147, 710, 809,
 810, 812, 11152, 16422, 37940, 37953
`\quark_if_nil_p:N` 147, 16404
`\quark_if_nil_p:n` 147, 16422
`\quark_if_no_value:N` .. 16412, 16420
`\quark_if_no_value:n` 16432
`\quark_if_no_value:NTF`
 147, 16404, 36492, 36496, 36565, 36569
`\quark_if_no_value:nTF` ... 147, 16422
`\quark_if_no_value_p:N` ... 147, 16404
`\quark_if_no_value_p:n` ... 147, 16422
`\quark_if_recursion_tail_`
`break:NN` ... 149, 811, 16392, 16392
`\quark_if_recursion_tail_`
`break:nN` ... 149, 811, 16392, 16398
`\quark_if_recursion_tail_stop:N` .
 148, 369, 811, 1294, 16360, 16360,
 33637, 33670, 34332, 34385, 34411
`\quark_if_recursion_tail_stop:n` .
 148,
 369, 810, 811, 5849, 5969, 16374,
 16374, 16390, 17439, 30997, 31198
`\quark_if_recursion_tail_stop_`
`do:Nn`
 .. 148, 369, 811, 16360, 16366, 30291
`\quark_if_recursion_tail_stop_`
`do:nn` 148, 369,
 811, 16374, 16381, 16391, 33957, 37171
`\quark_new:N`
 .. 147, 369, 370, 814, 4355, 4356,
 8323, 8324, 10466, 10965, 10967,
 10968, 12347, 12348, 12349, 12350,
 12351, 13409, 13410, 14176, 16347,
 16347, 16352, 16353, 16354, 16355,
 16356, 16357, 16359, 17458, 17459,
 19060, 19909, 19910, 21599, 31387,
 31389, 31390, 39026, 39027, 39647
`\q_recursion_stop` ... 26, 27, 148,
 149, 379, 808, 1553, 1557, 5845,
 5964, 16356, 17428, 30286, 31011,
 31194, 33657, 33692, 33953, 33962,
 33986, 34365, 34408, 34626, 37167
`\q_recursion_tail`
 148, 149, 808, 809, 5845,
 5963, 16356, 16362, 16368, 16377,
 16384, 16389, 16394, 16401, 17428,

- 30286, 31010, 31194, 33656, 33691,
33953, 34364, 34407, 34625, 37166
- \q_stop 26, 27, 39,
120, 146, 147, 379, 808, 1552, 1555,
9133, 9143, 11148, 11150, 12883,
16352, 31018, 31022, 31032, 31057,
31078, 31213, 31245, 31257, 31261,
31272, 31273, 31279, 31281, 31282,
31284, 31287, 31301, 31308, 31350,
31362, 31364, 31374, 31377, 37953
- quark internal commands:
- \q_bool_recursion_stop
..... 8323, 8326, 8439, 8465
- \q_bool_recursion_tail
..... 8323, 8439, 8465
- \q_char_no_value 19060
- \q_cs_nil 3018
- \q_cs_recursion_stop
..... 2700, 2704, 2715, 3011
- \q_debug_recursion_stop
..... 39026, 39029, 39228, 39233
- \q_debug_recursion_tail
..... 39026, 39228, 39233
- \q_file_nil
.. 10965, 11032, 11046, 11172, 11178
- \q_file_recursion_stop
..... 10967, 11011, 11022
- \q_file_recursion_tail
..... 10967, 11011, 11015
- \q_int_recursion_stop
.. 17458, 18167, 18184, 18227, 18254
- \q_int_recursion_tail
..... 17458, 18167, 18184, 18227
- \q_iow_nil 10466, 10713, 10720
- \q_keys_no_value
.. 997, 21581, 21599, 22250, 22270,
22273, 22287, 22292, 22307, 22312
- \q_prg_recursion_stop
..... 386, 1596, 1669, 1756
- \q_prg_recursion_tail
..... 386, 1669, 1679, 1756, 1775
- \q_prop_recursion_stop
..... 19909, 20650, 20738
- \q_prop_recursion_tail
..... 19909, 20649, 20737
- __quark_if_empty_if:n
.. 16422, 16424, 16434, 16444, 16584
- __quark_if_nil:w
..... 810, 16422, 16425, 16431
- __quark_if_no_value:w
..... 16422, 16435, 16441
- __quark_if_recursion_tail:w 809,
814, 16374, 16377, 16384, 16388, 16401
- __quark_module_name:N
..... 815, 16450, 16473, 16602, 16604
- __quark_module_name:w
..... 16602, 16606, 16609
- __quark_module_name_end:w
..... 16602, 16617, 16620
- __quark_module_name_loop:w
..... 16602, 16610, 16611, 16615
- __quark_new_conditional:Nnnn ...
..... 16449, 16471, 16475, 16492
- __quark_new_conditional_N:Nnnn .
..... 16569, 16574
- __quark_new_conditional_n:Nnnn .
..... 16569, 16569
- __quark_new_conditional_N_-
aux:Nnnn 16569, 16576, 16591
- __quark_new_conditional_n_-
aux:Nnnn 16569, 16571, 16579
- __quark_new_test:NNNn
..... 16449, 16457, 16462, 16468
- __quark_new_test_aux:Nn
..... 16450, 16451, 16461
- __quark_new_test_aux:nnNNnnnn ..
..... 16449, 16464, 16485, 16493
- __quark_new_test_aux_do:nNNnnnnNNn
.. 813, 814, 16504, 16509, 16514,
16519, 16524, 16530, 16533, 16533
- __quark_new_test_define_break_-
ifx:nNNNn ... 16531, 16546, 16567
- __quark_new_test_define_break_-
tl:nNNNn ... 16515, 16546, 16565
- __quark_new_test_define_-
ifx:nNnNNn 813,
814, 16520, 16525, 16546, 16555, 16568
- __quark_new_test_define_-
tl:nNnNNn 813,
814, 16505, 16510, 16546, 16546, 16566
- __quark_new_test_N:Nnnn 16502, 16517
- __quark_new_test_n:Nnnn 16502, 16502
- __quark_new_test_NN:Nnnn
..... 16502, 16528
- __quark_new_test_Nn:Nnnn
..... 16502, 16522
- __quark_new_test_nN:Nnnn 16512
- __quark_new_test_nn:Nnnn
..... 16502, 16507
- \q_quark_nil 16359
- __quark_quark_conditional_-
name:N ... 816, 16472, 16624, 16626
- __quark_quark_conditional_-
name:w ... 816, 16624, 16628, 16631
- __quark_test_define_aux:NNNnnNNn
..... 814, 16533, 16535, 16540

- _quark_tmp:w
 [816](#), [16602](#), [16623](#), [16624](#), [16634](#)
 - \q_regex_nil
 ... [4326](#), [4331](#), [4356](#), [4361](#), [4969](#),
 [4973](#), [5629](#), [5647](#), [5648](#), [5743](#), [5753](#)
 - \q_regex_recursion_stop
 .. [4355](#), [4358](#), [4360](#), [5629](#), [5648](#), [7634](#)
 - \q_str_nil
 ... [786](#), [14176](#), [15265](#), [15272](#), [15287](#), [15314](#)
 - \q_str_recursion_stop
 [13409](#), [14013](#), [14021](#), [14026](#)
 - \q_str_recursion_tail
 [741](#), [13409](#), [13656](#),
 [13665](#), [13682](#), [13702](#), [13724](#), [14013](#)
 - \q_text_nil [31387](#), [31962](#), [31963](#)
 - \q_text_recursion_stop
 [31389](#), [31392](#), [31774](#), [31869](#),
 [31893](#), [31913](#), [32141](#), [32155](#), [32164](#),
 [32169](#), [32230](#), [32246](#), [32255](#), [32302](#),
 [32365](#), [32374](#), [32466](#), [32475](#), [32714](#),
 [32723](#), [32743](#), [32751](#), [32852](#), [32861](#),
 [32944](#), [32949](#), [33192](#), [33197](#), [33224](#),
 [33236](#), [33289](#), [33296](#), [33426](#), [33431](#),
 [33489](#), [33494](#), [33532](#), [33537](#), [33572](#),
 [33584](#), [33711](#), [33714](#), [33723](#), [33730](#),
 [33773](#), [33781](#), [33807](#), [33827](#), [33860](#),
 [33923](#), [33931](#), [33965](#), [33987](#), [34020](#),
 [34025](#), [34071](#), [34084](#), [34093](#), [34111](#),
 [34124](#), [34126](#), [34143](#), [34152](#), [34180](#)
 - \q_text_recursion_tail
 [31389](#), [31538](#), [31773](#),
 [31869](#), [31893](#), [31913](#), [32141](#), [32169](#),
 [32229](#), [32302](#), [33711](#), [33730](#), [33807](#),
 [33860](#), [33965](#), [34071](#), [34110](#), [34180](#)
 - \q_text_stop
 [32115](#), [32121](#), [32123](#), [32124](#)
 - \q_tl_mark [704](#),
 ... [705](#), [12347](#), [12455](#), [12457](#), [12459](#), [12461](#)
 - \q_tl_nil [705](#), [12347](#), [12493](#)
 - \q_tl_recursion_stop [12350](#)
 - \q_tl_recursion_tail . [12350](#), [13111](#)
 - \q_tl_stop [705](#), [12347](#), [12492](#)
 - \quitvmode [679](#)
- R**
- \r [32057](#), [34380](#),
 ... [34399](#), [34423](#), [34449](#), [34573](#), [34574](#)
 - \radical [381](#)
 - \raise [382](#)
 - \rand [279](#)
 - \randint [279](#)
 - \randomseed [954](#)
 - \read [383](#)
 - \readline [519](#)
 - \readpapersizespecial [1180](#)
 - \ref [31662](#), [31672](#)
 - regex commands:
 - \regex_const:Nn [55](#), [7234](#), [7244](#)
 - \regex_count:NnN . [56](#), [7284](#), [7286](#), [7289](#)
 - \regex_count:nnN
 [56](#), [560](#), [7284](#), [7284](#), [7288](#)
 - \regex_extract_all:NnN [57](#), [7304](#), [7320](#)
 - \regex_extract_all:nnN
 [48](#), [57](#), [471](#), [7304](#), [7320](#)
 - \regex_extract_all:NnNTF ... [57](#), [7304](#)
 - \regex_extract_all:nnNTF ... [57](#), [7304](#)
 - \regex_extract_once:NnN [57](#), [7304](#), [7318](#)
 - \regex_extract_once:nnN [57](#), [7304](#), [7318](#)
 - \regex_extract_once:NnNTF .. [57](#), [7304](#)
 - \regex_extract_once:nnNTF [51](#), [57](#), [7304](#)
 - \regex_gset:Nn [55](#), [7234](#), [7239](#)
 - \regex_log:N [55](#), [514](#), [7249](#), [7260](#)
 - \regex_log:n [55](#), [7249](#), [7250](#)
 - \regex_match:Nn [7278](#), [7283](#)
 - \regex_match:nn [7272](#), [7277](#)
 - \regex_match:NnTF [56](#), [7272](#)
 - \regex_match:nnTF .. [56](#), [562](#), [571](#), [7272](#)
 - \regex_match_case:nn
 ... [56](#), [59](#), [493](#), [523](#), [7290](#), [7298](#), [7428](#)
 - \regex_match_case:nnTF .. [56](#), [7290](#),
 ... [7290](#), [7299](#), [7300](#), [7301](#), [7302](#), [7303](#)
 - \regex_new:N [55](#),
 ... [474](#), [7228](#), [7228](#), [7230](#), [7231](#), [7232](#), [7233](#)
 - \regex_replace:nnN [196](#)
 - \regex_replace_all:NnN [58](#), [7304](#), [7324](#)
 - \regex_replace_all:nnN
 [48](#), [58](#), [559](#), [7304](#), [7324](#)
 - \regex_replace_all:NnNTF ... [58](#), [7304](#)
 - \regex_replace_all:nnNTF ... [58](#), [7304](#)
 - \regex_replace_case_all:nN
 [59](#), [7349](#), [7354](#), [7366](#)
 - \regex_replace_case_all:nNTF ...
 [59](#), [7349](#),
 ... [7349](#), [7367](#), [7368](#), [7369](#), [7370](#), [7371](#)
 - \regex_replace_case_once:nN
 [59](#), [7326](#), [7331](#), [7343](#)
 - \regex_replace_case_once:nNTF ...
 [59](#), [7326](#),
 ... [7326](#), [7344](#), [7345](#), [7346](#), [7347](#), [7348](#)
 - \regex_replace_once:NnN [58](#), [7304](#), [7322](#)
 - \regex_replace_once:nnN
 [57-59](#), [210](#), [558](#), [7304](#), [7322](#)
 - \regex_replace_once:NnNTF .. [58](#), [7304](#)
 - \regex_replace_once:nnNTF
 [58](#), [575](#), [7304](#)
 - \regex_set:Nn .. [47](#), [55](#), [56](#), [7234](#), [7234](#)
 - \regex_show:N [55](#), [502](#), [514](#), [7249](#), [7259](#)
 - \regex_show:n .. [48](#), [53](#), [55](#), [7249](#), [7249](#)

- \regex_split:NnN 58, [7304](#), [7325](#)
- \regex_split:nnN 58, [7304](#), [7325](#)
- \regex_split:NnNTF 58, [7304](#)
- \regex_split:nnNTF 58, [7304](#)
- \g_tmpa_regex 60, [7230](#)
- \l_tmpa_regex 60, [7230](#)
- \g_tmpb_regex 60, [7230](#)
- \l_tmpb_regex 60, [7230](#)
- regex internal commands:
 - __regex_A_test: 486, 5237, 5259, 5875, 5878, 5884, 6002, [6483](#), 6516
 - __regex_action_cost:n 522, 526, 6272, 6273, 6281, 6730, [6756](#), [6756](#)
 - __regex_action_free:n 522, 534, 6295, 6301, 6302, 6313, 6371, 6375, 6400, 6425, 6429, 6432, 6460, 6468, 6478, 6492, 6535, 6728, [6732](#), 6732
 - __regex_action_free_aux:nn 6732, 6733, 6735, 6736
 - __regex_action_free_group:n 522, 534, 6321, 6440, 6443, [6732](#), 6734
 - __regex_action_start_wildcard:N 522, 6156, 6176, [6725](#), 6725
 - __regex_action_submatch:nN 522, 6180, 6202, 6394, 6395, 6533, 6781, [6783](#), 6783
 - __regex_action_submatch_aux:w 6783, 6785, 6788
 - __regex_action_submatch_auxii:w 6783, 6794, 6799
 - __regex_action_submatch_ auxiii:w [6783](#), 6795, 6800, 6801, 6802
 - __regex_action_submatch_ auxiv:w [6783](#)
 - __regex_action_success: 522, 6159, 6205, 6223, [6804](#), 6804
 - __regex_action_wildcard: 539
 - \l_regex_added_begin_int [7383](#), 7522, 7530, 7534, 7588, 7717, 7722, 7726, 7737, 7752
 - \l_regex_added_end_int [7383](#), 7524, 7530, 7535, 7589, 7719, 7722, 7727, 7739, 7753
 - \c_regex_all_catcodes_int 4783, 4895, 4999, 5959
 - \c_regex_ascii_lower_int 4354, 4415, 4420
 - \c__regex_ascii_max_control_int 4351, 4531
 - \c__regex_ascii_max_int 4351, 4524, 4532, 4723
 - \c_regex_ascii_min_int 4351, 4523, 4530
 - __regex_assertion:Nn 486, 500, 532, 5233, 5255, 5864, 5995, [6483](#), 6483
 - __regex_b_test: 486, 532, 5245, 5247, 5881, 6000, [6483](#), 6501
 - \l_regex_balance_int 474, 546, 569, [4350](#), 6880, 6912, 7171, 7188, 7395, 7408, 7410, 7411, 7668, 7694, 7718, 7720
 - \g__regex_balance_intarray 471, 560, 6859, 6866, [7382](#), 7407
 - \g__regex_balance_tl 546, [6822](#), 6881, 6911, 6937, 6954, 6964, 7039
 - \l__regex_begin_flag 7373, 7513, 7523, 7566
 - __regex_branch:n 486, 504, 528, 4347, 4900, 4975, 5405, 5458, 5643, 5753, 5761, 5845, 5847, 5850, 5977, [6366](#), 6366, 39749, 39750, 39751
 - __regex_break_point:TF 475, 499, 526, [4363](#), 4364, 4365, 4369, 6272, 6273, 6489, 6506
 - __regex_break_true:w 475, 476, [4363](#), 4363, 4369, 4374, 4381, 4388, 4392, 4399, 4405, 4452, 4464, 4480, 5208, 6513, 6519, 6525
 - __regex_build:N 558, [6139](#), 6141, 7280, 7287, 7307, 7311, 39718, 39721, 39723
 - __regex_build:n 523, 558, [6139](#), 6139, 7274, 7285, 7306, 7309
 - __regex_build_aux:NN 570, [6139](#), 6142, 6146, 6148, 7774, 7793, 7863
 - __regex_build_aux:Nn 570, [6139](#), 6140, 6143, 7765, 7783, 7855
 - __regex_build_for_cs:n 4475, [6212](#), 6212, 39725, 39728, 39730
 - __regex_build_new_state: 6153, 6154, 6173, 6174, 6178, 6215, 6216, [6245](#), 6245, 6254, 6286, 6320, 6324, 6368, 6383, 6388, 6427, 6446, 6481, 6485, 6530, 39743
 - \l_regex_build_tl 504, 575, 4344, 4892, 4899, 4917, 4922, 4925, 4926, 4929, 4930, 4933, 4993, 4996, 5036, 5050, 5054, 5177, 5191, 5232, 5254, 5267, 5299, 5312, 5316, 5398, 5401, 5404, 5410, 5411, 5414, 5457, 5747, 5751, 5758, 5764, 5785, 5801, 5819, 5976, 6033, 6036, 6047, 6077, 6092, 6096, 6099, 6105, 6879, 6902, 6913, 6916, 6967, 7036, 7093, 7096, 7110, 7178, 7921, 7924, 7932, 7935
 - __regex_build_transition_ left:NNN [6241](#), 6241, 6429, 6443, 6460

- _regex_build_transition_-right:nNn [6241](#),
[6243](#), [6287](#), [6321](#), [6371](#), [6375](#),
[6400](#), [6425](#), [6432](#), [6440](#), [6468](#), [6478](#)
- _regex_build_transitions_-laziness:NNNNN
..... [6252](#), [6252](#), [6294](#), [6300](#), [6312](#)
- \l_regex_capturing_group_int ...
..... [471](#), [521](#), [568](#),
[6138](#), [6151](#), [6189](#), [6194](#), [6197](#), [6337](#),
[6339](#), [6350](#), [6351](#), [6359](#), [6360](#), [6363](#),
[6627](#), [6700](#), [6701](#), [6774](#), [6793](#), [7027](#),
[7031](#), [7619](#), [7640](#), [7648](#), [7699](#), [7707](#)
- \g_regex_case_balance_tl
..... [6942](#), [6945](#), [6951](#), [6955](#), [6963](#)
- _regex_case_build:n
[562](#), [6163](#), [6163](#), [6168](#), [7337](#), [7360](#), [7434](#)
- _regex_case_build_aux:Nn
..... [6163](#), [6165](#), [6169](#)
- _regex_case_build_loop:n
..... [6163](#), [6187](#), [6192](#)
- \l_regex_case_changed_char_int .
..... [476](#), [4391](#),
[4403](#), [4404](#), [4411](#), [4415](#), [4420](#), [6548](#)
- \g_regex_case_int
..... [558](#), [559](#), [6161](#), [6166](#), [6183](#),
[6186](#), [6203](#), [6204](#), [7294](#), [7338](#), [7630](#)
- \l_regex_case_max_group_int ...
..... [6162](#), [6182](#), [6189](#), [6196](#), [6197](#)
- _regex_case_replacement:n
..... [6941](#), [6943](#), [6959](#), [7361](#)
- _regex_case_replacement_aux:n .
..... [6953](#), [6960](#)
- \g_regex_case_replacement_tl ...
..... [6941](#), [6951](#), [6957](#), [6962](#)
- \c_regex_catcode_A_int [4783](#)
- \c_regex_catcode_B_int [4783](#)
- \c_regex_catcode_C_int [4783](#)
- \c_regex_catcode_D_int [4783](#)
- \c_regex_catcode_E_int [4783](#)
- \c_regex_catcode_in_class_mode_-int [4773](#), [4884](#), [5266](#), [5427](#), [5520](#), [5549](#)
- \c_regex_catcode_L_int [4783](#)
- \c_regex_catcode_M_int [4783](#)
- \c_regex_catcode_mode_int
.. [4773](#), [4880](#), [4953](#), [5298](#), [5518](#), [5547](#)
- \c_regex_catcode_O_int [4783](#)
- \c_regex_catcode_P_int [4783](#)
- \c_regex_catcode_S_int [4783](#)
- \c_regex_catcode_T_int [4783](#)
- \c_regex_catcode_U_int [4783](#)
- \l_regex_catcodes_bool
..... [4780](#), [5554](#), [5558](#), [5593](#)
- \l_regex_catcodes_int
..... [487](#), [4780](#), [4896](#), [4998](#),
[5000](#), [5006](#), [5285](#), [5302](#), [5402](#), [5415](#),
[5514](#), [5551](#), [5586](#), [5588](#), [5594](#), [5595](#)
- _regex_char_if_alphanumeric:N [4746](#)
- _regex_char_if_alphanumeric:NTF
..... [4717](#), [4946](#), [7145](#)
- _regex_char_if_special:N ... [4717](#)
- _regex_char_if_special:NTF ...
..... [4717](#), [4942](#)
- _regex_chk_c_allowed:TF
..... [4866](#), [4866](#), [5507](#)
- _regex_class:NnnnN
..... [486](#), [494](#), [495](#), [501](#),
[4348](#), [4994](#), [5293](#), [5294](#), [5300](#), [5660](#),
[5793](#), [5803](#), [5865](#), [5992](#), [6266](#), [6266](#)
- \c_regex_class_mode_int
..... [4773](#), [4870](#), [4885](#)
- _regex_class_repeat:n
... [527](#), [6276](#), [6282](#), [6282](#), [6298](#), [6307](#)
- _regex_class_repeat:nN
..... [6277](#), [6291](#), [6291](#)
- _regex_class_repeat:nnN
..... [6278](#), [6305](#), [6305](#)
- _regex_clean_assertion:Nn ...
..... [5822](#), [5864](#), [5872](#)
- _regex_clean_bool:n
.. [5822](#), [5822](#), [5874](#), [5889](#), [5893](#), [5901](#)
- _regex_clean_branch:n
..... [5822](#), [5850](#), [5853](#)
- _regex_clean_branch_loop:n [5822](#),
[5855](#), [5858](#), [5863](#), [5885](#), [5894](#), [5902](#)
- _regex_clean_class:n
..... [5822](#), [5890](#), [5904](#), [5915](#), [5936](#)
- _regex_clean_class:NnnnN
..... [5822](#), [5865](#), [5887](#)
- _regex_clean_class_loop:nnn ...
..... [5822](#),
[5905](#), [5906](#), [5917](#), [5927](#), [5937](#), [5951](#)
- _regex_clean_exact_cs:n
..... [5822](#), [5912](#), [5958](#)
- _regex_clean_exact_cs:w
..... [5822](#), [5962](#), [5967](#), [5971](#)
- _regex_clean_group:nnnN
..... [5822](#), [5866](#), [5867](#), [5868](#), [5896](#)
- _regex_clean_int:n
... [5822](#), [5828](#), [5831](#), [5891](#), [5892](#),
[5899](#), [5900](#), [5913](#), [5914](#), [5926](#), [5936](#)
- _regex_clean_int_aux:N
..... [5822](#), [5832](#), [5834](#)
- _regex_clean_regex:n
..... [5822](#), [5842](#), [5898](#), [5911](#), [7264](#)
- _regex_clean_regex_loop:w ...
..... [5822](#), [5844](#), [5847](#), [5851](#)

- _regex_command_K:
... [486](#), [5819](#), [5863](#), [5993](#), [6528](#), [6528](#)
- _regex_compile:n ... [4935](#), [4935](#),
[4971](#), [6145](#), [7236](#), [7241](#), [7246](#), [7253](#)
- _regex_compile:w
..... [492](#), [4889](#), [4889](#), [4937](#), [5600](#)
- _regex_compile_\$: [5228](#)
- _regex_compile_(: [5422](#)
- _regex_compile_) : [5461](#)
- _regex_compile_.. : [5199](#)
- _regex_compile_/A: [5228](#)
- _regex_compile_/B: [5228](#)
- _regex_compile_/b: [5228](#)
- _regex_compile_/c: [5506](#)
- _regex_compile_/D: [5211](#)
- _regex_compile_/d: [5211](#)
- _regex_compile_/G: [5228](#)
- _regex_compile_/H: [5211](#)
- _regex_compile_/h: [5211](#)
- _regex_compile_/K: [5816](#)
- _regex_compile_/N: [5211](#)
- _regex_compile_/S: [5211](#)
- _regex_compile_/s: [5211](#)
- _regex_compile_/u: [5680](#)
- _regex_compile_/V: [5211](#)
- _regex_compile_/v: [5211](#)
- _regex_compile_/W: [5211](#)
- _regex_compile_/w: [5211](#)
- _regex_compile_/Z: [5228](#)
- _regex_compile_/z: [5228](#)
- _regex_compile_[: [5277](#)
- _regex_compile_] : [5261](#)
- _regex_compile_~ : [5228](#)
- _regex_compile_abort_tokens:n .
.. [5009](#), [5009](#), [5017](#), [5043](#), [5382](#), [5392](#)
- _regex_compile_anchor_letter:NNN
..... [5228](#), [5228](#),
[5237](#), [5239](#), [5241](#), [5243](#), [5245](#), [5247](#)
- _regex_compile_c[:w [5543](#)
- _regex_compile_cC:NN
..... [5522](#), [5531](#), [5531](#)
- _regex_compile_c_lbrack_add:N .
..... [5543](#), [5569](#), [5584](#)
- _regex_compile_c_lbrack_end: . .
..... [5543](#), [5576](#), [5580](#), [5591](#)
- _regex_compile_c_lbrack_-
loop:NN [5543](#), [5555](#), [5559](#), [5563](#), [5571](#)
- _regex_compile_c_test:NN
..... [5506](#), [5507](#), [5508](#)
- _regex_compile_class:NN
..... [5307](#), [5313](#), [5317](#), [5320](#)
- _regex_compile_class:TFNN
..... [501](#), [5292](#), [5303](#), [5307](#), [5307](#)
- _regex_compile_class_catcode:w
..... [5284](#), [5296](#), [5296](#)
- _regex_compile_class_normal:w .
..... [5287](#), [5290](#), [5290](#)
- _regex_compile_class_posix:NNNNw
..... [5326](#), [5332](#), [5345](#)
- _regex_compile_class_posix_-
end:w [5326](#), [5363](#), [5365](#)
- _regex_compile_class_posix_-
loop:w . [5326](#), [5351](#), [5356](#), [5359](#), [5362](#)
- _regex_compile_class_posix_-
test:w [5280](#), [5326](#), [5326](#)
- _regex_compile_cs_aux:Nn
..... [5615](#), [5628](#), [5641](#), [5649](#)
- _regex_compile_cs_aux:NNnnnN . .
..... [5615](#), [5646](#), [5656](#), [5669](#)
- _regex_compile_end:
..... [492](#), [4889](#), [4902](#), [4962](#), [5624](#)
- _regex_compile_end_cs:
..... [4958](#), [5615](#), [5619](#), [5622](#)
- _regex_compile_escaped:N
..... [4947](#), [4978](#), [4983](#)
- _regex_compile_group_begin:N . .
.. [5396](#), [5396](#), [5444](#), [5449](#), [5467](#), [5469](#)
- _regex_compile_group_end:
..... [5396](#), [5407](#), [5464](#)
- _regex_compile_if_quantifier:TFw
..... [5018](#), [5018](#), [5744](#), [5756](#)
- _regex_compile_lparen:w [5431](#), [5435](#)
- _regex_compile_one:n
..... [4988](#), [4988](#), [5145](#), [5151](#),
[5203](#), [5214](#), [5217](#), [5227](#), [5373](#), [5631](#)
- _regex_compile_quantifier:w
..... [5007](#),
[5025](#), [5025](#), [5272](#), [5416](#), [5749](#), [5765](#)
- _regex_compile_quantifier_*:w [5059](#)
- _regex_compile_quantifier_+:w [5059](#)
- _regex_compile_quantifier_?:w [5059](#)
- _regex_compile_quantifier_-
abort:nNN
.. [5034](#), [5039](#), [5069](#), [5088](#), [5101](#), [5124](#)
- _regex_compile_quantifier_-
braced_auxi:w [5065](#), [5068](#), [5071](#)
- _regex_compile_quantifier_-
braced_auxii:w [5065](#), [5084](#), [5093](#)
- _regex_compile_quantifier_-
braced_auxiii:w [5065](#), [5083](#), [5106](#)
- _regex_compile_quantifier_-
laziness:nnNN [496](#), [5046](#), [5046](#),
[5060](#), [5062](#), [5064](#), [5077](#), [5097](#), [5119](#)
- _regex_compile_quantifier_-
none: [5030](#), [5032](#), [5034](#), [5034](#), [5041](#)
- _regex_compile_range:Nw
..... [5143](#), [5156](#), [5170](#)

- __regex_compile_raw:N 4822, 4943, 4947, 4949, 4981, 4986, 5014, 5136, 5138, 5138, 5158, 5202, 5252, 5275, 5323, 5343, 5361, 5419, 5424, 5429, 5445, 5455, 5463, 5481, 5482, 5483, 5489, 5500, 5501, 5502, 5510, 5565, 5613, 5620, 5685, 5701, 5702, 5708
- __regex_compile_raw_error:N ... 5133, 5133, 5230, 5683, 5820
- __regex_compile_special:N . 488, 4943, 4978, 4978, 5020, 5027, 5048, 5075, 5080, 5095, 5108, 5142, 5160, 5310, 5328, 5347, 5367, 5368, 5437, 5472, 5490, 5533, 5552, 5692, 5711
- __regex_compile_special_group_ - :w 5470
- __regex_compile_special_group_ :w 5466
- __regex_compile_special_group_ i:w 5470, 5470
- __regex_compile_special_group_ l:w 5466
- __regex_compile_u_brace:NNN ... 5686, 5687, 5690, 5690
- __regex_compile_u_end: 5687, 5754, 5754
- __regex_compile_u_in_cs: 5775, 5778, 5778
- __regex_compile_u_in_cs_aux:n .. 5788, 5791
- __regex_compile_u_loop:NN 5696, 5706, 5706, 5709, 5721
- __regex_compile_u_not_cs: 5773, 5797, 5797
- __regex_compile_u_payload: 512, 5754, 5763, 5767, 5769
- __regex_compile_ur:n 512, 5732, 5739, 5741
- __regex_compile_ur_aux:w 5732, 5743, 5753
- __regex_compile_ur_end: 5686, 5700, 5732, 5732
- __regex_compile_use:n 4964, 4964, 6195
- __regex_compile_use_aux:w 4968, 4973
- __regex_compile_l: 5453
- __regex_compute_case_changed_ - char: 4409, 4409, 4425, 6671
- __regex_count:nnN 7285, 7287, 7440, 7440
- \l_regex_cs_flag 5615
- \c_regex_cs_in_class_mode_int .. 4773, 5606
- \c_regex_cs_mode_int ... 4773, 5604
- \l_regex_curr_analysis_tl 536, 6562, 6608, 6635, 6642, 6676, 6677
- \l_regex_curr_catcode_int 4431, 4450, 4458, 4470, 6548, 6674
- \l_regex_curr_char_int 538, 4373, 4379, 4380, 4387, 4397, 4398, 4411, 4412, 4413, 4414, 4419, 4451, 5207, 6222, 6504, 6512, 6548, 6631, 6670, 6673, 6689
- __regex_curr_cs_to_str: 4307, 4307, 4461, 4478
- \l_regex_curr_pos_int 473, 538, 6524, 6543, 6619, 6630, 6669, 6803, 6811, 7396, 7401, 7405, 7406, 7408, 7906, 7911, 7915, 7916
- \l_regex_curr_state_int 535, 541, 6554, 6707, 6708, 6710, 6715, 6718, 6740, 6745, 6750, 6751, 6759, 39773
- \l_regex_curr_submatches_tl ... 6555, 6626, 6720, 6752, 6753, 6764, 6786, 6790, 6815
- \l_regex_curr_token_tl 4310, 6548, 6672
- \l_regex_default_catcodes_int .. 487, 4780, 4894, 4896, 5006, 5302, 5402, 5415
- __regex_disable_submatches: 4474, 5601, 6778, 6778, 7417, 7443, 7804
- \l_regex_empty_success_bool ... 6565, 6611, 6615, 6809, 7504
- \l_regex_end_flag 7373, 7514, 7525, 7574
- __regex_escape_ :w 4597
- __regex_escape_/\scan_stop::w 4597
- __regex_escape_/a:w 4597
- __regex_escape_/e:w 4597
- __regex_escape_/f:w 4597
- __regex_escape_/n:w 4597
- __regex_escape_/r:w 4597
- __regex_escape_/t:w 4597
- __regex_escape_/x:w 4616
- __regex_escape_\ :w 4581
- __regex_escape_\scan_stop::w . 4597
- __regex_escape_escaped:N 4567, 4591, 4594, 4595
- __regex_escape_loop:N 481, 4574, 4581, 4581, 4585, 4588, 4592, 4616, 4655, 4666, 4667, 4687, 4696
- __regex_escape_raw:N 482, 4568, 4594, 4596, 4605, 4607, 4609, 4611, 4613, 4615, 4629
- __regex_escape_unescaped:N 4566, 4584, 4594, 4594
- __regex_escape_use:nnn 39716

- _regex_escape_use:nnnn [480](#), [492](#),
[4562](#), [4562](#), [4940](#), [6882](#), [39709](#), [39712](#)
- _regex_escape_x:N
..... [482](#), [4654](#), [4658](#), [4658](#)
- _regex_escape_x_end:w
..... [482](#), [4616](#), [4618](#), [4621](#)
- _regex_escape_x_large:n [4616](#)
- _regex_escape_x_loop:N
... [482](#), [4651](#), [4670](#), [4670](#), [4679](#), [4682](#)
- _regex_escape_x_loop_error: . [4670](#)
- _regex_escape_x_loop_error:n . .
..... [4676](#), [4688](#), [4693](#)
- _regex_escape_x_test:N
..... [482](#), [4619](#), [4633](#), [4633](#), [4641](#)
- _regex_escape_x_testii:N
..... [4633](#), [4643](#), [4648](#)
- \l_regex_every_match_tl
..... [6564](#), [6646](#), [6656](#), [6693](#)
- _regex_extract:
..... [562](#), [574](#), [7458](#), [7465](#),
[7478](#), [7615](#), [7615](#), [7665](#), [7689](#), [7879](#)
- _regex_extract_all:nnN
..... [7319](#), [7452](#), [7462](#)
- _regex_extract_aux:w
..... [7615](#), [7632](#), [7637](#), [7653](#)
- _regex_extract_check:n
..... [7579](#), [7581](#), [7584](#)
- _regex_extract_check:w
... [564](#), [565](#), [7526](#), [7579](#), [7579](#), [7590](#)
- _regex_extract_check_end:w ...
..... [566](#), [7579](#), [7595](#), [7607](#)
- _regex_extract_check_loop:w ...
..... [7579](#), [7593](#), [7600](#), [7605](#), [7608](#)
- _regex_extract_once:nnN
..... [7317](#), [7452](#), [7452](#)
- _regex_extract_seq:N
..... [7511](#), [7538](#), [7540](#)
- _regex_extract_seq:NNn
..... [7511](#), [7544](#), [7548](#)
- _regex_extract_seq_aux:n
..... [7519](#), [7555](#), [7555](#)
- _regex_extract_seq_aux:ww
..... [7555](#), [7558](#), [7561](#)
- _regex_extract_seq_loop:Nw ...
..... [7511](#), [7543](#), [7550](#), [7553](#)
- \l_regex_fresh_thread_bool
..... [536](#), [541](#), [6534](#),
[6540](#), [6565](#), [6687](#), [6727](#), [6729](#), [6810](#)
- _regex_G_test:
... [486](#), [5239](#), [5879](#), [6003](#), [6483](#), [6522](#)
- _regex_get_digits:NNTFw
..... [4808](#), [4808](#), [5067](#), [5082](#)
- _regex_get_digits_loop:nw
..... [4811](#), [4814](#), [4817](#)
- _regex_get_digits_loop:w ... [4808](#)
- _regex_group:nnnN
..... [486](#), [504](#), [5444](#), [5449](#),
[5735](#), [5866](#), [5986](#), [6157](#), [6334](#), [6334](#)
- _regex_group_aux:nnnnN
..... [528](#), [6317](#), [6317](#),
[6336](#), [6344](#), [6347](#), [39745](#), [39746](#), [39747](#)
- _regex_group_aux:nnnnnN [528](#)
- _regex_group_end_extract_seq:N
... [565](#), [7460](#), [7469](#), [7509](#), [7511](#), [7511](#)
- _regex_group_end_replace:N ...
..... [7680](#), [7713](#), [7715](#), [7715](#)
- _regex_group_end_replace_
check:n [569](#), [7715](#), [7745](#), [7748](#)
- _regex_group_end_replace_
check:w [569](#), [7715](#), [7734](#), [7743](#)
- _regex_group_end_replace_try: .
..... [569](#), [7715](#), [7721](#), [7732](#), [7754](#)
- \l_regex_group_level_int . [4772](#),
[4893](#), [4911](#), [4913](#), [4915](#), [5407](#), [5409](#)
- _regex_group_no_capture:nnnN ..
..... [486](#), [5467](#), [5735](#), [5736](#),
[5748](#), [5760](#), [5867](#), [5988](#), [6334](#), [6343](#)
- _regex_group_repeat:nn
..... [6329](#), [6378](#), [6378](#)
- _regex_group_repeat:nnN
..... [6330](#), [6418](#), [6418](#)
- _regex_group_repeat:nnnN
..... [6331](#), [6449](#), [6449](#)
- _regex_group_repeat_aux:n
[529](#), [531](#), [6385](#), [6398](#), [6398](#), [6436](#), [6453](#)
- _regex_group_resetting:nnnN ...
[486](#), [5469](#), [5736](#), [5868](#), [5990](#), [6345](#), [6345](#)
- _regex_group_resetting_
loop:nnNn .. [6345](#), [6349](#), [6357](#), [6362](#)
- _regex_group_submatches:nnN ...
.. [6386](#), [6391](#), [6391](#), [6421](#), [6437](#), [6451](#)
- _regex_hexadecimal_use:NNTF ...
..... [4653](#), [4665](#), [4678](#), [4698](#), [4698](#)
- _regex_if_end_range:NNTF
..... [5156](#), [5156](#), [5172](#)
- _regex_if_in_class: [4829](#)
- _regex_if_in_class:TF ... [4829](#),
[4904](#), [4991](#), [5007](#), [5140](#), [5201](#), [5263](#),
[5279](#), [5424](#), [5455](#), [5463](#), [7999](#), [8012](#)
- _regex_if_in_class_or_catcode:TF
..... [4847](#), [4847](#), [5230](#), [5252](#), [5682](#)
- _regex_if_in_cs:TF
.. [4837](#), [4837](#), [5611](#), [5618](#), [7997](#), [8006](#)
- _regex_if_match:nn
..... [7274](#), [7280](#), [7414](#), [7414](#), [7433](#)
- _regex_if_raw_digit:NNTF
..... [4810](#), [4816](#), [4820](#), [4820](#)

_regex_if_two_empty_matches:TF
 ... 536, 6565, 6567, 6616, 6622, 6806
 _regex_if_within_catcode: .. 4858
 _regex_if_within_catcode:TF ...
 4858, 5282
 _regex_input_item:n
 570, 574, 575, 7760,
 7761, 7821, 7843, 7884, 7907, 7916
 \l_regex_input_tl
 571, 573, 574, 7760,
 7816, 7820, 7842, 7844, 7905, 7909
 _regex_int_eval:w
 4271, 4271, 4313, 4440,
 4704, 5586, 6242, 6244, 6258, 6259,
 6261, 6262, 6404, 6494, 6537, 6711,
 6759, 6772, 6836, 6837, 6848, 6858,
 7037, 7040, 7642, 7646, 7933, 7938
 _regex_intarray_item:NnTF
 4312, 4312, 6859, 6866
 _regex_intarray_item_aux:nNTF .
 4312, 4313, 4314
 \l_regex_internal_a_int 496, 550,
 4336, 5067, 5078, 5089, 5098, 5102,
 5110, 5113, 5117, 5120, 5127, 6299,
 6302, 6308, 6313, 6387, 6402, 6408,
 6414, 6423, 6426, 6430, 6433, 6438,
 6441, 6444, 6459, 6467, 6476, 7046,
 7067, 7631, 7640, 7642, 7647, 7652
 \l_regex_internal_a_tl 480, 512,
 513, 517, 569, 4336, 4460, 4463,
 4565, 4572, 4579, 5350, 5355, 5371,
 5376, 5381, 5385, 5391, 5392, 5626,
 5637, 5695, 5739, 5771, 5783, 5799,
 5980, 5983, 6036, 6057, 6099, 6106,
 6198, 6199, 6236, 6237, 6238, 6239,
 6369, 6370, 6374, 6376, 6632, 6635,
 7257, 7269, 7670, 7703, 7738, 39711
 \l_regex_internal_b_int
 4336, 5082,
 5111, 5114, 5115, 5117, 5121, 5128,
 6403, 6408, 6413, 6459, 6467, 6476
 \l_regex_internal_b_tl
 4336, 5694, 5714, 5727
 \l_regex_internal_bool
 4336, 5349, 5354, 5375, 5384
 \l_regex_internal_c_int
 4336, 6405, 6410, 6411, 6415
 \l_regex_internal_regex
 ... 491, 4796, 4933, 4971, 5628,
 5634, 6146, 7237, 7242, 7247, 7254
 \l_regex_internal_seq 4336, 6112,
 6113, 6118, 6125, 6126, 6127, 6129
 \g_regex_internal_tl
 564, 565, 4336,
 4570, 4574, 5780, 5787, 7516, 7527,
 7528, 7546, 7591, 7594, 7730, 7735
 _regex_item_caseful_equal:n ...
 486, 4371,
 4371, 4491, 4492, 4496, 4497, 4498,
 4499, 4500, 4509, 4514, 4532, 4550,
 4897, 5494, 5662, 5794, 5913, 6004
 _regex_item_caseful_range:nn ..
 486,
 4371, 4377, 4488, 4503, 4506, 4507,
 4508, 4522, 4529, 4536, 4538, 4540,
 4543, 4544, 4545, 4546, 4551, 4554,
 4559, 4560, 4898, 5496, 5921, 6006
 _regex_item_caseless_equal:n ..
 ... 486, 4385, 4385, 5475, 5914, 6011
 _regex_item_caseless_range:nn .
 ... 486, 4385, 4395, 5477, 5922, 6013
 _regex_item_catcode:
 4428, 4428, 4440
 _regex_item_catcode:n 4438
 _regex_item_catcode:nTF .. 486,
 501, 4428, 4447, 5000, 5304, 5932, 6018
 _regex_item_catcode_reverse:nTF
 ... 486, 4428, 4446, 5305, 5933, 6020
 _regex_item_cs:n
 ... 486, 4468, 4468, 5634, 5911, 6027
 _regex_item_equal:n
 4426, 4426, 4897, 5146,
 5152, 5180, 5193, 5194, 5474, 5493
 _regex_item_exact:nn
 486, 513, 4448, 4448, 5809, 5923, 6024
 _regex_item_exact_cs:n ... 486,
 509, 4448, 4456, 5636, 5806, 5912, 6026
 _regex_item_range:nn
 .. 4426, 4427, 4898, 5182, 5476, 5495
 _regex_item_reverse:n
 486, 502, 4366, 4366, 4447,
 4513, 5218, 5375, 5915, 6022, 6507
 \l_regex_last_char_int
 6504, 6518, 6548, 6670, 6812
 \l_regex_last_char_success_int .
 6548, 6606, 6631, 6812
 \l_regex_left_state_int
 6134, 6155,
 6175, 6179, 6230, 6237, 6248, 6255,
 6258, 6259, 6261, 6262, 6288, 6296,
 6299, 6322, 6370, 6372, 6382, 6402,
 6422, 6424, 6452, 6455, 6458, 6461,
 6473, 6486, 6495, 6531, 6538, 39736
 \l_regex_left_state_seq
 6134, 6229, 6236, 6369
 _regex_maplike_break:
 473, 571, 4321, 4321, 4322,
 6578, 6592, 6637, 6651, 6659, 7824

- _regex_match:n
 6571, 6571, 7420, 7447, 7457, 7467,
 7493, 7662, 7691, 39754, 39757, 39758
- _regex_match_case:nnTF
 7292, 7423, 7423
- _regex_match_case_aux:nn 7423, 7439
- \l_regex_match_count_int
 560, 562, 7372, 7444, 7445, 7450
- _regex_match_cs:n
 4478, 6571, 6580, 39761, 39764, 39765
- _regex_match_init:
 . 6571, 6573, 6583, 6594, 7815, 39769
- _regex_match_once_init:
 .. 6574, 6584, 6613, 6613, 6663, 7817
- _regex_match_once_init_aux: ...
 6633, 6639
- _regex_match_one_active:n
 6666, 6684, 6695
- _regex_match_one_token:nnN ...
 . 538, 541, 571, 6576, 6577, 6588,
 6589, 6591, 6636, 6666, 6666, 7822
- \l_regex_match_success_bool ...
 ... 536, 6568, 6625, 6650, 6658, 6808
- \l_regex_matched_analysis_tl ...
 536, 6562, 6607, 6632, 6641, 6675, 6813
- \l_regex_max_pos_int
 545, 6543, 7401,
 7499, 7505, 7678, 7711, 7897, 7911
- \l_regex_max_state_int 521, 524,
 583, 6131, 6152, 6172, 6207, 6209,
 6210, 6214, 6247, 6249, 6250, 6309,
 6381, 6401, 6403, 6411, 6455, 6461,
 6469, 6479, 6598, 8271, 39738, 39739
- \l_regex_max_thread_int
 6558, 6582,
 6628, 6680, 6683, 6688, 6765, 6773
- _regex_maybe_compute_ccc:
 4390, 4402, 4423, 4425, 6671
- \l_regex_min_pos_int
 545, 6543, 6604, 6605
- \l_regex_min_state_int 524, 6131,
 6152, 6172, 6214, 6598, 6629, 8270
- \l_regex_min_submatch_int
 560, 564,
 568, 6609, 6610, 7375, 7518, 7698, 7706
- \l_regex_min_thread_int
 .. 6558, 6582, 6628, 6680, 6682, 6688
- \l_regex_mode_int 4773,
 4831, 4839, 4841, 4849, 4851, 4860,
 4868, 4870, 4880, 4881, 4883, 4885,
 4939, 4953, 4955, 5265, 5269, 5270,
 5271, 5298, 5309, 5426, 5516, 5517,
 5545, 5546, 5602, 5603, 5772, 5818
- _regex_mode_quit_c:
 4878, 4878, 4990, 5399
- _regex_msg_repeated:nnN
 6072, 6093, 6103, 8240, 8240
- _regex_multi_match:n
 536, 6644, 6654, 7445, 7465, 7474, 7689
- \c_regex_no_match_regex
 4345, 4796, 7229
- \c_regex_outer_mode_int
 4773, 4841, 4851, 4860,
 4868, 4881, 4939, 4955, 5772, 5818
- _regex_peek:nnTF
 573, 7764, 7773, 7782, 7792, 7800, 7800
- _regex_peek_aux:nnTF
 7800, 7802, 7808, 7873
- _regex_peek_end:
 570, 572, 7766, 7775, 7828, 7828
- \l_regex_peek_false_tl
 7757, 7812, 7832, 7838, 7901
- _regex_peek_reinsert:N ... 572,
 574, 7831, 7832, 7838, 7840, 7840, 7901
- _regex_peek_remove_end:n
 570, 572, 7784, 7794, 7828, 7834
- _regex_peek_replace:nnTF
 7855, 7863, 7870, 7870
- _regex_peek_replace_end:
 7873, 7875, 7875
- _regex_peek_replacement_put:n .
 7881, 7918, 7918
- _regex_peek_replacement_put_-
 submatch_aux:n .. 7883, 7929, 7929
- _regex_peek_replacement_-
 token:n 575, 7885, 7927, 7927
- _regex_peek_replacement_var:N .
 7886, 7943, 7943
- \l_regex_peek_true_tl
 572, 574, 7757, 7811, 7831, 7837, 7890
- _regex_pop_lr_states:
 6190, 6219, 6227, 6234, 6327
- _regex_posix_alnum: ... 4516, 4516
- _regex_posix_alpha:
 516, 4516, 4517, 4518
- _regex_posix_ascii: ... 4516, 4520
- _regex_posix_blank: ... 4516, 4526
- _regex_posix_cntrl: ... 4516, 4527
- _regex_posix_digit:
 4516, 4517, 4534, 4558
- _regex_posix_graph: ... 4516, 4535
- _regex_posix_lower: 4516, 4519, 4537
- _regex_posix_print: ... 4516, 4539
- _regex_posix_punct: ... 4516, 4541
- _regex_posix_space: ... 4516, 4548
- _regex_posix_upper: 4516, 4519, 4553
- _regex_posix_word: 4516, 4555

- `__regex_posix_xdigit`: ... [4516](#), [4556](#)
- `__regex_prop_:` [499](#), [5199](#)
- `__regex_prop_d`:
 - [499](#), [516](#), [4487](#), [4487](#), [4534](#)
- `__regex_prop_h`: [4487](#), [4489](#), [4526](#)
- `__regex_prop_N`: [4487](#), [4511](#), [5227](#)
- `__regex_prop_s`: [4487](#), [4494](#)
- `__regex_prop_v`: [4487](#), [4502](#)
- `__regex_prop_w`:
 - .. [4487](#), [4504](#), [4555](#), [6505](#), [6507](#), [6508](#)
- `__regex_push_lr_states`:
 - [6181](#), [6217](#), [6227](#), [6227](#), [6325](#)
- `__regex_quark_if_nil:N` [4362](#)
- `__regex_quark_if_nil:NTF` [5652](#), [5672](#)
- `__regex_quark_if_nil:nTF` [4362](#)
- `__regex_quark_if_nil_p:n` [4362](#)
- `__regex_query_range:nn`
 - [545](#), [574](#), [6827](#), [6833](#),
 - [6833](#), [6852](#), [6923](#), [7673](#), [7710](#), [7892](#)
- `__regex_query_range_loop:ww` ...
 - [6833](#), [6835](#), [6840](#), [6847](#)
- `__regex_query_set:n` [7393](#),
- [7393](#), [7459](#), [7468](#), [7494](#), [7666](#), [7692](#)
- `__regex_query_set_aux:nN`
 - [7393](#), [7397](#), [7399](#), [7400](#), [7403](#)
- `__regex_query_set_from_input_-tl:` [7880](#), [7903](#), [7903](#)
- `__regex_query_set_item:n`
 - [7903](#), [7907](#), [7908](#), [7910](#), [7913](#)
- `__regex_query_submatch:n`
 - .. [6850](#), [6850](#), [7037](#), [7570](#), [7933](#), [7938](#)
- `__regex_reinsert_item:n`
 - [573](#), [574](#), [7840](#), [7843](#), [7846](#), [7884](#), [7922](#)
- `__regex_replace_all:nnN`
 - [7323](#), [7684](#), [7684](#)
- `__regex_replace_all_aux:nnN` ...
 - [7359](#), [7685](#), [7686](#)
- `__regex_replace_once:nnN`
 - [7321](#), [7655](#), [7655](#)
- `__regex_replace_once_aux:nnN` ...
 - [7336](#), [7655](#), [7656](#), [7657](#)
- `__regex_replacement:n`
 - [574](#), [6874](#), [6874](#), [6918](#), [7338](#),
 - [7656](#), [7685](#), [7887](#), [39778](#), [39779](#), [39780](#)
- `__regex_replacement_apply:Nn` ...
 - [6874](#), [6875](#), [6876](#), [6953](#)
- `__regex_replacement_balance_-one_match:n`
 - ... [544](#), [6823](#), [6823](#), [6935](#), [7669](#), [7701](#)
- `__regex_replacement_c:w` . [7076](#), [7076](#)
- `__regex_replacement_c_A:w`
 - [548](#), [7008](#), [7164](#), [7165](#)
- `__regex_replacement_c_B:w`
 - [6996](#), [7167](#), [7168](#)
- `__regex_replacement_c_C:w` [7176](#), [7176](#)
- `__regex_replacement_c_D:w`
 - [7003](#), [7181](#), [7182](#)
- `__regex_replacement_c_E:w`
 - [6997](#), [7184](#), [7185](#)
- `__regex_replacement_c_L:w`
 - [7006](#), [7193](#), [7194](#)
- `__regex_replacement_c_M:w`
 - [6998](#), [7196](#), [7197](#)
- `__regex_replacement_c_O:w` [6995](#),
- [7000](#), [7004](#), [7007](#), [7009](#), [7199](#), [7200](#)
- `__regex_replacement_c_P:w`
 - [7001](#), [7202](#), [7203](#)
- `__regex_replacement_c_S:w`
 - [6991](#), [7005](#), [7208](#), [7208](#)
- `__regex_replacement_c_T:w`
 - [6999](#), [7216](#), [7217](#)
- `__regex_replacement_c_U:w`
 - [7002](#), [7219](#), [7220](#)
- `__regex_replacement_cat:NNN` ...
 - [7081](#), [7124](#), [7124](#)
- `\l__regex_replacement_category_-seq` [6820](#), [6905](#), [6908](#), [6909](#), [6978](#), [7138](#)
- `\l__regex_replacement_category_-tl` [548](#),
- [6820](#), [6973](#), [6979](#), [6982](#), [7139](#), [7140](#)
- `__regex_replacement_char:nNN` ...
 - [555](#),
 - [7159](#), [7159](#), [7166](#), [7173](#), [7183](#), [7190](#),
 - [7195](#), [7198](#), [7201](#), [7205](#), [7218](#), [7221](#)
- `\l__regex_replacement_csnames_-int` [543](#), [6819](#), [6899](#), [6901](#), [6903](#),
- [6970](#), [7038](#), [7092](#), [7099](#), [7109](#), [7111](#),
- [7118](#), [7129](#), [7170](#), [7187](#), [7920](#), [7931](#)
- `__regex_replacement_cu_aux:Nw` ...
 - [7086](#), [7090](#), [7090](#), [7104](#)
- `__regex_replacement_do_one_-match:n` [574](#),
- [575](#), [6825](#), [6825](#), [6921](#), [7672](#), [7709](#), [7891](#)
- `__regex_replacement_error:NNN` ...
 - [7047](#), [7059](#),
 - [7070](#), [7082](#), [7087](#), [7105](#), [7223](#), [7223](#)
- `__regex_replacement_escaped:N` ...
 - [6895](#), [7014](#), [7014](#), [7143](#)
- `__regex_replacement_exp_not:N` ...
 - ... [551](#), [6831](#), [6831](#), [7086](#), [7179](#), [7885](#)
- `__regex_replacement_exp_not:n` ...
 - [6832](#), [6832](#), [7104](#), [7886](#)
- `__regex_replacement_g:w` . [7043](#), [7043](#)
- `__regex_replacement_g_digits:NN`
 - [7043](#), [7046](#), [7049](#), [7056](#)
- `__regex_replacement_lbrace:N` ...
 - .. [6888](#), [7045](#), [7085](#), [7103](#), [7116](#), [7116](#)

- __regex_replacement_normal:n . . .
 6890, 6896, 6968, 6968,
 7021, 7051, 7078, 7113, 7121, 7136
- __regex_replacement_normal_
 aux:N 6968, 6974, 6988
- __regex_replacement_put:n
 6966, 6966, 6971, 7162, 7214, 7881
- __regex_replacement_put_
 submatch:n 7019, 7025, 7025, 7066
- __regex_replacement_put_
 submatch_aux:n
 7025, 7028, 7034, 7882
- __regex_replacement_rbrace:N
 6885, 7065, 7107, 7107
- __regex_replacement_set:n
 6874, 6875, 6919, 6956
- \l__regex_replacement_tl
 7759, 7872, 7887
- __regex_replacement_u:w 7101, 7101
- __regex_return:
 558, 7275, 7281, 7309, 7311, 7385, 7385
- \l__regex_right_state_int
 6134, 6158, 6199, 6200, 6220,
 6232, 6239, 6248, 6249, 6288, 6295,
 6301, 6314, 6322, 6372, 6376, 6387,
 6401, 6410, 6422, 6426, 6430, 6433,
 6438, 6441, 6444, 6452, 6466, 6469,
 6472, 6475, 6479, 6495, 6538, 39737
- \l__regex_right_state_seq
 6134, 6198, 6208, 6231, 6238, 6374
- \l__regex_saved_success_bool
 536, 4476, 4483, 6568
- __regex_show:N
 556, 5973, 5973, 7254, 7266
- __regex_show:NN 7249, 7259, 7260, 7261
- __regex_show:Nn 7249, 7249, 7250, 7251
- __regex_show_char:n 6005,
 6009, 6012, 6016, 6025, 6038, 6038
- __regex_show_class:NnnnN
 5992, 6074, 6074
- __regex_show_group_aux:nnnnN
 5987, 5989, 5991, 6065, 6065
- __regex_show_item_catcode:NnTF
 6019, 6021, 6110, 6110
- __regex_show_item_exact_cs:n
 6026, 6123, 6123
- \l__regex_show_lines_int
 4798, 6046, 6078, 6081, 6088
- __regex_show_one:n
 5981, 5994, 5997, 6005, 6008,
 6012, 6015, 6025, 6029, 6044, 6044,
 6060, 6067, 6071, 6084, 6100, 6128
- __regex_show_pop:
 6054, 6056, 6063, 6070
- \l__regex_show_prefix_seq 4797,
 5979, 5982, 6030, 6050, 6055, 6057
- __regex_show_push:n
 6031, 6054, 6054, 6061, 6068, 6079
- __regex_show_scope:nn
 6023, 6028, 6054, 6058, 6115
- __regex_single_match: 536, 4473,
 6644, 6644, 7418, 7455, 7660, 7813
- __regex_split:nnN 7325, 7471, 7471
- __regex_standard_escapechar:
 4272, 4272, 4569, 4938, 6150, 6171
- \l__regex_start_pos_int
 6524, 6543, 6619, 6624, 6630,
 7477, 7489, 7502, 7505, 7628, 7711
- \g__regex_state_active_intarray
 471, 524, 535–
 537, 6560, 6601, 6706, 6709, 6717, 6744
- \l__regex_step_int 471, 6557, 6603,
 6668, 6707, 6711, 6719, 6733, 6735
- __regex_store_state:n
 535, 6629, 6758, 6761, 6761
- __regex_store_submatches: 6761
- __regex_store_submatches:n 6780
- __regex_store_submatches:nn
 6763, 6767
- __regex_submatch_balance:n
 6824, 6856, 6856, 6938, 7040, 7559
- \g__regex_submatch_begin_
 intarray
 471, 544, 567, 6829, 6853, 6869,
 6930, 7378, 7484, 7487, 7500, 7641
- \g__regex_submatch_case_intarray
 6949, 7378, 7623, 7629
- \g__regex_submatch_end_intarray
 471, 567, 6854, 6862,
 7378, 7481, 7497, 7644, 7675, 7894
- \l__regex_submatch_int
 471, 560, 563, 564, 568, 6610, 7375,
 7496, 7498, 7501, 7503, 7506, 7519,
 7618, 7622, 7624, 7625, 7700, 7708
- \g__regex_submatch_prev_intarray
 471, 560, 566, 6828,
 6926, 7378, 7479, 7495, 7621, 7627
- \g__regex_success_bool
 536, 4477, 4479, 4482, 6568,
 6596, 6649, 6661, 7340, 7363, 7387,
 7436, 7617, 7663, 7830, 7836, 7877
- \l__regex_success_pos_int
 6543, 6605, 6624, 6811, 7477
- \l__regex_success_submatches_tl
 535, 567, 6555, 6814, 7632
- __regex_tests_action_cost:n
 6266, 6268, 6281, 6287, 6296, 6314

- \g_regex_thread_info_intarray
 . 471, 534–536, 542, 6560, 6699, 6770
 - __regex_tl_even_items:n
 4323, 4323, 4324, 7361
 - __regex_tl_even_items_loop:nn
 4323, 4326, 4329, 4333
 - __regex_tl_odd_items:n
 4323, 4323, 7337, 7360, 7434
 - __regex_tmp:w 564, 4335, 4335, 5211,
 5221, 5222, 5223, 5224, 5225, 5248,
 5259, 5260, 7304, 7317, 7319, 7321,
 7323, 7325, 7515, 7520, 7543, 7550,
 7557, 7595, 7600, 7604, 7608, 7613
 - __regex_toks_clear:N
 4275, 4275, 6207, 6247
 - __regex_toks_memcpy:Nn
 4280, 4280, 6412
 - __regex_toks_put_left:Nn
 4289, 4290,
 4292, 6179, 6200, 6242, 6394, 6395
 - __regex_toks_put_right:Nn
 472, 4289,
 4296, 4298, 4302, 4304, 6155, 6158,
 6175, 6220, 6244, 6255, 6486, 6531
 - __regex_toks_set:Nn
 4275, 4277, 4278, 7406, 7916
 - __regex_toks_use:w
 4274, 4274, 6708, 6846, 8274
 - __regex_trace:nnn
 . . . 8256, 8257, 8259, 8260, 8273,
 39733, 39755, 39762, 39767, 39772
 - __regex_trace_pop:nnN
 . 8256, 8258, 39712, 39721, 39728,
 39746, 39750, 39757, 39764, 39779
 - __regex_trace_push:nnN
 . 8256, 8256, 39709, 39718, 39725,
 39745, 39749, 39754, 39761, 39778
 - \g_regex_trace_regex_int 8266
 - __regex_trace_states:n
 8267, 8267, 39720, 39727
 - __regex_two_if_eq:NNNNTF
 4799, 4799, 5048, 5095, 5108, 5142,
 5310, 5347, 5367, 5368, 5437, 5472,
 5489, 5490, 5552, 5685, 5692, 7136
 - __regex_use_i_delimit_by_q_-
 recursion_stop:nw 4357, 4359, 5675
 - __regex_use_none_delimit_by_q_-
 nil:w 4331, 4357, 4361
 - __regex_use_none_delimit_by_q_-
 recursion_stop:w
 4357, 4357, 5653, 5677, 7633
 - __regex_use_state:
 6704, 6704, 6721, 6747, 39776
 - __regex_use_state_and_submatches:w
 539, 6697, 6713, 6713
 - __regex_Z_test: 486, 5241,
 5243, 5260, 5880, 6001, 6483, 6510
 - \l_regex_zerOTH_submatch_int
 560, 566, 7375, 7480, 7482,
 7485, 7488, 7618, 7628, 7630, 7642,
 7647, 7669, 7672, 7676, 7891, 7895
 - register commands:
 register_luaDATA 12063
 - \relax . . . 4, 8, 13, 17, 53, 54, 61, 85, 86,
 87, 88, 89, 90, 91, 92, 93, 94, 96, 97,
 98, 99, 100, 101, 102, 103, 104, 105, 384
 - \relpenalty 385
 - \resettimer 777
 - reverse commands:
 \reverse_if:N
 28, 692, 749, 848, 849, 1061, 1392,
 1397, 4379, 4380, 4397, 4398, 4403,
 4404, 8493, 12102, 13990, 17546,
 17697, 17699, 17701, 17703, 17766,
 20869, 20874, 20878, 20880, 24102,
 27736, 28558, 28581, 31248, 31272
 - \right 386
 - \rightghost 911
 - \righthyphenmin 387
 - \rightmarginkern 680
 - \rightskip 388
 - \rmfamily 34288
 - \romannumeral 389
 - round 276
 - \rPCODE 681
- S**
- \saveboxresource 958
 - \savecatcodetable 912
 - \saveimageresource 959
 - \savepos 957
 - \savingsphcodes 520
 - \savingsdiscards 521
 - scan commands:
 \scan_new:N 150,
 729, 817, 3126, 3127, 3536, 8538,
 8539, 9239, 9240, 10463, 10464,
 10964, 13018, 13295, 13296, 13297,
 13405, 13406, 14175, 16636, 16636,
 16663, 16664, 16665, 17455, 17456,
 18377, 18378, 19059, 19284, 19285,
 19709, 19710, 19907, 19908, 19913,
 20749, 20750, 21180, 21343, 21344,
 21345, 21346, 21596, 21597, 21598,
 23218, 23221, 23222, 23223, 23224,
 23226, 23227, 23228, 23229, 23230,

- 23329, 29496, 31386, 31395, 31396,
36693, 36707, 38442, 39024, 39651
- `\scan_stop`: *13, 22, 23, 150, 167, 207, 366, 370, 389, 393, 403, 409, 463, 464, 478, 486, 509, 585, 667, 695, 700, 707, 708, 710, 714, 720, 749, 816, 848, 853, 902, 910, 912, 914, 915, 933, 937, 944, 945, 950, 1057, 1061–1063, 1066, 1300, 1478, 121, 134, 1421, 1421, 1828, 1850, 1866, 1874, 1894, 1910, 1945, 1958, 2326, 2349, 2358, 2367, 2432, 2698, 2699, 2714, 2754, 2780, 2804, 2821, 3011, 3017, 3160, 3541, 3697, 3737, 3741, 3747, 3749, 3796, 3798, 4070, 4083, 4092, 4094, 4104, 4145, 4146, 4147, 4153, 4440, 4461, 4462, 4575, 4635, 4660, 4672, 4704, 4818, 5586, 5645, 5963, 5967, 5970, 6125, 6711, 6723, 6872, 7037, 7040, 7161, 7213, 7515, 7933, 7938, 8893, 8897, 9127, 9139, 9151, 9210, 10253, 10258, 10380, 10504, 10507, 11079, 11086, 11118, 11449, 11495, 12376, 12560, 12570, 12633, 12663, 12665, 12973, 12993, 13007, 13991, 14285, 15276, 16358, 16645, 16648, 17095, 17896, 19269, 19377, 19446, 19758, 19819, 19895, 19898, 19900, 20316, 20464, 20761, 20780, 20782, 20786, 20789, 20792, 20796, 20801, 20805, 21028, 21190, 21208, 21210, 21218, 21220, 21224, 21226, 21247, 21252, 21255, 21281, 21301, 21303, 21311, 21313, 21317, 21319, 21323, 22785, 22868, 22967, 23005, 23012, 23172, 23204, 23393, 24100, 24104, 24305, 24322, 24623, 24670, 24671, 24926, 24969, 24997, 25011, 25833, 27647, 27655, 28400, 28403, 28406, 28409, 28412, 28415, 28418, 28421, 28424, 29465, 29488, 29728, 29859, 30422, 30449, 30658, 31412, 31413, 34633, 34781, 36473, 38750, 38753, 39165, 39188, 39201, 39283, 39308, 39370, 39379, 39793*
- `\s_stop` *150, 817, 16648, 16659*
- scan internal commands:
- `\s__bool_mark` *8538, 8551, 8559*
- `\s__bool_stop` *8538, 8551, 8559*
- `\s__char_stop` *19059*
- `\s__clist_mark` *875, 877–879, 884, 18377, 18379, 18407, 18408, 18425, 18554, 18564, 18568, 18590, 18640, 18646, 18660, 18672, 18673, 18674, 18677, 18678, 18679, 18688, 18689, 18698, 18896, 18897, 18909, 18910, 18923, 18931, 18937, 18940*
- `\s__clist_stop` *878, 880, 884, 18377, 18380, 18381, 18393, 18397, 18539, 18542, 18554, 18557, 18565, 18568, 18576, 18590, 18646, 18674, 18677, 18678, 18690, 18698, 18741, 18742, 18749, 18753, 18755, 18757, 18764, 18770, 18786, 18787, 18813, 18814, 18821, 18826, 18828, 18830, 18836, 18843, 18871, 18876, 18898, 18909, 18910, 18911, 18924, 18937, 18940, 18970, 19005*
- `\s__color_mark` *36707, 36966, 36968, 36971, 36978, 37219, 37224, 37230, 37233, 37240, 37271, 37371, 37377, 37457, 37460, 37470, 37519, 37887, 37929, 37932, 37950, 37956, 38072, 38101, 38104, 38118, 38129, 38133, 38136, 38144, 38150, 38154, 38157, 38170, 38180*
- `\s__color_stop` *1415, 36693, 36736, 36742, 36743, 36750, 36754, 36755, 36758, 36764, 36766, 36768, 36770, 36772, 36774, 36793, 36795, 36801, 36821, 36850, 36867, 36873, 36890, 36966, 36968, 36971, 36978, 36985, 36987, 36996, 37007, 37008, 37010, 37012, 37014, 37054, 37061, 37062, 37063, 37072, 37081, 37146, 37151, 37179, 37219, 37224, 37230, 37233, 37240, 37248, 37271, 37371, 37377, 37408, 37411, 37445, 37451, 37457, 37460, 37470, 37519, 37538, 37542, 37567, 37569, 37571, 37573, 37591, 37706, 37719, 37723, 37734, 37741, 37749, 37755, 37763, 37765, 37771, 37775, 37776, 37797, 37799, 37878, 37884, 37887, 37895, 37909, 37923, 37926, 37929, 37933, 37936, 37946, 37950, 37956, 37983, 38012, 38067, 38068, 38075, 38086, 38087, 38101, 38104, 38118, 38129, 38133, 38136, 38144, 38150, 38154, 38157, 38170, 38180, 38224, 38225*
- `\s__cs_mark` *388, 389, 417, 419, 1816, 1817, 1820, 1821, 1822, 2698, 2728, 2729, 2731, 2737, 2741, 2763, 2772, 2791, 2819, 2822, 2830, 2845, 2877, 2891, 2895, 2904, 2923, 2932, 2937, 3012, 3015, 3031, 16655*
- `\s__cs_stop` *388, 419, 1817, 1820, 1821, 1822, 2698, 2701, 2702,*

- 2732, 2741, 2767, 2819, 2822, 2826,
2834, 2840, 2849, 2855, 2857, 2877,
2899, 2904, 2934, 2937, 3012, 16656
- `\s__debug_stop` 39024,
39025, 39151, 39153, 39344, 39358
- `\s__dim_mark` 20749, 20910, 20917
- `\s__dim_stop` 20749,
20751, 20857, 20881, 20910, 20917
- `\s__file_stop` .. 676, 10937, 10942,
10964, 11032, 11033, 11037, 11044,
11046, 11047, 11172, 11173, 11178,
11180, 11182, 11514, 11516, 11519,
11520, 11522, 11534, 11610, 11613,
11620, 11622, 11638, 11639, 11642
- `\s__fp` 1018–1020, 1025,
1026, 1051, 1057, 1059, 1061, 1075,
1077, 1078, 1109, 1113, 1115, 1117,
1123, 1126, 1216, 23218, 23231,
23232, 23233, 23234, 23235, 23245,
23250, 23252, 23253, 23268, 23281,
23284, 23286, 23296, 23308, 23328,
23345, 23348, 23355, 23362, 23378,
23405, 23511, 23513, 23515, 23516,
23517, 23519, 23520, 23521, 23523,
23539, 23699, 23704, 23931, 23985,
23994, 23996, 24672, 24827, 25309,
25324, 25348, 25368, 25369, 25467,
25482, 25484, 25502, 25507, 25508,
25567, 25603, 25604, 25618, 25619,
25656, 25657, 25760, 25761, 25762,
25771, 25787, 25791, 25855, 25856,
25859, 25870, 25871, 25879, 25880,
25882, 25883, 25884, 25886, 25887,
25888, 25900, 25903, 25907, 25910,
25930, 25980, 25983, 25986, 26006,
26007, 26009, 26010, 26011, 26019,
26022, 26033, 26034, 26036, 26045,
26121, 26273, 26307, 26308, 26311,
26392, 26530, 26538, 26540, 26717,
26726, 26728, 26733, 26741, 26743,
26745, 26748, 27252, 27264, 27266,
27475, 27492, 27494, 27675, 27694,
27696, 27697, 27700, 27717, 27720,
27723, 27747, 27748, 27750, 27766,
27855, 27868, 27870, 27873, 27878,
27911, 27927, 28010, 28023, 28025,
28038, 28040, 28053, 28055, 28068,
28070, 28083, 28085, 28098, 28108,
28609, 28625, 28626, 28630, 28641,
28748, 28761, 28763, 28779, 28782,
28792, 28815, 28826, 28828, 28842,
28844, 28849, 28911, 28932, 28935,
28965, 28986, 28989, 29039, 29055,
29058, 29133, 29134, 29218, 29220,
29252, 30060, 30068, 30071, 30150
- `\s__fp_(type)` 1051
- `\s__fp_division` 23226
- `\s__fp_exact` 23226, 23231,
23232, 23233, 23234, 23235, 25855
- `\s__fp_expr_mark`
... 1057, 1058, 1061, 1083, 1086,
23221, 24876, 24889, 24970, 25012
- `\s__fp_expr_stop`
1027, 23221, 23419, 24778, 24877,
24881, 24890, 25937, 25948, 25958,
25966, 29524, 29684, 29876, 29962
- `\s__fp_invalid` 23226
- `\s__fp_mark`
23223, 23368, 23369, 23373, 29439,
29441, 29449, 29508, 29509, 29513
- `\s__fp_overflow` 23226, 23252
- `\s__fp_stop`
..... 1025, 23223, 23225, 23269,
23345, 23356, 23363, 23369, 23373,
23387, 23406, 24200, 24204, 24708,
24713, 25309, 25331, 25485, 25490,
25495, 25502, 25513, 25519, 25566,
25567, 25603, 25604, 25760, 25761,
25762, 25929, 25930, 27551, 27566,
28885, 28889, 29443, 29510, 29513
- `\s__fp_symbolic` 1232, 1233, 25469,
25492, 25495, 25519, 25523, 29496,
29502, 29509, 29513, 29515, 29533,
29546, 29566, 29595, 29618, 29690,
29694, 29711, 29869, 29907, 29916
- `\s__fp_tuple` 1024,
23329, 23335, 23336, 23413, 23415,
25089, 25301, 25316, 25341, 25343,
25360, 25361, 25363, 25468, 25487,
25490, 25513, 25516, 25648, 25649,
26780, 26781, 26787, 26788, 28861
- `\s__fp_underflow` 23226, 23250
- `\s__int_mark`
.. 17455, 17670, 17673, 17747, 17754
- `\s__int_stop` 848, 860, 17455, 17457,
17649, 17665, 17667, 17671, 17684,
17747, 17754, 18160, 18166, 18183
- `\s__iow_mark` . 10463, 10827, 10834,
10846, 10920, 10921, 10922, 10923
- `\s__iow_stop`
.... 10463, 10465, 10713, 10754,
10812, 10850, 10863, 10920, 10923
- `\s__kernel_stop` 2339, 2347, 2356, 2365
- `\s__keys_mark` 21596,
21657, 21660, 21672, 21674, 21678,
22360, 22363, 22368, 22374, 22617,
22620, 22629, 22631, 22636, 22639

- `\s__keys_nil` [21596](#), [21652](#),
[21653](#), [21655](#), [21657](#), [21660](#), [21669](#),
[21670](#), [21672](#), [21674](#), [21677](#), [21678](#),
[21685](#), [22355](#), [22356](#), [22358](#), [22360](#),
[22363](#), [22366](#), [22374](#), [22375](#), [22616](#),
[22619](#), [22625](#), [22627](#), [22635](#), [22638](#)
- `\s__keys_stop`
[21596](#), [21695](#), [21705](#), [21840](#), [21887](#),
[21990](#), [21997](#), [22343](#), [22353](#), [22549](#),
[22569](#), [22692](#), [22699](#), [22704](#), [22709](#)
- `\s__keyval_mark`
[965–967](#), [970](#), [21343](#), [21357](#), [21368](#),
[21369](#), [21370](#), [21371](#), [21377](#), [21378](#),
[21380](#), [21385](#), [21386](#), [21389](#), [21390](#),
[21391](#), [21396](#), [21397](#), [21401](#), [21402](#),
[21405](#), [21406](#), [21409](#), [21410](#), [21413](#),
[21416](#), [21417](#), [21422](#), [21425](#), [21426](#),
[21430](#), [21431](#), [21434](#), [21437](#), [21441](#),
[21442](#), [21443](#), [21444](#), [21451](#), [21452](#),
[21461](#), [21462](#), [21464](#), [21468](#), [21482](#),
[21483](#), [21491](#), [21492](#), [21501](#), [21502](#),
[21503](#), [21504](#), [21506](#), [21527](#), [21528](#),
[21533](#), [21537](#), [21539](#), [21541](#), [21553](#)
- `\s__keyval_nil` [966](#),
[21343](#), [21376](#), [21384](#), [21389](#), [21391](#),
[21392](#), [21393](#), [21395](#), [21401](#), [21404](#),
[21409](#), [21413](#), [21415](#), [21422](#), [21424](#),
[21430](#), [21434](#), [21436](#), [21441](#), [21443](#),
[21451](#), [21462](#), [21482](#), [21491](#), [21526](#),
[21530](#), [21546](#), [21549](#), [21553](#), [21554](#)
- `\s__keyval_stop` [21343](#), [21369](#),
[21371](#), [21382](#), [21390](#), [21402](#), [21410](#),
[21413](#), [21419](#), [21431](#), [21434](#), [21436](#),
[21437](#), [21441](#), [21451](#), [21482](#), [21483](#),
[21491](#), [21492](#), [21501](#), [21504](#), [21506](#)
- `\s__keyval_tail` [966](#),
[21343](#), [21357](#), [21365](#), [21366](#), [21375](#),
[21459](#), [21461](#), [21467](#), [21468](#), [21503](#)
- `\s__msg_mark`
. [9239](#), [9576](#), [9659](#), [9660](#), [9665](#), [9668](#)
- `\s__msg_stop` [9239](#),
[9241](#), [9578](#), [9582](#), [9584](#), [9661](#), [10144](#)
- `\s__pdf_stop` [38442](#), [38642](#), [38643](#),
[38651](#), [38663](#), [38676](#), [38680](#), [38682](#)
- `\s__peek_mark`
. [19709](#), [19872](#), [19873](#), [19880](#)
- `\s__peek_stop`
. [19709](#), [19711](#), [19861](#), [19874](#), [19883](#)
- `\s__prg_mark` [1663](#), [1665](#), [1673](#)
- `\s__prg_stop` [1690](#), [1695](#), [1714](#), [1722](#),
[1730](#), [1786](#), [1790](#), [1792](#), [1794](#), [1796](#)
- `\s__prop` [915–](#)
[917](#), [922](#), [926](#), [934](#), [936](#), [937](#), [939–](#)
[941](#), [19913](#), [19921](#), [19923](#), [19924](#),
[19928](#), [19931](#), [19933](#), [19935](#), [19938](#),
[19969](#), [19988](#), [20011](#), [20014](#), [20020](#),
[20023](#), [20062](#), [20072](#), [20075](#), [20078](#),
[20081](#), [20084](#), [20088](#), [20152](#), [20209](#),
[20212](#), [20217](#), [20256](#), [20265](#), [20269](#),
[20292](#), [20303](#), [20323](#), [20338](#), [20339](#),
[20442](#), [20448](#), [20456](#), [20487](#), [20506](#),
[20510](#), [20547](#), [20548](#), [20549](#), [20550](#),
[20554](#), [20555](#), [20556](#), [20557](#), [20571](#),
[20585](#), [20586](#), [20587](#), [20588](#), [20592](#),
[20593](#), [20594](#), [20595](#), [20646](#), [20648](#),
[20649](#), [20652](#), [20663](#), [20666](#), [20669](#),
[20673](#), [20684](#), [20705](#), [20736](#), [20737](#)
- `\s__prop_mark`
. [926](#), [927](#), [19907](#), [19964](#), [19965](#),
[19968](#), [20209](#), [20211](#), [20213](#), [20264](#),
[20265](#), [20269](#), [20689](#), [20708](#), [20709](#)
- `\s__prop_stop` [926](#),
[19907](#), [19965](#), [19968](#), [20209](#), [20214](#),
[20222](#), [20223](#), [20266](#), [20269](#), [20506](#),
[20510](#), [20652](#), [20669](#), [20689](#), [20709](#)
- `\s__quark`
[16358](#), [16607](#), [16609](#), [16610](#), [16621](#),
[16624](#), [16629](#), [16632](#), [16634](#), [16653](#)
- `\g__scan_marks_tl`
. [817](#), [16638](#), [16644](#), [16648](#)
- `\s__seq`
[818](#), [822](#), [826](#), [830](#), [834](#), [836](#), [838](#),
[16663](#), [16674](#), [16704](#), [16709](#), [16714](#),
[16719](#), [16730](#), [16762](#), [16802](#), [16810](#),
[16814](#), [16918](#), [16930](#), [16932](#), [17097](#),
[17145](#), [17299](#), [17305](#), [17387](#), [17426](#)
- `\s__seq_mark`
. [16664](#), [17375](#), [17376](#), [17390](#), [17393](#)
- `\s__seq_stop`
[16664](#), [16921](#), [16932](#), [17050](#), [17053](#),
[17061](#), [17063](#), [17144](#), [17145](#), [17298](#),
[17299](#), [17301](#), [17305](#), [17309](#), [17311](#),
[17316](#), [17377](#), [17390](#), [17393](#), [17395](#)
- `\s__skip_stop`
. [21180](#), [21241](#), [21243](#), [40031](#)
- `\s__sort_mark` [436](#), [439–441](#),
[3126](#), [3322](#), [3326](#), [3332](#), [3336](#), [3342](#),
[3345](#), [3410](#), [3411](#), [3413](#), [3450](#), [3452](#),
[3455](#), [3459](#), [3462](#), [3465](#), [3467](#), [3470](#)
- `\s__sort_stop` [438](#), [440](#), [441](#), [3126](#),
[3398](#), [3407](#), [3411](#), [3413](#), [3450](#), [3451](#),
[3452](#), [3457](#), [3459](#), [3463](#), [3465](#), [3473](#)
- `\s__str` [760](#),
[768](#), [786](#), [789](#), [14175](#), [14324](#), [14328](#),
[14512](#), [14559](#), [14627](#), [14630](#), [15074](#),
[15086](#), [15091](#), [15101](#), [15106](#), [15111](#),
[15114](#), [15129](#), [15142](#), [15145](#), [15280](#),
[15281](#), [15298](#), [15304](#), [15320](#), [15326](#),

- 15327, 15432, 15447, 15456, 15457
- `\s__str_mark` 736,
740, 743, 749, 13405, 13605, 13640,
13649, 13732, 13749, 13997, 13999
- `\s__str_stop` 743, 747, 785,
789, 13405, 13407, 13408, 13512,
13605, 13640, 13649, 13732, 13741,
13747, 13749, 13755, 13772, 13791,
13853, 13910, 13922, 13960, 13976,
13983, 13991, 13993, 13997, 13999,
14324, 14330, 14372, 14377, 14387,
14582, 14585, 14604, 14610, 14989,
14991, 14999, 15087, 15123, 15237,
15239, 15243, 15255, 15395, 15397,
15401, 15413, 15422, 15429, 15450
- `\s__text_recursion_stop` .. 31395,
31398, 31725, 31739, 31748, 31790,
31799, 31941, 31949, 32028, 32036
- `\s__text_recursion_tail`
..... 31395, 31402, 31403, 31725
- `\s__text_stop`
.. 31386, 31480, 31482, 31962, 31963
- `\s__tl` 445-448,
456, 457, 3535, 3536, 3794, 3830,
3836, 3861, 3879, 3884, 3901, 3905,
3937, 3940, 4077, 4081, 4122, 4126
- `\s__tl_act_stop` 722, 13018,
13024, 13025, 13028, 13031, 13035,
13044, 13047, 13050, 13053, 13056,
13058, 13060, 13064, 13067, 13073
- `\s__tl_mark` 12803,
12804, 12807, 12810, 12811, 13295
- `\s__tl_nil` 716, 12838,
12842, 12861, 12864, 12867, 13295
- `\s__tl_stop`
702, 712, 715, 12438, 12440, 12665,
12671, 12689, 12690, 12699, 12703,
12705, 12707, 12709, 12718, 12719,
12729, 12730, 12739, 12744, 12746,
12748, 12805, 12807, 12812, 12814,
12844, 12867, 12884, 12899, 12913,
12939, 12964, 13259, 13269, 13295
- `\s__token_mark`
..... 908, 19284, 19688, 19689, 19698
- `\s__token_stop` .. 902, 904, 19284,
19434, 19437, 19467, 19502, 19610,
19614, 19620, 19643, 19690, 19698
- `\scantextokens` 913
- `\scantokens` 522
- `\scriptbaselineshiftfactor` 1181
- `\scriptfont` 390
- `\scriptscriptbaselineshiftfactor` . 1183
- `\scriptscriptfont` 391
- `\scriptscriptstyle` 392
- `\scriptsize` 34305
- `\scriptspace` 393
- `\scriptstyle` 394
- `\scrollmode` 395
- `\scshape` 34294
- `sec` 276
- `secd` 277
- `\selectfont` 34266
- seq commands:
- `\c_empty_seq` 163, 819, 16674, 16678,
16682, 16685, 16959, 17029, 17037
- `\seq_clear:N`
..... 151, 163, 6030, 6909, 7542,
9157, 9657, 9720, 11562, 11655,
16681, 16681, 16683, 16688, 16835
- `\seq_clear_new:N`
..... 151, 16687, 16687, 16689
- `\seq_concat:NNN` 153,
163, 11568, 16788, 16788, 16792, 39423
- `\seq_const_from_clist:Nn`
..... 152, 16727, 16727, 16732
- `\seq_count:N` .. 154, 160, 162, 256,
6908, 11676, 16890, 16972, 17156,
17170, 17339, 17339, 17362, 17367
- `\seq_elt:w` 818
- `\seq_elt_end:` 818
- `\seq_gclear:N`
..... 151, 433, 3263, 3272, 16681,
16684, 16686, 16691, 16984, 16992
- `\seq_gclear_new:N`
..... 151, 16687, 16690, 16692
- `\seq_gconcat:NNN`
153, 11581, 16788, 16790, 16793, 39424
- `\seq_get:NN` 161, 6369, 6374, 17406,
17406, 17407, 17412, 17413, 37347
- `\seq_get:NNTF` 161, 17412
- `\seq_get_left:NN` 153,
17045, 17045, 17055, 17115, 17116,
17119, 17406, 17407, 17412, 17413
- `\seq_get_left:NNTF` 155, 17115
- `\seq_get_right:NN` 154, 17070,
17070, 17087, 17117, 17118, 17121
- `\seq_get_right:NNTF` 155, 17115
- `\seq_gpop:NN`
..... 161, 11470, 17406, 17410,
17411, 17416, 17417, 30553, 37340
- `\seq_gpop:NNTF`
162, 10242, 10493, 17412, 30523, 30535
- `\seq_gpop_left:NN`
.. 154, 17056, 17058, 17069, 17126,
17137, 17410, 17411, 17416, 17417
- `\seq_gpop_left:NNTF` 155, 17123
- `\seq_gpop_right:NN`
154, 17088, 17090, 17114, 17132, 17141

`\seq_gpop_right:NNTF` [156](#), [17123](#)
`\seq_gpush:Nn` [31](#), [162](#), [10289](#),
[10533](#), [11455](#), [17400](#), [17403](#), [17404](#),
[17405](#), [30527](#), [30537](#), [30546](#), [37276](#)
`\seq_gput_left:Nn`
[153](#), [16798](#), [16806](#), [16817](#), [16818](#), [17403](#)
`\seq_gput_right:Nn`
. [153](#), [3267](#), [10939](#), [10946](#), [11444](#),
[16819](#), [16821](#), [16825](#), [16826](#), [16987](#)
`\seq_gremove_all:Nn`
. [156](#), [16845](#), [16847](#), [16871](#), [16872](#)
`\seq_gremove_duplicates:N`
. [156](#), [16829](#), [16831](#), [16844](#)
`\seq_greverse:N`
. [157](#), [16939](#), [16941](#), [16956](#)
`\seq_gset_eq:NN` [151](#), [3245](#),
[16685](#), [16693](#), [16697](#), [16698](#), [16699](#),
[16700](#), [16832](#), [16969](#), [39406](#), [39560](#)
`\seq_gset_filter:NNn` [153](#), [16778](#), [16780](#)
`\seq_gset_from_clist:NN`
. [152](#), [16701](#), [16711](#), [16724](#), [16725](#)
`\seq_gset_from_clist:Nn`
. [152](#), [16701](#), [16716](#), [16726](#)
`\seq_gset_item:Nnn`
[156](#), [16874](#), [16876](#), [16879](#), [16882](#), [16885](#)
`\seq_gset_item:NnnTF` [156](#), [16874](#)
`\seq_gset_map:NNn` [159](#), [17329](#), [17331](#)
`\seq_gset_map_e:NNn`
. [160](#), [17319](#), [17321](#), [38894](#), [38895](#)
`\seq_gset_map_x:NNn` [38892](#), [38895](#)
`\seq_gset_split:Nnn`
. [152](#), [16733](#), [16735](#), [16774](#), [16775](#)
`\seq_gset_split_keep_spaces:Nnn`
. [152](#), [16733](#), [16739](#), [16777](#)
`\seq_gshuffle:N`
. [157](#), [16967](#), [16969](#), [17005](#)
`\seq_gsort:Nn`
. [157](#), [3241](#), [3244](#), [3246](#), [16957](#)
`\seq_if_empty:N` [16957](#), [16965](#)
`\seq_if_empty:NTF`
. [157](#), [6905](#), [16957](#), [17169](#), [18456](#), [30587](#)
`\seq_if_empty_p:N` [157](#), [16957](#)
`\seq_if_exist:N` [16794](#), [16796](#)
`\seq_if_exist:NTF`
. [153](#), [16688](#), [16691](#), [16794](#), [17365](#)
`\seq_if_exist_p:N` [153](#), [16794](#)
`\seq_if_in:Nn` [879](#), [17006](#), [17025](#)
`\seq_if_in:NnTF` [157](#),
[162](#), [163](#), [10288](#), [10532](#), [16838](#), [17006](#)
`\seq_indexed_map_function:NN`
. [38886](#), [38889](#)
`\seq_indexed_map_inline:Nn`
. [38886](#), [38887](#)
`\seq_item:Nn` [57](#), [154](#), [833](#), [9738](#),
[9739](#), [9744](#), [17143](#), [17143](#), [17166](#), [17170](#)
`\seq_log:N` [164](#), [17418](#), [17420](#), [17421](#)
`\seq_map_break:` [153](#), [159](#), [160](#),
[17173](#), [17173](#), [17174](#), [17176](#), [17186](#),
[17227](#), [17237](#), [17260](#), [17267](#), [17276](#)
`\seq_map_break:n` [159](#), [833](#), [3242](#),
[3245](#), [9677](#), [9691](#), [11130](#), [17173](#), [17175](#)
`\seq_map_function:NN` [6](#), [87](#), [157](#),
[158](#), [835](#), [6050](#), [6118](#), [9742](#), [11571](#),
[17177](#), [17177](#), [17200](#), [17433](#), [18462](#)
`\seq_map_indexed_function:NN`
. [158](#), [17264](#), [17264](#), [38888](#), [38889](#)
`\seq_map_indexed_inline:Nn`
. [158](#), [17264](#), [17269](#), [38886](#), [38887](#)
`\seq_map_inline:Nn`
. [157](#), [158](#), [163](#), [822](#), [3242](#),
[3245](#), [9672](#), [16836](#), [17223](#), [17223](#), [17229](#)
`\seq_map_pairwise_function:NNN`
. [158](#), [17297](#), [17297](#), [17318](#), [38890](#), [38891](#)
`\seq_map_tokens:Nn` [157](#),
[158](#), [11129](#), [11680](#), [17230](#), [17230](#), [17239](#)
`\seq_map_variable:NNn`
. [158](#), [17252](#), [17252](#), [17262](#), [17263](#)
`\seq_mapthread_function:NNN`
. [38890](#), [38891](#)
`\seq_new:N` [6](#), [151](#), [3113](#), [4342](#), [4797](#),
[6136](#), [6137](#), [6821](#), [9627](#), [9628](#), [10194](#),
[10447](#), [10931](#), [10956](#), [10962](#), [10963](#),
[16675](#), [16675](#), [16680](#), [16688](#), [16691](#),
[16828](#), [16967](#), [17443](#), [17444](#), [17445](#),
[17446](#), [18601](#), [19156](#), [19159](#), [21585](#),
[21586](#), [30367](#), [30368](#), [30369](#), [37260](#)
`\seq_pop:NN`
. [161](#), [6198](#), [6236](#), [6238](#), [6978](#),
[17406](#), [17408](#), [17409](#), [17414](#), [17415](#)
`\seq_pop:NNTF` [162](#), [17412](#)
`\seq_pop_left:NN`
. [154](#), [17056](#), [17056](#), [17068](#), [17123](#),
[17135](#), [17408](#), [17409](#), [17414](#), [17415](#)
`\seq_pop_left:NNTF` [155](#), [17123](#)
`\seq_pop_right:NN` [154](#), [5979](#), [6057](#),
[17088](#), [17088](#), [17113](#), [17129](#), [17139](#)
`\seq_pop_right:NNTF` [156](#), [17123](#)
`\seq_push:Nn` [162](#), [6208](#), [6229](#),
[6231](#), [7138](#), [17400](#), [17400](#), [17401](#), [17402](#)
`\seq_put_left:Nn` [153](#), [9667](#),
[16798](#), [16798](#), [16815](#), [16816](#), [17400](#)
`\seq_put_right:Nn` [153](#), [162](#),
[163](#), [5982](#), [6055](#), [7552](#), [9728](#), [11657](#),
[16819](#), [16819](#), [16823](#), [16824](#), [16839](#)
`\seq_rand_item:N`
. [155](#), [17167](#), [17167](#), [17172](#)

- \seq_remove_all:Nn . [152](#), [156](#), [162](#),
[163](#), [16845](#), [16845](#), [16869](#), [16870](#), [18633](#)
- \seq_remove_duplicates:N [156](#),
[162](#), [163](#), [11569](#), [16829](#), [16829](#), [16843](#)
- \seq_reverse:N
. [157](#), [826](#), [16939](#), [16939](#), [16955](#)
- \seq_set_eq:NN [151](#), [163](#), [3242](#),
[16682](#), [16693](#), [16693](#), [16694](#), [16695](#),
[16696](#), [16830](#), [16968](#), [39405](#), [39479](#)
- \seq_set_filter:NNn
. [153](#), [836](#), [6113](#), [16778](#), [16778](#)
- \seq_set_from_clist:NN
[152](#), [16701](#), [16701](#), [16721](#), [16722](#), [18632](#)
- \seq_set_from_clist:Nn [152](#), [185](#),
[820](#), [11565](#), [11579](#), [16701](#), [16706](#), [16723](#)
- \seq_set_item:Nnn
[156](#), [16874](#), [16874](#), [16878](#), [16880](#), [16884](#)
- \seq_set_item:NnnTF [156](#), [16874](#)
- \seq_set_map:NNn [159](#), [17329](#), [17329](#)
- \seq_set_map_e:NNn [160](#),
[837](#), [6126](#), [17319](#), [17319](#), [38892](#), [38893](#)
- \seq_set_map_x:NNn [38892](#), [38893](#)
- \seq_set_split:Nnn
. [152](#), [6112](#), [6125](#), [9160](#), [16733](#),
[16733](#), [16772](#), [16773](#), [19157](#), [19160](#)
- \seq_set_split_keep_spaces:Nnn
. [152](#), [16733](#), [16737](#), [16776](#)
- \seq_show:N
. [164](#), [623](#), [728](#), [17418](#), [17418](#), [17419](#)
- \seq_shuffle:N [157](#), [16967](#), [16968](#), [17004](#)
- \seq_sort:Nn
. [45](#), [157](#), [3241](#), [3241](#), [3243](#), [16957](#)
- \seq_use:Nn
. [161](#), [6129](#), [17363](#), [17397](#), [17399](#)
- \seq_use:Nnnn
. [160](#), [17363](#), [17363](#), [17385](#), [17398](#)
- \g_tmpa_seq [164](#), [17443](#)
- \l_tmpa_seq [164](#), [17443](#)
- \g_tmpb_seq [164](#), [17443](#)
- \l_tmpb_seq [164](#), [17443](#)
- seq internal commands:
- __seq_count:w
[837](#), [17339](#), [17344](#), [17357](#), [17360](#), [17361](#)
- __seq_count_end:w [837](#),
[17339](#), [17346](#), [17347](#), [17348](#), [17349](#),
[17350](#), [17351](#), [17352](#), [17353](#), [17361](#)
- __seq_get_left:wnw
. [17045](#), [17049](#), [17053](#)
- __seq_get_right_end:NnN
. [17070](#), [17078](#), [17086](#)
- __seq_get_right_loop:nw
. [830](#), [17070](#), [17075](#), [17081](#), [17084](#)
- __seq_if_in: [17006](#), [17015](#), [17023](#)
- __seq_int_eval:w
. [16873](#), [16873](#), [16920](#), [16930](#)
- \l_seq_internal_a_int
. [16979](#), [16985](#), [16997](#), [16999](#), [17000](#)
- \l_seq_internal_a_tl
. [821](#), [826](#), [16671](#), [16745](#),
[16749](#), [16755](#), [16760](#), [16762](#), [16860](#),
[16865](#), [16888](#), [16935](#), [17010](#), [17014](#)
- \l_seq_internal_b_int
. [16998](#), [17001](#), [17002](#)
- \l_seq_internal_b_tl
. [16671](#), [16856](#), [16860](#), [17013](#), [17014](#)
- \g_seq_internal_seq [16967](#)
- __seq_item:n
. [818](#), [822](#), [823](#), [828](#), [829](#), [831](#)–
[834](#), [836](#)–[838](#), [16666](#), [16666](#), [16802](#),
[16810](#), [16820](#), [16822](#), [16827](#), [16888](#),
[16925](#), [16928](#), [16945](#), [16946](#), [16948](#),
[16953](#), [16981](#), [17011](#), [17050](#), [17053](#),
[17063](#), [17078](#), [17081](#), [17094](#), [17095](#),
[17106](#), [17150](#), [17159](#), [17184](#), [17189](#),
[17190](#), [17191](#), [17192](#), [17204](#), [17209](#),
[17215](#), [17219](#), [17235](#), [17241](#), [17242](#),
[17243](#), [17244](#), [17287](#), [17289](#), [17325](#),
[17335](#), [17346](#), [17347](#), [17348](#), [17349](#),
[17350](#), [17351](#), [17352](#), [17353](#), [17358](#),
[17359](#), [17374](#), [17389](#), [17392](#), [17395](#)
- __seq_item:nN [17143](#), [17148](#), [17153](#)
- __seq_item:nwn
. [17143](#), [17147](#), [17159](#), [17164](#)
- __seq_item:wNn [17143](#), [17144](#), [17145](#)
- __seq_map_function:Nw
. [833](#), [17177](#), [17180](#), [17188](#), [17198](#)
- __seq_map_indexed:NN
. [17266](#), [17274](#), [17279](#)
- __seq_map_indexed:nNN [17264](#)
- __seq_map_indexed:Nw
. [835](#), [17264](#), [17281](#), [17289](#), [17293](#)
- __seq_map_pairwise_function:Nnnwnn
. [17297](#), [17307](#), [17311](#), [17316](#)
- __seq_map_pairwise_function:wNN
. [17297](#), [17298](#), [17299](#)
- __seq_map_pairwise_function:wNw
. [17297](#), [17301](#), [17305](#)
- __seq_map_tokens:nw
. [17230](#), [17233](#), [17240](#), [17250](#)
- __seq_pop:NNNN [17027](#),
[17027](#), [17057](#), [17059](#), [17089](#), [17091](#)
- __seq_pop_item_def:
. [818](#), [16786](#), [16867](#), [16983](#), [17201](#),
[17217](#), [17227](#), [17260](#), [17327](#), [17337](#)
- __seq_pop_left:NNN [17056](#),
[17057](#), [17059](#), [17060](#), [17125](#), [17128](#)

- __seq_pop_left:wnwNNN
..... 17056, 17061, 17062
- __seq_pop_right:NNN . 824, 17088,
17089, 17091, 17092, 17131, 17134
- __seq_pop_right_loop:nn
..... 17088, 17099, 17108, 17111
- __seq_pop_TF:NNNN
..... 831, 17027, 17035, 17116,
17118, 17125, 17128, 17131, 17134
- __seq_push_item_def:
.. 16980, 17201, 17203, 17208, 17211
- __seq_push_item_def:n
.. 818, 16784, 16851, 17201, 17201,
17206, 17225, 17254, 17325, 17335
- __seq_put_left_aux:w
.... 822, 16798, 16803, 16811, 16814
- __seq_remove_all_aux:NNn
..... 16845, 16846, 16848, 16849
- __seq_remove_duplicates:NN
..... 16829, 16830, 16832, 16833
- \l__seq_remove_seq
.. 16828, 16835, 16838, 16839, 16841
- __seq_reverse:NN
..... 16939, 16940, 16942, 16943
- __seq_reverse_item:nw 827
- __seq_reverse_item:nwn
..... 16939, 16946, 16950
- __seq_set_filter:NNNn
..... 16778, 16779, 16781, 16782
- __seq_set_item:NnnNN ... 16874,
16875, 16877, 16881, 16883, 16886
- __seq_set_item:nnNNNN
..... 16874, 16889, 16892
- __seq_set_item:nNnnNNNN
.... 826, 16874, 16895, 16900, 16914
- __seq_set_item:wn
..... 16874, 16919, 16925, 16929
- __seq_set_item_end:w
..... 826, 16874, 16927, 16932
- __seq_set_item_false:nnNNNN ...
..... 825, 16874, 16903, 16905
- __seq_set_map:NNNn
..... 17329, 17330, 17332, 17333
- __seq_set_map_e:NNNn
..... 17319, 17320, 17322, 17323
- __seq_set_split:NNnn 16733
- __seq_set_split:NNNnn
.. 16734, 16736, 16738, 16740, 16741
- __seq_set_split:Nw
.... 821, 16733, 16751, 16758, 16764
- __seq_set_split:w
..... 821, 16733, 16766, 16770
- __seq_set_split_end: 821, 16733,
16753, 16757, 16764, 16768, 16770
- __seq_show:NN
..... 17418, 17418, 17420, 17422
- __seq_show_validate:nn
..... 17418, 17427, 17437, 17441
- __seq_shuffle:NN
..... 16967, 16968, 16969, 16970
- __seq_shuffle_item:n
..... 16967, 16981, 16995
- __seq_tmp:w 16673,
16673, 16945, 16948, 17094, 17106
- __seq_use:NNnNnn
..... 17363, 17370, 17371, 17386
- __seq_use:nwnn . 17363, 17376, 17395
- __seq_use:nwwwnwn
..... 17363, 17375, 17387, 17388
- __seq_use_setup:w 17363, 17374, 17387
- __seq_wrap_item:n
..... 821, 822, 16704, 16709,
16714, 16719, 16730, 16746, 16771,
16784, 16827, 16827, 16863, 17440
- \setbox 396
- \setfontid 914
- \setlanguage 397
- \setrandomseed 960
- \sfcode 398
- \sffamily 34289, 36468
- \shapemode 915
- \shbscode 682
- \shellescape 778
- \Shipout 1252
- \shipout 399, 1239, 1240
- \ShortText 37, 75
- \show 400
- \showbox 401
- \showboxbreadth 402
- \showboxdepth 403
- \showgroups 523
- \showifs 524
- \showlists 404
- \showmode 1185
- \showstream 1218
- \showthe 405
- \showtokens 525
- sign 276
- sin 276
- sind 277
- \sjis 1186
- \skewchar 406
- \skip 407, 19526
- skip commands:
- \c_max_skip 235, 21266
- \skip_add:Nn
..... 233, 21217, 21217, 21221, 39483, 39856

- \skip_const:Nn
..... [232](#), [962](#), [21187](#), [21187](#),
[21192](#), [21266](#), [21267](#), [39605](#), [39860](#)
- \skip_eval:n ... [234](#), [21190](#), [21231](#),
[21246](#), [21246](#), [21261](#), [21265](#), [39901](#)
- \skip_gadd:Nn
[233](#), [21217](#), [21219](#), [21222](#), [39564](#), [39857](#)
- .skip_gset:N [244](#), [22157](#)
- \skip_gset:Nn [233](#),
[958](#), [21207](#), [21209](#), [21212](#), [39562](#), [39855](#)
- \skip_gset_eq:NN
..... [233](#), [21213](#), [21215](#), [21216](#), [39563](#)
- \skip_gsub:Nn
[233](#), [21217](#), [21225](#), [21228](#), [39565](#), [39859](#)
- \skip_gzero:N
[232](#), [21193](#), [21194](#), [21196](#), [21200](#), [39561](#)
- \skip_gzero_new:N
..... [233](#), [21197](#), [21199](#), [21202](#)
- \skip_horizontal:N
..... [235](#), [21250](#), [21250](#), [21252](#), [21256](#)
- \skip_horizontal:n
..... [235](#), [21250](#), [21251](#), [39902](#)
- \skip_if_eq:nn [21229](#)
- \skip_if_eq:nnTF [234](#), [21229](#)
- \skip_if_eq_p:nn [234](#), [21229](#)
- \skip_if_exist:N [21203](#), [21205](#)
- \skip_if_exist:NTF
..... [233](#), [21198](#), [21200](#), [21203](#)
- \skip_if_exist_p:N [233](#), [21203](#)
- \skip_if_finite:n [21237](#), [40024](#), [40029](#)
- \skip_if_finite:nTF [234](#), [21235](#)
- \skip_if_finite_p:n [234](#), [21235](#)
- \skip_log:N .. [235](#), [21262](#), [21262](#), [21263](#)
- \skip_log:n [235](#), [21262](#), [21264](#)
- \skip_new:N [232](#), [233](#),
[21181](#), [21181](#), [21186](#), [21189](#), [21198](#),
[21200](#), [21268](#), [21269](#), [21270](#), [21271](#)
- .skip_set:N [244](#), [22157](#)
- \skip_set:Nn
[233](#), [21207](#), [21207](#), [21211](#), [39481](#), [39854](#)
- \skip_set_eq:NN
..... [233](#), [21213](#), [21213](#), [21214](#), [39482](#)
- \skip_show:N . [234](#), [21258](#), [21258](#), [21259](#)
- \skip_show:n .. [234](#), [961](#), [21260](#), [21260](#)
- \skip_sub:Nn
[233](#), [21217](#), [21223](#), [21227](#), [39484](#), [39858](#)
- \skip_use:N [234](#), [21240](#),
[21247](#), [21248](#), [21248](#), [21249](#), [40027](#)
- \skip_vertical:N
..... [236](#), [21250](#), [21253](#), [21255](#), [21257](#)
- \skip_vertical:n
..... [236](#), [21250](#), [21254](#), [39903](#)
- \skip_zero:N [232](#), [233](#), [236](#),
[944](#), [21193](#), [21193](#), [21195](#), [21198](#), [39480](#)
- \skip_zero_new:N
..... [233](#), [21197](#), [21197](#), [21201](#)
- \g_tmpa_skip [235](#), [21268](#)
- \l_tmpa_skip [235](#), [21268](#)
- \g_tmpb_skip [235](#), [21268](#)
- \l_tmpb_skip [235](#), [21268](#)
- \c_zero_skip
..... [235](#), [944](#), [20764](#), [20766](#), [21266](#)
- skip internal commands:
 _skip_if_finite:wwNw
 [21235](#), [21239](#), [21243](#), [40026](#)
 _skip_tmp:w
 [21235](#), [21245](#), [40022](#), [40034](#)
- \skipdef [408](#)
- \slshape [34295](#)
- \small [34306](#)
- sort commands:
 \sort_return_same:
 [44](#), [45](#), [436](#), [3325](#), [3325](#)
 \sort_return_swapped:
 [44](#), [45](#), [436](#), [3325](#), [3335](#)
- sort internal commands:
 _sort:nnNnn [437](#), [438](#)
 \l__sort_A_int .. [435](#), [3123](#), [3130](#),
 [3137](#), [3140](#), [3149](#), [3289](#), [3294](#), [3297](#),
 [3317](#), [3349](#), [3356](#), [3371](#), [3373](#), [3374](#)
 \l__sort_B_int [435](#), [3123](#),
 [3294](#), [3298](#), [3306](#), [3308](#), [3309](#), [3361](#),
 [3362](#), [3371](#), [3372](#), [3381](#), [3382](#), [3384](#)
 \l__sort_begin_int
 [430](#), [435](#), [3121](#), [3286](#), [3374](#), [3384](#)
 \l__sort_block_int . [429](#), [430](#), [434](#),
 [3120](#), [3132](#), [3137](#), [3141](#), [3144](#), [3149](#),
 [3150](#), [3215](#), [3277](#), [3280](#), [3287](#), [3290](#)
 \l__sort_C_int [435](#), [3123](#),
 [3295](#), [3299](#), [3306](#), [3307](#), [3318](#), [3350](#),
 [3357](#), [3361](#), [3363](#), [3364](#), [3381](#), [3383](#)
 _sort_compare:nn [432](#), [436](#), [3214](#), [3316](#)
 _sort_compute_range:
 [429-431](#), [3154](#),
 [3154](#), [3162](#), [3170](#), [3178](#), [3191](#), [3202](#)
 _sort_copy_block:
 [434](#), [3296](#), [3304](#), [3304](#), [3312](#)
 _sort_disable_toksdef:
 [3201](#), [3481](#), [3481](#)
 _sort_disabled_toksdef:n
 [3481](#), [3482](#), [3483](#)
 \l__sort_end_int [430](#),
 [434](#), [435](#), [3121](#), [3278](#), [3286](#), [3287](#),
 [3288](#), [3289](#), [3290](#), [3291](#), [3292](#), [3309](#)
 _sort_error: [3475](#), [3475](#), [3487](#), [3505](#)
 _sort_i:nnnnNn [439](#)
 \g_sort_internal_seq
 [432](#), [433](#), [3113](#), [3263](#), [3267](#), [3271](#), [3272](#)

- \g__sort_internal_tl 3113, 3226, 3229, 3230
- \l__sort_length_int 429, 430, 3115, 3212, 3277
- __sort_level: 432, 442, 3216, 3275, 3275, 3281, 3479
- __sort_loop:wNn 438, 439
- __sort_main:NNn 433, 3199, 3199, 3225, 3262
- \l__sort_max_int 429, 430, 3115, 3134, 3206
- \c__sort_max_length_int 3154
- __sort_merge_blocks: 3279, 3284, 3284, 3301, 3478
- __sort_merge_blocks_aux: 434, 3300, 3314, 3314, 3367, 3377, 3477
- __sort_merge_blocks_end: 437, 3375, 3379, 3379, 3387
- \l__sort_min_int 429, 430, 432, 3115, 3131, 3139, 3156, 3172, 3180, 3193, 3203, 3213, 3227, 3265, 3278, 3503, 3504
- __sort_quick_cleanup:w 3389, 3410, 3413
- __sort_quick_end:nnTFNn 440, 441, 3409, 3449, 3449, 3455, 3462, 3467, 3470
- __sort_quick_only_i:NnnnnNn ... 3414, 3417, 3421, 3424
- __sort_quick_only_i_end:nnwnw . 3425, 3449, 3452
- __sort_quick_only_ii:NnnnnNn ... 3414, 3416, 3428, 3430
- __sort_quick_only_ii_end:nnwnw . 3432, 3449, 3459
- __sort_quick_prepare:Nnnn 3389, 3395, 3402, 3405
- __sort_quick_prepare_end:NNnw . 3389, 3397, 3407
- __sort_quick_single_end:nnwnw . 3418, 3449, 3450
- __sort_quick_split:NnNn 439, 440, 3409, 3414, 3414, 3454, 3461, 3467, 3469
- __sort_quick_split_end:nnwnw . 3439, 3446, 3449, 3465
- __sort_quick_split_i:NnnnnNn ... 438, 3414, 3431, 3435, 3438, 3445
- __sort_quick_split_ii:NnnnnNn . 3414, 3423, 3437, 3442, 3444
- __sort_redefine_compute_range: . 3154, 3161, 3166, 3186
- __sort_return_mark:w 436, 3320, 3321, 3325, 3326, 3331, 3336, 3341, 3345
- __sort_return_none_error: 436, 3323, 3325, 3346, 3351, 3359, 3369
- __sort_return_same:w 436, 3333, 3351, 3359, 3359
- __sort_return_swapped:w 3343, 3369, 3369
- __sort_return_two_error: 436, 3325, 3330, 3340, 3353
- __sort_seq:NNNn 432, 3241, 3242, 3245, 3249, 3255, 3259
- __sort_shrink_range: 430, 431, 3128, 3128, 3158, 3174, 3182, 3195
- __sort_shrink_range_loop: 3128, 3133, 3147, 3151
- __sort_tl:NNn 432, 3218, 3218, 3220, 3222
- __sort_tl_toks:w 432, 3218, 3227, 3233, 3237
- __sort_too_long_error:NNw 3207, 3498, 3498
- \l__sort_top_int 429, 432, 435, 3115, 3203, 3206, 3209, 3210, 3213, 3235, 3265, 3288, 3291, 3292, 3295, 3364, 3504
- \l__sort_true_max_int 429, 430, 3115, 3131, 3144, 3157, 3173, 3181, 3194, 3503
- sp 280
- spac commands:
 - \spac_directions_normal_body_dir 1332
 - \spac_directions_normal_page_dir 1333
 - \spacefactor 409
 - \spaceskip 410
 - \span 411
 - \special 412
 - \splitbotmark 413
 - \splitbotmarks 526
 - \splitdiscards 527
 - \splitfirstmark 414
 - \splitfirstmarks 528
 - \splitmaxdepth 415
 - \splittopskip 416
 - sqrt 278
 - \SS 32070, 33654, 34366
 - \ss 32070, 33654, 34362
 - \stbscode 683
 - \stockheight .. 38709, 38717, 38721, 38725
 - \stockwidth ... 38710, 38718, 38721, 38726

- str commands:
- `\c_ampersand_str` [141](#), [14135](#)
 - `\c_atsign_str` [141](#), [14135](#)
 - `\c_backslash_str`
..... [141](#), [4587](#), [5190](#), [14135](#),
[14840](#), [14842](#), [14865](#), [14894](#), [14896](#),
[14928](#), [14937](#), [14941](#), [39342](#), [39352](#)
 - `\c_circumflex_str` [141](#), [14135](#)
 - `\c_colon_str` ... [141](#), [14135](#), [19437](#),
[19614](#), [19620](#), [21705](#), [37445](#), [37450](#)
 - `\c_dollar_str` [141](#), [14135](#)
 - `\c_empty_str` [141](#), [14148](#)
 - `\c_hash_str` [141](#), [14135](#),
[14808](#), [14911](#), [15486](#), [15487](#), [15490](#),
[15493](#), [31272](#), [31301](#), [31305](#), [31374](#),
[39240](#), [39808](#), [39810](#), [39870](#), [39918](#),
[39923](#), [39953](#), [39957](#), [39959](#), [39977](#)
 - `\c_left_brace_str` . [141](#), [488](#), [4650](#),
[5065](#), [5069](#), [5089](#), [5102](#), [5126](#), [5598](#),
[5609](#), [5613](#), [5692](#), [5716](#), [6887](#), [14135](#)
 - `\c_percent_str` [141](#), [14135](#), [14810](#), [14964](#)
 - `\c_right_brace_str`
.... [141](#), [4686](#), [5075](#), [5095](#), [5108](#),
[5616](#), [5620](#), [5713](#), [6884](#), [14135](#), [22709](#)
 - `\str_case:Nn` [132](#), [13584](#), [13609](#)
 - `\str_case:nn` [132](#),
[5330](#), [8719](#), [10111](#), [13584](#), [13584](#),
[13606](#), [13607](#), [13609](#), [37963](#), [38001](#)
 - `\str_case:NnTF`
.... [132](#), [13584](#), [13610](#), [13611](#), [13612](#)
 - `\str_case:nnTF`
.. [132](#), [593](#), [850](#), [947](#), [5940](#), [8795](#),
[8830](#), [9214](#), [9927](#), [13584](#), [13589](#),
[13594](#), [13599](#), [13610](#), [13611](#), [13612](#),
[21909](#), [21954](#), [25480](#), [29606](#), [29620](#)
 - `\str_case:e:nn`
.... [133](#), [13584](#), [13619](#), [13641](#), [13642](#)
 - `\str_case:e:nnTF` [133](#), [5073](#),
[13584](#), [13624](#), [13629](#), [13634](#), [14863](#)
 - `\str_casefold:n`
..... [139](#), [140](#), [297](#), [14000](#),
[14000](#), [14003](#), [24291](#), [37632](#), [38874](#),
[38875](#), [38876](#), [38877](#), [38878](#), [38879](#),
[38880](#), [38881](#), [38955](#), [38956](#), [38963](#),
[38964](#), [38971](#), [38972](#), [38981](#), [38982](#)
 - `\c_str_cctab` [291](#), [1261](#), [30663](#)
 - `\str_clear:N` [130](#), [13413](#),
[21687](#), [21836](#), [22341](#), [22342](#), [39485](#)
 - `\str_clear_new:N` [130](#), [13413](#)
 - `\str_compare:nNn` [13539](#), [13546](#)
 - `\str_compare:nNnTF` [133](#), [13539](#)
 - `\str_compare_p:nNn` [133](#), [13539](#)
 - `\str_concat:NNN`
.... [130](#), [13413](#), [13436](#), [13438](#), [39425](#)
 - `\str_const:Nn`
.... [130](#), [8648](#), [8667](#), [8685](#), [8717](#),
[8786](#), [9017](#), [9176](#), [11735](#), [11742](#),
[11746](#), [11750](#), [13440](#), [13444](#), [13471](#),
[14135](#), [14136](#), [14137](#), [14138](#), [14139](#),
[14140](#), [14141](#), [14142](#), [14143](#), [14144](#),
[14145](#), [14146](#), [14147](#), [14904](#), [14905](#),
[14927](#), [21561](#), [21562](#), [21563](#), [21564](#),
[21565](#), [21566](#), [21567](#), [30999](#), [39606](#)
 - `\str_convert_pdfname:n`
..... [144](#), [15462](#), [15462](#), [37729](#)
 - `\str_count:N` [135](#), [3998](#), [9356](#), [9357](#),
[9530](#), [9531](#), [10613](#), [10691](#), [13932](#),
[13932](#), [13933](#), [30201](#), [30206](#), [30282](#)
 - `\str_count:n`
..... [135](#), [3992](#), [13932](#), [13932](#), [13934](#)
 - `\str_count_ignore_spaces:n`
..... [135](#), [748](#), [3580](#), [13932](#), [13947](#)
 - `\str_count_spaces:N`
..... [135](#), [13912](#), [13912](#), [13914](#)
 - `\str_count_spaces:n`
[135](#), [748](#), [13912](#), [13913](#), [13915](#), [13938](#)
 - `\str_declare_eight_bit_encoding:nmm`
..... [38882](#), [38883](#)
 - `\str_fold_case:n` . [38866](#), [38875](#), [38877](#)
 - `\str_foldcase:n` . [38878](#), [38879](#), [38881](#)
 - `\str_gclear:N` [130](#), [13413](#), [39566](#)
 - `\str_gclear_new:N` [130](#), [13413](#)
 - `\str_gconcat:NNN`
.... [130](#), [13413](#), [13437](#), [13439](#), [39426](#)
 - `\str_gput_left:Nn`
.... [131](#), [13440](#), [13454](#), [13473](#), [39568](#)
 - `\str_gput_right:Nn`
.... [131](#), [13440](#), [13464](#), [13475](#), [39569](#)
 - `\str_gremove_all:Nn`
..... [138](#), [13522](#), [13524](#), [13527](#)
 - `\str_gremove_once:Nn`
..... [138](#), [13516](#), [13518](#), [13521](#)
 - `\str_greplace_all:Nnn`
.... [138](#), [13476](#), [13482](#), [13487](#), [13525](#)
 - `\str_greplace_once:Nnn`
.... [138](#), [13476](#), [13478](#), [13485](#), [13519](#)
 - `.str_gset:N` [244](#), [22165](#)
 - `\str_gset:Nn`
..... [131](#), [11476](#), [11477](#), [11478](#),
[13440](#), [13442](#), [13470](#), [30192](#), [30196](#)
 - `\str_gset_convert:Nnnn`
..... [144](#), [14344](#), [14346](#), [14358](#)
 - `\str_gset_convert:NnnnTF` . [144](#), [14344](#)
 - `.str_gset_e:N` [244](#), [22165](#)
 - `\str_gset_eq:NN`
.. [130](#), [11463](#), [11464](#), [11465](#), [13413](#),
[13433](#), [13435](#), [30276](#), [39408](#), [39567](#)
 - `.str_gset_x:N` [38809](#)

- \str_head:N
..... [136](#), [749](#), [13970](#), [13970](#), [13971](#)
- \str_head:n [136](#), [718](#),
[749](#), [12900](#), [12947](#), [13970](#), [13970](#), [13972](#)
- \str_head_ignore_spaces:n
..... [136](#), [13970](#), [13980](#)
- \str_if_empty:N [13532](#), [13534](#)
- \str_if_empty:n [13536](#)
- \str_if_empty:NTF
[131](#), [13528](#), [21642](#), [21665](#), [22575](#), [31354](#)
- \str_if_empty:nTF [131](#), [13528](#)
- \str_if_empty_p:N [131](#), [13528](#)
- \str_if_empty_p:n [131](#), [13528](#)
- \str_if_eq:NN [13563](#), [13568](#)
- \str_if_eq:nn [935](#), [13551](#), [13556](#), [13558](#)
- \str_if_eq:NNTF [131](#), [738](#), [13563](#)
- \str_if_eq:nnTF
..... [100](#), [112](#), [113](#), [132](#), [133](#),
[213](#), [219](#), [220](#), [824](#), [903](#), [4975](#), [8098](#),
[8242](#), [8703](#), [8779](#), [8810](#), [8988](#), [9688](#),
[9731](#), [10027](#), [10030](#), [10095](#), [11540](#),
[11603](#), [11618](#), [13551](#), [13615](#), [13645](#),
[15247](#), [15250](#), [15405](#), [15408](#), [16853](#),
[19440](#), [19497](#), [20284](#), [20538](#), [21231](#),
[21738](#), [24161](#), [24234](#), [25504](#), [25515](#),
[25521](#), [29595](#), [31283](#), [31301](#), [31303](#),
[31352](#), [31374](#), [31875](#), [31931](#), [32308](#),
[32409](#), [34225](#), [36917](#), [37235](#), [37399](#),
[37434](#), [37479](#), [37537](#), [37831](#), [37841](#)
- \str_if_eq_p:NN [131](#), [13563](#)
- \str_if_eq_p:nn [132](#), [8663](#),
[8690](#), [8691](#), [8818](#), [8819](#), [9184](#), [9186](#),
[11754](#), [13551](#), [31403](#), [31674](#), [31692](#),
[31936](#), [35577](#), [36465](#), [37229](#), [38445](#)
- \str_if_exist:N
..... [13528](#), [13530](#), [30186](#), [30188](#)
- \str_if_exist:NTF
..... [130](#), [8777](#), [8846](#), [13528](#)
- \str_if_exist_p:N [130](#), [13528](#)
- \str_if_in:Nn [13570](#), [13576](#)
- \str_if_in:nn [13578](#)
- \str_if_in:NnTF [132](#), [13570](#)
- \str_if_in:nnTF [132](#), [3047](#), [13570](#), [30647](#)
- \str_item:Nn
..... [136](#), [13774](#), [13774](#), [13775](#), [30306](#)
- \str_item:nn
.. [136](#), [744](#), [748](#), [13774](#), [13774](#), [13776](#)
- \str_item_ignore_spaces:nn
..... [136](#), [744](#), [13774](#), [13774](#), [13784](#)
- \str_log:N ... [140](#), [14153](#), [14161](#), [14166](#)
- \str_log:n [140](#), [14153](#), [14160](#)
- \str_lower_case:n [38866](#), [38867](#), [38869](#)
- \str_lowercase:n . [139](#), [297](#), [14000](#),
[14001](#), [14004](#), [38866](#), [38867](#), [38868](#),
[38869](#), [38957](#), [38958](#), [38973](#), [38974](#)
- \str_map_break: [134](#), [13651](#),
[13657](#), [13666](#), [13683](#), [13691](#), [13703](#),
[13709](#), [13715](#), [13716](#), [13718](#), [13725](#)
- \str_map_break:n
.. [134](#), [135](#), [3051](#), [5839](#), [13651](#), [13717](#)
- \str_map_function:NN
..... [133](#), [742](#), [13651](#), [13659](#), [13670](#)
- \str_map_function:nN .. [133](#), [134](#),
[741](#), [5832](#), [13651](#), [13651](#), [13660](#), [15465](#)
- \str_map_inline:Nn
..... [134](#), [13651](#), [13686](#), [13688](#)
- \str_map_inline:nn
.. [134](#), [3045](#), [6585](#), [13651](#), [13671](#), [13687](#)
- \str_map_tokens:Nn
..... [134](#), [13719](#), [13727](#), [13728](#)
- \str_map_tokens:nn
..... [134](#), [13719](#), [13719](#), [13727](#)
- \str_map_variable:NNn
..... [134](#), [13651](#), [13705](#), [13714](#)
- \str_map_variable:nNn
..... [134](#), [13651](#), [13695](#), [13706](#)
- \str_mdfive_hash:n
..... [140](#), [14133](#), [14133](#), [14134](#)
- \str_new:N
... [130](#), [9237](#), [9238](#), [10928](#), [10929](#),
[10930](#), [10959](#), [10960](#), [10961](#), [11772](#),
[13413](#), [14149](#), [14150](#), [14151](#), [14152](#),
[21572](#), [21574](#), [21577](#), [21579](#), [21582](#)
- \str_put_left:Nn
..... [131](#), [735](#), [13440](#), [13449](#), [13472](#), [39487](#)
- \str_put_right:Nn
..... [131](#), [735](#), [13440](#), [13459](#), [13474](#), [39488](#)
- \str_range:Nnn
..... [137](#), [13835](#), [13835](#), [13836](#), [30213](#), [30215](#)
- \str_range:nnn [100](#), [137](#),
[748](#), [3995](#), [5942](#), [13835](#), [13835](#), [13837](#)
- \str_range_ignore_spaces:nnn ...
..... [137](#), [13835](#), [13845](#)
- \str_remove_all:Nn
..... [138](#), [13522](#), [13522](#), [13526](#)
- \str_remove_once:Nn
..... [138](#), [13516](#), [13516](#), [13520](#)
- \str_replace_all:Nnn
... [138](#), [13476](#), [13480](#), [13486](#), [13523](#)
- \str_replace_once:Nnn
... [138](#), [13476](#), [13476](#), [13484](#), [13517](#)
- .str_set:N [244](#), [22165](#)
- \str_set:Nn [131](#), [138](#),
[244](#), [976](#), [9319](#), [9320](#), [9518](#), [9519](#),
[11553](#), [11554](#), [11555](#), [13440](#), [13440](#),

- 13469, 13710, 21621, 21623, 22243,
 22245, 22350, 22472, 30190, 30194
 \str_set_convert:Nnnn 144,
 145, 759, 770, 14344, 14344, 14349
 \str_set_convert:NnnnTF
 144, 759, 14344
 .str_set_e:N 244, 22165
 \str_set_eq:NN 130, 13413,
 13432, 13434, 30272, 39407, 39486
 .str_set_x:N 38809
 \str_show:N . . 140, 14153, 14154, 14159
 \str_show:n 140, 14153, 14153
 \str_tail:N . . 136, 13985, 13985, 13986
 \str_tail:n
 136, 451, 13985, 13985, 13987, 31454
 \str_tail_ignore_spaces:n
 136, 13985, 13994
 \str_titlecase:n 38961, 38962
 \str_upper_case:n 38866, 38871, 38873
 \str_uppercase:n . 139, 297, 14000,
 14002, 14005, 38870, 38871, 38872,
 38873, 38959, 38960, 38979, 38980
 \str_use:N 135, 13413
 \c_tilde_str 141, 14135
 \g_tmpa_str 141, 14149
 \l_tmpa_str 138, 141, 14149
 \g_tmpb_str 141, 14149
 \l_tmpb_str 141, 14149
 \c_underscore_str . . 141, 14135, 29902
 \c_zero_str
 141, 14135, 30166, 30172, 30272, 30276
 str internal commands:
 \g__str_alias_prop . 762, 14177, 14417
 \c__str_byte_1_tl 14258
 \c__str_byte_0_tl 14258
 \c__str_byte_1_tl 14258
 \c__str_byte_255_tl 14258
 \c__str_byte_⟨number⟩_tl 757
 \l__str_byte_flag
 764, 14205, 14485, 14499,
 14502, 14767, 14776, 14820, 14835
 __str_case:nnTF 13584,
 13587, 13592, 13597, 13602, 13604
 __str_case:nw
 13584, 13605, 13613, 13617
 __str_case_e:nnTF 13584,
 13622, 13627, 13632, 13637, 13639
 __str_case_e:nw
 13584, 13640, 13643, 13647
 __str_case_end:nw
 13584, 13616, 13646, 13649
 __str_change_case:nn
 . . 14000, 14000, 14001, 14002, 14006
 __str_change_case_aux:nn
 14000, 14008, 14011
 __str_change_case_char:nN
 14000, 14025, 14034
 __str_change_case_char:nnn
 14000, 14045, 14074,
 14077, 14083, 14092, 14105, 14114
 __str_change_case_char:nnnnn
 14000, 14117, 14119
 __str_change_case_char_aux:nnn
 14000, 14109, 14115
 __str_change_case_char_auxi:nN
 14000, 14052, 14056, 14062
 __str_change_case_char_auxii:nN
 14000, 14055, 14059, 14073
 __str_change_case_codepoint:nN
 14000, 14038, 14044, 14047
 __str_change_case_codepoint:nNN
 14000, 14065, 14075
 __str_change_case_codepoint:nNNN
 14000, 14068, 14081
 __str_change_case_codepoint:nNNNN
 14000, 14090
 __str_change_case_codepoint:nNNNNN
 14069
 __str_change_case_end:nw 14000
 __str_change_case_end:wn
 14019, 14037
 __str_change_case_loop:nw
 . . 14000, 14013, 14021, 14032, 14112
 __str_change_case_output:nw
 . . 14000, 14016, 14018, 14031, 14107
 __str_change_case_result:n
 . . 14000, 14014, 14016, 14017, 14019
 __str_change_case_space:n
 14000, 14024, 14029
 __str_collect_delimit_by_q_ -
 stop:w 13863, 13886, 13886
 __str_collect_end:nnnnnnnw
 747, 13886, 13905, 13910
 __str_collect_end:wn
 13886, 13893, 13903
 __str_collect_loop:wn
 13886, 13887, 13888, 13899
 __str_collect_loop:wnNNNNNN
 13886, 13891, 13897
 __str_convert:nnn
 761, 762, 14389, 14390, 14404, 14404
 __str_convert:nnnn
 762, 14404, 14408, 14413
 __str_convert:NNnNN
 14386, 14391, 14394
 __str_convert:nNNnnn . . . 14344,
 14345, 14347, 14352, 14361, 14366

- __str_convert:wwwnn
 [761](#), [14371](#), [14376](#), [14386](#), [14386](#)
- __str_convert_decode:
 [14375](#), [14509](#), [14509](#)
- __str_convert_decode_clist: ...
 [14549](#), [14549](#)
- __str_convert_decode_eight_
 bit:n [14570](#), [14614](#), [14614](#)
- __str_convert_decode_utf16: . [15236](#)
- __str_convert_decode_utf16be: [15236](#)
- __str_convert_decode_utf16le: [15236](#)
- __str_convert_decode_utf32: . [15394](#)
- __str_convert_decode_utf32be: [15394](#)
- __str_convert_decode_utf32le: [15394](#)
- __str_convert_decode_utf8: . . [15055](#)
- __str_convert_encode:
 [14380](#), [14513](#), [14519](#), [14525](#)
- __str_convert_encode_clist: ...
 [14560](#), [14560](#)
- __str_convert_encode_eight_
 bit:n [14572](#), [14641](#), [14642](#)
- __str_convert_encode_utf16: . [15151](#)
- __str_convert_encode_utf16be: [15151](#)
- __str_convert_encode_utf16le: [15151](#)
- __str_convert_encode_utf32: . [15334](#)
- __str_convert_encode_utf32be: [15334](#)
- __str_convert_encode_utf32le: [15334](#)
- __str_convert_encode_utf8: . . [14980](#)
- __str_convert_escape:
 [14507](#), [14507](#), [14508](#)
- __str_convert_escape_bytes: ...
 [14507](#), [14508](#)
- __str_convert_escape_hex:
 [14900](#), [14900](#)
- __str_convert_escape_name:
 [777](#), [14904](#), [14906](#)
- __str_convert_escape_string: ...
 [14927](#), [14929](#)
- __str_convert_escape_url:
 [14959](#), [14959](#)
- __str_convert_gmap:N
 [14302](#), [14302](#), [14510](#),
 [14622](#), [14901](#), [14907](#), [14930](#), [14960](#)
- __str_convert_gmap_internal:N ..
 [14318](#),
 [14318](#), [14520](#), [14528](#), [14562](#), [14651](#),
 [14981](#), [15164](#), [15336](#), [15340](#), [15342](#)
- __str_convert_gmap_internal_
 loop:Nw [14318](#)
- __str_convert_gmap_internal_
 loop:Nww [14322](#), [14328](#), [14332](#)
- __str_convert_gmap_loop:NN
 [14302](#), [14306](#), [14312](#), [14316](#)
- __str_convert_lowercase_
 alphanum:n ... [14409](#), [14441](#), [14441](#)
- __str_convert_lowercase_
 alphanum_loop:N
 [14441](#), [14443](#), [14447](#), [14465](#)
- __str_convert_pdfname:n
 [15462](#), [15465](#), [15471](#), [15498](#)
- __str_convert_pdfname_bytes:n ..
 [15462](#), [15474](#), [15477](#)
- __str_convert_pdfname_bytes_
 aux:n [15462](#), [15479](#), [15482](#)
- __str_convert_pdfname_bytes_
 aux:nnn [15462](#)
- __str_convert_pdfname_bytes_
 aux:nnnn [15483](#), [15484](#)
- __str_convert_unescape:
 [14491](#), [14497](#), [14505](#), [14506](#)
- __str_convert_unescape_bytes: ..
 [14491](#), [14506](#)
- __str_convert_unescape_hex: ...
 [14716](#), [14716](#)
- __str_convert_unescape_name: ...
 [772](#), [14762](#)
- __str_convert_unescape_string: .
 [14812](#), [14817](#)
- __str_convert_unescape_url: . [14762](#)
- __str_count:n
 [748](#), [13790](#), [13850](#), [13932](#), [13942](#)
- __str_count_aux:n
 .. [13932](#), [13936](#), [13944](#), [13949](#), [13952](#)
- __str_count_loop:NNNNNNNN [13932](#),
 [13939](#), [13945](#), [13950](#), [13963](#), [13968](#)
- __str_count_spaces_loop:w
 [13912](#), [13919](#), [13925](#), [13930](#)
- __str_declare_eight_bit_
 aux:NNnnn [14566](#), [14573](#), [14576](#)
- __str_declare_eight_bit_
 encoding:nnnn
 [766](#), [14566](#), [14566](#), [15501](#),
 [15508](#), [15572](#), [15614](#), [15671](#), [15772](#),
 [15859](#), [15945](#), [16019](#), [16032](#), [16085](#),
 [16183](#), [16246](#), [16284](#), [16299](#), [38884](#)
- __str_declare_eight_bit_loop:Nn
 [14566](#), [14584](#), [14608](#), [14612](#)
- __str_declare_eight_bit_
 loop:Nnn [14566](#), [14581](#), [14602](#), [14606](#)
- __str_decode_clist_char:n
 [14549](#), [14555](#), [14558](#)
- __str_decode_eight_bit_aux:n ...
 [14614](#), [14628](#), [14632](#)
- __str_decode_eight_bit_aux:Nn ..
 [14614](#), [14618](#), [14625](#)
- __str_decode_native_char:N
 [14509](#), [14510](#), [14511](#)

__str_decode_utf_viii_aux:wNnnwN
 [15055](#), [15098](#), [15110](#)
 __str_decode_utf_viii_continuation:wwN
 [15055](#), [15083](#), [15090](#), [15126](#)
 __str_decode_utf_viii_end:
 [15055](#), [15065](#), [15140](#)
 __str_decode_utf_viii_overflow:w
 [15055](#), [15124](#), [15133](#)
 __str_decode_utf_viii_start:N ..
 [15055](#), [15064](#), [15070](#),
 [15088](#), [15091](#), [15108](#), [15111](#), [15131](#)
 __str_decode_utf_xvi:Nw
 [785](#), [15236](#), [15237](#),
 [15239](#), [15248](#), [15251](#), [15252](#), [15255](#)
 __str_decode_utf_xvi_bom:NN ...
 [15236](#), [15242](#), [15245](#)
 __str_decode_utf_xvi_error:nNN .
 [15270](#),
 [15288](#), [15307](#), [15316](#), [15321](#), [15322](#)
 __str_decode_utf_xvi_extra:NNw .
 [15270](#), [15278](#), [15320](#)
 __str_decode_utf_xvi_pair:NN ...
 [785](#), [786](#), [15264](#),
 [15270](#), [15270](#), [15282](#), [15285](#), [15309](#)
 __str_decode_utf_xvi_pair_
 end:Nw .. [15270](#), [15273](#), [15289](#), [15311](#)
 __str_decode_utf_xvi_quad:NNwNN
 [15270](#), [15277](#), [15284](#)
 __str_decode_utf_xxxii:Nw
 [789](#), [15394](#), [15395](#),
 [15397](#), [15406](#), [15409](#), [15410](#), [15413](#)
 __str_decode_utf_xxxii_bom:NNNN
 [15394](#), [15400](#), [15403](#)
 __str_decode_utf_xxxii_end:w ...
 [15394](#), [15430](#), [15450](#)
 __str_decode_utf_xxxii_loop:NNNN
 [15394](#), [15421](#), [15427](#), [15448](#)
 __str_encode_clist_char:n
 [14560](#), [14562](#), [14565](#)
 __str_encode_eight_bit_aux:NNn .
 [14641](#), [14646](#), [14654](#)
 __str_encode_eight_bit_aux:nnN .
 [14641](#), [14656](#), [14664](#)
 __str_encode_native_char:n
 .. [14513](#), [14520](#), [14521](#), [14528](#), [14532](#)
 __str_encode_utf_vii_loop:wnnw [778](#)
 __str_encode_utf_viii_char:n ...
 [14980](#), [14981](#), [14982](#)
 __str_encode_utf_viii_loop:wnnw
 [14980](#), [14984](#), [14991](#), [14997](#)
 __str_encode_utf_xvi_aux:N
 .. [15151](#), [15153](#), [15157](#), [15159](#), [15160](#)
 __str_encode_utf_xvi_be:nn ... [783](#)
 __str_encode_utf_xvi_char:n ...
 [15151](#), [15164](#), [15167](#)
 __str_encode_utf_xxxii_be:n ...
 [15334](#), [15336](#), [15340](#), [15343](#)
 __str_encode_utf_xxxii_be_
 aux:nn [15334](#), [15345](#), [15348](#)
 __str_encode_utf_xxxii_le:n ...
 [15334](#), [15342](#), [15354](#)
 __str_encode_utf_xxxii_le_
 aux:nn [15334](#), [15356](#), [15359](#)
 __str_end [15185](#), [15365](#)
 \l__str_end_flag
 [15187](#), [15202](#), [15229](#), [15260](#),
 [15366](#), [15373](#), [15387](#), [15416](#), [15454](#)
 \g__str_error_bool [14204](#),
 [14341](#), [14351](#), [14355](#), [14360](#), [14364](#)
 \l__str_error_flag
 [14205](#), [14527](#), [14529](#), [14535](#),
 [14621](#), [14623](#), [14635](#), [14650](#), [14652](#),
 [14668](#), [14719](#), [14730](#), [14739](#), [14752](#),
 [14768](#), [14777](#), [14790](#), [14795](#), [14821](#),
 [14836](#), [14876](#), [15057](#), [15068](#), [15079](#),
 [15103](#), [15117](#), [15137](#), [15144](#), [15162](#),
 [15165](#), [15174](#), [15257](#), [15268](#), [15324](#),
 [15417](#), [15425](#), [15435](#), [15440](#), [15455](#)
 __str_escape_hex_char:N
 [14900](#), [14901](#), [14902](#)
 __str_escape_name_char:n
 .. [14904](#), [14907](#), [14908](#), [15475](#), [15498](#)
 \c__str_escape_name_not_str
 [775](#), [14904](#)
 \c__str_escape_name_str .. [775](#), [14904](#)
 __str_escape_string_char:N
 [14927](#), [14930](#), [14931](#)
 \c__str_escape_string_str [14927](#)
 __str_escape_url_char:n
 [14959](#), [14960](#), [14961](#)
 __str_extra [15002](#), [15185](#)
 \l__str_extra_flag
 [15003](#), [15012](#), [15035](#), [15059](#),
 [15078](#), [15186](#), [15201](#), [15224](#), [15259](#)
 __str_filter_bytes:n ... [14467](#),
 [14473](#), [14490](#), [14501](#), [14782](#), [14844](#)
 __str_filter_bytes_aux:N
 [14467](#), [14475](#), [14479](#), [14487](#)
 __str_head:w [749](#), [13970](#), [13974](#), [13978](#)
 __str_hexadecimal_use:N [14239](#)
 __str_hexadecimal_use:NTF
 [772](#), [14239](#), [14736](#), [14746](#), [14785](#), [14787](#)
 __str_if_contains_char:Nn ... [14207](#)
 __str_if_contains_char:nn ... [14216](#)
 __str_if_contains_char:NnTF ...
 [14207](#), [14916](#), [14922](#), [14935](#)

- __str_if_contains_char:nnTF 756, 14207, 14969, 14975
- __str_if_contains_char_aux:nn 14207, 14209, 14214
- __str_if_contains_char_auxi:nN 14207, 14215, 14218, 14222, 14227
- __str_if_contains_char_true: 14207, 14225, 14229
- __str_if_eq:nn 738, 13538, 13538, 13542, 13548, 13553, 13560, 13565
- __str_if_escape_name:n 14913
- __str_if_escape_name:nTF 14904, 14910
- __str_if_escape_string:N 14947
- __str_if_escape_string:nTF 14927, 14933
- __str_if_escape_url:n 14966
- __str_if_escape_url:nTF 14959, 14963
- __str_if_flag_error:Nnn 759, 760, 14334, 14334, 14353, 14362, 14502, 14529, 14623, 14652, 14730, 14776, 14777, 14835, 14836, 15068, 15165, 15268, 15425
- __str_if_flag_no_error:Nnn 759, 14334, 14340, 14353, 14362
- __str_if_flag_times:nTF . 14342, 14342, 15011, 15012, 15013, 15014, 15200, 15201, 15202, 15372, 15373
- __str_if_recursion_tail_-break:NN 13411, 13411, 13691, 13709
- __str_if_recursion_tail_stop_-do:Nn 13411, 13412, 14036
- \l__str_internal_tl 762, 14170, 14259, 14260, 14262, 14417, 14418, 14419, 14421, 14425, 14429, 14436, 14568
- __str_item:nn 744, 13774, 13780, 13785, 13786
- __str_item:w 744, 13774, 13788, 13793
- __str_map_function:nn 741, 13651, 13654, 13663, 13668, 13722
- __str_map_function:w 741, 13651, 13653, 13661, 13662, 13722
- __str_map_inline:NN 13651, 13678, 13689, 13693
- __str_map_variable:NnN 13651, 13699, 13707, 13712
- \c__str_max_byte_int . . 14174, 14534
- __str_missing 15002, 15185
- \l__str_missing_flag 15002, 15011, 15029, 15058, 15102, 15143, 15185, 15200, 15219, 15258
- \l__str_modulo_int 14641
- __str_octal_use:N 14231
- __str_octal_use:nTF 756, 757, 14231, 14847, 14849, 14851
- __str_output_byte:n 787, 14270, 14270, 14299, 14300, 14462, 14667, 14994, 15000, 15352, 15361
- __str_output_byte:w 772, 14270, 14271, 14272, 14723, 14749, 14784, 14846
- __str_output_byte_pair:nnN 14286, 14288, 14293, 14296
- __str_output_byte_pair_be:n 14286, 14286, 15153, 15157, 15351
- __str_output_byte_pair_le:n 14286, 14291, 15159, 15362
- __str_output_end: 772, 14270, 14271, 14284, 14728, 14748, 14798, 14880, 14884
- __str_output_hexadecimal:n 14270, 14278, 14903, 14911, 14964, 15486, 15487, 15490, 15493
- __str_overflow 15002, 15365
- \l__str_overflow_flag . . . 15005, 15014, 15047, 15061, 15136, 15365, 15372, 15380, 15415, 15434, 15439
- __str_overlong 15002
- \l__str_overlong_flag 15004, 15013, 15040, 15060, 15116
- __str_range:nnn 13835, 13841, 13846, 13847
- __str_range:nnw . 13835, 13857, 13861
- __str_range:w . . 13835, 13849, 13855
- __str_range_normalize:nn 13858, 13859, 13867, 13867
- __str_replace:NNNnn 13476, 13477, 13479, 13481, 13483, 13488
- __str_replace_aux:NNNnnn 13476, 13497, 13503
- __str_replace_next:w . . . 13476, 13481, 13483, 13505, 13508, 13515
- \c__str_replacement_char_int 14173, 14636, 15080, 15104, 15118, 15138, 15145, 15175, 15327, 15436, 15441, 15457
- \g__str_result_tl 754, 758-760, 764, 766, 772, 785, 787, 789, 14172, 14304, 14308, 14320, 14324, 14370, 14381, 14382, 14384, 14500, 14501, 14551, 14552, 14555, 14563, 14721, 14725, 14770, 14772, 14823, 14826, 14829, 14832, 15062, 15064, 15154, 15237, 15239, 15243, 15262, 15337, 15395, 15397, 15400, 15419
- __str_skip_end:NNNNNNNN 745, 13814, 13831, 13834

- __str_skip_end:w [13814](#), [13819](#), [13829](#)
- __str_skip_exp_end:w
..... [745](#), [747](#), [13801](#),
[13810](#), [13814](#), [13814](#), [13826](#), [13865](#)
- __str_skip_loop:wNNNNNNNN
..... [13814](#), [13817](#), [13824](#)
- __str_tail_auxi:w [13985](#), [13989](#), [13993](#)
- __str_tail_auxii:w
..... [749](#), [13985](#), [13996](#), [13999](#)
- __str_tmp:n [13414](#), [13420](#), [13423](#)
- __str_tmp:w [768](#), [772](#),
[783](#), [785](#), [789](#), [14170](#), [14170](#), [14616](#),
[14622](#), [14644](#), [14651](#), [14762](#), [14808](#),
[14810](#), [14815](#), [14840](#), [15163](#), [15170](#),
[15175](#), [15177](#), [15180](#), [15181](#), [15261](#),
[15276](#), [15281](#), [15292](#), [15295](#), [15301](#),
[15302](#), [15418](#), [15433](#), [15438](#), [15444](#)
- __str_to_other_end:w
..... [743](#), [13729](#), [13744](#), [13749](#)
- __str_to_other_fast_end:w
..... [13752](#), [13767](#), [13772](#)
- __str_to_other_fast_loop:w
..... [13754](#), [13763](#), [13770](#)
- __str_to_other_loop:w
..... [743](#), [13729](#), [13731](#), [13740](#), [13746](#)
- __str_unescape_hex_auxi:N
.. [14716](#), [14724](#), [14733](#), [14740](#), [14749](#)
- __str_unescape_hex_auxii:N
..... [14716](#), [14737](#), [14743](#), [14753](#)
- __str_unescape_name_loop:wNN ...
..... [14762](#), [14809](#)
- __str_unescape_string_loop:wNNN
.. [14812](#), [14831](#), [14842](#), [14881](#), [14884](#)
- __str_unescape_string_newlines:wN
..... [14812](#), [14825](#), [14885](#), [14889](#)
- __str_unescape_string_repeat:NNNNNN
.. [14812](#), [14856](#), [14858](#), [14860](#), [14883](#)
- __str_unescape_url_loop:wNN ...
..... [14762](#), [14811](#)
- __str_use_i_delimit_by_s_
stop:nw [749](#), [13407](#), [13408](#),
[13800](#), [13809](#), [13928](#), [13979](#), [13982](#)
- __str_use_none_delimit_by_s_
stop:w [13407](#),
[13407](#), [13510](#), [13798](#), [13807](#), [13966](#),
[14330](#), [14604](#), [14610](#), [14995](#), [15087](#)
- \strcmp [5](#)
- \string [417](#)
- \sum [1430](#)
- \suppressfontnotfounderror [704](#)
- \suppressifcsnameerror [916](#)
- \suppresslongerror [917](#)
- \suppressmathparerror [918](#)
- \suppressoutererror [919](#)
- \suppressprimitiveerror [920](#)
- \synctex [684](#)
- sys commands:
 \c_sys_backend_str ... [79](#), [8774](#), [8846](#)
 \c_sys_day_int [74](#), [8983](#)
 \c_sys_engine_exec_str
 [75](#), [599](#), [8665](#), [11648](#)
 \c_sys_engine_format_str
 [75](#), [599](#), [8665](#), [11649](#)
 \c_sys_engine_str
 [75](#), [599](#), [679](#), [8648](#), [8719](#)
 \c_sys_engine_version_str .. [75](#), [8717](#)
 \sys_ensure_backend: . [79](#), [8844](#), [8844](#)
 \sys_finalise: .. [79](#), [8776](#), [9165](#), [9165](#)
 \sys_get_query:nN [78](#), [9088](#), [9088](#)
 \sys_get_query:nnN ... [78](#), [9088](#), [9090](#)
 \sys_get_query:nnnN
 [78](#), [9088](#), [9089](#), [9091](#), [9092](#), [9158](#)
 \sys_get_shell:nnN
 [77](#), [8869](#), [8869](#), [8874](#), [9117](#)
 \sys_get_shell:nnNTF [77](#), [91](#), [8869](#), [8871](#)
 \sys_gset_rand_seed:n
 [77](#), [279](#), [9028](#), [9030](#)
 \c_sys_hour_int [74](#), [8983](#)
 \sys_if_engine luatex:TF
 [75](#), [105](#), [8648](#), [8673](#), [8698](#),
 [8812](#), [8822](#), [8823](#), [8909](#), [8931](#), [8963](#),
 [9044](#), [9071](#), [10256](#), [11084](#), [11295](#),
 [11447](#), [11493](#), [11733](#), [11781](#), [19529](#),
 [30374](#), [30416](#), [30461](#), [30474](#), [30500](#),
 [30569](#), [30593](#), [30630](#), [38519](#), [38584](#)
 \sys_if_engine luatex_p: [75](#), [8648](#),
 [10442](#), [14195](#), [14469](#), [14493](#), [14515](#),
 [14685](#), [15468](#), [30685](#), [30711](#), [30775](#),
 [30955](#), [31546](#), [31586](#), [32582](#), [33698](#)
 \sys_if_engine pdftex:TF [75](#),
 [8648](#), [8669](#), [8693](#), [9198](#), [31556](#), [31608](#)
 \sys_if_engine pdftex_p:
 [75](#), [8648](#), [8707](#)
 \sys_if_engine ptex:TF
 [75](#), [8648](#), [8671](#), [8696](#), [32547](#)
 \sys_if_engine ptex_p:
 [75](#), [3598](#), [3620](#), [8648](#)
 \sys_if_engine uptex:TF
 [75](#), [8648](#), [8672](#), [8697](#)
 \sys_if_engine uptex_p:
 [75](#), [3599](#), [3621](#), [8648](#)
 \sys_if_engine xetex:TF
 .. [6](#), [75](#), [8648](#), [8670](#), [8695](#), [8793](#), [9193](#)
 \sys_if_engine xetex_p: . [75](#), [8648](#),
 [8994](#), [14196](#), [14470](#), [14494](#), [14516](#),
 [14686](#), [15469](#), [30685](#), [30711](#), [30776](#),
 [30956](#), [31547](#), [31587](#), [32583](#), [33699](#)
 \sys_if_output_dvi:TF [76](#), [9174](#)

- \sys_if_output_dvi_p: 76, 9174
 - \sys_if_output_pdf:TF
 - 76, 8808, 9174, 9196
 - \sys_if_output_pdf_p: 76, 9174
 - \sys_if_platform_unix:TF
 - 76, 8774, 11751
 - \sys_if_platform_unix_p:
 - 76, 8774, 11751
 - \sys_if_platform_windows:TF
 - 76, 8774, 11751
 - \sys_if_platform_windows_p:
 - 76, 8774, 11751
 - \sys_if_shell:TF
 - 77, 78, 8876, 9079, 9115, 10262, 10512
 - \sys_if_shell_p: 77, 9079
 - \sys_if_shell_restricted:TF
 - 78, 9079, 9105, 9107
 - \sys_if_shell_restricted_p: 78, 9079
 - \sys_if_shell_unrestricted:TF
 - 77, 9079
 - \sys_if_shell_unrestricted_p:
 - 77, 9079
 - \sys_if_timer_exist:TF 76, 9033
 - \sys_if_timer_exist_p: 76, 9033
 - \c_sys_jobname_str . 74, 99, 611, 8981
 - \sys_load_backend:n
 - 79, 8774, 8774, 8847
 - \sys_load_debug: 79,
 - 1571, 1577, 8850, 8850, 8862, 10129
 - \sys_load_deprecation: . 38897, 38898
 - \c_sys_minute_int 74, 8983
 - \c_sys_month_int 74, 8983
 - \c_sys_output_str 76, 9174
 - \c_sys_platform_str
 - 76, 8774, 11733, 11754
 - \sys_rand_seed: 76, 157, 279, 9024, 9026
 - \c_sys_shell_escape_int
 - 77, 9067, 9082, 9084, 9086
 - \sys_shell_now:n
 - 78, 8911, 8933, 8937, 8940
 - \sys_shell_shipout:n
 - 78, 8942, 8965, 8969, 8972
 - \sys_split_query:nN . . 79, 9146, 9146
 - \sys_split_query:nnN . 79, 9146, 9148
 - \sys_split_query:nnnN
 - 78, 79, 9146, 9147, 9149, 9155
 - \sys_timer:
 - 75, 9033, 9046, 9052, 9056, 9060
 - \c_sys_timestamp_str 74, 9015
 - \c_sys_year_int 74, 8983
 - sys internal commands:
 - \g__sys_backend_tl
 - 8784, 8785, 8786, 9188
 - __sys_const:nn . 8632, 8632, 8662,
 - 9065, 9081, 9083, 9085, 9183, 9185
 - \g__sys_debug_bool . . 8849, 8852, 8854
 - __sys_elapsedtime: . 9033, 9047, 9066
 - __sys_everyjob:n
 - 8973, 8978, 8981, 8983,
 - 9015, 9024, 9028, 9067, 9079, 9163
 - \g__sys_everyjob_tl 8973
 - __sys_finalise:n
 - 9165, 9171, 9174, 9189, 9206
 - \g__sys_finalise_tl 9165
 - __sys_get:nnN 8869, 8877, 8880
 - __sys_get_do:Nw 8869, 8894, 8903
 - __sys_get_query:Nw 9132, 9143
 - __sys_get_query_auxi:nnnN
 - 9088, 9095, 9097, 9112
 - __sys_get_query_auxii:nnnN
 - 9088, 9099, 9113, 9137
 - \l__sys_internal_tl 8867
 - __sys_load_backend_check:N
 - 8774, 8785, 8791
 - \c__sys_marker_tl . . . 8868, 8892, 8904
 - __sys_shell_now:n 8911, 8934
 - __sys_shell_shipout:n . . . 8942, 8966
 - \c__sys_shell_stream_int
 - 8909, 8938, 8970
 - __sys_tmp:w
 - 8986, 9007, 9009, 9010, 9011, 9012
 - \l__sys_tmp_tl 8631,
 - 9129, 9130, 9133, 9158, 9159, 9160
 - syst commands:
 - \c_syst_catcodes_n 30636, 30640
 - \c_syst_last_allocated_toks . . 3187
- ### T
- \T 65
 - \t 32057, 34380, 34406
 - \tabskip 418
 - \tagcode 685
 - tan 276
 - tand 277
 - \tate 1187
 - \tbaselineshift 1188
 - TeX and L^AT_EX 2_ε commands:
 - \@ 14136
 - \@@@hyph 359
 - \@end 1225, 1226
 - \@hyph 1229, 1232
 - \@input 1227
 - \@italiccorr 1233
 - \@shipout 1235, 1236
 - \@tracingfonts 360, 1271
 - \@underline 1234
 - \@addtofilelist 11443

- \@changed@cmd 31989, 34238
- \@classoptionslist . . 9208, 9210, 9212
- \@current@cmd 31988, 34237
- \@currnamestack
- 661, 10950, 10952, 10953
- \@expl@finalise@setup@@ 8856, 8858,
- 30158, 30160, 30676, 30677, 31675,
- 31677, 33661, 33663, 38446, 38448
- \@expl@luadata@bytecode 19
- \@filelist 104, 661, 674,
- 677, 11442, 11563, 11566, 11575, 11580
- \@firstofone 24
- \@firstoftwo 378
- \@gobble 26
- \@gobbletwo 26
- \@kernel@after@begindocument . . .
- 8860, 31679, 33665, 38450
- \@kernel@before@begindocument . . .
- 38703, 38705
- \@protected@testopt 1298, 31976
- \@secondoftwo 378
- \@sptoken 200
- \@tempa 1243, 1257, 1260
- \@tfor 360, 1243
- \@uclclist 1335, 33690
- \@unexpandable@protect 1062
- \@unusedoptionlist 9227
- \active@prefix 31837
- \afterassignment 565
- \aftergroup 14
- \AtBeginDocument 360
- \begingroup 13
- \bgroup 200
- \botmark 904
- \box 310
- \catcodetable 1258, 1262, 1264
- \char 212
- \chardef 203, 204, 585, 588, 845, 1287
- \conditionally@traceoff
- 653, 9631, 10671
- \conditionally@traceon 9649
- \copy 303
- \count 211, 431
- \cr 596
- \CROP@shipout 1244
- \csname 21, 366, 663, 664
- \csstring 388
- \currentgrouplevel . . . 401, 628, 1261
- \currentgrouptype 401, 628
- \day 74
- \declare@file@substitution . . . 30161
- \def 211
- \detokenize 115
- \development@branch@name 11651, 11652
- \dimen 903
- \dimendef 903
- \dimexpr 1359
- \directlua 105
- \dp 304, 1063, 1064
- \dup@shipout 1245
- \e@alloc@ccodetable@count 30634
- \e@alloc@top 431, 3173
- \edef 3, 6, 694
- \egroup 200
- \else 28
- \end 359, 621
- \endcsname 21
- \endgroup 13
- \endinput 84
- \endlinechar 93, 125, 289–
- 291, 700–702, 904, 1258, 1259, 1261
- \endtemplate 73, 596
- \errhelp 617
- \errmessage 617, 618
- \errorcontextlines 368, 618, 728, 1362
- \escapechar 115, 387, 402, 471, 652
- \everyeof 702
- \everyjob 606
- \everypar 30, 207, 404
- \expandafter 39, 41
- \expanded
- 3, 6, 26, 34, 335, 407, 410, 423, 701
- \fi 28, 210
- \firstmark 417, 904
- \fmtname 75
- \font 210, 902
- \fontdimen 61, 257, 1011–1014
- \frozen@everydisplay 1230
- \frozen@everymath 1231
- \futurelet
- 446, 449, 451, 462, 596, 909, 912
- \global 338
- \GPTorg@shipout 1246
- \halign 73, 103, 405, 596, 895
- \hskip 235
- \ht 304, 1063, 1064
- \hyphen 904
- \hyphenchar 1011
- \if 29
- \ifcase 179
- \ifcat 29
- \ifcsname 29
- \ifdefined 29
- \ifdim 238
- \ifeof 99
- \iffalse 28, 66, 692
- \ifhbox 314
- \ifhmode 29

- \ifincsname 335
- \ifinner 29
- \ifmode 29, 692
- \ifnum 179
- \ifodd 180, 913
- \iftrue 28, 66, 692
- \ifvbox 314
- \ifvmode 29
- \ifvoid 314
- \ifx 29
- \indent 404
- \infty 272
- \input 359
- \input@path 100, 666, 11131, 11133
- \italiccorr 904
- \jobname 74, 606
- \kcatcode 294
- \lastnamedcs 391
- \lccode 449, 454, 863
- \leavevmode 30
- \let 338
- \letcharcode 893
- \LL@shipout 1247
- \loctoks 431
- \long 5, 212, 728
- \lower 1378
- \lowercase 464, 553, 554
- \luaescapestring 106
- \makeatletter 9
- \mathchar 212
- \mathchardef 204, 845, 1287
- \mathop 1429
- \mathord 325
- \maxdimen 230
- \meaning 20,
201, 211, 212, 448, 902, 903, 912, 913
- \mem@oldshipout 1248
- \message 34
- \month 74
- \newcatcodetable 1258
- \newif 66, 108
- \newlinechar
125, 368, 392, 618, 650, 700–702, 728
- \newread 640
- \newtoks 44, 442, 471
- \newwrite 647
- \noexpand 40, 210
- \nullfont 904, 905
- \number 179, 842, 1119
- \numexpr 367
- \opem@shipout 1249
- \or 179
- \outer 7, 212, 446, 462–464,
640, 647, 895, 913, 914, 1465, 1466
- \parindent 30
- \pdfescapehex 771
- \pdfescapename 143, 771
- \pdfescapestring 143, 771
- \pdffeedback 607
- \pdffilesize 666
- \pdfmapfile 361
- \pdfmapline 361
- \pdfstrcmp 334, 349, 738
- \pdfuniformdeviate 279
- \pgfpages@originalshipout 1250
- \pi 272
- \pr@shipout 1251
- \primitive 360, 607
- \protect 654, 1061, 1062, 1348
- \protected 212, 728
- \protected@edef 1292
- \ProvidesClass 9
- \ProvidesFile 9
- \ProvidesPackage 9
- \quitvmode 404
- \read 93, 645
- \readline 93, 645
- \relax 28, 210, 384,
389, 402, 463, 589, 663, 865, 867,
919, 926, 928, 1018, 1020, 1045, 1077
- \RequirePackage 10, 661
- \romannumeral 42, 1018, 1286, 1293
- \savecatcodetable 1260
- \scantokens 125, 145, 665, 700
- \shipout 360
- \show 20, 117, 402
- \showbox 1362
- \showgroups 14, 402
- \showthe 401, 863, 957, 961, 963
- \showtokens 117, 623, 728
- \sin 272
- \skip 454, 455
- \space 904
- \splitbotmark 904
- \splitfirstmark 904
- \SS 1351
- \strcmp 334, 349
- \string 201, 449, 451, 452
- \tenrm 210
- \the
170, 210, 229, 234, 237, 409, 842, 1359
- \time 74
- \toks 44, 157, 179,
429–437, 442, 448, 450, 452, 454,
455, 458, 471, 472, 522, 529, 530,
534, 535, 545, 553, 561, 574, 583, 827
- \toksdef 442
- \topmark 211, 904

- \tracingfonts 360
- \tracingnesting 665, 700
- \tracingonline 1362
- \typeout 654
- \Ucharcat 894
- \undefined 928
- \unexpanded 40,
116, 117, 121, 122, 154, 155, 160,
161, 187, 191–193, 218, 694, 717, 848
- \unhbox 310
- \unhcopy 307
- \uniformdeviate 279
- \unless 28
- \unvbox 310
- \unvcopy 309
- \uppercase 553
- \usepackage 661
- \UTFviii@four@octets 31836
- \UTFviii@three@octets 31835
- \UTFviii@two@octets 31834
- \valign 596
- \value 166
- \verb 125
- \verso@orig@shipout 1253
- \vskip 236
- \vtop 1384
- \wd 304, 1063, 1064
- \write 97, 650
- \year 74
- tex commands:
 - \tex_above:D 151
 - \tex_abovedisplayshortskip:D .. 152
 - \tex_abovedisplayskip:D 153
 - \tex_abovewithdelims:D 154
 - \tex_accent:D 155
 - \tex_adjdemerits:D 156
 - \tex_adjustinterwordglue:D 628
 - \tex_adjustspacing:D 629, 932
 - \tex_advance:D .. 157, 3280, 3287,
3290, 3731, 3733, 3766, 3768, 5269,
17611, 17613, 17615, 17617, 17623,
17625, 17627, 17629, 20792, 20795,
20801, 20804, 21218, 21220, 21224,
21226, 21311, 21313, 21317, 21319
 - \tex_afterassignment:D . 158, 3672,
3715, 7526, 7590, 7734, 19718, 30223
 - \tex_aftergroup:D ... 1264, 159, 1427
 - \tex_alignmark:D 780
 - \tex_aligntab:D 781
 - \tex_appendkern:D 630
 - \tex_atop:D 160
 - \tex_atopwithdelims:D 161
 - \tex_attribute:D 782
 - \tex_attributedef:D 783
 - \tex_automaticdiscretionary:D .. 785
 - \tex_automatichyphenmode:D 786
 - \tex_automatichyphenpenalty:D .. 788
 - \tex_autospacing:D 1134
 - \tex_autoxspacing:D 1135
 - \tex_badness:D 162
 - \tex_baselineskip:D 163
 - \tex_batchmode:D 164, 9495
 - \tex_begincsname:D 789
 - \tex_begingroup:D 165, 1238, 1282, 1422
 - \tex_beginL:D 473
 - \tex_beginR:D 474
 - \tex_belowdisplayshortskip:D .. 166
 - \tex_belowdisplayskip:D 167
 - \tex_binoppenalty:D 168
 - \tex_bodydir:D 790, 1332
 - \tex_bodydirection:D 791
 - \tex_botmark:D 169
 - \tex_botmarks:D 475
 - \tex_boundary:D 792
 - \tex_box:D ... 170, 34661, 34663, 34706
 - \tex_boxdir:D 793
 - \tex_boxdirection:D 794
 - \tex_boxmaxdepth:D 171
 - \tex_breakafterdirmode:D 795
 - \tex_brokenpenalty:D 172
 - \tex_catcode:D 173, 2688,
6994, 8619, 8622, 12185, 19063, 19065
 - \tex_catcodetable:D 796, 30478, 30485
 - \tex_char:D 174
 - \tex_chardef:D 376,
175, 1415, 1444, 1446, 1798, 1799,
8286, 8308, 8313, 10253, 10504,
17586, 19599, 31707, 31709, 31711
 - \tex_cleaders:D 176
 - \tex_clearmarks:D 797
 - \tex_closein:D 177, 10286
 - \tex_closeout:D 178, 10530
 - \tex_clubpenalties:D 476
 - \tex_clubpenalty:D 179
 - \tex_compoundhyphenmode:D 799
 - \tex_copy:D 180, 34655,
34657, 34681, 34690, 34699, 34707
 - \tex_copyfont:D 631, 933
 - \tex_count:D 181, 3156, 3172,
3180, 3181, 10201, 10203, 10454, 10456
 - \tex_countdef:D 182
 - \tex_cr:D 183
 - \tex_crampeddisplaystyle:D 800
 - \tex_crampedscriptscriptstyle:D 802
 - \tex_crampedscriptstyle:D 803
 - \tex_crampedtextstyle:D 804
 - \tex_crcr:D 184
 - \tex_creationdate:D .. 632, 769, 9021

- `\tex_csname:D` 185, 1409
- `\tex_csstring:D` 805
- `\tex_currentcjktoken:D` ... 1136, 1200
- `\tex_currentgrouplevel:D`
..... 1263, 477, 20094, 20112,
30467, 30547, 30557, 30564, 37666
- `\tex_currentgroupstype:D` 478
- `\tex_currentifbranch:D` 479
- `\tex_currentiflevel:D` 480
- `\tex_currentifttype:D` 481
- `\tex_currentspacingmode:D` 1137
- `\tex_currentxspacingmode:D` ... 1138
- `\tex_day:D` 186, 1289, 1293
- `\tex_deadcycles:D` 187
- `\tex_def:D`
.. 188, 688, 689, 690, 1472, 1476,
1481, 1486, 39325, 39349, 39383, 39796
- `\tex_defaultthyphenchar:D` 189
- `\tex_defaultskewchar:D` 190
- `\tex_deferred:D` 806
- `\tex_delcode:D` 191
- `\tex_delimiter:D` 192
- `\tex_delimiterfactor:D` 193
- `\tex_delimitershortfall:D` 194
- `\tex_detokenize:D` 482, 1418, 1420
- `\tex_dimen:D` 195
- `\tex_dimendef:D` 196
- `\tex_dimexpr:D` 483, 20747, 34631
- `\tex_directlua:D` . 809, 1269, 1270,
8666, 9019, 9020, 9073, 11736, 11761
- `\tex_disablecjktoken:D` 1201
- `\tex_discretionary:D` 197
- `\tex_discretionaryligaturemode:D` 808
- `\tex_disinhibitglue:D` 1139
- `\tex_displayindent:D` 198
- `\tex_displaylimits:D` 199, 37311
- `\tex_displaystyle:D` 200
- `\tex_displaywidowpenalties:D` .. 484
- `\tex_displaywidowpenalty:D` 201
- `\tex_displaywidth:D` 202
- `\tex_divide:D` 203, 3150, 5270
- `\tex_doublehyphendemerits:D` ... 204
- `\tex_dp:D` 205, 34671
- `\tex_draftmode:D` 633, 934
- `\tex_dtou:D` 1140
- `\tex_dump:D` 206
- `\tex_dviextension:D` 810
- `\tex_dvifeedback:D` 811
- `\tex_dvivariable:D` 812
- `\tex_eachlinedepth:D` 634
- `\tex_eachlineheight:D` 635
- `\tex_edef:D` 207, 1239,
1240, 1256, 1283, 1284, 1289, 1290,
1295, 1296, 1301, 1302, 1473, 1474,
1478, 1483, 1488, 10763, 10821, 38786
- `\tex_efcode:D` 671
- `\tex_elapsedtime:D`
.... 636, 770, 9050, 9053, 9066, 9845
- `\tex_else:D` . 208, 1242, 1268, 1286,
1292, 1298, 1304, 1395, 1447, 1450
- `\tex_emergencystretch:D` 209
- `\tex_enablecjktoken:D` ... 1202, 8654
- `\tex_end:D` 210, 1226, 1315, 1967
- `\tex_endcsname:D` 211, 1410
- `\tex_endgroup:D`
..... 212, 1224, 1264, 1307, 1423
- `\tex_endinput:D` 213, 9504, 11427, 11661
- `\tex_endL:D` 485
- `\tex_endlinechar:D`
..... 120, 121, 134, 214, 9127,
10358, 10360, 10361, 12373, 12374,
12375, 12409, 30423, 30427, 30440,
30480, 30481, 30496, 30667, 30682,
30718, 39308, 39370, 39379, 39793
- `\tex_endlocalcontrol:D` 815
- `\tex_endR:D` 486
- `\tex_epTeXinputencoding:D` 1141
- `\tex_epTeXversion:D` . 1142, 8739, 8764
- `\tex_eqno:D` 215
- `\tex_errhelp:D` 216, 9372
- `\tex_errmessage:D` 217, 1959, 9392
- `\tex_errorcontextlines:D`
..... 218, 2325, 2332,
2333, 9387, 9406, 9593, 13262, 34770
- `\tex_errorstopmode:D` 219
- `\tex_escapechar:D` . 663, 220, 2310,
3637, 3699, 3700, 4044, 4142, 4143,
4146, 4174, 4273, 10380, 10625,
10672, 10678, 14720, 14769, 14822
- `\tex_escapehex:D` 637
- `\tex_escapename:D` 638
- `\tex_escapestring:D` 639
- `\tex_eTeXglueshrinkorder:D` 813
- `\tex_eTeXgluestretchorder:D` ... 814
- `\tex_eTeXrevision:D` 487
- `\tex_eTeXversion:D` 488
- `\tex_etoksapp:D` 816, 4295, 4296
- `\tex_etokspre:D` 817, 4289, 4290
- `\tex_euc:D` 1143
- `\tex_everycr:D` 221
- `\tex_everydisplay:D` 222, 1230
- `\tex_everyeof:D`
..... 489, 8892, 11077, 12382, 12434
- `\tex_everyhbox:D` 223
- `\tex_everyjob:D` 224, 1309, 1316
- `\tex_everymath:D` 225, 1231
- `\tex_everypar:D` 226

- `\tex_everyvbox:D` 227
- `\tex_exceptionpenalty:D` 818
- `\tex_exhyphenchar:D` 819
- `\tex_exhyphenpenalty:D` 228
- `\tex_expandafter:D`
229, 693, 1243, 1257, 1259, 1260, 1411
- `\tex_expanded:D` 423,
821, 1325, 1499, 2403, 2466, 2495,
2553, 2592, 2609, 2674, 2898, 4293,
4299, 12224, 12255, 12296, 12324,
12874, 20626, 21353, 21666, 30844
- `\tex_explicitdiscretionary:D` .. 822
- `\tex_explicitthyphenpenalty:D` .. 820
- `\tex_fam:D` 230
- `\tex_fi:D`
. 231, 694, 1228, 1237, 1261, 1263,
1272, 1273, 1274, 1276, 1277, 1281,
1288, 1294, 1300, 1306, 1310, 1313,
1326, 1334, 1339, 1396, 1452, 1453
- `\tex_filedump:D`
. 703, 771, 1382, 11287, 11300, 11307
- `\tex_filemoddate:D`
..... 702, 772, 1378, 11406
- `\tex_filesize:D`
... 668, 700, 773, 1365, 11096, 11307
- `\tex_finalhyphendemerits:D` 232
- `\tex_firstlineheight:D` 640
- `\tex_firstmark:D` 233
- `\tex_firstmarks:D` 490
- `\tex_firstvalidlanguage:D` 823
- `\tex_fixupboxesmode:D` 825
- `\tex_floatingpenalty:D` 234
- `\tex_font:D` 235, 23004
- `\tex_fontcharhp:D` 491
- `\tex_fontcharht:D` 492
- `\tex_fontcharic:D` 493
- `\tex_fontcharwd:D` 494
- `\tex_fontdimen:D` 236, 22996
- `\tex_fontexpand:D` 641, 935
- `\tex_fontid:D` 826
- `\tex_fontname:D` 237
- `\tex_fontsize:D` 642
- `\tex_forcecjktoken:D` 1203
- `\tex_formatname:D` 827
- `\tex_futurelet:D` . 238, 3645, 3703,
4148, 4161, 4222, 4245, 19713, 19715
- `\tex_gdef:D` 239, 1428, 1430,
1432, 1433, 1454, 1457, 1458, 1459,
1462, 1463, 1464, 1467, 1468, 1469
- `\tex_gleaders:D` 833
- `\tex_glet:D` 834
- `\tex_global:D` 945, 140, 144, 240, 695,
1259, 1287, 1293, 1299, 1305, 1392,
1393, 1394, 1395, 1396, 1397, 1398,
1399, 1400, 1401, 1402, 1403, 1404,
1405, 1406, 1407, 1408, 1409, 1410,
1411, 1412, 1413, 1414, 1415, 1416,
1417, 1418, 1419, 1420, 1421, 1422,
1423, 1425, 1426, 1427, 1443, 1444,
1446, 1449, 1451, 1454, 1455, 1456,
1461, 1466, 1471, 1472, 1473, 1474,
1479, 1484, 1489, 1798, 1799, 2052,
2059, 8286, 8313, 10253, 10504,
12148, 12160, 17564, 17570, 17574,
17587, 17590, 17593, 17604, 17615,
17617, 17627, 17629, 17637, 19336,
19338, 19348, 19599, 19715, 20761,
20766, 20782, 20789, 20795, 20804,
21190, 21210, 21215, 21220, 21226,
21281, 21287, 21303, 21308, 21313,
21319, 23004, 31707, 31708, 31709,
31710, 31711, 31712, 34657, 34663,
34736, 34789, 34801, 34814, 34834,
34881, 34893, 34905, 34918, 34939,
34954, 39324, 39348, 39382, 39796
- `\tex_globaldefs:D` 241
- `\tex_glueexpr:D` 495,
21208, 21210, 21218, 21220, 21224,
21226, 21240, 21247, 21252, 21255,
29107, 39862, 39905, 40027, 40029
- `\tex_glueshrink:D` 496
- `\tex_glueshrinkorder:D` 497
- `\tex_gluestretch:D` ... 498, 3862, 3868
- `\tex_gluestretchorder:D` 499
- `\tex_gluetomu:D` 500
- `\tex_glyphdimensionsmode:D` 835
- `\tex_gtoksapp:D` 836
- `\tex_gtokspre:D` 837
- `\tex_halign:D` 242
- `\tex_hangafter:D` 243
- `\tex_hangindent:D` 244
- `\tex_hbadness:D` 245
- `\tex_hbox:D` 246, 34781, 34784,
34789, 34796, 34801, 34808, 34814,
34828, 34834, 34842, 34847, 36408
- `\tex_hfi:D` 1144
- `\tex_hfil:D` 247
- `\tex_hfill:D` 248
- `\tex_hfilneg:D` 249
- `\tex_hfuzz:D` 250
- `\tex_hjcode:D` 828
- `\tex_hoffset:D` 251, 1328
- `\tex_holdinginserts:D` 252
- `\tex_hpack:D` 829
- `\tex_hrulerule:D` 253
- `\tex_hsize:D`
..... 254, 35554, 35579, 35580, 35630
- `\tex_hskip:D` 255, 21250

- \tex_hss:D
256, 34851, 34853, 34855, 35299, 35308
- \tex_ht:D 257, 34670
- \tex_hyphen:D 150, 1232
- \tex_hyphenation:D 258
- \tex_hyphenationbounds:D 830
- \tex_hyphenationmin:D 831
- \tex_hyphenchar:D 259, 22997
- \tex_hyphenpenalty:D 260
- \tex_hyphenpenaltymode:D 832
- \tex_if:D 261, 1398, 1399
- \tex_ifabsdim:D 624, 936
- \tex_ifabsnum:D
..... 1010, 625, 937, 23063, 23067
- \tex_ifcase:D 262, 17454
- \tex_ifcat:D 263, 1400
- \tex_ifcondition:D 838
- \tex_ifcsname:D 501, 1408
- \tex_ifdbox:D 1145
- \tex_ifddir:D 1146
- \tex_ifdefined:D
..... 502, 692, 1225, 1229, 1235,
1266, 1269, 1276, 1277, 1308, 1311,
1314, 1327, 1335, 1407, 1445, 1448
- \tex_ifdim:D 264, 20746
- \tex_ifeof:D 265, 10322
- \tex_iffalse:D 266, 1393
- \tex_iffontchar:D 503
- \tex_ifhbox:D 267, 34718
- \tex_ifhmode:D 268, 1404
- \tex_ifincsname:D 672
- \tex_ifinner:D 269, 1406
- \tex_ifjfont:D 1147
- \tex_ifmbox:D 1148
- \tex_ifmdir:D 1149
- \tex_ifmmode:D 270, 1403
- \tex_ifnum:D 271, 1275, 1425
- \tex_ifodd:D .. 272, 1402, 8279, 17453
- \tex_ifprimitive:D 626, 775
- \tex_iftbox:D 1150
- \tex_iftdir:D 1152
- \tex_iftfont:D 1151
- \tex_iftrue:D 273, 1392
- \tex_ifvbox:D 274, 34719
- \tex_ifvmode:D 275, 1405
- \tex_ifvoid:D 276, 34720
- \tex_ifx:D 277, 1241,
1258, 1285, 1291, 1297, 1303, 1401
- \tex_ifybox:D 1153
- \tex_ifydir:D 1154
- \tex_ignoreddimen:D 643
- \tex_ignoreligaturesinfont:D .. 938
- \tex_ignorespaces:D 278
- \tex_immediate:D
. 279, 1976, 1978, 10506, 10530, 10589
- \tex_immediateassigned:D 839
- \tex_immediateassignment:D 840
- \tex_indent:D 280, 2386
- \tex_inhibitglue:D 1155
- \tex_inhibitxspcode:D 1156
- \tex_initcatcodetable:D .. 841, 30384
- \tex_input:D 281,
1227, 1317, 8897, 11083, 11446, 11492
- \tex_inputlineno:D ... 282, 1974, 9310
- \tex_insert:D 283
- \tex_insertht:D 644, 939
- \tex_insertpenalties:D 284
- \tex_interactionmode:D 504,
2323, 2328, 2329, 34754, 34757, 34759
- \tex_interlinepenalties:D 505
- \tex_interlinepenalty:D 285
- \tex_italiccorrection:D
..... 149, 1233, 1329
- \tex_jcharwidowpenalty:D 1157
- \tex_jfam:D 1158
- \tex_jfont:D 1159
- \tex_jis:D 1160
- \tex_jobname:D
..... 286, 8982, 9164, 10941, 10942
- \tex_kanjiskip:D 1161, 8652
- \tex_kansuji:D 1162
- \tex_kansujichar:D 1163
- \tex_kcatcode:D 1164
- \tex_kchar:D 1204
- \tex_kchardef:D 1205
- \tex_kern:D 287, 34635
- \tex_knaccode:D 673
- \tex_knbccode:D 674
- \tex_knbscode:D 675
- \tex_kuten:D 1165, 1206
- \tex_language:D 288, 1318
- \tex_lastbox:D 289, 34734, 34736
- \tex_lastkern:D 290
- \tex_lastlinedepth:D 645
- \tex_lastlinefit:D 506
- \tex_lastmatch:D 646
- \tex_lastnamedcs:D
842, 1838, 1850, 1880, 1894, 1920, 1930
- \tex_lastnodechar:D 1166
- \tex_lastnodefont:D 1167
- \tex_lastnodesubtype:D 1168
- \tex_lastnodetype:D 507
- \tex_lastpenalty:D 291
- \tex_lastskip:D 292
- \tex_lastxpos:D 647, 946
- \tex_lastypos:D 648, 947
- \tex_latelua:D 843, 11762

- `\tex_lateluafunction:D` 844
- `\tex_lccode:D` 293, 3594, 3604,
3614, 3626, 3697, 3699, 3702, 3732,
4110, 7161, 7213, 9139, 9151, 13735,
13736, 13758, 13759, 19139, 19141
- `\tex_leaders:D` 294
- `\tex_left:D` 295, 1336
- `\tex_leftghost:D` 845
- `\tex_lefthyphenmin:D` 296
- `\tex_leftmarginkern:D` 676
- `\tex_leftskip:D` 297
- `\tex_leqno:D` 298
- `\tex_let:D` 1465, 141,
144, 299, 695, 1226, 1227, 1230,
1231, 1232, 1233, 1234, 1236, 1259,
1265, 1267, 1271, 1279, 1280, 1287,
1293, 1299, 1305, 1309, 1312, 1315,
1316, 1317, 1318, 1319, 1320, 1321,
1322, 1323, 1324, 1325, 1328, 1329,
1330, 1331, 1332, 1333, 1336, 1337,
1338, 1392, 1393, 1394, 1395, 1396,
1397, 1398, 1399, 1400, 1401, 1402,
1403, 1404, 1405, 1406, 1407, 1408,
1409, 1410, 1411, 1412, 1413, 1414,
1416, 1417, 1418, 1419, 1420, 1421,
1422, 1423, 1425, 1426, 1427, 1443,
1454, 1455, 1456, 1461, 1466, 1471,
1472, 1473, 1474, 1479, 1484, 1489,
2048, 3595, 3605, 3616, 3628, 4070,
4139, 12146, 12148, 12158, 12160,
19336, 19338, 19348, 38750, 38753
- `\tex_letcharcode:D` 846
- `\tex_letterspacefont:D` 677
- `\tex_limits:D` 300, 37309
- `\tex_linedir:D` 847
- `\tex_linedirection:D` 848
- `\tex_lineendmode:D` 1175
- `\tex_linepenalty:D` 301
- `\tex_lineskip:D` 302
- `\tex_lineskiplimit:D` 303
- `\tex_localbrokenpenalty:D` 849
- `\tex_localinterlinepenalty:D` 850
- `\tex_localleftbox:D` 855
- `\tex_localrightbox:D` 856
- `\tex_long:D` 304, 688, 689,
690, 1428, 1430, 1433, 1457, 1458,
1459, 1460, 1462, 1464, 1467, 1468,
1469, 1470, 1476, 1478, 1486, 1488
- `\tex_looseness:D` 305
- `\tex_lower:D` 306, 34717
- `\tex_lowercase:D` 895, 896,
307, 3595, 3605, 3615, 3627, 3698,
4111, 7162, 7214, 9140, 9152, 9379,
13737, 13760, 19170, 19253, 19280
- `\tex_lpcode:D` 678
- `\tex_luabytecode:D` 851
- `\tex_luabytecodecall:D` 852
- `\tex_luacopyinputnodes:D` 853
- `\tex_luadef:D` 854
- `\tex_luaescapestring:D` 857, 11760
- `\tex_luafunction:D` 858
- `\tex_luafunctioncall:D` 859
- `\tex luatexbanner:D` 860
- `\tex luatexrevision:D` 861, 8748
- `\tex luatexversion:D`
. 862, 1277, 1445, 8650,
8744, 8746, 10443, 14042, 17581, 18269
- `\tex_mag:D` 308, 38713
- `\tex_mark:D` 309
- `\tex_marks:D` 508
- `\tex_match:D` 649
- `\tex_mathaccent:D` 310
- `\tex_mathbin:D` 311
- `\tex_mathchar:D` 312
- `\tex_mathchardef:D` 376,
313, 1451, 17589, 31708, 31710, 31712
- `\tex_mathchoice:D` 314
- `\tex_mathclose:D` 315
- `\tex_mathcode:D` 316, 19133, 19135
- `\tex_mathdefaultsmode:D` 863
- `\tex_mathdelimitersmode:D` 864
- `\tex_mathdir:D` 865
- `\tex_mathdirection:D` 866
- `\tex_mathdisplayskipmode:D` 867
- `\tex_matheqdirmode:D` 868
- `\tex_matheqnogapstep:D` 869
- `\tex_mathflattenmode:D` 870
- `\tex_mathinner:D` 317
- `\tex_mathitalicsmode:D` 871
- `\tex_mathnolimitsmode:D` 872
- `\tex_mathop:D` 318, 1319
- `\tex_mathopen:D` 319
- `\tex_mathoption:D` 873
- `\tex_mathord:D` 320
- `\tex_mathpenaltiesmode:D` 874
- `\tex_mathpunct:D` 321
- `\tex_mathrel:D` 322
- `\tex_mathrulesfam:D` 875
- `\tex_mathrulesmode:D` 877
- `\tex_mathrulethicknessmode:D` 879
- `\tex_mathscriptboxmode:D` 881
- `\tex_mathscriptcharmode:D` 882
- `\tex_mathscriptsmode:D` 880
- `\tex_mathstyle:D` 883
- `\tex_mathsurround:D` 323
- `\tex_mathsurroundmode:D` 884
- `\tex_mathsurroundskip:D` 885
- `\tex_maxdeadcycles:D` 324

<code>\tex_maxdepth:D</code>	325	<code>\tex_openout:D</code>	350, 10506
<code>\tex_mdffivesum:D</code>		<code>\tex_or:D</code>	351, 1394
. 701, 774, 1369, 11242, 14133, 14134		<code>\tex_oradical:D</code>	1215
<code>\tex_meaning:D</code> ..	326, 1240, 1257,	<code>\tex_outer:D</code>	352, 1321, 38786
1283, 1289, 1295, 1301, 1416, 1417		<code>\tex_output:D</code>	353
<code>\tex_medmskip:D</code>	327	<code>\tex_outputbox:D</code>	891
<code>\tex_message:D</code>	328	<code>\tex_outputpenalty:D</code>	354
<code>\tex_middle:D</code>	509, 1337	<code>\tex_over:D</code>	355, 1322
<code>\tex_mkern:D</code>	329	<code>\tex_overfullrule:D</code>	356
<code>\tex_month:D</code> ...	330, 1295, 1299, 1320	<code>\tex_overline:D</code>	357
<code>\tex_moveleft:D</code>	331, 34711	<code>\tex_overwithdelims:D</code>	358
<code>\tex_moveright:D</code>	332, 34713	<code>\tex_pagebottomoffset:D</code>	892
<code>\tex_mskip:D</code>	333	<code>\tex_pagedepth:D</code>	359
<code>\tex_muexpr:D</code>		<code>\tex_pagedir:D</code>	893, 1333
. 510, 21301, 21303, 21311, 21313,		<code>\tex_pagedirection:D</code>	894
21317, 21319, 21323, 39851, 39910		<code>\tex_pagediscards:D</code>	513
<code>\tex_multiply:D</code>	334	<code>\tex_pagefillllstretch:D</code>	360
<code>\tex_muskip:D</code>	335	<code>\tex_pagefillstretch:D</code>	361
<code>\tex_muskipdef:D</code>	336	<code>\tex_pagefilstretch:D</code>	362
<code>\tex_mutogluue:D</code>		<code>\tex_pagefistretch:D</code>	1171
. 365, 1479, 511, 39851, 39910		<code>\tex_pagegoal:D</code>	363
<code>\tex_newlinechar:D</code>		<code>\tex_pageheight:D</code>	652, 950
. 337, 1958, 9385, 9591,		<code>\tex_pageleftoffset:D</code>	895
10588, 12375, 12401, 12405, 13260		<code>\tex_pagerightoffset:D</code>	896
<code>\tex_noalign:D</code>	338	<code>\tex_pageshrink:D</code>	364
<code>\tex_noautospadding:D</code>	1169	<code>\tex_pagestretch:D</code>	365
<code>\tex_noautoxspacing:D</code>	1170	<code>\tex_pagetopoffset:D</code>	897
<code>\tex_noboundary:D</code>	339	<code>\tex_pagetotal:D</code>	366
<code>\tex_noexpand:D</code>	340, 1412	<code>\tex_pagewidth:D</code>	653, 951
<code>\tex_nohrule:D</code>	886	<code>\tex_par:D</code>	367
<code>\tex_noindent:D</code>	341	<code>\tex_pardir:D</code>	898
<code>\tex_nokerns:D</code>	887	<code>\tex_pardirection:D</code>	899
<code>\tex_noligatures:D</code>	650	<code>\tex_parfillskip:D</code>	368
<code>\tex_noligs:D</code>	888	<code>\tex_parindent:D</code>	369
<code>\tex_nolimits:D</code>	342, 37310	<code>\tex_parshape:D</code>	370
<code>\tex_nonscript:D</code>	343	<code>\tex_parshapedimen:D</code>	514
<code>\tex_nonstopmode:D</code>	344	<code>\tex_parshapeindent:D</code>	515
<code>\tex_normaldeviate:D</code>	651, 948	<code>\tex_parshapelength:D</code>	516
<code>\tex_nospaces:D</code>	889	<code>\tex_parskip:D</code>	371
<code>\tex_novrule:D</code>	890	<code>\tex_partokencontext:D</code>	1216
<code>\tex_nulldelimiterspace:D</code>	345	<code>\tex_partokenname:D</code>	1217
<code>\tex_nullfont:D</code>	346, 19628	<code>\tex_patterns:D</code>	372
<code>\tex_number:D</code>	347, 17450, 35457	<code>\tex_pausing:D</code>	373
<code>\tex_numexpr:D</code>		<code>\tex_pdfannot:D</code>	539
. 512, 4271, 16873, 17451, 19267, 23203		<code>\tex_pdfcatalog:D</code>	540
<code>\tex_odelcode:D</code>	1209	<code>\tex_pdfcolorstack:D</code>	542
<code>\tex_odelimiter:D</code>	1210	<code>\tex_pdfcolorstackinit:D</code>	543
<code>\tex_omathaccent:D</code>	1211	<code>\tex_pdfcompresslevel:D</code>	541
<code>\tex_omathchar:D</code>	1212	<code>\tex_pdfdecimaldigits:D</code>	544
<code>\tex_omathchardef:D</code>		<code>\tex_pdfdest:D</code>	545
. 1213, 1448, 1449, 17582, 17584, 17585		<code>\tex_pdfdestmargin:D</code>	546
<code>\tex_omathcode:D</code>	1214	<code>\tex_pdfendlink:D</code>	547
<code>\tex_omit:D</code>	348	<code>\tex_pdfendthread:D</code>	548
<code>\tex_openin:D</code>	349, 10255	<code>\tex_pdfextension:D</code>	900

<code>\tex_pdffakespace:D</code>	549	<code>\tex_pdfsetmatrix:D</code>	604
<code>\tex_pdffeedback:D</code>	901	<code>\tex_pdfstartlink:D</code>	605
<code>\tex_pdffontattr:D</code>	550	<code>\tex_pdfstartthread:D</code>	606
<code>\tex_pdffontname:D</code>	551	<code>\tex_pdfsuppressptexinfo:D</code>	607
<code>\tex_pdffontobjnum:D</code>	552	<code>\tex_pdfsuppresswarningdupdest:D</code>	609
<code>\tex_pdfgamma:D</code>	553	<code>\tex_pdfsuppresswarningdupmap:D</code>	611
<code>\tex_pdfgentounicode:D</code>	554	<code>\tex_pdfsuppresswarningpagegroup:D</code>	613
<code>\tex_pdfglyphtounicode:D</code>	555	613
<code>\tex_pdfhorigin:D</code>	556	<code>\tex_pdftexbanner:D</code>	668
<code>\tex_pdfimageapplygamma:D</code>	557	<code>\tex_pdftexrevision:D</code>	669, 8727
<code>\tex_pdfimagegamma:D</code>	558	<code>\tex_pdftexversion:D</code>	
<code>\tex_pdfimagehicolor:D</code>	559	..	670, 1276, 8651, 8723, 8725, 14051
<code>\tex_pdfimageresolution:D</code>	560	<code>\tex_pdfthread:D</code>	614
<code>\tex_pdfincludechars:D</code>	561	<code>\tex_pdfthreadmargin:D</code>	615
<code>\tex_pdfinclusioncopyfonts:D</code> ..	562	<code>\tex_pdftrailer:D</code>	616
<code>\tex_pdfinclusionerrorlevel:D</code> ..	564	<code>\tex_pdftrailerid:D</code>	617
<code>\tex_pdfinfo:D</code>	565	<code>\tex_pdfuniquefilename:D</code>	618
<code>\tex_pdfinfoomitdate:D</code>	566	<code>\tex_pdfvariable:D</code>	902
<code>\tex_pdfinterwordspaceoff:D</code> ...	567	<code>\tex_pdfvorigin:D</code>	619
<code>\tex_pdfinterwordspaceon:D</code>	568	<code>\tex_pdfxform:D</code>	620, 958
<code>\tex_pdflastannot:D</code>	569	<code>\tex_pdfxformname:D</code>	621
<code>\tex_pdflastlink:D</code>	570	<code>\tex_pdfximage:D</code>	622, 959
<code>\tex_pdflastobj:D</code>	571	<code>\tex_pdfximagebbox:D</code>	623
<code>\tex_pdflastxform:D</code>	572, 941	<code>\tex_penalty:D</code>	374
<code>\tex_pdflastximage:D</code>	573, 943	<code>\tex_pkmode:D</code>	654
<code>\tex_pdflastximagecolordepth:D</code> .	575	<code>\tex_pkreolution:D</code>	655
<code>\tex_pdflastximagepages:D</code> ..	576, 945	<code>\tex_postbreakpenalty:D</code>	1172
<code>\tex_pdflinkmargin:D</code>	577	<code>\tex_postdisplaypenalty:D</code>	375
<code>\tex_pdfliteral:D</code>	578	<code>\tex_postexhyphenchar:D</code>	903
<code>\tex_pdfmajorversion:D</code>	581	<code>\tex_postthyphenchar:D</code>	904
<code>\tex_pdfmapfile:D</code>	579, 1279	<code>\tex_prebinoppenalty:D</code>	905
<code>\tex_pdfmapline:D</code>	580, 1280	<code>\tex_prebreakpenalty:D</code>	1173
<code>\tex_pdfminorversion:D</code>	582	<code>\tex_predisplaydirection:D</code>	517
<code>\tex_pdfnames:D</code>	583	<code>\tex_predisplaygapfactor:D</code>	906
<code>\tex_pdfnobuiltintounicode:D</code> ..	584	<code>\tex_predisplaypenalty:D</code>	376
<code>\tex_pdfobj:D</code>	585	<code>\tex_predisplaysize:D</code>	377
<code>\tex_pdfobjcompresslevel:D</code>	586	<code>\tex_preexhyphenchar:D</code>	907
<code>\tex_pdfomitchar:D</code>	587	<code>\tex_prehyphenchar:D</code>	908
<code>\tex_pdfoutline:D</code>	588	<code>\tex_prependkern:D</code>	657
<code>\tex_pdfoutput:D</code>	599,	<code>\tex_prerelpenalty:D</code>	909
589, 949, 1312, 8694, 8700, 8708, 9179		<code>\tex_pretolerance:D</code>	378
<code>\tex_pdfpageattr:D</code>	590	<code>\tex_prevdepth:D</code>	379
<code>\tex_pdfpagebox:D</code>	592	<code>\tex_prevgraf:D</code>	380
<code>\tex_pdfpageref:D</code>	593	<code>\tex_primitive:D</code> .	656, 776, 8991, 9001
<code>\tex_pdfpageresources:D</code>	594	<code>\tex_protected:D</code>	
<code>\tex_pdfpagesattr:D</code>	591, 595	518, 1457, 1459, 1462,
<code>\tex_pdfrefobj:D</code>	596	1463, 1464, 1465, 1467, 1468, 1469,	
<code>\tex_pdfrefxform:D</code>	597, 955	1470, 1481, 1483, 1486, 1488, 38786	
<code>\tex_pdfrefximage:D</code>	598, 956	<code>\tex_protrudechars:D</code> ..	658, 779, 952
<code>\tex_pdfrestore:D</code>	599	<code>\tex_protrusionboundary:D</code>	910
<code>\tex_pdfretval:D</code>	600	<code>\tex_ptexfontname:D</code>	1174
<code>\tex_pdfrunninglinkoff:D</code>	601	<code>\tex_ptexminorversion:D</code>	
<code>\tex_pdfrunninglinkon:D</code>	602	1176, 8736, 8757
<code>\tex_pdfsave:D</code>	603	<code>\tex_ptexrevision:D</code> .	1177, 8737, 8758

- \tex_ptextracingfonts:D 1178
- \tex_ptexversion:D
..... 1179, 8731, 8734, 8752, 8755
- \tex_pxdimen:D 659, 953
- \tex_quitvmode:D 679
- \tex_radical:D 381
- \tex_raise:D 382, 34715
- \tex_randomseed:D 660, 954, 9026
- \tex_read:D 383, 9496, 10342
- \tex_readline:D 519, 10359
- \tex_readpapersizespecial:D .. 1180
- \tex_relax:D 365,
1020, 1479, 384, 1421, 17452, 20748
- \tex_relpenny:D 385
- \tex_resettimer:D 661, 777
- \tex_right:D 386, 1338
- \tex_rightghost:D 911
- \tex_rightthyphenmin:D 387
- \tex_rightmarginkern:D 680
- \tex_rightskip:D 388
- \tex_romannumeral:D 388,
415, 416, 1479, 389, 1414, 1426,
1803, 19182, 23205, 29902, 29919,
29921, 30467, 30945, 30999, 39199
- \tex_rrcode:D 681
- \tex_savecatcodetable:D
..... 912, 30422, 30484
- \tex_savepos:D 662, 957
- \tex_savinghyphcodes:D 520
- \tex_savingvdiscards:D 521
- \tex_scantextokens:D 913
- \tex_scantokens:D 522, 12387,
12448, 39322, 39346, 39380, 39794
- \tex_scriptbaselineshiftfactor:D
..... 1182
- \tex_scriptfont:D 390
- \tex_scriptscriptbaselineshiftfactor:D
..... 1184
- \tex_scriptscriptfont:D 391
- \tex_scriptscriptstyle:D 392
- \tex_scriptspace:D 393
- \tex_scriptstyle:D 394
- \tex_scrollmode:D 395
- \tex_setbox:D 396,
34655, 34657, 34661, 34663, 34681,
34690, 34699, 34734, 34736, 34784,
34789, 34796, 34801, 34808, 34814,
34828, 34834, 34876, 34881, 34888,
34893, 34900, 34905, 34912, 34918,
34933, 34939, 34950, 34954, 36408
- \tex_setfontid:D 914
- \tex_setlanguage:D 397
- \tex_setrandomseed:D . 663, 960, 9031
- \tex_sfcode:D 398, 19151, 19153
- \tex_shapemode:D 915
- \tex_shbscode:D 682
- \tex_shellescape:D ... 664, 778, 9076
- \tex_shipout:D 399, 1236, 1260
- \tex_show:D 400
- \tex_showbox:D 401, 34771
- \tex_showboxbreadth:D ... 402, 34767
- \tex_showboxdepth:D 403, 34768
- \tex_showgroups:D 523, 2327
- \tex_showifs:D 524
- \tex_showlists:D 404
- \tex_showmode:D 1185
- \tex_showstream:D 1218
- \tex_showthe:D 405
- \tex_showtokens:D
..... 728, 525, 1331, 9595, 13264
- \tex_sjis:D 1186
- \tex_skewchar:D 406
- \tex_skip:D
407, 3735, 3764, 3783, 3846, 3862, 3868
- \tex_skipdef:D 408
- \tex_space:D 148
- \tex_spacefactor:D 409
- \tex_spaceskip:D 410
- \tex_span:D 411
- \tex_special:D 412
- \tex_splitbotmark:D 413
- \tex_splitbotmarks:D 526
- \tex_splitdiscards:D 527
- \tex_splitfirstmark:D 414
- \tex_splitfirstmarks:D 528
- \tex_splitmaxdepth:D 415
- \tex_splittopskip:D 416
- \tex_stbscode:D 683
- \tex_strcmp:D
..... 699, 1342, 11363, 13538, 23576
- \tex_string:D 417, 1239,
1243, 1284, 1290, 1296, 1302, 1419
- \tex_suppressfontnotfounderror:D 705
- \tex_suppressifcsnameerror:D .. 916
- \tex_suppresslongerror:D 917
- \tex_suppressmathparerror:D ... 918
- \tex_suppressoutererror:D 919
- \tex_suppressprimitiveerror:D .. 921
- \tex_synctex:D 684
- \tex_tabskip:D 418
- \tex_tagcode:D 685
- \tex_tate:D 1187
- \tex_tbaselineshift:D 1188
- \tex_textbaselineshiftfactor:D 1190
- \tex_textdir:D 922
- \tex_textdirection:D 923
- \tex_textfont:D 419
- \tex_textstyle:D 420

<code>\tex_TeXeTstate:D</code>	529	<code>\tex_uchyph:D</code>	440
<code>\tex_tfont:D</code>	1191	<code>\tex_ucs:D</code>	1194
<code>\tex_the:D</code>		<code>\tex_Udelcode:D</code>	965
. 365, 409, 1057, 1063, 1064, 121,		<code>\tex_Udelcodenum:D</code>	966
421, 1974, 2296, 2438, 2442, 3236,		<code>\tex_Udelimiter:D</code>	967
3268, 3317, 3318, 3349, 3350, 3356,		<code>\tex_Udelimiterover:D</code>	968
3357, 3861, 3975, 4274, 4293, 4299,		<code>\tex_Udelimiterunder:D</code>	969
4305, 4313, 6242, 6244, 6258, 6259,		<code>\tex_Uhextensible:D</code>	970
6261, 6262, 6494, 6537, 6759, 6858,		<code>\tex_Uleft:D</code>	971
9026, 16988, 17463, 17464, 17640,		<code>\tex_Umathaccent:D</code>	972
17641, 19065, 19135, 19141, 19147,		<code>\tex_Umathaxis:D</code>	973
19153, 20978, 20979, 20980, 21050,		<code>\tex_Umathbinbinspacing:D</code>	974
21051, 24196, 24684, 34754, 39188		<code>\tex_Umathbinclonespacing:D</code>	975
<code>\tex_thickmuskip:D</code>	422	<code>\tex_Umathbininnerspacing:D</code>	976
<code>\tex_thinmuskip:D</code>	423	<code>\tex_Umathbinopenspacing:D</code>	977
<code>\tex_time:D</code>	424, 1283, 1287	<code>\tex_Umathbinopspacing:D</code>	978
<code>\tex_tojis:D</code>	1192	<code>\tex_Umathbinordspacing:D</code>	979
<code>\tex_toks:D</code>		<code>\tex_Umathbinpunctspacing:D</code>	980
.... 425, 3209, 3236, 3268, 3306,		<code>\tex_Umathbinrelspacing:D</code>	981
3317, 3318, 3349, 3350, 3356, 3357,		<code>\tex_Umathchar:D</code>	982
3361, 3371, 3381, 3680, 3698, 3861,		<code>\tex_Umathcharclass:D</code>	983
4274, 4276, 4277, 4279, 4284, 4293,		<code>\tex_Umathchardef:D</code>	984
4299, 4305, 16988, 17000, 17001, 17002		<code>\tex_Umathcharfam:D</code>	985
<code>\tex_toksapp:D</code>	924, 4301, 4302	<code>\tex_Umathcharnum:D</code>	986
<code>\tex_toksdef:D</code>	426, 3488	<code>\tex_Umathcharnumdef:D</code>	987
<code>\tex_tokspre:D</code>	925	<code>\tex_Umathcharslot:D</code>	988
<code>\tex_tolerance:D</code>	427	<code>\tex_Umathclosebinspacing:D</code>	989
<code>\tex_topmark:D</code>	428	<code>\tex_Umathcloseclonespacing:D</code>	991
<code>\tex_topmarks:D</code>	530	<code>\tex_Umathcloseinnerspacing:D</code>	993
<code>\tex_topskip:D</code>	429	<code>\tex_Umathcloseopenspacing:D</code>	994
<code>\tex_toucs:D</code>	1193	<code>\tex_Umathcloseopspacing:D</code>	995
<code>\tex_tpack:D</code>	926	<code>\tex_Umathcloseordspacing:D</code>	996
<code>\tex_tracingassigns:D</code>	531	<code>\tex_Umathclosepunctspacing:D</code>	998
<code>\tex_tracingcommands:D</code>	430	<code>\tex_Umathcloserelspacing:D</code>	999
<code>\tex_tracingfonts:D</code>		<code>\tex_Umathcode:D</code>	1000
..... 665, 961, 1265, 1267, 1271		<code>\tex_Umathcodenum:D</code>	1001
<code>\tex_tracinggroups:D</code>	532	<code>\tex_Umathconnectoroverlapmin:D</code>	1003
<code>\tex_tracingifs:D</code>	533	<code>\tex_Umathfractiondelsize:D</code>	1004
<code>\tex_tracinglostchars:D</code>	431	<code>\tex_Umathfractiondenomdown:D</code>	1006
<code>\tex_tracingmacros:D</code>	432	<code>\tex_Umathfractiondenomvgap:D</code>	1008
<code>\tex_tracingnesting:D</code>		<code>\tex_Umathfractionnumup:D</code>	1009
..... 534, 8891, 11076, 12372		<code>\tex_Umathfractionnumvgap:D</code>	1010
<code>\tex_tracingonline:D</code>		<code>\tex_Umathfractionrule:D</code>	1011
..... 433, 2324, 2330, 2331, 34769		<code>\tex_Umathinnerbinspacing:D</code>	1012
<code>\tex_tracingoutput:D</code>	434	<code>\tex_Umathinnerclonespacing:D</code>	1014
<code>\tex_tracingpages:D</code>	435	<code>\tex_Umathinnerinnerspacing:D</code>	1016
<code>\tex_tracingparagraphs:D</code>	436	<code>\tex_Umathinneropenspacing:D</code>	1017
<code>\tex_tracingrestores:D</code>	437	<code>\tex_Umathinneropspacing:D</code>	1018
<code>\tex_tracingscantokens:D</code>	535	<code>\tex_Umathinnerordspacing:D</code>	1019
<code>\tex_tracingstacklevels:D</code>	1219	<code>\tex_Umathinnerpunctspacing:D</code>	1021
<code>\tex_tracingstats:D</code>	438	<code>\tex_Umathinnerrelspacing:D</code>	1022
<code>\tex_uccode:D</code>	439, 19145, 19147	<code>\tex_Umathlimitabovebgap:D</code>	1023
<code>\tex_Uchar:D</code>	963	<code>\tex_Umathlimitabovekern:D</code>	1024
<code>\tex_Ucharcat:D</code> 964, <u>1354</u> , 19218, 19223		<code>\tex_Umathlimitabovevgap:D</code>	1025

<code>\tex_Umathlimitbelowbgap:D</code> . . .	1026	<code>\tex_Umathreltopspacing:D</code>	1088
<code>\tex_Umathlimitbelowkern:D</code> . . .	1027	<code>\tex_Umathreltopspacing:D</code>	1089
<code>\tex_Umathlimitbelowvgap:D</code> . . .	1028	<code>\tex_Umathrelpunctspacing:D</code> . .	1090
<code>\tex_Umathnolimitsubfactor:D</code> . .	1029	<code>\tex_Umathrelrelspacing:D</code>	1091
<code>\tex_Umathnolimitsupfactor:D</code> . .	1030	<code>\tex_Umathskewedfractionhgap:D</code>	1093
<code>\tex_Umathopbinspacing:D</code>	1031	<code>\tex_Umathskewedfractionvgap:D</code>	1095
<code>\tex_Umathopclonespacing:D</code>	1032	<code>\tex_Umathspaceafterscript:D</code> . .	1096
<code>\tex_Umathopenbinspacing:D</code>	1033	<code>\tex_Umathstackdenomdown:D</code> . . .	1097
<code>\tex_Umathopenclospacespacing:D</code> . .	1034	<code>\tex_Umathstacknumup:D</code>	1098
<code>\tex_Umathopeninnerspacing:D</code> . . .	1035	<code>\tex_Umathstackvgap:D</code>	1099
<code>\tex_Umathopenopenspacing:D</code> . . .	1036	<code>\tex_Umathsubshiftdown:D</code>	1100
<code>\tex_Umathopenopspacing:D</code>	1037	<code>\tex_Umathsubshiftdrop:D</code>	1101
<code>\tex_Umathopenordspacing:D</code>	1038	<code>\tex_Umathsubsupshiftdown:D</code> . .	1102
<code>\tex_Umathopenpunctspacing:D</code> . . .	1039	<code>\tex_Umathsubsupvgap:D</code>	1103
<code>\tex_Umathopenrelspacing:D</code>	1040	<code>\tex_Umathsubtopmax:D</code>	1104
<code>\tex_Umathoperatorbottomsize:D</code>	1041	<code>\tex_Umathsupbottommin:D</code>	1105
<code>\tex_Umathopinnerspacing:D</code>	1042	<code>\tex_Umathsupshiftdrop:D</code>	1106
<code>\tex_Umathopopenspacing:D</code>	1043	<code>\tex_Umathsupshiftdown:D</code>	1107
<code>\tex_Umathopopspacing:D</code>	1044	<code>\tex_Umathsupsubbottommax:D</code> . .	1108
<code>\tex_Umathopordspacing:D</code>	1045	<code>\tex_Umathunderbarkern:D</code>	1109
<code>\tex_Umathoppunctspacing:D</code>	1046	<code>\tex_Umathunderbarrule:D</code>	1110
<code>\tex_Umathoprelspacing:D</code>	1047	<code>\tex_Umathunderbarvgap:D</code>	1111
<code>\tex_Umathordbinspacing:D</code>	1048	<code>\tex_Umathunderdelimitervgap:D</code>	1113
<code>\tex_Umathordclonespacing:D</code>	1049	<code>\tex_Umathunderdelimitervgap:D</code>	1115
<code>\tex_Umathordinnerspacing:D</code>	1050	<code>\tex_Umiddle:D</code>	1116
<code>\tex_Umathordopspacing:D</code>	1051	<code>\tex_undefine:D</code>	29972
<code>\tex_Umathordopspacing:D</code>	1052	<code>\tex_undefined:D</code>	
<code>\tex_Umathordordspacing:D</code>	1053 465, 904, 905, 975, 1265,	
<code>\tex_Umathordpunctspacing:D</code>	1054	1279, 1280, 1287, 1293, 1299, 1305,	
<code>\tex_Umathordrelspacing:D</code>	1055	2065, 3159, 3595, 3605, 3615, 3616,	
<code>\tex_Umathoverbarkern:D</code>	1056	3627, 3628, 3707, 3808, 4109, 18301,	
<code>\tex_Umathoverbarrule:D</code>	1057	20025, 20325, 21614, 29745, 29762	
<code>\tex_Umathoverbarvgap:D</code>	1058	<code>\tex_underline:D</code>	441, 1234
<code>\tex_Umathoverdelimitervgap:D</code> . .	1060	<code>\tex_unescapehex:D</code>	666
<code>\tex_Umathoverdelimitervgap:D</code> . .	1062	<code>\tex_unexpanded:D</code>	
<code>\tex_Umathpunctbinspacing:D</code>	1063 415, 536, 1324, 1413, 2670	
<code>\tex_Umathpunctclonespacing:D</code>	1065	<code>\tex_unhbox:D</code>	442, 34857
<code>\tex_Umathpunctinnerspacing:D</code>	1067	<code>\tex_unhcopy:D</code>	443, 34856
<code>\tex_Umathpunctopspacing:D</code>	1068	<code>\tex_uniformdeviate:D</code>	
<code>\tex_Umathpunctopspacing:D</code>	1069 827, 1220, 1221, 667,	
<code>\tex_Umathpunctordspacing:D</code>	1070	962, 16999, 29176, 29177, 29358, 29361	
<code>\tex_Umathpunctpunctspacing:D</code> . . .	1072	<code>\tex_unkern:D</code>	444
<code>\tex_Umathpunctrelspacing:D</code>	1073	<code>\tex_unless:D</code>	537, 1397
<code>\tex_Umathquad:D</code>	1074	<code>\tex_Unosubscript:D</code>	1117
<code>\tex_Umathradicaldegreeafter:D</code>	1076	<code>\tex_Unosuperscript:D</code>	1118
<code>\tex_Umathradicaldegreebefore:D</code>	1078	<code>\tex_unpenalty:D</code>	445
<code>\tex_Umathradicaldegreeraise:D</code>	1080	<code>\tex_unskip:D</code>	446
<code>\tex_Umathradicalkern:D</code>	1081	<code>\tex_unvbox:D</code>	447, 34946
<code>\tex_Umathradicalrule:D</code>	1082	<code>\tex_unvcopy:D</code>	448, 34945
<code>\tex_Umathradicalvgap:D</code>	1083	<code>\tex_Uoverdelimiter:D</code>	1119
<code>\tex_Umathrelbinspacing:D</code>	1084	<code>\tex_uppercase:D</code>	449
<code>\tex_Umathrelclonespacing:D</code>	1085	<code>\tex_uptexrevision:D</code>	1207, 8762
<code>\tex_Umathrelinnerspacing:D</code>	1086	<code>\tex_uptexversion:D</code>	1208, 8761
<code>\tex_Umathreltopspacing:D</code>	1087	<code>\tex_Uradical:D</code>	1120

<code>\tex_Uright:D</code>	1121	<code>\tex_XeTeXfindvariationbyname:D</code>	721
<code>\tex_Uroot:D</code>	1122	<code>\tex_XeTeXfirstfontchar:D</code>	722
<code>\tex_Uskewed:D</code>	1123	<code>\tex_XeTeXfonttype:D</code>	723
<code>\tex_Uskewedwithdelims:D</code>	1124	<code>\tex_XeTeXgenerateactualtext:D</code> .	725
<code>\tex_Ustack:D</code>	1125	<code>\tex_XeTeXglyph:D</code>	726
<code>\tex_Ustartdisplaymath:D</code>	1126	<code>\tex_XeTeXglyphbounds:D</code>	727
<code>\tex_Ustartmath:D</code>	1127	<code>\tex_XeTeXglyphindex:D</code>	728
<code>\tex_Ustopdisplaymath:D</code>	1128	<code>\tex_XeTeXglyphname:D</code>	729
<code>\tex_Ustopmath:D</code>	1129	<code>\tex_XeTeXhyphenatablelength:D</code> .	768
<code>\tex_Usubscript:D</code>	1130	<code>\tex_XeTeXinputencoding:D</code>	730
<code>\tex_Usuperscript:D</code>	1131	<code>\tex_XeTeXinputnormalization:D</code> .	732
<code>\tex_Uunderdelimitter:D</code>	1132	<code>\tex_XeTeXinterchartokenstate:D</code>	734
<code>\tex_Uvextensible:D</code>	1133	<code>\tex_XeTeXinterchartoks:D</code>	735
<code>\tex_vadjust:D</code>	450	<code>\tex_XeTeXinterwordspaceshaping:D</code>	
<code>\tex_valign:D</code>	451	766
<code>\tex_variablefam:D</code>	927	<code>\tex_XeTeXisdefaultselector:D</code> ..	737
<code>\tex_vbadness:D</code>	452	<code>\tex_XeTeXisexclusivefeature:D</code> .	739
<code>\tex_vbox:D</code>	453, 34861, 34866, 34871, 34876, 34881, 34900, 34905, 34912, 34918, 34933, 34939	<code>\tex_XeTeXlastfontchar:D</code>	740
<code>\tex_vcenter:D</code>	454, 1323	<code>\tex_XeTeXlinebreaklocale:D</code> ...	742
<code>\tex_vfi:D</code>	1199	<code>\tex_XeTeXlinebreakpenalty:D</code> ..	743
<code>\tex_vfil:D</code>	455	<code>\tex_XeTeXlinebreakskip:D</code>	741
<code>\tex_vfill:D</code>	456	<code>\tex_XeTeXOTcountfeatures:D</code> ...	744
<code>\tex_vfilneg:D</code>	457	<code>\tex_XeTeXOTcountlanguages:D</code> ..	745
<code>\tex_vfuzz:D</code>	458	<code>\tex_XeTeXOTcountscritps:D</code>	746
<code>\tex_voffset:D</code>	459, 1330	<code>\tex_XeTeXOTfeaturetag:D</code>	747
<code>\tex_vpack:D</code>	928	<code>\tex_XeTeXOTlanguagetag:D</code>	748
<code>\tex_vrule:D</code>	460, 36473	<code>\tex_XeTeXOTscripttag:D</code>	749
<code>\tex_vsize:D</code>	461	<code>\tex_XeTeXpdiffile:D</code>	750
<code>\tex_vskip:D</code>	462, 21253	<code>\tex_XeTeXpdiffpagecount:D</code>	751
<code>\tex_vsplit:D</code>	463, 34950, 34955	<code>\tex_XeTeXpicfile:D</code>	752
<code>\tex_vss:D</code>	464	<code>\tex_XeTeXrevision:D</code> . 753, 8770, 8997	
<code>\tex_vtop:D</code> ..	465, 34863, 34888, 34893	<code>\tex_XeTeXselectorcode:D</code>	764
<code>\tex_wd:D</code>	466, 34672	<code>\tex_XeTeXselectorname:D</code>	754
<code>\tex_widowpenalties:D</code>	538	<code>\tex_XeTeXtracingfonts:D</code>	755
<code>\tex_widowpenalty:D</code>	467	<code>\tex_XeTeXupwardsmode:D</code>	756
<code>\tex_wordboundary:D</code>	929	<code>\tex_XeTeXuseglyphmetrics:D</code> ...	757
<code>\tex_write:D</code>		<code>\tex_XeTeXvariation:D</code>	758
..	468, 1976, 1978, 10567, 10570, 10589	<code>\tex_XeTeXvariationdefault:D</code> ..	759
<code>\tex_xdef:D</code>		<code>\tex_XeTeXvariationmax:D</code>	760
...	469, 1455, 1456, 1460, 1465, 1470	<code>\tex_XeTeXvariationmin:D</code>	761
<code>\tex_XeTeXcharclass:D</code>	706	<code>\tex_XeTeXvariationname:D</code>	762
<code>\tex_XeTeXcharglyph:D</code>	707	<code>\tex_XeTeXversion:D</code>	
<code>\tex_XeTeXcountfeatures:D</code>	708	..	763, 8658, 8769, 14041, 17583, 18270
<code>\tex_XeTeXcountglyphs:D</code>	709	<code>\tex_xkanjiskip:D</code>	1195
<code>\tex_XeTeXcountselectors:D</code>	710	<code>\tex_xleaders:D</code>	470
<code>\tex_XeTeXcountvariations:D</code> ...	711	<code>\tex_xspaceskip:D</code>	471
<code>\tex_XeTeXdashbreakstate:D</code>	713	<code>\tex_xspcode:D</code>	1196
<code>\tex_XeTeXdefaultencoding:D</code> ...	712	<code>\tex_xtoksapp:D</code>	930
<code>\tex_XeTeXfeaturecode:D</code>	714	<code>\tex_xtokspre:D</code>	931
<code>\tex_XeTeXfeaturename:D</code>	715	<code>\tex_ybaselineshift:D</code>	1197
<code>\tex_XeTeXfindfeaturebyname:D</code> ..	717	<code>\tex_year:D</code>	472, 1301, 1305
<code>\tex_XeTeXfindselectorbyname:D</code> .	719	<code>\tex_yoko:D</code>	1198
		<code>\text</code>	34271

- text commands:
- `\l_text_accents_tl` 31657, 31892
 - `\l_text_case_exclude_arg_tl`
.. 296, 297, 299, 31659, 31856, 32295
 - `\text_case_switch:nnnn`
.. 298, 31849, 32345, 32646, 32646
 - `\text_declare_case_equivalent:Nn`
.. 298, 32608, 32608
 - `\text_declare_expand_equivalent:Nn`
296, 32050, 32050, 32055, 32058, 32073
 - `\text_declare_lowercase_mapping:n`
.. 298, 32613, 32613
 - `\text_declare_lowercase_mapping:nn`
.. 298, 32613, 32629
 - `\text_declare_purify_equivalent:Nn`
.. 299, 34251, 34251,
34256, 34264, 34265, 34266, 34267,
34284, 34309, 34310, 34312, 34313,
34315, 34318, 34319, 34325, 34327,
34328, 34329, 34333, 34366, 34381
 - `\text_declare_titlecase_mapping:n`
.. 298, 32613, 32615
 - `\text_declare_titlecase_mapping:nn`
.. 298, 32613, 32631
 - `\text_declare_uppercase_mapping:n`
.. 298, 32613, 32617, 33701
 - `\text_declare_uppercase_mapping:nn`
.. 298, 32613, 32633
 - `\text_expand:n` 296, 297, 299,
300, 31713, 31713, 32103, 33707, 34064
 - `\l_text_expand_exclude_tl`
.. 296, 299, 31670, 31855
 - `\l_text_letterlike_tl` . . . 31657, 31912
 - `\text_lowercase:n` . . . 139, 198, 297,
32079, 32079, 38902, 38905, 38907,
38909, 38949, 38950, 38965, 38966
 - `\text_lowercase:nn`
.. 297, 32079, 32087, 38910, 38912
 - `\text_map_break:` 300,
33706, 33712, 33731, 33747, 33788,
33938, 34040, 34041, 34043, 34051
 - `\text_map_break:n` . . . 300, 33706, 34042
 - `\text_map_function:nN`
.. 300, 33706, 33706, 34049
 - `\text_map_inline:n` 300, 34044, 34044
 - `\l_text_math_arg_tl`
.. 296, 299, 31666, 31854, 32294, 34180
 - `\l_text_math_delims_tl`
.. 296, 299, 31668, 31772, 32224, 34109
 - `\text_purify:n` 299, 34058, 34058
 - `\text_titlecase:n` 38900, 38901
 - `\text_titlecase:nn` 38900, 38904
 - `\text_titlecase_all:n`
.. 139, 297, 32079, 32083
 - `\text_titlecase_all:nn`
.. 297, 32079, 32091
 - `\l_text_titlecase_check_letter_-
bool` 298, 299, 32077, 32495
 - `\text_titlecase_first:n`
.. 297, 32079, 32085,
38900, 38902, 38919, 38921, 38953,
38954, 38969, 38970, 38976, 38978
 - `\text_titlecase_first:nn`
.. 297, 32079,
32093, 38903, 38905, 38922, 38924
 - `\text_uppercase:n`
139, 199, 297, 32079, 32081, 38913,
38915, 38951, 38952, 38967, 38968
 - `\text_uppercase:nn`
.. 297, 32079, 32089, 38916, 38918
- text internal commands:
- `__text_case_switch_marker:`
.. 32646, 32648, 32651
 - `__text_change_case:nnn`
.. 32079, 32080, 32082,
32084, 32088, 32090, 32092, 32095
 - `__text_change_case:nnnn`
.. 32086, 32094, 32096, 32097, 32097
 - `__text_change_case_auxi:nnnn`
.. 32097, 32102, 32107
 - `__text_change_case_auxii:nnnn`
.. 32097, 32129, 32132,
32133, 32136, 32181, 32195, 32325
 - `__text_change_case_BCP:nnnn`
.. 32097, 32109, 32112
 - `__text_change_case_BCP:nnnnnw`
.. 32097, 32123, 32124
 - `__text_change_case_BCP:nnnw`
.. 32097, 32114, 32119
 - `__text_change_case_boundary_-
upper_el-x-iota:Nnnnw` 33191
 - `__text_change_case_boundary_-
upper_el:nnnN` . . . 33191, 33195, 33201
 - `__text_change_case_boundary_-
upper_el:nnnn` . . . 33191, 33207, 33211
 - `__text_change_case_boundary_-
upper_el:Nnnnw` 33191, 33191, 33200
 - `__text_change_case_boundary_-
upper_el:nnnnw` 33191, 33220, 33223
 - `__text_change_case_break:w`
.. 32097, 32166, 32214
 - `__text_change_case_break_aux:w`
.. 32097, 32167, 32168
 - `__text_change_case_breathing:nnnn`
.. 33221, 33238, 33238
 - `__text_change_case_breathing:nnnnn`
.. 33238, 33242, 33251

- _text_change_case_breathing:nnnnnw
..... 33238, 33263, 33267, 33276
- _text_change_case_breathing:nnnnw
..... 33238, 33254, 33257
- _text_change_case_breathing_
aux:nnnN 33238, 33294, 33298
- _text_change_case_breathing_
aux:nnnnnn ... 33238, 33271, 33280
- _text_change_case_breathing_
aux:nnnnw 33238, 33285, 33288
- _text_change_case_breathing_
dialytika:nnnn 33238, 33301, 33303
- _text_change_case_catcode:nn ..
..... 32097, 32568, 32585,
32589, 32661, 32843, 32847, 33232,
33321, 33323, 33337, 33339, 33404,
33406, 33408, 33421, 33458, 33484,
33565, 33580, 33604, 33612, 33624
- _text_change_case_codepoint:nn
..... 32097, 32532,
32536, 32720, 32730, 32816, 32830,
32858, 32868, 32881, 32905, 33292
- _text_change_case_codepoint:nnn
..... 32097,
32538, 32541, 32546, 32550, 32551
- _text_change_case_codepoint:nnnnn
.... 32097, 32456, 32463, 32520,
32524, 32666, 32704, 33313, 33328,
33344, 33347, 33359, 33370, 33416,
33479, 33517, 33530, 33569, 33628
- _text_change_case_codepoint_
aux:nn 32097, 32543, 32558
- _text_change_case_codepoint_
aux:nnn 32097, 32549, 32555
- _text_change_case_codepoint_
aux:nnnn 32560, 32562
- _text_change_case_codepoint_
lower:nnnn 32097, 32447
- _text_change_case_codepoint_
title:nnn 32097, 32503, 32509, 32513
- _text_change_case_codepoint_
title:nnnn 32097, 32493
- _text_change_case_codepoint_
title_auxi:nnnn 32097, 32497, 32506
- _text_change_case_codepoint_
title_auxii:nnnn
..... 32097, 32510, 32514, 32515
- _text_change_case_codepoint_
upper:nnnn 32097, 32453
- _text_change_case_cs_check:nnnN
..... 32097, 32235, 32280
- _text_change_case_custom:nnnnn
..... 32097
- _text_change_case_custom:nnnnnn
..... 32418, 32425, 32427, 32431
- _text_change_case_custom_
lower:nnnn ... 32097, 32416, 32422
- _text_change_case_custom_
title:nnnn 32097, 32423
- _text_change_case_custom_
upper:nnnn 32097, 32421
- _text_change_case_end:w 32097,
32149, 32172, 32218, 32260, 32379
- _text_change_case_exclude:nnnN
..... 32097, 32283, 32290
- _text_change_case_exclude:nnnNN
..... 32097, 32301, 32304, 32313
- _text_change_case_exclude:nnnnN
..... 32097, 32292, 32299
- _text_change_case_exclude:nnnNnn
..... 32097, 32316, 32317
- _text_change_case_exclude:nnnNw
..... 32097, 32311, 32315
- _text_change_case_generate:n ..
..... 32652, 32652, 32889, 32911
- _text_change_case_group_
lower:nnnn ... 32097, 32174, 32187
- _text_change_case_group_
title:nnnn 32097, 32188
- _text_change_case_group_
upper:nnnn 32097, 32186
- _text_change_case_if_greek:n ..
..... 32670, 32969, 32971, 32974
- _text_change_case_if_greek:nTF
..... 32670, 33240
- _text_change_case_if_greek_
accent:n 32670, 32998, 33000, 33003
- _text_change_case_if_greek_
accent:nTF 32670, 32794
- _text_change_case_if_greek_
accent_p:n 32775, 32964
- _text_change_case_if_greek_
breathing:n
..... 32670, 33109, 33112, 33115
- _text_change_case_if_greek_
breathing:nTF 32670, 32797
- _text_change_case_if_greek_
breathing_p:n 32776, 32965
- _text_change_case_if_greek_p:n
..... 32673
- _text_change_case_if_greek_
spacing_diacritic:n
..... 32670, 33031, 33034, 33037
- _text_change_case_if_greek_
spacing_diacritic:nTF 32670, 32680
- _text_change_case_if_greek_
stress:n 32670, 33127, 33130, 33133

- _text_change_case_if_greek_-
 stress:nTF 32670, 32807
- _text_change_case_if_takes_-
 dialytika:n
 32670, 33145, 33147, 33150
- _text_change_case_if_takes_-
 dialytika:nTF
 32670, 32826, 32878, 33305
- _text_change_case_if_takes_-
 ypogegrammeni:n
 32670, 33170, 33172, 33175
- _text_change_case_if_takes_-
 ypogegrammeni:nTF .. 32670, 32734
- _text_change_case_letterlike:nnnnN
 32097, 32391, 32395, 32396
- _text_change_case_letterlike_-
 lower:nnnN ... 32097, 32390, 32393
- _text_change_case_letterlike_-
 title:nnnN 32097, 32394
- _text_change_case_letterlike_-
 upper:nnnN 32097, 32392
- _text_change_case_loop:nnnw ...
 32097, 32140, 32155,
 32184, 32209, 32263, 32329, 32341,
 32353, 32358, 32413, 32473, 32491,
 32601, 32683, 32702, 32721, 32731,
 32804, 32811, 32817, 32859, 32869,
 32948, 32954, 32967, 33196, 33204,
 33227, 33234, 33249, 33260, 33286,
 33295, 33308, 33310, 33410, 33430,
 33461, 33567, 33582, 33606, 33614
- _text_change_case_lower_-
 az:nnnnn 33630, 33630
- _text_change_case_lower_-
 la-x-medieval:nnnnn 33350
- _text_change_case_lower_-
 lt:nnnN 33372, 33429, 33433
- _text_change_case_lower_-
 lt:nnnn 33372, 33436, 33438
- _text_change_case_lower_-
 lt:nnnnn 33372
- _text_change_case_lower_-
 lt:nnnw 33372, 33423, 33426
- _text_change_case_lower_lt_-
 auxi:nnnnn 33374, 33385
- _text_change_case_lower_lt_-
 auxii:nnnnn 33389, 33413
- _text_change_case_lower_-
 sigma:nnnnN ... 32097, 32469, 32477
- _text_change_case_lower_-
 sigma:nnnnn ... 32097, 32450, 32459
- _text_change_case_lower_-
 sigma:nnnnw ... 32097, 32462, 32466
- _text_change_case_lower_-
 tr:NnnnN 33555, 33575, 33586
- _text_change_case_lower_-
 tr:Nnnnn 33555, 33589, 33591
- _text_change_case_lower_-
 tr:nnnnn 33555, 33555, 33631
- _text_change_case_lower_-
 tr:nnnNw 33555, 33558, 33572
- _text_change_case_math_-
 group:nnnNn ... 32097, 32252, 32266
- _text_change_case_math_-
 loop:nnnNw 32097,
 32241, 32246, 32264, 32269, 32278
- _text_change_case_math_N_-
 type:nnnNN ... 32097, 32249, 32257
- _text_change_case_math_-
 search:nnnNNN
 32097, 32228, 32232, 32244
- _text_change_case_math_-
 space:nnnNw ... 32097, 32253, 32273
- _text_change_case_N_type:nnnN .
 32097, 32158, 32215
- _text_change_case_N_type:nnnnN
 32097, 32223, 32226
- _text_change_case_N_type_-
 aux:nnnN 32097, 32219, 32221
- _text_change_case_next_end:nnn
 32097, 32606
- _text_change_case_next_-
 lower:nnn 32097, 32600, 32603, 32605
- _text_change_case_next_-
 title:nnn 32097, 32604
- _text_change_case_next_-
 upper:nnn 32097, 32602
- _text_change_case_replace:nnnN
 32097, 32307, 32331
- _text_change_case_replace:nnnn
 32097,
 32335, 32340, 32342, 32435, 32441
- _text_change_case_result:n ...
 .. 32097, 32142, 32147, 32148, 32149
- _text_change_case_setup:NN ...
 33635, 33642, 33644
- _text_change_case_setup:Nn ...
 33668, 33688, 33690
- _text_change_case_skip:nnw ...
 32097, 32198,
 32363, 32365, 32381, 32386, 32607
- _text_change_case_skip_-
 group:nnn ... 32097, 32371, 32383
- _text_change_case_skip_N_-
 type:nnN 32097, 32368, 32376
- _text_change_case_skip_-
 space:nnw 32097, 32372, 32388

- _text_change_case_space:nnw [32097](#), [32162](#), [32202](#), [32389](#)
- _text_change_case_space_-break:nnn [32213](#)
- _text_change_case_space_-break:nnnw [32097](#)
- _text_change_case_store:n [32097](#), [32144](#), [32146](#), [32171](#), [32176](#), [32190](#), [32205](#), [32240](#), [32261](#), [32268](#), [32277](#), [32320](#), [32322](#), [32352](#), [32357](#), [32362](#), [32380](#), [32385](#), [32400](#), [32405](#), [32471](#), [32479](#), [32529](#), [32531](#), [32658](#), [32682](#), [32697](#), [32719](#), [32729](#), [32802](#), [32809](#), [32815](#), [32829](#), [32836](#), [32857](#), [32867](#), [32880](#), [33229](#), [33291](#), [33318](#), [33334](#), [33354](#), [33365](#), [33401](#), [33418](#), [33455](#), [33481](#), [33526](#), [33549](#), [33562](#), [33577](#), [33601](#), [33609](#), [33621](#)
- _text_change_case_store:nw [32097](#), [32145](#), [32147](#)
- _text_change_case_switch:nnnN [32097](#), [32338](#), [32343](#)
- _text_change_case_switch_-lower:nnnNnnnn [32097](#), [32350](#)
- _text_change_case_switch_-title:nnnNnnnn [32097](#), [32360](#)
- _text_change_case_switch_-upper:nnnNnnnn [32097](#), [32355](#)
- _text_change_case_title_-el:nnnnn [33312](#), [33312](#)
- _text_change_case_title_-hy-x-yiwn:nnnnn [33314](#)
- _text_change_case_title_-hy:nnnnn [33314](#), [33330](#)
- _text_change_case_title_-nl:nnnN [33513](#), [33535](#), [33539](#)
- _text_change_case_title_-nl:nnnnn [33513](#), [33513](#)
- _text_change_case_title_-nl:nnnw [33513](#), [33528](#), [33532](#)
- _text_change_case_title_nl_-aux:nnnnn [33513](#), [33516](#), [33520](#)
- _text_change_case_upper_-az:nnnnn [33630](#), [33632](#)
- _text_change_case_upper_-de-alt:nnnnn [32654](#)
- _text_change_case_upper_-de-x-eszett:nnnnn [32654](#)
- _text_change_case_upper_-el-x-iota:nnnnn [32670](#)
- _text_change_case_upper_-el-x-iota ypogegrammeni:n . [32670](#)
- _text_change_case_upper_-el:nnnn [32670](#), [32686](#), [32709](#), [32798](#), [32885](#)
- _text_change_case_upper_-el:nnnnN [32670](#), [32717](#), [32725](#)
- _text_change_case_upper_-el:nnnnn [32670](#), [32670](#), [32708](#)
- _text_change_case_upper_-el:nnnnw [32670](#), [32712](#), [32714](#)
- _text_change_case_upper_el_-aux:nnnnN [32670](#), [32739](#), [32750](#), [32756](#), [32781](#), [32784](#)
- _text_change_case_upper_el_-aux:nnnnn [32670](#), [32787](#), [32789](#)
- _text_change_case_upper_el_-dialytika:n [32670](#), [32827](#), [32834](#), [32882](#), [33307](#)
- _text_change_case_upper_el_-dialytika:nnnn . [32670](#), [32792](#), [32824](#)
- _text_change_case_upper_el_-gobble:nnnN [32670](#), [32947](#), [32951](#)
- _text_change_case_upper_el_-gobble:nnnn [32670](#), [32957](#), [32961](#)
- _text_change_case_upper_el_-gobble:nnnw [32670](#), [32832](#), [32883](#), [32943](#), [32966](#)
- _text_change_case_upper_el_-hiatus:nnnnN [32670](#), [32855](#), [32863](#)
- _text_change_case_upper_el_-hiatus:nnnnn [32670](#), [32873](#), [32876](#)
- _text_change_case_upper_el_-hiatus:nnnnw [32670](#), [32795](#), [32851](#)
- _text_change_case_upper_el_-stress:nn [32670](#), [32810](#), [32909](#)
- _text_change_case_upper_el_-ypogegrammeni:n [32670](#), [32887](#)
- _text_change_case_upper_el_-ypogegrammeni:nnnnnnN [32670](#), [32747](#), [32753](#)
- _text_change_case_upper_el_-ypogegrammeni:nnnnnnn [32670](#), [32760](#), [32766](#)
- _text_change_case_upper_el_-ypogegrammeni:nnnnnnw [32670](#), [32736](#), [32742](#), [32770](#), [32778](#)
- _text_change_case_upper_-hy-x-yiwn:nnnnn [33314](#)
- _text_change_case_upper_-hy:nnnnn [33314](#), [33314](#)
- _text_change_case_upper_-la-x-medieval:nnnnn [33350](#)
- _text_change_case_upper_-lt:nnnN [33463](#), [33492](#), [33496](#)
- _text_change_case_upper_-lt:nnnn [33463](#), [33499](#), [33501](#)

- _text_change_case_upper_-
 lt:nnnnn 33463
- _text_change_case_upper_-
 lt:nnnw 33463, 33486, 33489
- _text_change_case_upper_lt_-
 aux:nnnnn 33465, 33476
- _text_change_case_upper_-
 tr:nnnnn 33617, 33617, 33633
- _text_change_cases_lower_-
 lt:nnnnn 33372
- _text_change_cases_lower_lt_-
 auxi:nnnnn 33372
- _text_change_cases_lower_lt_-
 auxii:nnnnn 33372
- _text_change_cases_upper_-
 lt:nnnnn 33463
- _text_change_cases_upper_lt_-
 aux:nnnnn 33463
- _text_char_catcode:N . . . 31498,
 31498, 32472, 32488, 32489, 32586,
 32592, 33355, 33366, 33527, 33550
- \c_text_chardef_group_begin_-
 token 31707
- \c_text_chardef_group_end_token
 31707
- \c_text_chardef_space_token . 31707
- _text_codepoint_compare:nNn . . .
 31590, 31599
- _text_codepoint_compare:nNnTF .
 31585, 32461, 32564, 32591,
 32656, 32695, 32768, 32791, 32800,
 33316, 33332, 33352, 33363, 33557,
 33560, 33619, 33762, 33768, 33813,
 33819, 33866, 33872, 33971, 33977
- _text_codepoint_compare_p:nNn .
 31585, 32676, 32677, 32839, 32840,
 33215, 33216, 33217, 33218, 33283,
 33284, 33449, 33450, 33451, 33509,
 33599, 33845, 33846, 34013, 34014
- _text_codepoint_from_chars:N . .
 31585, 31617, 31625, 31644
- _text_codepoint_from_chars:NN .
 31585, 31632, 31645
- _text_codepoint_from_chars:NNN
 31585, 31636, 31647
- _text_codepoint_from_chars:NNNN
 31585, 31639, 31649
- _text_codepoint_from_chars:Nw .
 31585, 31595,
 31601, 31605, 32500, 32539, 32689,
 32892, 32914, 32918, 32930, 32972,
 33001, 33035, 33113, 33131, 33148,
 33173, 33245, 33376, 33391, 33467
- _text_codepoint_from_chars_-
 aux:Nw . . 31585, 31611, 31621, 31629
- _text_codepoint_process:nN . . .
 31545, 31549, 31552, 32285, 32711,
 32758, 32786, 32872, 32956, 33206,
 33253, 33262, 33275, 33300, 33435,
 33498, 33588, 33756, 33946, 34035
- _text_codepoint_process:nNN . . .
 31545, 31570, 31578
- _text_codepoint_process:nNNN . .
 31545, 31573, 31580
- _text_codepoint_process:nNNNN .
 31545, 31574, 31582
- _text_codepoint_process_aux:nN
 31545, 31557, 31561, 31567
- _text_data_auxi:w . . . 31350, 31377
- _text_data_auxii:w . . 31362, 31364
- \g_text_data_ior
 31345, 31347, 31372, 31380
- _text_declare_case_mapping:nmn
 . . 32613, 32614, 32616, 32618, 32619
- _text_declare_case_mapping:nnnn
 . . 32613, 32630, 32632, 32634, 32635
- _text_declare_case_mapping_-
 aux:nnn 32613, 32621, 32624
- _text_declare_case_mapping_-
 aux:nnnn 32613, 32637, 32640
- _text_end_env:n 34312, 34313, 34314
- _text_expand:n
 31713, 31718, 31721, 31757
- _text_expand_accent:N
 31713, 31874, 31889
- _text_expand_accent:NN
 31713, 31891, 31895, 31907
- _text_expand_cs:N
 31713, 31918, 31929
- _text_expand_cs_expand:N
 31713, 32007, 32010
- _text_expand_encoding:N
 31713, 31978, 31985
- _text_expand_encoding_escape:N
 31713
- _text_expand_encoding_escape:NN
 31990, 31993
- _text_expand_end:w
 . . 31713, 31733, 31770, 31804, 31956
- _text_expand_exclude:N
 31713, 31825, 31847
- _text_expand_exclude:NN
 31713, 31868, 31871, 31880
- _text_expand_exclude:nN
 31713, 31852, 31866
- _text_expand_exclude:Nnn
 31713, 31883, 31884

- _text_expand_exclude:Nw
..... 31713, 31878, 31882
- _text_expand_exclude_switch:Nnnnn
..... 31713, 31850, 31861
- _text_expand_explicit:N
..... 31713, 31779, 31822
- _text_expand_group:n
..... 31713, 31745, 31750
- _text_expand_letterlike:N
..... 31713, 31898, 31909
- _text_expand_letterlike:NN ...
..... 31713, 31911, 31915, 31927
- _text_expand_loop:w
31713, 31724, 31739, 31760, 31765,
31808, 31840, 31843, 31864, 31887,
31904, 31924, 31947, 31972, 31983,
31990, 32009, 32016, 32020, 32048
- _text_expand_math_group:Nn ...
..... 31713, 31796, 31811
- _text_expand_math_loop:Nw 31713,
31785, 31790, 31809, 31814, 31820
- _text_expand_math_N_type:NN ...
..... 31713, 31793, 31801
- _text_expand_math_search:NNN ..
..... 31713, 31771, 31776, 31788
- _text_expand_math_space:Nw ...
..... 31713, 31797, 31816
- _text_expand_N_type:N
..... 31713, 31742, 31767
- _text_expand_protect:N
..... 31713, 31944, 31951
- _text_expand_protect:nN
..... 31713, 31958, 31961
- _text_expand_protect:Nw
..... 31713, 31962, 31963
- _text_expand_protect:w
..... 31713, 31932, 31941
- _text_expand_replace:N
..... 31713, 31938, 31991, 31994
- _text_expand_replace:n
..... 31713, 32004, 32009
- _text_expand_result:n
.. 31713, 31726, 31731, 31732, 31733
- _text_expand_space:w
..... 31713, 31746, 31762
- _text_expand_store:n
..... 31713, 31728,
31730, 31752, 31764, 31784, 31806,
31813, 31819, 31842, 31863, 31886,
31903, 31923, 31946, 31955, 31968,
31969, 31971, 31982, 32019, 32047
- _text_expand_store:nw
..... 31713, 31729, 31731
- _text_expand_testopt:N
..... 31713, 31937, 31974
- _text_expand_testopt:NNn
..... 31713, 31977, 31980
- _text_expand_unexpanded:N
..... 1300, 31713, 32034, 32038
- _text_expand_unexpanded:n
..... 31713, 32031, 32045
- _text_expand_unexpanded:w
..... 31713, 32015, 32023, 32033
- _text_expand_unexpanded_test:w
..... 31713, 32025, 32028
- \c_text_grapheme_Control_clist .
..... 33853
- _text_if_expandable:N 31530
- _text_if_expandable:NTF
..... 31530, 32012, 34241
- _text_if_q_recursion_tail_-
stop_do:Nn
31393, 31393, 31778, 31873, 31897,
31917, 32217, 32234, 32259, 32306,
32378, 33744, 33785, 33935, 34029,
34103, 34115, 34156, 34184, 34231
- _text_if_q_recursion_tail_-
stop_do:nn
.. 31393, 31394, 33811, 33864, 33969
- _text_if_recursion_tail_stop:N
..... 34057, 34057
- _text_if_s_recursion_tail_-
stop_do:Nn
.. 31399, 31399, 31769, 31803, 31953
- _text_loop:Nn 34330,
34338, 34340, 34383, 34388, 34390
- _text_loop:NNn . 34409, 34415, 34417
- _text_map_class:Nnnn
..... 33706, 33770,
33799, 33881, 33883, 33885, 33887,
33889, 33891, 33893, 33895, 33897
- _text_map_class:nNnn
..... 33706, 33801, 33804
- _text_map_class_end:nw . 33706,
33815, 33822, 33827, 33868, 33875
- _text_map_class_loop:Nnnnw ...
..... 33706, 33806, 33809, 33820
- _text_map_codepoint:Nnn
..... 33706, 33757, 33760
- _text_map_Control:Nnn 33706, 33828
- _text_map_CR:NnN 33706, 33776, 33783
- _text_map_CR:Nnw 33706, 33765, 33773
- _text_map_Extend:Nnn
..... 33706, 33834, 33836
- _text_map_function:nN
..... 33706, 33707, 33708

- _text_map_group:Nnn [33706](#), [33720](#), [33725](#)
- _text_map_hangul:NnnN [33706](#), [33926](#), [33933](#)
- _text_map_hangul:Nnnn [33706](#), [33947](#), [33950](#)
- _text_map_hangul:nNnnnw [33706](#), [33959](#), [33962](#)
- _text_map_hangul:Nnnw [33706](#), [33906](#), [33912](#),
[33919](#), [33923](#), [33990](#), [33995](#), [34001](#)
- _text_map_hangul_aux:Nnnnw . [33706](#)
- _text_map_hangul_aux:Nnnw [33952](#), [33955](#), [33986](#)
- _text_map_hangul_end:nw [33706](#), [33973](#), [33980](#), [33987](#)
- _text_map_hangul_L:Nnn [33706](#), [33988](#)
- _text_map_hangul_loop:Nnnnw [33706](#), [33964](#), [33967](#), [33978](#)
- _text_map_hangul_LV:Nnn [33706](#), [33993](#), [33998](#)
- _text_map_hangul_LVT:Nnn [33706](#), [33999](#), [34004](#)
- _text_map_hangul_next:Nnnn [33706](#), [33970](#), [33974](#), [33985](#)
- _text_map_hangul_T:Nnn [33706](#), [34004](#)
- _text_map_hangul_V:Nnn [33706](#), [33998](#)
- _text_map_L:Nnn [33706](#), [33903](#)
- _text_map_lookahead:NnNN [33706](#), [34023](#), [34027](#)
- _text_map_lookahead:NnNw [33706](#), [33840](#), [34008](#), [34020](#)
- _text_map_loop:Nnw [33706](#), [33710](#), [33714](#), [33729](#), [33733](#),
[33740](#), [33753](#), [33769](#), [33779](#), [33795](#),
[33797](#), [33832](#), [33835](#), [33849](#), [33865](#),
[33869](#), [33876](#), [33901](#), [33929](#), [33943](#),
[33958](#), [34016](#), [34018](#), [34024](#), [34033](#)
- _text_map_LV:Nnn [33706](#), [33909](#), [33915](#)
- _text_map_LVT:Nnn [33706](#), [33916](#), [33922](#)
- _text_map_N_type:NnN [33706](#), [33717](#), [33742](#)
- _text_map_not_Control:Nnn [33706](#), [33880](#)
- _text_map_not_Extend:Nnn [33706](#), [33882](#)
- _text_map_not_L:Nnn . [33706](#), [33888](#)
- _text_map_not_LV:Nnn . [33706](#), [33890](#)
- _text_map_not_LVT:Nnn [33706](#), [33894](#)
- _text_map_not_Prepend:Nnn [33706](#), [33886](#)
- _text_map_not_Regional_
Indicator:Nnn [33898](#)
- _text_map_not_SpacingMark:Nnn [33706](#), [33884](#)
- _text_map_not_T:Nnn . [33706](#), [33896](#)
- _text_map_not_V:Nnn . [33706](#), [33892](#)
- _text_map_output:Nn . . . [33706](#),
[33727](#), [33738](#), [33746](#), [33751](#), [33764](#),
[33794](#), [33830](#), [33831](#), [33839](#), [33900](#),
[33905](#), [33911](#), [33918](#), [34007](#), [34038](#)
- _text_map_Prepend:Nnn [33706](#), [33837](#)
- _text_map_Prepend:Nnnn [33706](#), [33852](#), [33857](#)
- _text_map_Prepend_aux:Nnn [33706](#), [33840](#), [33842](#)
- _text_map_Prepend_loop:Nnnw [33706](#), [33859](#), [33862](#), [33873](#)
- _text_map_Regional_Indicator:Nnn [33706](#), [34005](#)
- _text_map_Regional_Indicator_
aux:Nnn [33706](#), [34008](#), [34010](#)
- _text_map_space:Nnw [33706](#), [33721](#), [33736](#)
- _text_map_SpacingMark:Nnn [33706](#), [33836](#)
- _text_map_T:Nnn [33706](#), [33922](#)
- _text_map_V:Nnn [33706](#), [33915](#)
- \l_text_math_mode_tl [31706](#)
- \c_text_mathchardef_group_
begin_token [31707](#)
- \c_text_mathchardef_group_end_
token [31707](#)
- \c_text_mathchardef_space_token [31707](#)
- _text_purify:n . [34058](#), [34063](#), [34067](#)
- _text_purify_accent:NN [34367](#), [34367](#), [34381](#)
- _text_purify_encoding:N [34058](#), [34227](#), [34234](#)
- _text_purify_encoding_escape:NN [34058](#), [34239](#), [34246](#)
- _text_purify_end:w [34058](#), [34078](#), [34103](#), [34141](#), [34231](#)
- _text_purify_expand:N [34058](#), [34217](#), [34223](#)
- _text_purify_group:n [34058](#), [34090](#), [34095](#)
- _text_purify_loop:w [34058](#), [34070](#),
[34084](#), [34095](#), [34099](#), [34136](#), [34213](#),
[34220](#), [34232](#), [34242](#), [34243](#), [34249](#)
- _text_purify_math_cmd:N [34058](#), [34116](#), [34177](#)
- _text_purify_math_cmd:n [34189](#), [34193](#)
- _text_purify_math_cmd:NN [34058](#), [34179](#), [34182](#), [34191](#)

- `__text_purify_math_cmd:Nn` ... [34058](#)
- `__text_purify_math_end:w`
..... [34058](#), [34133](#), [34159](#), [34194](#)
- `__text_purify_math_group:NNn` ...
..... [34058](#), [34149](#), [34165](#)
- `__text_purify_math_loop:NNw` ...
..... [34058](#),
[34126](#), [34143](#), [34162](#), [34168](#), [34175](#)
- `__text_purify_math_N_type:NNN` ..
..... [34058](#), [34146](#), [34154](#)
- `__text_purify_math_result:n` ...
..... [34127](#),
[34131](#), [34132](#), [34133](#), [34138](#), [34194](#)
- `__text_purify_math_search:NNN` ..
..... [34058](#), [34108](#), [34113](#), [34122](#)
- `__text_purify_math_space:NNw` ...
..... [34058](#), [34150](#), [34170](#)
- `__text_purify_math_start:NNw` ...
..... [34058](#), [34120](#), [34124](#)
- `__text_purify_math_stop:Nw`
..... [34138](#), [34157](#)
- `__text_purify_math_store:n`
.. [34058](#), [34129](#), [34161](#), [34167](#), [34174](#)
- `__text_purify_math_store:nw` ...
..... [34058](#), [34130](#), [34131](#)
- `__text_purify_N_type:N`
..... [34058](#), [34087](#), [34101](#)
- `__text_purify_N_type_aux:N`
..... [34058](#), [34104](#), [34106](#)
- `__text_purify_protect:N`
..... [34058](#), [34226](#), [34229](#)
- `__text_purify_replace:N`
..... [34058](#), [34185](#), [34195](#)
- `__text_purify_replace_auxi:n` ...
..... [34058](#), [34205](#), [34213](#)
- `__text_purify_replace_auxii:n` ..
..... [34058](#), [34209](#), [34214](#)
- `__text_purify_result:n`
..... [34072](#), [34076](#), [34077](#), [34078](#)
- `__text_purify_space:w`
..... [34058](#), [34091](#), [34096](#)
- `__text_purify_store:n`
..... [34058](#), [34074](#),
[34098](#), [34135](#), [34140](#), [34219](#), [34248](#)
- `__text_purify_store:nw`
..... [34058](#), [34075](#), [34076](#)
- `__text_quark_if_nil:n` [31388](#)
- `__text_quark_if_nil:nTF` [31388](#), [31965](#)
- `__text_quark_if_nil_p:n` [31388](#)
- `__text_tmp:w` [31682](#), [31700](#)
- `\l__text_tmpa_str`
.. [31348](#), [31357](#), [31358](#), [31366](#), [31368](#)
- `\l__text_tmpb_str`
.. [31349](#), [31352](#), [31354](#), [31356](#), [31360](#)
- `__text_token_to_explicit:N`
..... [31407](#), [31409](#), [34210](#)
- `__text_token_to_explicit:n`
..... [31407](#), [31461](#), [31465](#)
- `__text_token_to_explicit_auxi:w`
..... [31407](#), [31467](#), [31482](#)
- `__text_token_to_explicit_-auxii:w` [31407](#), [31487](#), [31495](#)
- `__text_token_to_explicit_-auxiii:w` [31407](#), [31489](#), [31497](#)
- `__text_token_to_explicit_char:N`
..... [31407](#), [31419](#), [31451](#)
- `__text_token_to_explicit_cs:N` ..
..... [31407](#), [31417](#), [31424](#)
- `__text_token_to_explicit_cs_-aux:N` [31407](#), [31428](#), [31434](#)
- `__text_use_i_delimit_by_q_-recursion_stop:nw`
[31391](#), [31391](#), [31782](#), [31877](#), [31901](#),
[31921](#), [32238](#), [32310](#), [34119](#), [34188](#)
- `__text_use_i_delimit_by_s_-recursion_stop:nw`
..... [31397](#), [31397](#), [31404](#)
- `\textbaselineshiftfactor` [1189](#)
- `\textbf` [34276](#)
- `\textdir` [922](#)
- `\textdirection` [923](#)
- `\textfont` [419](#)
- `\textit` [34278](#)
- `\textmd` [34277](#)
- `\textnormal` [34272](#)
- `\textrm` [34273](#)
- `\textsc` [34281](#)
- `\textsf` [34274](#)
- `\textsl` [34279](#)
- `\textstyle` [420](#)
- `\texttt` [34275](#)
- `\textulc` [34282](#)
- `\textup` [34280](#)
- `\TeXeTstate` [529](#)
- `\tfont` [1191](#)
- `\TH` [32071](#), [33655](#), [34350](#)
- `\th` [32071](#), [33655](#), [34363](#)
- `\the` [29](#), [85](#), [86](#), [87](#), [88](#), [89](#), [90](#), [91](#), [92](#), [93](#), [421](#)
- `\thickmuskip` [422](#)
- `\thinmuskip` [423](#)
- `\time` [424](#), [1284](#), [9007](#), [9009](#)
- `\tiny` [34307](#), [36468](#)
- tl commands:
 - `\c_catcode_active_space_tl` [196](#), [19277](#)
 - `\c_catcode_active_tl`
..... [200](#), [900](#), [19352](#), [19412](#)

- \c_catcode_other_space_tl 6105, 6675, 6902, 6967, 7036, 7093, 7096, 7110, 7178, 7820, 7921, 7924, 7932, 7935, 13321, 13321, 13326, 39503
- \c_empty_tl 125, 856, 870, 9514, 12135, 12146, 12148, 12183, 12550, 13340, 13382, 13395, 14148, 17993, 17999, 18375, 18391
- \c_novalue_tl 113, 126, 12184, 12655
- \c_space_tl 126, 3548, 9104, 9314, 9602, 9604, 11649, 12188, 13064, 14029, 18881, 22780, 31210, 31305, 31694, 31762, 31817, 32202, 32274, 33736, 34096, 34172, 34325, 36253, 36297, 36364, 36788, 36789, 36790, 36797, 36798, 36973, 36974, 36980, 36981, 36982, 37776, 37889, 37932, 37933, 37955
- \tl_analysis_log:N 46, 3911, 3913
- \tl_analysis_log:n 46, 3925, 3927
- \tl_analysis_map_inline:Nn 46, 3887, 3887, 5799
- \tl_analysis_map_inline:mn 46, 208, 536, 570, 571, 3887, 3888, 3889, 6575, 7398
- \tl_analysis_show:N 46, 3911, 3911
- \tl_analysis_show:n 46, 3925, 3925
- \tl_build_begin:N 127, 128, 517, 730, 4892, 5401, 5976, 6077, 6607, 6641, 6813, 6879, 7816, 13306, 13306, 38936, 38937, 39502
- \tl_build_clear:N 38936, 38937
- \tl_build_end:N 127, 128, 517, 730, 4922, 4930, 5411, 6033, 6096, 6913, 7842, 7905, 13370, 13370
- \tl_build_gbegin:N 127, 128, 13306, 13308, 38938, 38939, 39583
- \tl_build_gclear:N 38936, 38939
- \tl_build_gend:N 127, 128, 13370, 13375
- \tl_build_get:NN 38940, 38941
- \tl_build_get_intermediate:NN 128, 6632, 13388, 13388, 38940, 38941
- \tl_build_gput_left:Nn 127, 13353, 13356, 13358, 39585
- \tl_build_gput_right:Nn 127, 13321, 13327, 13332, 39584
- \tl_build_put_left:Nn 127, 13353, 13353, 13355, 39504
- \tl_build_put_right:Nn 127, 548, 731, 4899, 4917, 4925, 4929, 4993, 4996, 5036, 5050, 5054, 5177, 5191, 5232, 5254, 5267, 5299, 5312, 5316, 5398, 5404, 5410, 5414, 5457, 5747, 5751, 5758, 5764, 5785, 5801, 5819, 6047, 6092, 6105, 6675, 6902, 6967, 7036, 7093, 7096, 7110, 7178, 7820, 7921, 7924, 7932, 7935, 13321, 13321, 13326, 39503
- \tl_case:Nn 38925, 38926, 38933, 38934
- \tl_case:NnTF 38925, 38928, 38930, 38932
- \tl_clear:N 111, 4221, 4565, 6608, 6642, 9094, 10729, 10730, 10733, 10742, 10852, 10855, 10915, 12145, 12145, 12149, 12152, 12363, 13373, 14429, 18434, 18435, 21988, 22318, 22427, 31684, 37393, 37822, 39490
- \tl_clear_new:N 111, 11589, 11590, 11591, 11592, 11593, 12151, 12151, 12155, 18438, 18439, 32052, 32610, 32626, 32642, 32644, 34253, 37402, 37436, 37477
- \tl_concat:NNN 111, 12163, 12163, 12179, 13436, 39427
- \tl_const:Nn 111, 584, 3542, 4259, 4345, 7247, 8868, 9242, 9243, 9278, 9283, 9285, 9287, 9289, 9291, 9296, 9297, 9304, 10626, 10632, 11053, 12138, 12138, 12143, 12144, 12183, 12186, 12188, 12354, 14264, 14269, 16674, 16729, 18431, 19255, 19280, 19282, 19354, 19969, 23231, 23232, 23233, 23234, 23235, 23243, 23332, 25425, 26796, 27243, 27244, 27245, 27246, 27247, 27248, 27249, 27250, 27251, 31024, 31064, 31251, 31263, 31291, 33638, 33640, 33658, 33659, 33676, 33683, 34386, 34412, 37545, 37546, 37547, 37548, 37549, 37595, 37596, 37597, 37607, 37608, 37609, 37610, 37611, 37612, 37613, 37614, 37733, 37748, 37762, 37796, 38052, 38057, 38062, 38223, 39607
- \tl_count:N 33, 113, 116, 12773, 12778, 12785
- \tl_count:n 33, 113, 116, 397, 748, 843, 1025, 1643, 1647, 2113, 2164, 5663, 7328, 7332, 7351, 7355, 7425, 7429, 12773, 12773, 12784, 13117, 13132, 13144
- \tl_count_tokens:n 116, 12786, 12786, 12799
- \tl_gclear:N 111, 433, 3230, 6881, 8976, 9169, 12145, 12147, 12150, 12154, 13378, 18436, 18437, 39571
- \tl_gclear_new:N 111, 12151, 12153, 12156, 18440, 18441
- \tl_gconcat:NNN 111, 12163, 12171, 12180, 13437, 39428

- \tl_gput_left:Nn 11376, 11385, 11409, 11485, 12581,
 . 111, 12205, 12236, 12241, 12246,
 12251, 12259, 12273, 12274, 12275,
 12276, 12277, 12278, 15154, 15337,
 39573, 39574, 39575, 39576, 39577
- \tl_gput_right:Nn
 112, 1587, 1588, 6911,
 6962, 6963, 7039, 8858, 8860, 8979,
 9172, 12279, 12307, 12309, 12314,
 12319, 12327, 12341, 12342, 12343,
 12344, 12345, 12346, 16644, 16822,
 29987, 30160, 30677, 31677, 31679,
 33663, 33665, 38448, 38450, 38705,
 39578, 39579, 39580, 39581, 39582
- \tl_gremove_all:Nn
 124, 12540, 12542, 12546, 12547
- \tl_gremove_once:Nn
 124, 12534, 12536, 12539
- \tl_greplace_all:Nnn
 124, 12454, 12460, 12474, 12476, 12543
- \tl_greplace_once:Nnn
 123, 12454, 12456, 12466, 12468, 12537
- \tl_greverse:N 116, 13100, 13102, 13105
- .tl_gset:N 245, 22181
- \tl_gset:Nn ... 111, 128, 153, 694,
 703, 732, 6945, 6954, 8784, 8798,
 8804, 8813, 8814, 8824, 8825, 8839,
 9217, 9219, 9221, 9223, 9225, 12189,
 12193, 12195, 12201, 12202, 12203,
 12204, 12367, 16791, 17059, 17128
- .tl_gset_e:N 245, 22181
- \tl_gset_eq:NN 111, 3220,
 6951, 7242, 8302, 12157, 12159,
 12162, 13433, 14347, 14363, 16697,
 16698, 16699, 16700, 18446, 18447,
 18448, 18449, 25430, 39410, 39572
- \tl_gset_rescan:Nnn
 125, 12355, 12366, 12397, 12398
- .tl_gset_x:N 38817
- \tl_gsort:Nn
 123, 3218, 3220, 3221, 12872
- \tl_gtrim_spaces:N
 117, 12816, 12828, 12831
- \tl_head:N 120, 12872, 12885
- \tl_head:n
 120, 717, 725, 12872, 12872, 12882,
 12885, 13141, 25480, 31247, 31256
- \tl_head:w ... 120, 718, 719, 12872,
 12883, 31272, 31301, 31374, 37953
- \tl_if_blank:n 12582, 12590
- \tl_if_blank:nTF 112,
 120, 3393, 6587, 8783, 9100, 9102,
 9431, 10381, 10935, 11105, 11127,
 11160, 11195, 11235, 11299, 11306,
 11376, 11385, 11409, 11485, 12581,
 12889, 13131, 13416, 14125, 14128,
 15488, 15491, 18877, 18993, 22337,
 22552, 22702, 30810, 30813, 30816,
 30848, 30851, 30977, 31015, 31028,
 31048, 31080, 31267, 31295, 31376,
 32319, 32569, 32573, 33259, 33269,
 33387, 33415, 33478, 34039, 36854,
 36875, 36884, 36889, 36904, 37009,
 37238, 37454, 37462, 38090, 38093,
 38096, 38120, 38146, 38172, 38798
- \tl_if_blank_p:n ... 112, 11255, 12581
- \tl_if_empty:N 12548,
 12556, 13532, 13534, 18681, 18683
- \tl_if_empty:n
 12558, 12566, 12573, 13536
- \tl_if_empty:NTF . 112, 6973, 9130,
 9159, 9330, 9340, 10746, 10836,
 10871, 10952, 11215, 11342, 11357,
 12548, 22365, 22370, 22528, 36826,
 37184, 37466, 37828, 37849, 39077
- \tl_if_empty:nTF ... 112, 512, 708,
 710, 711, 870, 879, 885, 1683, 1779,
 2158, 4263, 4264, 5743, 7586, 7750,
 8235, 8390, 9588, 9698, 9713, 9806,
 9810, 9874, 9981, 10023, 10026,
 10058, 10086, 10087, 10097, 10104,
 10110, 10117, 10740, 10987, 11518,
 11524, 11526, 11528, 11731, 12480,
 12558, 12568, 12636, 12677, 12981,
 13242, 13277, 13490, 14397, 15452,
 16376, 16383, 16400, 16550, 16743,
 18385, 18404, 18413, 18416, 18694,
 18727, 18841, 18928, 19616, 19890,
 20691, 20695, 21706, 21798, 22751,
 23935, 24788, 25478, 29150, 29192,
 29657, 30356, 31289, 38434, 39197
- \tl_if_empty_p:N
 112, 11652, 12548, 37228
- \tl_if_empty_p:n ... 112, 12558, 12568
- \tl_if_eq:NN 12592, 12593
- \tl_if_eq:Nn 12597, 12609
- \tl_if_eq:nn 12610, 12623
- \tl_if_eq:NNTF
 112, 113, 131, 146, 738, 824,
 9679, 9733, 12592, 13281, 16860,
 31138, 36531, 36534, 36915, 37187
- \tl_if_eq:NnTF 112, 12597
- \tl_if_eq:nnTF 100,
 113, 132, 156, 187, 824, 12610, 18715
- \tl_if_eq_p:NN 112, 12592
- \tl_if_exist:N
 12181, 12182, 13528, 13530

- `\tl_if_exist:NnTF` [111](#), [3917](#), [11667](#),
[12152](#), [12154](#), [12181](#), [12766](#), [31340](#),
[32433](#), [32439](#), [33661](#), [37509](#), [37793](#)
`\tl_if_exist_p:N`
[111](#), [11651](#), [12181](#), [31675](#), [32128](#), [38446](#)
`\tl_if_head_eq_catcode:nN`
..... [718](#), [720](#), [12909](#), [12925](#)
`\tl_if_head_eq_catcode:nNTF`
..... [114](#), [12895](#)
`\tl_if_head_eq_catcode_p:nN`
..... [114](#), [12895](#)
`\tl_if_head_eq_charcode:nN`
..... [718](#), [720](#), [12895](#), [12907](#)
`\tl_if_head_eq_charcode:nNTF` ...
..... [114](#), [12895](#), [31017](#), [32508](#)
`\tl_if_head_eq_charcode_p:nN` ...
..... [114](#), [12895](#)
`\tl_if_head_eq_meaning:nN`
..... [719](#), [12927](#), [12934](#)
`\tl_if_head_eq_meaning:nNTF`
..... [114](#), [5830](#), [12895](#)
`\tl_if_head_eq_meaning_p:nN`
..... [114](#), [5662](#), [12895](#),
[31385](#), [31834](#), [31835](#), [31836](#), [31837](#)
`\tl_if_head_is_group:n` [12988](#)
`\tl_if_head_is_group:nTF`
. [114](#), [12915](#), [12955](#), [12988](#), [13040](#),
[18411](#), [31744](#), [31795](#), [32030](#), [32160](#),
[32251](#), [32370](#), [33719](#), [34089](#), [34148](#)
`\tl_if_head_is_group_p:n` . [114](#), [12988](#)
`\tl_if_head_is_N_type:n` .. [718](#), [12968](#)
`\tl_if_head_is_N_type:nTF`
..... [114](#), [12674](#),
[12898](#), [12912](#), [12929](#), [12968](#), [13202](#),
[31741](#), [31792](#), [31943](#), [32034](#), [32157](#),
[32248](#), [32367](#), [32468](#), [32716](#), [32745](#),
[32854](#), [32946](#), [33194](#), [33226](#), [33293](#),
[33428](#), [33491](#), [33534](#), [33574](#), [33716](#),
[33775](#), [33925](#), [34022](#), [34086](#), [34145](#)
`\tl_if_head_is_N_type_p:n` [114](#), [12968](#)
`\tl_if_head_is_space:n` [13003](#)
`\tl_if_head_is_space:nTF` ... [115](#),
[121](#), [13003](#), [13184](#), [13193](#), [14023](#), [20693](#)
`\tl_if_head_is_space_p:n` . [115](#), [13003](#)
`\tl_if_in:Nn` [879](#), [12629](#)
`\tl_if_in:nn` [12631](#), [12640](#)
`\tl_if_in:NnTF` [113](#), [12502](#),
[12626](#), [12626](#), [12627](#), [12628](#), [16638](#)
`\tl_if_in:nnTF` [113](#),
[709](#), [710](#), [739](#), [4462](#), [8882](#), [9574](#),
[9576](#), [10268](#), [10518](#), [12404](#), [12486](#),
[12488](#), [12626](#), [12627](#), [12628](#), [12631](#),
[13573](#), [13581](#), [19888](#), [29660](#), [36344](#)
`\tl_if_novalue:n` [12644](#)
- `\tl_if_novalue:nTF` [113](#), [12642](#)
`\tl_if_novalue_p:n` [113](#), [12642](#)
`\tl_if_single:N` [12660](#)
`\tl_if_single:n` [12661](#)
`\tl_if_single:NnTF`
..... [113](#), [12656](#), [12657](#), [12658](#), [12659](#)
`\tl_if_single:nTF` [113](#),
[588](#), [710](#), [5824](#), [5860](#), [5875](#), [5908](#),
[12657](#), [12658](#), [12659](#), [12661](#), [33515](#)
`\tl_if_single_p:N` [113](#), [12656](#)
`\tl_if_single_p:n` [113](#), [12656](#),
[12661](#), [32527](#), [33443](#), [33506](#), [33596](#)
`\tl_if_single_token:n` [12672](#)
`\tl_if_single_token:nTF`
..... [113](#), [4966](#), [12672](#), [33671](#)
`\tl_if_single_token_p:n` .. [113](#), [12672](#)
`\tl_item:Nn`
..... [121](#), [942](#), [13106](#), [13127](#), [13128](#), [20739](#)
`\tl_item:nn` [121](#), [559](#), [725](#),
[7294](#), [7338](#), [13106](#), [13106](#), [13127](#), [13132](#)
`\tl_log:N`
..... [117](#), [3914](#), [13234](#), [13236](#), [13237](#), [14164](#)
`\tl_log:n` [117](#), [401](#),
[402](#), [1097](#), [2288](#), [2304](#), [8354](#), [9756](#),
[10296](#), [10540](#), [13236](#), [13270](#), [13270](#),
[13272](#), [14160](#), [18310](#), [18363](#), [25455](#)
`\tl_lower_case:n` [38907](#), [38908](#)
`\tl_lower_case:nn` [38907](#), [38911](#)
`\tl_map_break:`
..... [61](#), [119](#), [458](#), [473](#), [3901](#),
[3902](#), [12691](#), [12705](#), [12720](#), [12731](#),
[12746](#), [12757](#), [12757](#), [12758](#), [12760](#)
`\tl_map_break:n`
..... [119](#), [3225](#), [11134](#), [12757](#), [12759](#), [37323](#)
`\tl_map_function:NN`
..... [118](#), [5787](#), [12686](#), [12693](#), [12695](#), [12781](#)
`\tl_map_function:nN` [118](#), [2157](#), [5013](#),
[12686](#), [12686](#), [12694](#), [12776](#), [16746](#)
`\tl_map_inline:Nn` [118](#), [3225](#), [12710](#),
[12723](#), [12725](#), [14260](#), [14262](#), [37319](#)
`\tl_map_inline:nn` [118](#), [119](#),
[149](#), [473](#), [3187](#), [6184](#), [6634](#), [8660](#),
[10629](#), [12710](#), [12710](#), [12724](#), [19846](#),
[19848](#), [19850](#), [24841](#), [27983](#), [29561](#),
[29569](#), [31685](#), [32056](#), [32059](#), [34257](#),
[34268](#), [34285](#), [34316](#), [34380](#), [38983](#),
[39309](#), [39371](#), [39804](#), [39865](#), [39913](#)
`\tl_map_tokens:Nn`
..... [118](#), [11133](#), [12726](#), [12733](#), [12735](#)
`\tl_map_tokens:nn`
..... [118](#), [6952](#), [12726](#), [12726](#), [12734](#), [12751](#)
`\tl_map_variable:NNn`
..... [118](#), [12750](#), [12754](#), [12756](#)

- `\tl_map_variable:nNn`
..... [119](#), [713](#), [12750](#), [12750](#), [12755](#)
- `\tl_mixed_case:n` [38907](#), [38920](#)
- `\tl_mixed_case:nn` [38907](#), [38923](#)
- `\tl_new:N` [110](#),
[111](#), [201](#), [696](#), [3114](#), [3539](#), [3558](#),
[4336](#), [4337](#), [4343](#), [4344](#), [6550](#), [6555](#),
[6556](#), [6562](#), [6563](#), [6564](#), [6820](#), [6822](#),
[6941](#), [6942](#), [7757](#), [7758](#), [7759](#), [7760](#),
[8631](#), [8867](#), [8980](#), [9173](#), [9188](#), [9236](#),
[9624](#), [9625](#), [10192](#), [10195](#), [10217](#),
[10437](#), [10448](#), [10482](#), [10603](#), [10605](#),
[10618](#), [10620](#), [10621](#), [10623](#), [10927](#),
[10957](#), [10958](#), [12132](#), [12132](#), [12137](#),
[12152](#), [12154](#), [12595](#), [12596](#), [13298](#),
[13299](#), [13300](#), [13301](#), [14171](#), [14172](#),
[16671](#), [16672](#), [18376](#), [18428](#), [18429](#),
[19214](#), [19704](#), [19906](#), [21569](#), [21573](#),
[21578](#), [21580](#), [21588](#), [21590](#), [21591](#),
[21593](#), [29983](#), [30371](#), [30372](#), [31657](#),
[31658](#), [31659](#), [31666](#), [31668](#), [31670](#),
[31706](#), [35426](#), [35451](#), [35452](#), [36462](#),
[36703](#), [36706](#), [36803](#), [36804](#), [36805](#),
[36806](#), [37181](#), [37258](#), [37385](#), [37504](#),
[39067](#), [39213](#), [39214](#), [39215](#), [40069](#)
- `\tl_put_left:Nn` [111](#), [12205](#), [12205](#),
[12210](#), [12215](#), [12220](#), [12228](#), [12267](#),
[12268](#), [12269](#), [12270](#), [12271](#), [12272](#),
[39492](#), [39493](#), [39494](#), [39495](#), [39496](#)
- `\tl_put_right:Nn` [112](#), [127](#),
[731](#), [4113](#), [4180](#), [4190](#), [4229](#), [4251](#),
[4572](#), [10877](#), [10880](#), [10885](#), [12279](#),
[12279](#), [12281](#), [12286](#), [12291](#), [12299](#),
[12335](#), [12336](#), [12337](#), [12338](#), [12339](#),
[12340](#), [16820](#), [18613](#), [19229](#), [19231](#),
[19232](#), [19233](#), [19235](#), [19237](#), [19239](#),
[19240](#), [19242](#), [19244](#), [19246](#), [19248](#),
[21604](#), [31697](#), [37985](#), [38031](#), [39070](#),
[39497](#), [39498](#), [39499](#), [39500](#), [39501](#)
- `\tl_rand_item:N`
..... [121](#), [13129](#), [13134](#), [13135](#)
- `\tl_rand_item:n`
..... [121](#), [13129](#), [13129](#), [13134](#)
- `\tl_range:Nnn` [122](#), [13136](#), [13136](#), [13137](#)
- `\tl_range:nnn`
..... [122](#), [137](#), [13136](#), [13136](#), [13138](#)
- `\tl_remove_all:Nn`
..... [124](#), [12540](#), [12540](#), [12544](#), [12545](#)
- `\tl_remove_once:Nn`
..... [124](#), [12534](#), [12534](#), [12538](#)
- `\tl_replace_all:Nnn`
..... [124](#), [821](#), [877](#), [12454](#),
[12458](#), [12470](#), [12472](#), [12541](#), [16755](#)
- `\tl_replace_once:Nnn`
[123](#), [12454](#), [12454](#), [12462](#), [12464](#), [12535](#)
- `\tl_rescan:nn`
.. [125](#), [290](#), [700](#), [12355](#), [12355](#), [12361](#)
- `\tl_reverse:N` [116](#), [13100](#), [13100](#), [13104](#)
- `\tl_reverse:n`
[116](#), [13081](#), [13081](#), [13093](#), [13101](#), [13103](#)
- `\tl_reverse_items:n` [116](#), [12800](#), [12800](#)
- `.tl_set:N` [245](#), [22181](#)
- `\tl_set:Nn` [111](#), [124](#), [125](#),
[127](#), [128](#), [153](#), [245](#), [417](#), [463](#), [625](#),
[694](#), [696](#), [703](#), [732](#), [920](#), [929](#), [4039](#),
[4933](#), [5694](#), [5771](#), [6036](#), [6099](#), [6626](#),
[6646](#), [6656](#), [6672](#), [6677](#), [6720](#), [6752](#),
[6790](#), [7140](#), [7811](#), [7812](#), [7872](#), [8872](#),
[8907](#), [9145](#), [9369](#), [9590](#), [9636](#), [9719](#),
[10340](#), [10353](#), [10386](#), [10694](#), [10731](#),
[11057](#), [11094](#), [11208](#), [11311](#), [11314](#),
[11317](#), [11320](#), [11349](#), [11602](#), [11605](#),
[11606](#), [11607](#), [11608](#), [11615](#), [11616](#),
[11617](#), [11619](#), [11623](#), [12189](#), [12189](#),
[12191](#), [12197](#), [12198](#), [12199](#), [12200](#),
[12365](#), [12600](#), [12613](#), [12614](#), [12753](#),
[13259](#), [13280](#), [14418](#), [14568](#), [16745](#),
[16749](#), [16789](#), [16856](#), [16865](#), [16888](#),
[17010](#), [17013](#), [17030](#), [17038](#), [17057](#),
[17066](#), [17125](#), [17256](#), [17888](#), [18537](#),
[18543](#), [18552](#), [18559](#), [18802](#), [19227](#),
[19779](#), [19945](#), [20080](#), [20126](#), [20232](#),
[20240](#), [20359](#), [20365](#), [20375](#), [20381](#),
[20439](#), [20658](#), [21020](#), [21581](#), [21774](#),
[21843](#), [21989](#), [22216](#), [22433](#), [25825](#),
[30505](#), [30515](#), [31068](#), [31121](#), [31140](#),
[31151](#), [31156](#), [31660](#), [31667](#), [31669](#),
[31671](#), [31702](#), [32053](#), [32611](#), [32627](#),
[32643](#), [34254](#), [35697](#), [36345](#), [36346](#),
[36467](#), [36704](#), [36733](#), [36809](#), [36836](#),
[36843](#), [36860](#), [36868](#), [36871](#), [36921](#),
[36936](#), [37192](#), [37198](#), [37199](#), [37203](#),
[37247](#), [37255](#), [37259](#), [37395](#), [37403](#),
[37421](#), [37424](#), [37465](#), [37481](#), [37505](#),
[37514](#), [37534](#), [37568](#), [37570](#), [37572](#),
[37575](#), [37592](#), [37786](#), [37974](#), [40067](#)
- `.tl_set_e:N` [245](#), [22181](#)
- `\tl_set_eq:NN` [111](#),
[185](#), [585](#), [3218](#), [6814](#), [7237](#), [7730](#),
[8301](#), [9682](#), [9690](#), [12157](#), [12157](#),
[12161](#), [13432](#), [14345](#), [14354](#), [16693](#),
[16694](#), [16695](#), [16696](#), [18442](#), [18443](#),
[18444](#), [18445](#), [21682](#), [21842](#), [22321](#),
[22345](#), [22422](#), [22443](#), [22493](#), [25429](#),
[31135](#), [36723](#), [36838](#), [36935](#), [37467](#),
[37487](#), [37510](#), [37829](#), [39409](#), [39491](#)
- `\tl_set_rescan:Nnn` . [125](#), [290](#), [665](#),

- [701](#), [12355](#), [12357](#), [12364](#), [12395](#), [12396](#)
`.tl_set_x:N` [38817](#)
`\tl_show:N` [117](#), [185](#),
[459](#), [3912](#), [13234](#), [13234](#), [13235](#), [14157](#)
`\tl_show:n` . [87](#), [117](#), [401](#), [402](#), [628](#),
[728](#), [1097](#), [2284](#), [2301](#), [8352](#), [9754](#),
[10294](#), [10538](#), [13234](#), [13254](#), [13254](#),
[13256](#), [14153](#), [18308](#), [18362](#), [19067](#),
[19137](#), [19143](#), [19149](#), [19155](#), [25453](#)
`\tl_sort:Nn` [123](#), [3218](#), [3218](#), [3219](#), [12872](#)
`\tl_sort:nN`
..... [123](#), [438](#), [439](#), [3389](#), [3389](#), [12872](#)
`\tl_tail:N`
... [120](#), [766](#), [5637](#), [12872](#), [12894](#),
[14563](#), [20715](#), [20723](#), [20733](#), [39076](#)
`\tl_tail:n`
..... [120](#), [12872](#), [12886](#), [12893](#), [12894](#)
`\tl_to_str:N`
[98](#), [115](#), [129](#), [551](#), [654](#), [736](#), [10689](#),
[10700](#), [11566](#), [11580](#), [12762](#), [12762](#),
[12763](#), [13285](#), [13286](#), [13498](#), [13565](#),
[13573](#), [14156](#), [14163](#), [14725](#), [19040](#)
`\tl_to_str:n`
[52](#), [54](#), [77](#), [98](#), [115](#), [125](#), [129](#), [139](#),
[140](#), [213](#), [215](#), [216](#), [240](#), [371](#), [383](#),
[551](#), [707](#), [711](#), [736](#), [743](#), [749](#), [903](#),
[939](#), [995](#), [1237](#), [1418](#), [1418](#), [1441](#),
[1668](#), [1755](#), [2339](#), [2713](#), [2727](#), [2730](#),
[2737](#), [2741](#), [3011](#), [3043](#), [3061](#), [5013](#),
[5970](#), [7097](#), [7256](#), [7332](#), [7355](#), [7429](#),
[8340](#), [8346](#), [8877](#), [9100](#), [9106](#), [9465](#),
[9466](#), [9602](#), [9604](#), [9608](#), [9610](#), [9615](#),
[9617](#), [10155](#), [10156](#), [10157](#), [10158](#),
[10263](#), [10513](#), [10611](#), [10627](#), [10992](#),
[11110](#), [11121](#), [11187](#), [12377](#), [12483](#),
[12560](#), [12761](#), [12761](#), [13255](#), [13271](#),
[13499](#), [13573](#), [13581](#), [13732](#), [13754](#),
[13778](#), [13785](#), [13839](#), [13846](#), [13920](#),
[13939](#), [13950](#), [13975](#), [13983](#), [13991](#),
[13997](#), [14009](#), [14020](#), [14122](#), [14133](#),
[14259](#), [14372](#), [14377](#), [14382](#), [14444](#),
[15464](#), [16623](#), [16634](#), [18165](#), [18182](#),
[18226](#), [18315](#), [18369](#), [19433](#), [19437](#),
[19467](#), [19468](#), [19502](#), [19517](#), [19519](#),
[19521](#), [19610](#), [19883](#), [19888](#), [19922](#),
[20204](#), [20258](#), [20260](#), [20277](#), [20305](#),
[20307](#), [20441](#), [20459](#), [20527](#), [20666](#),
[20672](#), [20673](#), [20860](#), [21060](#), [21245](#),
[21651](#), [21812](#), [22215](#), [22615](#), [22689](#),
[22691](#), [22697](#), [22698](#), [23172](#), [23369](#),
[23373](#), [23390](#), [23584](#), [23585](#), [24198](#),
[24199](#), [24204](#), [24208](#), [28906](#), [28960](#),
[29034](#), [29441](#), [29448](#), [29449](#), [29627](#),
[29735](#), [29750](#), [29770](#), [29787](#), [29821](#),
[29886](#), [29897](#), [29966](#), [30057](#), [30224](#),
[30262](#), [30361](#), [30580](#), [31496](#), [32110](#),
[32115](#), [32120](#), [32433](#), [32436](#), [32439](#),
[32442](#), [36488](#), [36561](#), [36815](#), [37438](#),
[37631](#), [38237](#), [38768](#), [38790](#), [38793](#),
[38976](#), [38978](#), [39033](#), [39041](#), [39328](#),
[39331](#), [39340](#), [39341](#), [39342](#), [39351](#),
[39353](#), [39386](#), [39388](#), [39798](#), [40034](#)
`\tl_trim_spaces:N`
..... [117](#), [12816](#), [12826](#), [12830](#)
`\tl_trim_spaces:n` . [117](#), [716](#), [1000](#),
[11040](#), [12816](#), [12816](#), [12822](#), [12827](#),
[12829](#), [16734](#), [16736](#), [31369](#), [38798](#)
`\tl_trim_spaces_apply:nN` ... [117](#),
[971](#), [11037](#), [12816](#), [12823](#), [12825](#),
[18387](#), [18844](#), [18932](#), [19006](#), [30224](#)
`\tl_upper_case:n` [38907](#), [38914](#)
`\tl_upper_case:nn` [38907](#), [38917](#)
`\tl_use:N` [115](#),
[229](#), [234](#), [237](#), [8975](#), [9168](#), [12764](#),
[12764](#), [12772](#), [21904](#), [31136](#), [31143](#),
[31148](#), [31152](#), [31157](#), [31160](#), [31178](#),
[31179](#), [31321](#), [31341](#), [39326](#), [39330](#)
`\g_tmpa_tl` [126](#), [13298](#)
`\l_tmpa_tl` [6](#), [59](#), [124](#), [126](#),
[1239](#), [1241](#), [1258](#), [1283](#), [1285](#), [1289](#),
[1291](#), [1295](#), [1297](#), [1301](#), [1303](#), [13300](#)
`\g_tmpb_tl` [126](#), [13298](#)
`\l_tmpb_tl` [126](#), [1240](#),
[1241](#), [1256](#), [1258](#), [1284](#), [1285](#), [1290](#),
[1291](#), [1296](#), [1297](#), [1302](#), [1303](#), [13300](#)
tl internal commands:
`__tl_act:NNNn`
..... [722](#), [723](#), [12790](#), [13019](#), [13070](#), [13086](#)
`__tl_act_count_group:n` [12792](#), [12799](#)
`__tl_act_count_group:nn` [12786](#)
`__tl_act_count_normal:N` [12791](#), [12797](#)
`__tl_act_count_normal:nN` [12786](#)
`__tl_act_count_space:` . [12793](#), [12798](#)
`__tl_act_count_space:n` [12786](#)
`__tl_act_end:wn`
..... [715](#), [13019](#), [13054](#), [13058](#)
`__tl_act_group:nwNNN`
..... [13019](#), [13041](#), [13056](#)
`__tl_act_if_head_is_space:nTF` ..
..... [722](#), [13019](#), [13021](#), [13037](#), [13046](#)
`__tl_act_if_head_is_space:w` ...
..... [13019](#), [13023](#), [13027](#)
`__tl_act_if_head_is_space_-true:w` [13019](#), [13024](#), [13030](#)
`__tl_act_loop:w` [722](#), [13019](#),
[13035](#), [13050](#), [13060](#), [13067](#), [13073](#)
`__tl_act_normal:NwNNN`
..... [13019](#), [13042](#), [13047](#)

- _tl_act_output:n . 723, 13019, 13077
- _tl_act_result:n . . . 723, 13054, 13075, 13077, 13078, 13079, 13080
- _tl_act_reverse 723
- _tl_act_reverse_output:n 13019, 13079, 13095, 13097, 13099
- _tl_act_space:wwNNN 722, 13019, 13038, 13064
- _tl_analysis:n 449, 459, 3581, 3581, 3891, 3919, 3931
- _tl_analysis_a:n . . 3585, 3634, 3634
- _tl_analysis_a_bgroup:w 3665, 3687, 3689
- _tl_analysis_a_cs:ww 3744, 3758, 3761
- _tl_analysis_a_egroup:w 3667, 3687, 3692
- _tl_analysis_a_group:nw 3687, 3690, 3693, 3695
- _tl_analysis_a_group_aux:w 3687, 3703, 3705
- _tl_analysis_a_group_auxii:w 3687, 3710, 3713
- _tl_analysis_a_group_test:w 3687, 3715, 3720
- _tl_analysis_a_loop:w . . . 3641, 3644, 3644, 3685, 3727, 3741, 3759
- _tl_analysis_a_safe:N 3666, 3708, 3744, 3744
- _tl_analysis_a_space:w 3664, 3670, 3670
- _tl_analysis_a_space_test:w 452, 3670, 3672, 3677
- _tl_analysis_a_store: 452, 3681, 3723, 3729, 3729
- _tl_analysis_a_type:w 3645, 3646, 3646
- _tl_analysis_b:n . . 3586, 3772, 3772
- _tl_analysis_b_char:Nn 465, 3799, 3806, 3806, 4114
- _tl_analysis_b_char_aux:nww 456, 3800, 3806, 3828
- _tl_analysis_b_cs:Nww 3802, 3834, 3834
- _tl_analysis_b_cs_test:ww 3834, 3837, 3839
- _tl_analysis_b_loop:w 458, 3772, 3776, 3780, 3880, 3885
- _tl_analysis_b_normal:wwN 3785, 3790, 3792, 3855
- _tl_analysis_b_normals:ww 456, 457, 3782, 3785, 3785, 3831, 3841
- _tl_analysis_b_special:w 3788, 3852, 3854
- _tl_analysis_b_special_char:wN 3852, 3869, 3877
- _tl_analysis_b_special_space:w 3852, 3871, 3882
- _tl_analysis_char_arg:Nw 4010, 4010, 4167, 4226
- _tl_analysis_char_arg_aux:Nw 4010, 4013, 4016
- \l_tl_analysis_char_token . 446, 452, 453, 3537, 3674, 3679, 3717, 3722
- _tl_analysis_cs_space_count:NN 3565, 3565, 3758, 3837
- _tl_analysis_cs_space_count:w 3565, 3569, 3573, 3577
- _tl_analysis_cs_space_count_end:w 3565, 3571, 3579
- _tl_analysis_disable:n 450, 3590, 3592, 3601, 3636, 3702
- _tl_analysis_disable_char:N 3610, 3612, 3623, 3755
- _tl_analysis_extract_charcode: 3559, 3559, 3697, 4141
- _tl_analysis_extract_charcode_aux:w 3559, 3561, 3564
- \l_tl_analysis_index_int 454, 455, 3555, 3639, 3642, 3680, 3698, 3735, 3738, 3764, 3766, 3858
- _tl_analysis_map:Nn 3887, 3893, 3896
- _tl_analysis_map:NwNw 3887, 3899, 3905, 3909
- \l_tl_analysis_nesting_int 451, 3556, 3640, 3731, 3740
- \l_tl_analysis_normal_int 3554, 3638, 3683, 3725, 3736, 3739, 3756, 3765, 3770
- \g_tl_analysis_result_tl 458, 3558, 3774, 3900, 3936
- _tl_analysis_show: 3921, 3932, 3934, 3934
- _tl_analysis_show:Nn 3925, 3926, 3928, 3929
- _tl_analysis_show:NNN 3911, 3912, 3914, 3915
- _tl_analysis_show_active:n 3949, 3978, 3980
- _tl_analysis_show_cs:n 3945, 3978, 3978
- \c_tl_analysis_show_etc_str 461, 3998, 4000, 4259
- _tl_analysis_show_long:nn 3978, 3979, 3981, 3982
- _tl_analysis_show_long_aux:nnnn 3978, 3984, 3989, 4004

- _tl_analysis_show_loop:wNw . . .
 [3934](#), [3936](#), [3940](#), [3956](#)
- _tl_analysis_show_normal:n . . .
 [3952](#), [3958](#), [3958](#)
- _tl_analysis_show_value:N
 [3963](#), [3963](#), [3987](#)
- \l_tl_analysis_token
 [446](#), [447](#), [451](#)–
 [453](#), [462](#), [3537](#), [3562](#), [3645](#), [3649](#),
 [3652](#), [3655](#), [3703](#), [3707](#), [3722](#), [4012](#),
 [4139](#), [4148](#), [4153](#), [4162](#), [4222](#), [4245](#)
- \l_tl_analysis_type_int . . . [451](#),
 [454](#), [3557](#), [3648](#), [3663](#), [3731](#), [3733](#), [3737](#)
- _tl_build_begin:NN
 . . . [13306](#), [13307](#), [13309](#), [13310](#), [13341](#)
- _tl_build_begin:NNN
 [730](#), [13306](#), [13311](#), [13312](#)
- _tl_build_end_loop:NN
 . . . [13370](#), [13373](#), [13378](#), [13380](#), [13386](#)
- _tl_build_get:NNN
 . . . [13372](#), [13377](#), [13389](#), [13390](#), [13390](#)
- _tl_build_get:w
 [13390](#), [13391](#), [13392](#), [13398](#)
- _tl_build_get_end:w
 [13390](#), [13396](#), [13400](#)
- _tl_build_last:NNn
 [730](#), [731](#), [13318](#), [13321](#),
 [13333](#), [13337](#), [13351](#), [13352](#), [13392](#)
- _tl_build_put:nn
 [731](#), [13321](#), [13348](#), [13350](#), [13365](#)
- _tl_build_put:nw
 [731](#), [13321](#), [13350](#), [13351](#)
- _tl_build_put_left:NNn
 [13353](#), [13354](#), [13357](#), [13359](#)
- _tl_count:n
 [714](#), [12773](#), [12776](#), [12781](#), [12783](#)
- _tl_head_aux:n [12875](#), [12877](#)
- _tl_head_aux:nw [12872](#)
- _tl_head_auxii:n [12872](#)
- _tl_head_exp_not:w
 [720](#), [12895](#), [12899](#), [12913](#), [12964](#)
- _tl_if_blank_p:NNw [12581](#)
- _tl_if_empty_if:n
 . . . [708](#), [811](#), [12568](#), [12568](#), [12575](#),
 [12584](#), [12647](#), [12675](#), [12679](#), [13015](#)
- _tl_if_head_eq_empty_arg:w . . .
 . . . [718](#), [720](#), [12895](#), [12899](#), [12913](#), [12966](#)
- _tl_if_head_eq_meaning_
 normal:nN [12930](#), [12936](#)
- _tl_if_head_eq_meaning_
 special:nN [12931](#), [12945](#)
- _tl_if_head_is_group_fi_
 false:w [12988](#), [12994](#), [13002](#)
- _tl_if_head_is_N_type_auxi:w . . .
 [720](#), [12968](#), [12971](#), [12979](#)
- _tl_if_head_is_N_type_auxii:n . . .
 [12968](#), [12983](#), [12986](#)
- _tl_if_head_is_space:w
 [13003](#), [13006](#), [13013](#)
- _tl_if_novalue:w
 [710](#), [12642](#), [12647](#), [12653](#)
- _tl_if_recursion_tail_break:nN
 [724](#), [12352](#), [12352](#), [13122](#)
- _tl_if_recursion_tail_stop:nTF
 [12352](#)
- _tl_if_recursion_tail_stop_p:n
 [12352](#)
- _tl_if_single:nnw
 [711](#), [12661](#), [12663](#), [12671](#)
- \l_tl_internal_a_tl [728](#),
 [4221](#), [4229](#), [4236](#), [4237](#), [4253](#), [12357](#),
 [12359](#), [12363](#), [12595](#), [12613](#), [12617](#),
 [13259](#), [13265](#), [13280](#), [13281](#), [13286](#)
- \l_tl_internal_b_tl
 . . . [12595](#), [12600](#), [12603](#), [12614](#), [12617](#)
- _tl_item:nn
 [13106](#), [13108](#), [13120](#), [13125](#)
- _tl_item_aux:nn [13106](#), [13109](#), [13114](#)
- _tl_map_function:Nnnnnnnnn . . .
 . . . [712](#), [12686](#), [12688](#), [12696](#), [12701](#), [12715](#)
- _tl_map_function_end:w
 [712](#), [12686](#), [12699](#), [12703](#), [12707](#)
- _tl_map_tokens:nnnnnnnn
 [12726](#), [12728](#), [12736](#), [12742](#)
- _tl_map_tokens_end:w
 [12726](#), [12739](#), [12744](#), [12748](#)
- _tl_map_variable:Nnn
 [12750](#), [12751](#), [12752](#)
- _tl_peek_analysis_active_str:n
 [4017](#), [4176](#), [4178](#)
- _tl_peek_analysis_char:N
 [465](#), [4017](#), [4095](#), [4105](#)
- _tl_peek_analysis_char:w
 [465](#), [4017](#), [4118](#), [4126](#)
- _tl_peek_analysis_collect:n . . .
 [4017](#), [4226](#), [4227](#)
- _tl_peek_analysis_collect:w . . .
 [4017](#), [4223](#), [4225](#), [4246](#)
- _tl_peek_analysis_collect_
 end:NNNN [4017](#), [4243](#), [4248](#)
- _tl_peek_analysis_collect_
 loop: [4017](#), [4230](#), [4232](#)
- _tl_peek_analysis_collect_
 test: [4017](#)
- _tl_peek_analysis_cs:N
 [4017](#), [4097](#), [4101](#)

- _tl_peek_analysis_escape: 4017, 4174, 4219
- _tl_peek_analysis_exp:N 4017, 4058, 4066
- _tl_peek_analysis_exp_aux:N . 4017
- _tl_peek_analysis_exp_aux:Nw 4076, 4081
- _tl_peek_analysis_explicit:n 4017, 4173, 4188
- _tl_peek_analysis_loop:NNn 4017, 4027, 4034, 4036
- _tl_peek_analysis_nonexp:N 4017, 4061, 4089, 4154
- _tl_peek_analysis_retest: 464, 465, 4017, 4149, 4151
- _tl_peek_analysis_special: 4017, 4063, 4137
- _tl_peek_analysis_str: 4017, 4156, 4159
- _tl_peek_analysis_str:n 4017, 4167, 4168
- _tl_peek_analysis_str:w 4017, 4163, 4166
- _tl_peek_analysis_test: 463, 4017, 4045, 4047
- \c_tl_peek_catcodes_tl 3540
- \l_tl_peek_charcode_int 4009, 4140, 4142, 4145, 4172
- \l_tl_peek_code_tl 463, 3539, 4039, 4068, 4071, 4079, 4102, 4113, 4122, 4180, 4186, 4190, 4217, 4251, 4257
- _tl_quark_if_nil:n 12353
- _tl_quark_if_nil:nTF 12491
- _tl_range:Nnnn . 13136, 13138, 13139
- _tl_range:nnNn . 13136, 13149, 13159
- _tl_range:nnnNn 13136, 13143, 13147
- _tl_range:w 725, 13136, 13138, 13177
- _tl_range_braced:w 725
- _tl_range_collect:nn . . . 13136, 13179, 13188, 13195, 13200, 13214
- _tl_range_collect_braced:w . . 725
- _tl_range_collect_group:nN . 13136
- _tl_range_collect_group:nn 13204, 13213
- _tl_range_collect_N:nN 13136, 13203, 13212
- _tl_range_collect_space:nw 13136, 13196, 13211
- _tl_range_items:nnNn 725
- _tl_range_normalize:nn 13151, 13155, 13215, 13215
- _tl_range_skip:w 725, 13136, 13166, 13168, 13171
- _tl_range_skip_spaces:n 13136, 13180, 13182, 13185
- _tl_replace:NnNNNnn 703, 705, 12455, 12457, 12459, 12461, 12478, 12478, 12489
- _tl_replace_auxi:NnnNNNnn 705, 12478, 12492, 12493, 12500, 12503
- _tl_replace_auxii:nNNNnn 704, 705, 12478, 12496, 12504, 12506
- _tl_replace_next:w 703, 705, 706, 12459, 12461, 12478, 12511, 12531, 12533
- _tl_replace_next_aux:w 12478, 12520, 12531
- _tl_replace_wrap:w 703, 705, 706, 12455, 12457, 12478, 12509, 12513, 12532
- _tl_rescan:NNw 700, 12355, 12383, 12390, 12440, 12445
- _tl_rescan_aux: 12355, 12358, 12362
- \c_tl_rescan_marker_tl 702, 12354, 12382, 12390, 12420, 12452
- _tl_reverse_group_preserve:n 13088, 13096
- _tl_reverse_group_preserve:mn 13081
- _tl_reverse_items:nwNwn 12800, 12802, 12803, 12807, 12810
- _tl_reverse_items:wn 12800, 12804, 12811, 12814
- _tl_reverse_normal:N . 13087, 13094
- _tl_reverse_normal:nN 13081
- _tl_reverse_space: . . 13089, 13098
- _tl_reverse_space:n 13081
- _tl_set_rescan:nNN 700, 702, 12377, 12399, 12399
- _tl_set_rescan:NNnn 700, 12355, 12365, 12367, 12368
- _tl_set_rescan_multi:nNN 700, 702, 12355, 12380, 12407, 12429
- _tl_set_rescan_single:nnNN 702, 12399, 12410, 12414, 12426
- _tl_set_rescan_single:NNww . . 702
- _tl_set_rescan_single_aux:nnnNN 12399, 12419, 12432
- _tl_set_rescan_single_aux:w 702, 12399, 12437, 12451
- _tl_show:n 13254, 13255, 13257
- _tl_show:NN 13234, 13234, 13236, 13238
- _tl_show:w 13254, 13259, 13269
- _tl_tl_head:w . 12872, 12884, 12939

- __tl_tmp:w 710,
716, 12635, 12636, 12642, 12655,
12832, 12871, 13019, 13034, 13302
- __tl_trim_mark: 715, 716,
12819, 12824, 12832, 12839, 12840,
12847, 12851, 12853, 12856, 12869
- __tl_trim_spaces:nn
.... 971, 12818, 12824, 12832, 12834
- __tl_trim_spaces_auxi:w
716, 12832, 12836, 12847, 12850, 12856
- __tl_trim_spaces_auxii:w
..... 716, 12832, 12840, 12855
- __tl_trim_spaces_auxiii:w
716, 12832, 12841, 12858, 12861, 12865
- __tl_trim_spaces_auxiv:w
..... 716, 12832, 12843, 12867
- __tl_use_none_delimit_by_q_act_-
stop:w 13019
- __tl_use_none_delimit_by_s_act_-
stop:w 13053, 13058
- __tl_use_none_delimit_by_s_-
stop:w 712, 12686,
12698, 12705, 12709, 12738, 12746
- \tojis 1192
- token commands:
 - \c_alignment_token
..... 200, 899, 1429, 3816,
19299, 19334, 19373, 31503, 37287
 - \c_catcode_letter_token 200,
900, 3812, 19290, 19334, 19402, 31515
 - \c_catcode_other_token 200,
900, 3810, 19293, 19334, 19407, 31518
 - \c_group_begin_token 200
 - \c_group_end_token 200
 - \c_math_subscript_token
..... 200, 900, 3820, 19308,
19334, 19392, 31474, 31509, 37302
 - \c_math_superscript_token
..... 200, 899, 3818,
19305, 19334, 19387, 31506, 37303
 - \c_math_toggle_token
..... 200, 899, 3814,
19296, 19334, 19368, 31471, 31500
 - \c_parameter_token
.. 200, 544, 899, 19334, 19377, 19380
 - \c_space_token
..... 40, 115, 126, 200, 207,
718, 900, 3649, 3679, 3822, 4012,
4052, 4192, 4640, 4681, 6990, 10801,
12917, 12957, 19317, 19334, 19397,
19730, 19755, 19868, 31475, 31512
 - \token_case_catcode:Nn
..... 205, 19661, 19661
 - \token_case_catcode:NnTF
..... 205, 19661, 19663, 19665, 19667, 37300
 - \token_case_charcode:Nn
..... 205, 19661, 19669
 - \token_case_charcode:NnTF
.... 205, 19661, 19671, 19673, 19675
 - \token_case_meaning:Nn
.... 205, 19661, 19677, 38925, 38926
 - \token_case_meaning:NnTF
..... 205, 5861,
5876, 5909, 5919, 5930, 8360, 19661,
19679, 19681, 19683, 37307, 38927,
38928, 38929, 38930, 38931, 38932
 - \token_if_active:N 19410
 - \token_if_active:NnTF 202, 19410
 - \token_if_active_p:N . 202, 19410,
31828, 32001, 32485, 32528, 34202
 - \token_if_alignment:N 19371
 - \token_if_alignment:NnTF .. 202, 19371
 - \token_if_alignment_p:N .. 202, 19371
 - \token_if_chardef:NnTF 203, 3967, 19480
 - \token_if_chardef_p:N
..... 203, 19480, 31437
 - \token_if_cs:N 19444
 - \token_if_cs:NnTF
..... 203, 19320, 19444, 31824,
32282, 32727, 32755, 32865, 32953,
33203, 33749, 33940, 34030, 34216
 - \token_if_cs_p:N
..... 203, 19444, 32000, 33444,
33507, 33542, 33597, 33791, 34201
 - \token_if_dim_register:NnTF
..... 204, 3969, 19480
 - \token_if_dim_register_p:N 204, 19480
 - \token_if_eq_catcode:NN 19417
 - \token_if_eq_catcode:NnTF
..... 202, 205,
206, 19417, 19662, 19664, 19666, 19668
 - \token_if_eq_catcode_p:NN 202, 19417
 - \token_if_eq_charcode:NN 19422
 - \token_if_eq_charcode:NnTF
..... 203, 205,
206, 4681, 4686, 5322, 5522, 5535,
5537, 5575, 5713, 6976, 6990, 9929,
10801, 19422, 19670, 19672, 19674,
19676, 21102, 21122, 21125, 27506
 - \token_if_eq_charcode_p:NN 203, 19422
 - \token_if_eq_meaning:NN 19415
 - \token_if_eq_meaning:NnTF
..... 203, 205,
206, 5020, 5027, 5328, 5361, 5510,
5533, 5565, 5700, 5708, 5711, 7045,
7051, 7078, 7085, 7103, 7126, 7143,
10853, 19415, 19678, 19680, 19682,

- 19684, 23705, 24716, 24775, 25506,
 25773, 25775, 25780, 25844, 26030,
 28103, 29465, 29488, 29613, 29688,
 31780, 31805, 31807, 31976, 32014,
 32236, 32262, 34117, 34158, 37321
 \token_if_eq_meaning_p:NN
 [203](#), [19415](#), [31538](#)
 \token_if_expandable:N [19449](#)
 \token_if_expandable:NTF
 [203](#), [3965](#), [19449](#), [31532](#)
 \token_if_expandable_p:N . [203](#), [19449](#)
 \token_if_font_selection:NTF ...
 [204](#), [19480](#)
 \token_if_font_selection_p:N ...
 [204](#), [19480](#)
 \token_if_group_begin:N [19356](#)
 \token_if_group_begin:NTF [201](#), [19356](#)
 \token_if_group_begin_p:N [201](#), [19356](#)
 \token_if_group_end:N [19361](#)
 \token_if_group_end:NTF .. [201](#), [19361](#)
 \token_if_group_end_p:N .. [201](#), [19361](#)
 \token_if_int_register:NTF
 [204](#), [3970](#), [19480](#)
 \token_if_int_register_p:N [204](#), [19480](#)
 \token_if_letter:N [902](#), [19400](#)
 \token_if_letter:NTF [202](#), [19400](#)
 \token_if_letter_p:N [202](#), [19400](#), [32482](#)
 \token_if_long_macro:NTF . [203](#), [19480](#)
 \token_if_long_macro_p:N . [203](#), [19480](#)
 \token_if_macro:N [19429](#)
 \token_if_macro:NTF
 . [203](#), [2344](#), [2353](#), [2362](#), [19427](#), [19605](#)
 \token_if_macro_p:N [203](#), [19427](#)
 \token_if_math_subscript:N ... [19390](#)
 \token_if_math_subscript:NTF ...
 [202](#), [19390](#)
 \token_if_math_subscript_p:N ...
 [202](#), [19390](#)
 \token_if_math_superscript:N . [19384](#)
 \token_if_math_superscript:NTF ..
 [202](#), [19384](#)
 \token_if_math_superscript_p:N ..
 [202](#), [19384](#)
 \token_if_math_toggle:N [19366](#)
 \token_if_math_toggle:NTF [202](#), [19366](#)
 \token_if_math_toggle_p:N [202](#), [19366](#)
 \token_if_mathchardef:NTF
 [204](#), [3968](#), [19480](#)
 \token_if_mathchardef_p:N
 [204](#), [19480](#), [31438](#)
 \token_if_muskip_register:NTF ...
 [204](#), [19480](#)
 \token_if_muskip_register_p:N ...
 [204](#), [19480](#)
 \token_if_other:N [19405](#)
 \token_if_other:NTF [202](#), [19405](#)
 \token_if_other_p:N [202](#), [19405](#)
 \token_if_parameter:N [19378](#)
 \token_if_parameter:NTF .. [202](#), [19376](#)
 \token_if_parameter_p:N .. [202](#), [19376](#)
 \token_if_primitive:N . [19593](#), [19602](#)
 \token_if_primitive:NTF .. [204](#), [19529](#)
 \token_if_primitive_p:N .. [204](#), [19529](#)
 \token_if_protected_long_
 macro:NTF [203](#), [19480](#)
 \token_if_protected_long_macro_
 p:N [203](#), [19480](#), [31537](#), [31690](#), [31833](#)
 \token_if_protected_macro:NTF ...
 [203](#), [19480](#)
 \token_if_protected_macro_p:N ...
 [203](#), [19480](#), [31536](#), [31689](#), [31832](#)
 \token_if_skip_register:NTF
 [204](#), [3971](#), [19480](#)
 \token_if_skip_register_p:N
 [204](#), [19480](#)
 \token_if_space:N [19395](#)
 \token_if_space:NTF [202](#), [19395](#)
 \token_if_space_p:N [202](#), [19395](#)
 \token_if_toks_register:NTF
 [204](#), [3972](#), [19480](#)
 \token_if_toks_register_p:N
 [204](#), [19480](#)
 \token_to_catcode:N [201](#), [19286](#), [19286](#)
 \token_to_meaning:N
 [20](#), [201](#), [211](#), [901](#), [904](#),
[1416](#), [1416](#), [1432](#), [1443](#), [1984](#), [2347](#),
[2356](#), [2365](#), [2727](#), [3562](#), [3961](#), [3986](#),
[4969](#), [8367](#), [13250](#), [13291](#), [19286](#),
[19433](#), [19501](#), [19609](#), [19871](#), [31480](#)
 \token_to_str:N
 [7](#), [22](#), [98](#), [129](#), [201](#), [211](#),
[388](#), [464–466](#), [663](#), [720](#), [721](#), [848](#),
[903](#), [1058](#), [1060](#), [1418](#), [1419](#), [1432](#),
[1432](#), [1646](#), [1655](#), [1687](#), [1710](#), [1763](#),
[1768](#), [1783](#), [1804](#), [1805](#), [1825](#), [1984](#),
[2125](#), [2161](#), [2168](#), [2280](#), [2300](#), [2313](#),
[2747](#), [2832](#), [2847](#), [2862](#), [2869](#), [2895](#),
[2904](#), [2941](#), [3007](#), [3028](#), [3486](#), [3502](#),
[3548](#), [3549](#), [3550](#), [3551](#), [3570](#), [3675](#),
[3718](#), [3748](#), [3797](#), [3809](#), [3811](#), [3813](#),
[3823](#), [3863](#), [3874](#), [3921](#), [3960](#), [3985](#),
[4077](#), [4084](#), [4093](#), [4164](#), [4184](#), [4193](#),
[4260](#), [4583](#), [4590](#), [4700](#), [4704](#), [5440](#),
[6971](#), [7268](#), [7331](#), [7354](#), [7428](#), [8053](#),
[8257](#), [8259](#), [8362](#), [8363](#), [8367](#), [8425](#),
[8868](#), [8988](#), [10305](#), [10307](#), [10549](#),
[10551](#), [10673](#), [10674](#), [10675](#), [10676](#),
[10677](#), [10684](#), [10989](#), [11006](#), [11053](#),

- 12186, 12354, 12972, 12992, 13246,
13250, 13285, 13291, 13692, 14233,
14241, 14244, 16455, 16479, 16483,
16498, 16641, 16910, 16975, 17432,
17678, 19034, 19040, [19286](#), 19515,
19516, 19521, 19523, 19524, 19525,
19526, 19527, 20101, 20119, 20679,
20689, 20708, 20709, 20716, 20717,
20718, 22874, 22922, 23041, 23083,
23193, 23368, 23383, 23590, 23591,
24082, 24083, 24112, 24279, 24330,
24362, 24382, 24397, 24409, 24410,
24423, 24424, 24449, 24458, 24460,
24485, 24488, 24513, 24515, 24529,
24545, 24563, 24633, 24643, 24644,
24659, 24660, 24987, 25029, 25221,
25461, 29439, 30031, 30261, 30325,
30342, 30575, 30620, 30626, 31454,
31455, 31997, 32005, 32052, 32053,
32333, 32336, 32398, 32401, 32410,
32610, 32611, 33638, 33640, 33658,
33659, 33674, 33677, 33681, 33684,
34198, 34206, 34253, 34254, 34370,
34373, 34377, 34386, 34413, 34776,
35476, 35696, 36631, 38767, 38790,
38793, 39098, 39107, 39656, 39666
- token internal commands:
- `\c_token_A_int` [19599](#), [19636](#)
 - `__token_case:NNnTF`
..... [19661](#), [19662](#), [19664](#), [19666](#),
[19668](#), [19670](#), [19672](#), [19674](#), [19676](#),
[19678](#), [19680](#), [19682](#), [19684](#), [19685](#)
 - `__token_case:NNw`
..... [19661](#), [19687](#), [19692](#), [19696](#)
 - `__token_case_end:nw`
..... [19661](#), [19695](#), [19698](#)
 - `__token_delimit_by_ufont:w` .. [19461](#)
 - `__token_delimit_by_char":w` .. [19461](#)
 - `__token_delimit_by_count:w` .. [19461](#)
 - `__token_delimit_by_dimen:w` .. [19461](#)
 - `__token_delimit_by_macro:w` .. [19461](#)
 - `__token_delimit_by_muskip:w` .. [19461](#)
 - `__token_delimit_by_skip:w` ... [19461](#)
 - `__token_delimit_by_toks:w` ... [19461](#)
 - `__token_if_macro_p:w`
..... [19427](#), [19432](#), [19436](#)
 - `__token_if_primitive:NNw`
..... [19529](#), [19608](#), [19613](#)
 - `__token_if_primitive:Nw`
..... [19529](#), [19637](#), [19643](#)
 - `__token_if_primitive_loop:N` ...
..... [19529](#), [19619](#), [19634](#), [19640](#)
 - `__token_if_primitive_lua:N`
..... [19529](#), [19595](#)
 - `__token_if_primitive_nullfont:N`
..... [19529](#), [19622](#), [19626](#)
 - `__token_if_primitive_space:w` ...
..... [19529](#), [19617](#), [19625](#)
 - `__token_if_primitive_undefined:N`
..... [19529](#), [19646](#), [19652](#)
 - `__token_tmp:w` [903](#), [19462](#),
[19471](#), [19472](#), [19473](#), [19474](#), [19475](#),
[19476](#), [19477](#), [19478](#), [19481](#), [19515](#),
[19516](#), [19517](#), [19518](#), [19520](#), [19522](#),
[19523](#), [19524](#), [19525](#), [19526](#), [19527](#)
 - `__token_to_catcode:N`
..... [19286](#), [19287](#), [19288](#)
 - `\toks` [425](#), [19527](#)
 - `\toksapp` [924](#)
 - `\toksdef` [426](#), [3482](#)
 - `\tokspre` [925](#)
 - `\tolerance` [427](#)
 - `\topmark` [428](#)
 - `\topmarks` [530](#)
 - `\topskip` [429](#)
 - `\toucs` [1193](#)
 - `\tpack` [926](#)
 - `\tracingassigns` [531](#)
 - `\tracingcommands` [430](#)
 - `\tracingfonts` [961](#)
 - `\tracinggroups` [532](#)
 - `\tracingifs` [533](#)
 - `\tracinglostchars` [431](#)
 - `\tracingmacros` [432](#)
 - `\tracingnesting` [534](#)
 - `\tracingonline` [433](#)
 - `\tracingoutput` [434](#)
 - `\tracingpages` [435](#)
 - `\tracingparagraphs` [436](#)
 - `\tracingrestores` [437](#)
 - `\tracingscantokens` [535](#)
 - `\tracingstacklevels` [1219](#)
 - `\tracingstats` [438](#)
 - `true` [280](#)
 - `trunc` [276](#)
- try commands:
- `try_require` [12024](#)
 - `\ttfamily` [34290](#)
- U**
- `\u` [32057](#),
[34380](#), [34396](#), [34477](#), [34478](#), [34493](#),
[34494](#), [34503](#), [34504](#), [34517](#), [34518](#),
[34519](#), [34545](#), [34546](#), [34571](#), [34572](#)
 - `\uccode` [439](#)
 - `\Uchar` [963](#)
 - `\Ucharcat` [964](#)
 - `\uchyph` [440](#)

<code>\ucs</code>	1194	<code>\Umathlimitbelowkern</code>	1027
<code>\Udelcode</code>	965	<code>\Umathlimitbelowvgap</code>	1028
<code>\Udelcodenum</code>	966	<code>\Umathnolimitsubfactor</code>	1029
<code>\Udelimiter</code>	967	<code>\Umathnolimitsupfactor</code>	1030
<code>\Udelimiterover</code>	968	<code>\Umathopbinspacing</code>	1031
<code>\Udelimiterunder</code>	969	<code>\Umathopclosespacing</code>	1032
<code>\Uhexensible</code>	970	<code>\Umathopenbinspacing</code>	1033
<code>\Uleft</code>	971	<code>\Umathopenclosespacing</code>	1034
<code>\Umathaccent</code>	972	<code>\Umathopeninnerspacing</code>	1035
<code>\Umathaxis</code>	973	<code>\Umathopenopenspacing</code>	1036
<code>\Umathbinbinspacing</code>	974	<code>\Umathopenopspacing</code>	1037
<code>\Umathbinclosespacing</code>	975	<code>\Umathopenordspacing</code>	1038
<code>\Umathbininnerspacing</code>	976	<code>\Umathopenpunctspacing</code>	1039
<code>\Umathbinopenspacing</code>	977	<code>\Umathopenrelspacing</code>	1040
<code>\Umathbinopspacing</code>	978	<code>\Umathoperatorsize</code>	1041
<code>\Umathbinordspacing</code>	979	<code>\Umathopinnerspacing</code>	1042
<code>\Umathbinpunctspacing</code>	980	<code>\Umathopopenspacing</code>	1043
<code>\Umathbinrelspacing</code>	981	<code>\Umathopopspacing</code>	1044
<code>\Umathchar</code>	982	<code>\Umathopordspacing</code>	1045
<code>\Umathcharclass</code>	983	<code>\Umathoppunctspacing</code>	1046
<code>\Umathchardef</code>	984	<code>\Umathoprelspacing</code>	1047
<code>\Umathcharfam</code>	985	<code>\Umathordbinspacing</code>	1048
<code>\Umathcharnum</code>	986	<code>\Umathordclosespacing</code>	1049
<code>\Umathcharnumdef</code>	987	<code>\Umathordinnerspacing</code>	1050
<code>\Umathcharslot</code>	988	<code>\Umathordopenspacing</code>	1051
<code>\Umathclosebinspacing</code>	989	<code>\Umathordopspacing</code>	1052
<code>\Umathcloseclosespacing</code>	990	<code>\Umathordordspacing</code>	1053
<code>\Umathcloseinnerspacing</code>	992	<code>\Umathordpunctspacing</code>	1054
<code>\Umathcloseopenspacing</code>	994	<code>\Umathordrelspacing</code>	1055
<code>\Umathcloseopspacing</code>	995	<code>\Umathoverbarkern</code>	1056
<code>\Umathcloseordspacing</code>	996	<code>\Umathoverbarrule</code>	1057
<code>\Umathclosepunctspacing</code>	997	<code>\Umathoverbarvgap</code>	1058
<code>\Umathcloserelspacing</code>	999	<code>\Umathoverdelimiterbgap</code>	1059
<code>\Umathcode</code>	1000	<code>\Umathoverdelimitervgap</code>	1061
<code>\Umathcodenum</code>	1001	<code>\Umathpunctbinspacing</code>	1063
<code>\Umathconnectoroverlapmin</code>	1002	<code>\Umathpunctclosespacing</code>	1064
<code>\Umathfractiondelsize</code>	1004	<code>\Umathpunctinnerspacing</code>	1066
<code>\Umathfractiondenomdown</code>	1005	<code>\Umathpunctopenspacing</code>	1068
<code>\Umathfractiondenomvgap</code>	1007	<code>\Umathpunctopspacing</code>	1069
<code>\Umathfractionnumup</code>	1009	<code>\Umathpunctordspacing</code>	1070
<code>\Umathfractionnumvgap</code>	1010	<code>\Umathpunctpunctspacing</code>	1071
<code>\Umathfractionrule</code>	1011	<code>\Umathpunctrelspacing</code>	1073
<code>\Umathinnerbinspacing</code>	1012	<code>\Umathquad</code>	1074
<code>\Umathinnerclosespacing</code>	1013	<code>\Umathradicaldegreeafter</code>	1075
<code>\Umathinnerinnerspacing</code>	1015	<code>\Umathradicaldegreebefore</code>	1077
<code>\Umathinneropenspacing</code>	1017	<code>\Umathradicaldegreeraise</code>	1079
<code>\Umathinneropspacing</code>	1018	<code>\Umathradicalkern</code>	1081
<code>\Umathinnerordspacing</code>	1019	<code>\Umathradicalrule</code>	1082
<code>\Umathinnerpunctspacing</code>	1020	<code>\Umathradicalvgap</code>	1083
<code>\Umathinnerrelspacing</code>	1022	<code>\Umathrelbinspacing</code>	1084
<code>\Umathlimitabovebgap</code>	1023	<code>\Umathrelclosespacing</code>	1085
<code>\Umathlimitabovekern</code>	1024	<code>\Umathrelinnerspacing</code>	1086
<code>\Umathlimitabovevgap</code>	1025	<code>\Umathrelopenspacing</code>	1087
<code>\Umathlimitbelowbgap</code>	1026	<code>\Umathrelopspacing</code>	1088

<code>\Umathrelordspacing</code>	1089	11236, 11341, 17676, 18122, 18132,
<code>\Umathrelpunctspacing</code>	1090	18237, 18241, 18243, 18245, 18246,
<code>\Umathrelrelspacing</code>	1091	18250, 20863, 21696, 21702, 22000,
<code>\Umathskewedfractionhgap</code>	1092	29684, 31045, 31324, 32161, 32286,
<code>\Umathskewedfractionvgap</code>	1094	32346, 32347, 32402, 32412, 32419,
<code>\Umathspaceafterscript</code>	1096	32428, 32534, 32663, 32803, 33325,
<code>\Umathstackdenomdown</code>	1097	33341, 33356, 33367, 33493, 33510,
<code>\Umathstacknumup</code>	1098	33511, 33536, 33551, 33553, 33626,
<code>\Umathstackvgap</code>	1099	33812, 33816, 33823, 33981, 34312,
<code>\Umathsubshiftdown</code>	1100	36702, 36738, 36741, 36742, 36747,
<code>\Umathsubshiftdrop</code>	1101	36749, 36753, 36754, 36965, 37046,
<code>\Umathsubsupshiftdown</code>	1102	37248, 37382, 37644, 37669, 37725,
<code>\Umathsubsupvgap</code>	1103	38014, 38047, 38226, 38642, 39173
<code>\Umathsubtopmax</code>	1104	<code>\use:n</code> .. 24, 26, 110, 210, 377, 423,
<code>\Umathsupbottommin</code>	1105	429, 439, 536, 573, 574, 636, 700,
<code>\Umathsupshiftdrop</code>	1106	834, 918, 997, 1058, 1362, 1493,
<code>\Umathsupshiftdown</code>	1107	1493, 1499, 1499, 1501, 1501, 1611,
<code>\Umathsupsubbottommax</code>	1108	1632, 1658, 1718, 1727, 1738, 1739,
<code>\Umathunderbarkern</code>	1109	1749, 2145, 2321, 2336, 2577, 2708,
<code>\Umathunderbarrule</code>	1110	2757, 2894, 2925, 2997, 3975, 4638,
<code>\Umathunderbarvgap</code>	1111	4663, 4876, 5011, 5375, 5539, 5984,
<code>\Umathunderdelimiterbgap</code>	1112	6050, 6118, 6567, 6622, 6678, 6738,
<code>\Umathunderdelimitervgap</code>	1114	6934, 7888, 7950, 8636, 8643, 9462,
<code>\Umiddle</code>	1116	10144, 10356, 10576, 11680, 12572,
undefine commands:		12581, 12740, 12741, 12744, 12886,
<code>.undefine:</code>	245, 22197	12979, 13013, 13035, 13495, 13572,
<code>\underline</code>	441	13580, 13676, 13697, 13711, 14117,
<code>\unexpanded</code>	536	14490, 17246, 17247, 17248, 17249,
<code>\unhbox</code>	442	18822, 18823, 18826, 19232, 19348,
<code>\unhcopy</code>	443	19427, 19464, 19483, 19600, 19925,
<code>\uniformdeviate</code>	962	20280, 20597, 20598, 20599, 20600,
<code>\unkern</code>	444	20686, 21057, 21605, 22217, 22510,
<code>\unless</code>	537	22544, 22565, 22694, 22706, 23628,
<code>\Unosubscript</code>	1117	23636, 23645, 23662, 23670, 23698,
<code>\Unosuperscript</code>	1118	24163, 25765, 29445, 29631, 29639,
<code>\unpenalty</code>	445	29648, 29926, 29931, 30577, 30678,
<code>\unskip</code>	446	30800, 30832, 31034, 31281, 31282,
<code>\unvbox</code>	447	31284, 31560, 31564, 32117, 32200,
<code>\unvcopy</code>	448	32271, 32560, 33735, 34284, 34322,
<code>\Uoverdelimiter</code>	1119	34771, 35909, 36881, 37035, 37447,
<code>\uppercase</code>	449	37993, 38108, 38161, 38588, 38603,
<code>\upshape</code>	34296	38990, 39071, 39240, 39337, 39377
<code>\uptexrevision</code>	1207	<code>\use:nn</code>
<code>\uptexversion</code>	1208	24, 1501, 1502, 2410,
<code>\Uradical</code>	1120	3943, 7837, 8902, 9607, 9609, 9614,
<code>\Uright</code>	1121	9616, 11089, 12389, 12450, 13721,
<code>\Uroot</code>	1122	14289, 20859, 24193, 24202, 24206,
usage commands:		27686, 30076, 31123, 31427, 37773
<code>.usage:n</code>	248, 22199	<code>\use:nnn</code>
use commands:		24, 1501, 1503, 2122
<code>\use:N</code>	21, 181, 384, 1492,	<code>\use:nnnn</code>
1492, 1682, 1778, 4907, 5739, 6982,		24, 1501, 1504
7153, 8401, 8423, 9326, 9336, 9339,		<code>\use_i:nn</code>
9524, 9556, 9562, 9569, 9641, 10758,		25, 376, 381, 383,
		384, 825, 919, 1185, 1188, 1201,
		1205, 1206, 1436, 1505, 1505, 1592,
		1676, 1698, 1740, 1948, 2101, 2928,
		2985, 3310, 3365, 3375, 3385, 3750,

- 4715, 4856, 5162, 5168, 5745, 6690,
8638, 12453, 14791, 14796, 14877,
14881, 16789, 16791, 16875, 16877,
17233, 17284, 17343, 17427, 19965,
20312, 20539, 23035, 23037, 23349,
23973, 24163, 25603, 25929, 26224,
26712, 26996, 27515, 27681, 27993,
28003, 28007, 28515, 28720, 29255,
29280, 29508, 36719, 37325, 39151
- \use_i:nmn 25, 772, 922, 927, 936,
1507, 1507, 2347, 3019, 4636, 4661,
4873, 7055, 14275, 14799, 15305,
17098, 20069, 20211, 20546, 20584,
24132, 26181, 27656, 30010, 30226
- \use_i:nmnn 25, 365, 589,
590, 1507, 1510, 1829, 2070, 8395,
8397, 8412, 8417, 8433, 8435, 21612,
25763, 26199, 26206, 26399, 29266
- \use_i:nmnnn 25, 1507, 1514
- \use_i:nmnnnn 25, 1507, 1519
- \use_i:nmnnnnn 25, 1507, 1525
- \use_i:nmnnnnnn 25, 1507, 1532
- \use_i:nmnnnnnnn 25, 1507, 1540
- \use_i_delimit_by_q_nil:nw
. 27, 1554, 1554
- \use_i_delimit_by_q_recursion_
stop:nw
. 27, 1554, 1556, 16369, 16385, 37175
- \use_i_delimit_by_q_recursion_
stop:w 148
- \use_i_delimit_by_q_stop:nw
. 27, 1554, 1555
- \use_i_ii:nnn 26, 383, 384, 1549,
1549, 1667, 2436, 16613, 17074, 17179
- \use_ii:nn 25,
72, 376, 381, 512, 519, 825, 919,
1185, 1188, 1201, 1205, 1206, 1218,
688, 693, 1438, 1505, 1506, 1594,
1700, 1735, 1740, 1927, 1929, 2099,
2371, 3752, 4319, 4675, 4806, 4827,
4845, 5022, 5369, 5491, 5668, 5751,
6069, 7333, 7356, 7430, 8644, 12402,
12948, 13025, 13031, 16881, 16883,
19964, 20698, 23550, 23573, 23975,
25339, 25603, 25604, 26226, 27517,
27999, 28005, 28009, 28517, 28722,
29152, 29257, 29509, 29613, 36720
- \use_ii:nmn 25, 384,
927, 1507, 1508, 2356, 4317, 4673,
4803, 4824, 4842, 4976, 20213, 39219
- \use_ii:nmnn
. 25, 589, 590, 1507, 1511, 8412
- \use_ii:nmnnn 25, 1507, 1515
- \use_ii:nmnnnn 25, 1507, 1520
- \use_ii:nmnnnnn 25, 1507, 1526
- \use_ii:nmnnnnnn 25, 1507, 1533
- \use_ii:nmnnnnnnn 25, 1507, 1541
- \use_ii_i:nn 26, 761,
1550, 1550, 14294, 14379, 18844, 18932
- \use_iii:nnn 25, 1507,
1509, 1938, 1940, 1946, 2365, 2376,
4712, 4853, 5165, 20310, 20317, 23355
- \use_iii:nmnn 25, 589, 590, 1507,
1512, 8412, 8434, 8436, 8437, 39112
- \use_iii:nmnnn 25, 1507, 1516
- \use_iii:nmnnnn 25, 1507, 1521
- \use_iii:nmnnnnn 25, 1507, 1527
- \use_iii:nmnnnnnn 25, 1507, 1534
- \use_iii:nmnnnnnnn 25, 1507, 1542
- \use_iv:nmnn 25,
589, 590, 1507, 1513, 8412, 8432, 25327
- \use_iv:nmnnn 25, 1507, 1517
- \use_iv:nmnnnn 25, 1507, 1522
- \use_iv:nmnnnnn 25, 1507, 1528
- \use_iv:nmnnnnnn 25, 1507, 1535
- \use_iv:nmnnnnnnn 25, 1507, 1543
- \use_ix:nmnnnnnnn 25, 1507, 1548
- \use_none:n 26,
365, 456, 458, 618, 655, 708, 780,
815, 867, 875, 878, 916, 918, 920,
922, 928, 930, 934, 935, 938, 941,
1055, 1056, 1059, 1429, 689, 695,
1558, 1558, 1666, 1718, 1719, 1738,
1739, 1858, 1865, 1902, 1909, 2126,
2316, 2952, 2953, 3748, 3797, 3907,
3942, 4093, 4114, 5536, 5831, 5960,
8637, 8642, 8997, 9395, 9589, 9806,
9810, 10060, 10731, 10786, 10842,
11120, 11178, 11540, 11543, 12436,
12525, 12584, 12675, 12880, 12891,
12952, 12987, 12991, 13016, 13193,
13202, 14230, 14253, 14269, 14281,
14314, 14449, 14481, 14735, 14745,
14783, 14845, 15009, 15093, 15198,
15370, 16371, 16386, 16510, 16526,
16610, 16669, 17083, 17313, 17314,
17992, 17998, 18343, 18346, 18416,
18461, 18564, 18653, 18680, 18719,
19649, 19993, 20015, 20075, 20082,
20084, 20088, 20090, 20265, 20323,
20333, 20340, 20345, 20351, 20471,
20480, 20484, 20487, 20491, 20533,
20628, 20724, 20735, 21793, 22475,
23133, 23148, 23344, 23493, 23497,
23501, 23505, 24790, 25039, 25046,
25063, 25082, 25105, 25173, 25214,
25339, 25354, 25375, 25376, 25665,
25666, 26200, 26203, 27184, 28876,

- 29161, 29671, 31123, 31357, 31405,
34264, 34319, 37313, 38240, 39079,
39082, 39180, 39181, 39200, 39239
- `\use_none:nn` 26, 706, 711, 826, 831,
1479, 1558, 1559, 1648, 1656, 3051,
6359, 7068, 8390, 10787, 10831,
12510, 12665, 12815, 12982, 14338,
16861, 16890, 17105, 18385, 18694,
18841, 18928, 20154, 20572, 23409,
23492, 23496, 23500, 23504, 28871,
29520, 29699, 29921, 34265, 39177
- `\use_none:nnn` .. 26, 719, 924, 932,
1558, 1560, 2833, 2848, 3973, 7546,
7823, 10788, 12939, 16499, 20143,
20392, 20394, 21940, 21949, 23491,
23495, 23499, 23503, 24132, 34311,
37529, 39094, 39103, 39122, 39713
- `\use_none:nnnn` 26,
918, 920, 1558, 1561, 1872, 10789,
19933, 20020, 20990, 34267, 37836
- `\use_none:nnnnn`
..... 26, 380, 656, 1039, 1558,
1562, 10790, 10800, 23623, 23657,
23683, 23691, 25789, 39127, 39133
- `\use_none:nnnnnn`
.. 26, 1558, 1563, 1785, 10791, 13383
- `\use_none:nnnnnnn`
26, 1039, 1558, 1564, 23625, 23659,
23685, 23693, 24016, 26240, 39186
- `\use_none:nnnnnnnn`
..... 26, 384, 1558, 1565, 1689, 2914
- `\use_none:nnnnnnnnn` .. 26, 1558, 1566
- `\use_none_delimit_by_q_nil:w` ...
..... 26, 1551, 1551
- `\use_none_delimit_by_q_recursion_-
stop:w` 26,
148, 382, 1551, 1553, 16363, 16378
- `\use_none_delimit_by_q_stop:w` ...
..... 26, 780, 817, 1551, 1552
- `\use_none_delimit_by_s_stop:w` ...
..... 150, 16659, 16659
- `\use_v:nnnnn` 25, 1507, 1518
- `\use_v:nnnnnn` 25, 1507, 1523
- `\use_v:nnnnnnn` 25, 1507, 1529
- `\use_v:nnnnnnnn` 25, 1507, 1536
- `\use_v:nnnnnnnnn` 25, 1507, 1544
- `\use_vi:nnnnnn` 25, 1507, 1524
- `\use_vi:nnnnnnn` 25, 1507, 1530
- `\use_vi:nnnnnnnn` 25, 1507, 1537
- `\use_vi:nnnnnnnnn` 25, 1507, 1545
- `\use_vii:nnnnnnn` 25, 1507, 1531
- `\use_vii:nnnnnnnn` 25, 1507, 1538
- `\use_viii:nnnnnnnn` ... 25, 1507, 1546
- `\use_viii:nnnnnnnnn` ... 25, 1507, 1539
- `\use_viii:nnnnnnnnnn` .. 25, 1507, 1547
- `\useboxresource` 955
- `\usefont` 34267
- `\useimageresource` 956
- `\Uskewed` 1123
- `\Uskewedwithdelims` 1124
- `\Ustack` 1125
- `\Ustartdisplaymath` 1126
- `\Ustartmath` 1127
- `\Ustopdisplaymath` 1128
- `\Ustopmath` 1129
- `\Usubscript` 1130
- `\Usuperscript` 1131
- `\Uunderdelimiter` 1132
- `\Uvextensible` 1133
- ### V
- `\v` 32057, 33701, 34380, 34401,
34487, 34488, 34489, 34490, 34499,
34500, 34533, 34534, 34541, 34542,
34553, 34554, 34561, 34562, 34565,
34566, 34588, 34589, 34590, 34591,
34592, 34593, 34594, 34595, 34596,
34597, 34598, 34599, 34600, 34601,
34602, 34605, 34606, 34615, 34616
- `\vadjust` 450
- `\valign` 451
- value commands:
- `.value_forbidden:n` 245, 22201
 - `.value_required:n` 245, 22201
- `\variablefam` 927
- `\vbadness` 452
- `\vbox` 1430, 453
- vbox commands:
- `\vbox:n` 303, 308, 34860, 34860
 - `\vbox_gset:Nn`
308, 34874, 34879, 34885, 35545, 39586
 - `\vbox_gset:Nw`
308, 34910, 34916, 34923, 35620, 39589
 - `\vbox_gset_end:`
..... 308, 34910, 34930, 35622
 - `\vbox_gset_split_to_ht:NNn`
.... 309, 34949, 34952, 34957, 39591
 - `\vbox_gset_to_ht:Nnn`
.... 308, 34898, 34903, 34909, 39588
 - `\vbox_gset_to_ht:Nnw`
.... 309, 34931, 34937, 34944, 39590
 - `\vbox_gset_top:Nn`
.... 308, 34886, 34891, 34897, 39587
 - `\vbox_set:Nn`
308, 34874, 34874, 34884, 35539, 39505
 - `\vbox_set:Nw`
308, 34910, 34910, 34922, 35613, 39508

<code>\vbox_set_end:</code>	<code>\XeTeXcharclass</code>	706
308, 309, 34910, 34924, 34930, 35615	<code>\XeTeXcharglyph</code>	707
<code>\vbox_set_split_to_ht:Nn</code>	<code>\XeTeXcountfeatures</code>	708
309, 34949, 34949, 34951, 39510	<code>\XeTeXcountglyphs</code>	709
<code>\vbox_set_to_ht:Nnn</code>	<code>\XeTeXcountselectors</code>	710
308, 309, 34898, 34898, 34908, 39507	<code>\XeTeXcountvariations</code>	711
<code>\vbox_set_to_ht:Nnw</code>	<code>\XeTeXdashbreakstate</code>	713
309, 34931, 34931, 34943, 39509	<code>\XeTeXdefaultencoding</code>	712
<code>\vbox_set_top:Nn</code>	<code>\XeTeXfeaturecode</code>	714
308, 34886, 34886, 34896, 35559, 35636, 39506	<code>\XeTeXfeaturename</code>	715
<code>\vbox_to_ht:mn</code>	<code>\XeTeXfindfeaturebyname</code>	716
308, 34864, 34864	<code>\XeTeXfindselectorbyname</code>	718
<code>\vbox_to_zero:n</code>	<code>\XeTeXfindvariationbyname</code>	720
308, 34864, 34869	<code>\XeTeXfirstfontchar</code>	722
<code>\vbox_top:n</code>	<code>\XeTeXfonttype</code>	723
308, 34860, 34862	<code>\XeTeXgenerateactualtext</code>	724
<code>\vbox_unpack:N</code>	<code>\XeTeXglyph</code>	726
309, 34945, 34945, 34947, 35559, 35636	<code>\XeTeXglyphbounds</code>	727
<code>\vbox_unpack_drop:N</code>	<code>\XeTeXglyphindex</code>	728
310, 34945, 34946, 34948	<code>\XeTeXglyphname</code>	729
<code>\vcenter</code>	<code>\XeTeXhyphenatablelength</code>	767
454	<code>\XeTeXinputencoding</code>	730
<code>vcoffin</code> commands:	<code>\XeTeXinputnormalization</code>	731
<code>\vcoffin_gset:Nnn</code>	<code>\XeTeXinterchartokenstate</code>	733
316, 35536, 35542, 35547	<code>\XeTeXinterchartoks</code>	735
<code>\vcoffin_gset:Nnw</code>	<code>\XeTeXinterwordspaceshaping</code>	765
316, 35611, 35618, 35624	<code>\XeTeXisdefaultselector</code>	736
<code>\vcoffin_gset_end:</code>	<code>\XeTeXisexclusivefeature</code>	738
316, 35611, 35621, 35652	<code>\XeTeXlastfontchar</code>	740
<code>\vcoffin_set:Nnn</code>	<code>\XeTeXlinebreaklocale</code>	742
316, 35536, 35536, 35541	<code>\XeTeXlinebreakpenalty</code>	743
<code>\vcoffin_set:Nnw</code>	<code>\XeTeXlinebreakskip</code>	741
316, 35611, 35611, 35617	<code>\XeTeXOTcountfeatures</code>	744
<code>\vcoffin_set_end:</code>	<code>\XeTeXOTcountlanguages</code>	745
316, 35611, 35614, 35651	<code>\XeTeXOTcountscripts</code>	746
<code>\vfi</code>	<code>\XeTeXOTfeaturetag</code>	747
1199	<code>\XeTeXOTlanguagetag</code>	748
<code>\vfil</code>	<code>\XeTeXOTscripttag</code>	749
455	<code>\XeTeXpdffile</code>	750
<code>\vfill</code>	<code>\XeTeXpdfpagecount</code>	751
456	<code>\XeTeXpicfile</code>	752
<code>\vfilneg</code>	<code>\XeTeXprotrudechars</code>	779
457	<code>\XeTeXrevision</code>	753
<code>\vfuzz</code>	<code>\XeTeXselectorcode</code>	764
458	<code>\XeTeXselectorname</code>	754
<code>\voffset</code>	<code>\XeTeXtracingfonts</code>	755
459	<code>\XeTeXupwardsmode</code>	756
<code>\vpack</code>	<code>\XeTeXuseglyphmetrics</code>	757
928	<code>\XeTeXvariation</code>	758
<code>\vrule</code>	<code>\XeTeXvariationdefault</code>	759
460	<code>\XeTeXvariationmax</code>	760
<code>\vsize</code>	<code>\XeTeXvariationmin</code>	761
461	<code>\XeTeXvariationname</code>	762
<code>\vskip</code>	<code>\XeTeXversion</code>	763
462		
<code>\vsplit</code>		
463		
<code>\vss</code>		
464		
<code>\vtop</code>		
465		
W		
<code>\wd</code>		466
<code>\widowpenalties</code>		538
<code>\widowpenalty</code>		467
<code>\wordboundary</code>		929
<code>\write</code>		68, 468
X		
<code>\xdef</code>		469

\kanjiskip	1195	\xtokspre	931
\xleaders	470		
\xspaceskip	471	Y	
\xspcode	1196	\ybaselineshift	1197
\xtoksapp	930	\year	472, 1302, 9012
		\yoko	1198