March 17, 2021 at 13:06

2* The present implementation has a long ancestry, beginning in the summer of 1977, when Michael F. Plass and Frank M. Liang designed and coded a prototype based on some specifications that the author had made in May of that year. This original protoTEX included macro definitions and elementary manipulations on boxes and glue, but it did not have line-breaking, page-breaking, mathematical formulas, alignment routines, error recovery, or the present semantic nest; furthermore, it used character lists instead of token lists, so that a control sequence like `\halign` was represented by a list of seven characters. A complete version of TEX was designed and coded by the author in late 1977 and early 1978; that program, like its prototype, was written in the SAIL language, for which an excellent debugging system was available. Preliminary plans to convert the SAIL code into a form somewhat like the present "web" were developed by Luis Trabb Pardo and the author at the beginning of 1979, and a complete implementation was created by Ignacio A. Zabala in 1979 and 1980. The TEX82 program, which was written by the author during the latter part of 1981 and the early part of 1982, also incorporates ideas from the 1979 implementation of TEX in MESA that was written by Leonidas Guibas, Robert Sedgewick, and Douglas Wyatt at the Xerox Palo Alto Research Center. Several hundred refinements were introduced into TEX82 based on the experiences gained with the original implementations, so that essentially every part of the system has been substantially improved. After the appearance of "Version 0" in September 1982, this program benefited greatly from the comments of many other people, notably David R. Fuchs and Howard W. Trickey. A final revision in September 1989 extended the input character set to eight-bit codes and introduced the ability to hyphenate words from different languages, based on some ideas of Michael J. Ferguson.

No doubt there still is plenty of room for improvement, but the author is firmly committed to keeping TEX82 "frozen" from now on; stability and reliability are to be its main virtues.

On the other hand, the `WEB` description can be extended without changing the core of TEX82 itself, and the program has been designed so that such extensions are not extremely difficult to make. The *banner* string defined here should be changed whenever TEX undergoes any modifications, so that it will be clear which version of TEX might be the guilty party when a problem arises.

This program contains code for various features extending TEX, therefore this program is called 'ε-TEX' and not 'TEX'; the official name 'TEX' by itself is reserved for software systems that are fully compatible with each other. A special test suite called the "`TRIP` test" is available for helping to determine whether a particular implementation deserves to be known as 'TEX' [cf. Stanford Computer Science report CS1027, November 1984].

MLTEX will add new primitives changing the behaviour of TEX. The *banner* string has to be changed. We do not change the *banner* string, but will output an additional line to make clear that this is a modified TEX version.

A similar test suite called the "`e-TRIP` test" is available for helping to determine whether a particular implementation deserves to be known as 'ε-TEX'.

define *eTeX_version* = 2 { `\eTeXversion` }
define *eTeX_revision* ≡ `".6"` { `\eTeXrevision` }
define *eTeX_version_string* ≡ `´-2.6´` { current ε-TEX version }
define *eTeX_banner* ≡ `´This␣is␣e-TeX,␣Version␣3.141592653´`, *eTeX_version_string* { printed when ε-TEX starts }
define *pdftex_version* ≡ 140 { `\pdftexversion` }
define *pdftex_revision* ≡ `"22"` { `\pdftexrevision` }
define *pdftex_version_string* ≡ `´-1.40.22´` { current pdfTEX version }
define *pdfTeX_banner* ≡ `´This␣is␣pdfTeX,␣Version␣3.141592653´`, *eTeX_version_string*, *pdftex_version_string* { printed when pdfTEX starts }
define *TeX_banner_k* ≡ `´This␣is␣TeXk,␣Version␣3.141592653´` { printed when TEX starts }
define *TeX_banner* ≡ `´This␣is␣TeX,␣Version␣3.141592653´` { printed when TEX starts }
define *banner* ≡ *pdfTeX_banner*
define *banner_k* ≡ *pdfTeX_banner*

define *TEX* ≡ *PDFTEX* { change program name into *PDFTEX* }
define *TeXXeT_code* = 0 { the TEX--XeT feature is optional }
define *eTeX_states* = 1 { number of ε-TEX state variables in *eqtb* }

4* The program begins with a normal Pascal program heading, whose components will be filled in later, using the conventions of `WEB`. For example, the portion of the program called '⟨ Global variables 13 ⟩' below will be replaced by a sequence of variable declarations that starts in §13 of this documentation. In this way, we are able to define each individual global variable when we are prepared to understand what it means; we do not have to define all of the globals at once. Cross references in §13, where it says "See also sections 20, 26, ...," also make it possible to look at the set of all global variables, if desired. Similar remarks apply to the other portions of the program heading.

define *mtype* ≡ *t*`@&`*y*`@&`*p*`@&`*e* { this is a `WEB` coding trick: }
format *mtype* ≡ *type* { '**mtype**' will be equivalent to '**type**' }
format *type* ≡ *true* { but '*type*' will not be treated as a reserved word }

⟨ Compiler directives 9 ⟩
program *TEX*; { all file names are defined dynamically }
const ⟨ Constants in the outer block 11* ⟩
mtype ⟨ Types in the outer block 18 ⟩
var ⟨ Global variables 13 ⟩
procedure *initialize*; { this procedure gets things started properly }
var ⟨ Local variables for initialization 19* ⟩
begin ⟨ Initialize whatever TEX might access 8* ⟩
end;
⟨ Basic printing procedures 57 ⟩
⟨ Error handling procedures 78 ⟩

6* For Web2c, labels are not declared in the main program, but we still have to declare the symbolic names.

define *start_of_TEX* = 1 { go here when TEX's variables are initialized }
define *final_end* = 9999 { this label marks the ending of the program }

7* Some of the code below is intended to be used only when diagnosing the strange behavior that sometimes occurs when TEX is being installed or when system wizards are fooling around with TEX without quite knowing what they are doing. Such code will not normally be compiled; it is delimited by the codewords '**debug** ... **gubed**', with apologies to people who wish to preserve the purity of English.

Similarly, there is some conditional code delimited by '**stat** ... **tats**' that is intended for use when statistics are to be kept about TEX's memory usage. The **stat** ... **tats** code also implements diagnostic information for `\tracingparagraphs`, `\tracingpages`, and `\tracingrestores`.

define *debug* ≡ *ifdef* (`´TEXMF_DEBUG´`)
define *gubed* ≡ *endif* (`´TEXMF_DEBUG´`)
format *debug* ≡ *begin*
format *gubed* ≡ *end*

define *stat* ≡ *ifdef* (`´STAT´`)
define *tats* ≡ *endif* (`´STAT´`)
format *stat* ≡ *begin*
format *tats* ≡ *end*

8* This program has two important variations: (1) There is a long and slow version called `INITEX`, which does the extra calculations needed to initialize TEX's internal tables; and (2) there is a shorter and faster production version, which cuts the initialization to a bare minimum. Parts of the program that are needed in (1) but not in (2) are delimited by the codewords '**init** ... **tini**' for declarations and by the codewords '**Init** ... **Tini**' for executable code. This distinction is helpful for implementations where a run-time switch differentiates between the two versions of the program.

define *init* ≡ *ifdef* (´`INITEX`´)
define *tini* ≡ *endif* (´`INITEX`´)
define *Init* ≡
  init
  if *ini_version* **then**
   begin
define *Tini* ≡
  end ; **tini**
format *Init* ≡ *begin*
format *Tini* ≡ *end*
format *init* ≡ *begin*
format *tini* ≡ *end*

⟨ Initialize whatever TEX might access 8* ⟩ ≡
 ⟨ Set initial values of key variables 21 ⟩
 Init ⟨ Initialize table entries (done by `INITEX` only) 182 ⟩ **Tini**

See also section 1909*.

This code is used in section 4*.

11* The following parameters can be changed at compile time to extend or reduce TEX's capacity. They may have different values in `INITEX` and in production versions of TEX.

define *file_name_size* ≡ *maxint*
define *ssup_error_line* = 255
define *ssup_max_strings* ≡ 2097151
{ Larger values than 65536 cause the arrays to consume much more memory. }
define *ssup_trie_opcode* ≡ 65535
define *ssup_trie_size* ≡ "3FFFFF
define *ssup_hyph_size* ≡ 65535 { Changing this requires changing (un)dumping! }
define *iinf_hyphen_size* ≡ 610 { Must be not less than *hyph_prime*! }
define *max_font_max* = 9000 { maximum number of internal fonts; this can be increased, but *hash_size* + *max_font_max* should not exceed 29000. }
define *font_base* = 0 { smallest internal font number; must be ≥ *min_quarterword*; do not change this without modifying the dynamic definition of the font arrays. }

⟨ Constants in the outer block 11* ⟩ ≡
hash_offset = 514; { smallest index in hash array, i.e., *hash_base* }
{ Use *hash_offset* = 0 for compilers which cannot decrement pointers. }
trie_op_size = 35111;
{ space for "opcodes" in the hyphenation patterns; best if relatively prime to 313, 361, and 1009. }
neg_trie_op_size = −35111; { for lower *trie_op_hash* array bound; must be equal to −*trie_op_size*. }
min_trie_op = 0; { first possible trie op code for any language }
max_trie_op = *ssup_trie_opcode*; { largest possible trie opcode for any language }
pool_name = *TEXMF_POOL_NAME*; { this is configurable, for the sake of ML-TEX }
{ string of length *file_name_size*; tells where the string pool appears }
engine_name = *TEXMF_ENGINE_NAME*; { the name of this engine }

inf_mem_bot = 0; *sup_mem_bot* = 1; *inf_main_memory* = 3000; *sup_main_memory* = 256000000;
inf_trie_size = 8000; *sup_trie_size* = *ssup_trie_size*; *inf_max_strings* = 3000;
sup_max_strings = *ssup_max_strings*; *inf_strings_free* = 100; *sup_strings_free* = *sup_max_strings*;
inf_buf_size = 500; *sup_buf_size* = 30000000; *inf_nest_size* = 40; *sup_nest_size* = 4000;
inf_max_in_open = 6; *sup_max_in_open* = 127; *inf_param_size* = 60; *sup_param_size* = 32767;
inf_save_size = 600; *sup_save_size* = 80000; *inf_stack_size* = 200; *sup_stack_size* = 30000;
inf_dvi_buf_size = 800; *sup_dvi_buf_size* = 65536; *inf_font_mem_size* = 20000;
sup_font_mem_size = 147483647; { *integer*-limited, so 2 could be prepended? }
sup_font_max = *max_font_max*; *inf_font_max* = 50; { could be smaller, but why? }
inf_pool_size = 32000; *sup_pool_size* = 40000000; *inf_pool_free* = 1000; *sup_pool_free* = *sup_pool_size*;
inf_string_vacancies = 8000; *sup_string_vacancies* = *sup_pool_size* − 23000;
sup_hash_extra = *sup_max_strings*; *inf_hash_extra* = 0; *sup_hyph_size* = *ssup_hyph_size*;
inf_hyph_size = *iinf_hyphen_size*; { Must be not less than *hyph_prime*! }
inf_expand_depth = 10; *sup_expand_depth* = 10000000;

See also sections 675, 679, 695, 721, and 1628.

This code is used in section 4*.

12* Like the preceding parameters, the following quantities can be changed at compile time to extend or reduce TEX's capacity. But if they are changed, it is necessary to rerun the initialization program `INITEX` to generate new tables for the production TEX program. One can't simply make helter-skelter changes to the following constants, since certain rather complex initialization numbers are computed from them. They are defined here using `WEB` macros, instead of being put into Pascal's **const** list, in order to emphasize this distinction.

define *hash_size* = 15000 { maximum number of control sequences; it should be at most about ($mem\_max - mem\_min$)/10; see also *font_max* }
define *hash_prime* = 8501 { a prime number equal to about 85% of *hash_size* }
define *hyph_prime* = 607 { another prime for hashing `\hyphenation` exceptions; if you change this, you should also change *iinf_hyphen_size*. }

16* Here are some macros for common programming idioms.

define *negate*(#) ≡ # ← −# { change the sign of a variable }
define *loop* ≡ **while** *true* **do** { repeat over and over until a **goto** happens }
format *loop* ≡ *xclause* { `WEB`'s **xclause** acts like '**while** *true* **do**' }
define *do_nothing* ≡ { empty statement }
define *return* ≡ **goto** *exit* { terminate a procedure call }
format *return* ≡ *nil*
define *empty* = 0 { symbolic name for a null constant }

19* The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lowercase letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, especially in a program for typesetting; so the present specification of TEX has been written under the assumption that the Pascal compiler and run-time system permit the use of text files with more than 64 distinguishable characters. More precisely, we assume that the character set contains at least the letters and symbols associated with ASCII codes ´*40* through ´*176*; all of these characters are now available on most computer terminals.

Since we are dealing with more characters than were present in the first Pascal compilers, we have to decide what to call the associated data type. Some Pascals use the original name *char* for the characters in text files, even though there now are more than 64 such characters, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name.

In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters that are converted to and from *ASCII_code* when they are input and output. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define *text_char* ≡ *ASCII_code* { the data type of characters in text files }
define *first_text_char* = 0 { ordinal number of the smallest element of *text_char* }
define *last_text_char* = 255 { ordinal number of the largest element of *text_char* }

⟨ Local variables for initialization 19* ⟩ ≡
i: *integer*;

See also sections 181 and 1102.

This code is used in section 4*.

20* The TEX processor converts between ASCII code and the user's external character set by means of arrays *xord* and *xchr* that are analogous to Pascal's *ord* and *chr* functions.

⟨ Global variables 13 ⟩ +≡
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [*ASCII_code*] **of** *text_char*; { specifies conversion of output characters }
xprn: **array** [*ASCII_code*] **of** *ASCII_code*; { non zero iff character is printable }
mubyte_read: **array** [*ASCII_code*] **of** *pointer*; { non zero iff character begins the multi byte code }
mubyte_write: **array** [*ASCII_code*] **of** *str_number*;
  { non zero iff character expands to multi bytes in log and write files }
mubyte_cswrite: **array** [0 .. 127] **of** *pointer*;
  { non null iff cs mod 128 expands to multi bytes in log and write files }
mubyte_skip: *integer*; { the number of bytes to skip in *buffer* }
mubyte_keep: *integer*; { the number of chars we need to keep unchanged }
mubyte_skeep: *integer*; { saved *mubyte_keep* }
mubyte_prefix: *integer*; { the type of mubyte prefix }
mubyte_tablein: *boolean*; { the input side of table will be updated }
mubyte_tableout: *boolean*; { the output side of table will be updated }
mubyte_relax: *boolean*; { the relax prefix is used }
mubyte_start: *boolean*; { we are making the token at the start of the line }
mubyte_sstart: *boolean*; { saved *mubyte_start* }
mubyte_token: *pointer*; { the token returned by *read_buffer* }
mubyte_stoken: *pointer*; { saved first token in mubyte primitive }
mubyte_sout: *integer*; { saved value of *mubyte_out* }
mubyte_slog: *integer*; { saved value of *mubyte_log* }
spec_sout: *integer*; { saved value of *spec_out* }
no_convert: *boolean*; { conversion supressed by noconvert primitive }
active_noconvert: *boolean*; { true if noconvert primitive is active }
write_noexpanding: *boolean*; { true only if we need not write expansion }
cs_converting: *boolean*; { true only if we need csname converting }
special_printing: *boolean*; { true only if we need converting in special }
message_printing: *boolean*; { true if message or errmessage prints to string }

23* The ASCII code is "standard" only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. Appendix C of *The TEXbook* gives a complete specification of the intended correspondence between characters and TEX's internal representation.

If TEX is being used on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, it doesn't really matter what codes are specified in *xchr*[0 .. ´37], but the safest policy is to blank everything out by using the code shown below.

However, other settings of *xchr* will make TEX more friendly on computers that have an extended character set, so that users can type things like '≠' instead of '\ne'. People with extended character sets can assign codes arbitrarily, giving an *xchr* equivalent to whatever characters the users of TEX are allowed to have in their input files. It is best to make the codes correspond to the intended interpretations as shown in Appendix C whenever possible; but this is not necessary. For example, in countries with an alphabet of more than 26 letters, it is usually best to map the additional letters into codes less than ´40. To get the most "permissive" character set, change ´␣´ on the right of these assignment statements to *chr*(*i*).

⟨Set initial values of key variables 21⟩ +≡
{ Initialize *xchr* to the identity mapping. }
for $i \leftarrow 0$ **to** ´37 **do** $xchr[i] \leftarrow i$;
for $i \leftarrow$ ´177 **to** ´377 **do** $xchr[i] \leftarrow i$; { Initialize encTEX data. }
for $i \leftarrow 0$ **to** 255 **do** *mubyte_read*[*i*] ← *null*;
for $i \leftarrow 0$ **to** 255 **do** *mubyte_write*[*i*] ← 0;
for $i \leftarrow 0$ **to** 127 **do** *mubyte_cswrite*[*i*] ← *null*;
mubyte_keep ← 0; *mubyte_start* ← *false*; *write_noexpanding* ← *false*; *cs_converting* ← *false*;
special_printing ← *false*; *message_printing* ← *false*; *no_convert* ← *false*; *active_noconvert* ← *false*;

24* The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*. Note that if $xchr[i] = xchr[j]$ where $i < j <$ ´177, the value of *xord*[*xchr*[*i*]] will turn out to be *j* or more; hence, standard ASCII code numbers will be used instead of codes below ´40 in case there is a coincidence.

⟨Set initial values of key variables 21⟩ +≡
for *i* ← *first_text_char* **to** *last_text_char* **do** *xord*[*chr*(*i*)] ← *invalid_code*;
for *i* ← ´200 **to** ´377 **do** *xord*[*xchr*[*i*]] ← *i*;
for *i* ← 0 **to** ´176 **do** *xord*[*xchr*[*i*]] ← *i*; { Set *xprn* for printable ASCII, unless *eight_bit_p* is set. }
for *i* ← 0 **to** 255 **do** *xprn*[*i*] ← (*eight_bit_p* ∨ ((*i* ≥ "␣") ∧ (*i* ≤ "~"))); { The idea for this dynamic translation comes from the patch by Libor Skarvada `<libor@informatics.muni.cz>` and Petr Sojka `<sojka@informatics.muni.cz>`. I didn't use any of the actual code, though, preferring a more general approach. }
{ This updates the *xchr*, *xord*, and *xprn* arrays from the provided *translate_filename*. See the function definition in `texmfmp.c` for more comments. }
if *translate_filename* **then** *read_tcx_file*;

26* Most of what we need to do with respect to input and output can be handled by the I/O facilities that are standard in Pascal, i.e., the routines called *get*, *put*, *eof*, and so on. But standard Pascal does not allow file variables to be associated with file names that are determined at run time, so it cannot be used to implement TEX; some sort of extension to Pascal's ordinary *reset* and *rewrite* is crucial for our purposes. We shall assume that *name_of_file* is a variable of an appropriate type such that the Pascal run-time system being used to implement TEX can open a file whose external name is specified by *name_of_file*.

⟨ Global variables 13 ⟩ +≡
name_of_file: ↑*text_char*;
name_length: 0 .. *file_name_size*;
{ this many characters are actually relevant in *name_of_file* (the rest are blank) }

27* All of the file opening functions are defined in C.

28* And all the file closing routines as well.

30* Input from text files is read one line at a time, using a routine called *input_ln*. This function is defined in terms of global variables called *buffer*, *first*, and *last* that will be described in detail later; for now, it suffices for us to know that *buffer* is an array of *ASCII_code* values, and that *first* and *last* are indices into this array representing the beginning and ending of a line of text.

⟨ Global variables 13 ⟩ +≡
buffer: ↑*ASCII_code*; { lines of characters being read }
first: 0 .. *buf_size*; { the first unused position in *buffer* }
last: 0 .. *buf_size*; { end of the line just input to *buffer* }
max_buf_stack: 0 .. *buf_size*; { largest index used in *buffer* }

31* The *input_ln* function brings the next line of input from the specified file into available positions of the buffer array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false* and sets $last \leftarrow first$. In general, the *ASCII_code* numbers that represent the next line of the file are input into $buffer[first]$, $buffer[first+1]$, ..., $buffer[last-1]$; and the global variable *last* is set equal to *first* plus the length of the line. Trailing blanks are removed from the line; thus, either $last = first$ (in which case the line was entirely blank) or $buffer[last-1] \neq$ "␣".

An overflow error is given, however, if the normal actions of *input_ln* would make $last \geq buf\_size$; this is done so that other parts of TEX can safely look at the contents of $buffer[last+1]$ without overstepping the bounds of the *buffer* array. Upon entry to *input_ln*, the condition $first < buf\_size$ will always hold, so that there is always room for an "empty" line.

The variable *max_buf_stack*, which is used to keep track of how large the *buf_size* parameter must be to accommodate the present job, is also kept up to date by *input_ln*.

If the *bypass_eoln* parameter is *true*, *input_ln* will do a *get* before looking at the first character of the line; this skips over an *eoln* that was in $f\uparrow$. The procedure does not do a *get* when it reaches the end of the line; therefore it can be used to acquire input from the user's terminal as well as from ordinary text files.

Standard Pascal says that a file should have *eoln* immediately before *eof*, but TEX needs only a weaker restriction: If *eof* occurs in the middle of a line, the system function *eoln* should return a *true* result (even though $f\uparrow$ will be undefined).

Since the inner loop of *input_ln* is part of TEX's "inner loop"—each character of input comes in at this place—it is wise to reduce system overhead by making use of special routines that read in an entire array of characters at once, if such routines are available. The following code uses standard Pascal to illustrate what needs to be done, but finer tuning is often possible at well-developed Pascal sites.

We define *input_ln* in C, for efficiency. Nevertheless we quote the module 'Report overflow of the input buffer, and abort' here in order to make `WEAVE` happy, since part of that module is needed by e-TeX.

`@{`⟨ Report overflow of the input buffer, and abort 35* ⟩`@}`

32* The user's terminal acts essentially like other files of text, except that it is used both for input and for output. When the terminal is considered an input file, the file variable is called *term_in*, and when it is considered an output file the file variable is *term_out*.

define *term_in* ≡ *stdin* { the terminal as an input file }
define *term_out* ≡ *stdout* { the terminal as an output file }

⟨ Global variables 13 ⟩ +≡
init *ini_version*: *boolean*; { are we `INITEX`? }
dump_option: *boolean*; { was the dump name option used? }
dump_line: *boolean*; { was a `%&format` line seen? }
tini

dump_name: *const_cstring*; { format name for terminal display }

bound_default: *integer*; { temporary for setup }
bound_name: *const_cstring*; { temporary for setup }

mem_bot: *integer*;
{ smallest index in the *mem* array dumped by `INITEX`; must not be less than *mem_min* }
main_memory: *integer*; { total memory words allocated in initex }
extra_mem_bot: *integer*; { $mem\_min \leftarrow mem\_bot - extra\_mem\_bot$ except in `INITEX` }
mem_min: *integer*; { smallest index in TEX's internal *mem* array; must be *min_halfword* or more; must be equal to *mem_bot* in `INITEX`, otherwise $\le mem\_bot$ }
mem_top: *integer*; { largest index in the *mem* array dumped by `INITEX`; must be substantially larger than *mem_bot*, equal to *mem_max* in `INITEX`, else not greater than *mem_max* }
extra_mem_top: *integer*; { $mem\_max \leftarrow mem\_top + extra\_mem\_top$ except in `INITEX` }
mem_max: *integer*; { greatest index in TEX's internal *mem* array; must be strictly less than *max_halfword*; must be equal to *mem_top* in `INITEX`, otherwise $\ge mem\_top$ }
error_line: *integer*; { width of context lines on terminal error messages }
half_error_line: *integer*; { width of first lines of contexts in terminal error messages; should be between 30 and $error\_line - 15$ }
max_print_line: *integer*; { width of longest text lines output; should be at least 60 }
max_strings: *integer*; { maximum number of strings; must not exceed *max_halfword* }
strings_free: *integer*; { strings available after format loaded }
string_vacancies: *integer*; { the minimum number of characters that should be available for the user's control sequences and font names, after TEX's own error messages are stored }
pool_size: *integer*; { maximum number of characters in strings, including all error messages and help texts, and the names of all fonts and control sequences; must exceed *string_vacancies* by the total length of TEX's own strings, which is currently about 23000 }
pool_free: *integer*; { pool space free after format loaded }
font_mem_size: *integer*; { number of words of *font_info* for all fonts }
font_max: *integer*; { maximum internal font number; ok to exceed *max_quarterword* and must be at most $font\_base + max\_font\_max$ }
font_k: *integer*; { loop variable for initialization }
hyph_size: *integer*; { maximum number of hyphen exceptions }
trie_size: *integer*; { space for hyphenation patterns; should be larger for `INITEX` than it is in production versions of TEX. 50000 is needed for English, German, and Portuguese. }
buf_size: *integer*; { maximum number of characters simultaneously present in current lines of open files and in control sequences between `\csname` and `\endcsname`; must not exceed *max_halfword* }
stack_size: *integer*; { maximum number of simultaneous input sources }
max_in_open: *integer*;
{ maximum number of input files and error insertions that can be going on simultaneously }
param_size: *integer*; { maximum number of simultaneous macro parameters }
nest_size: *integer*; { maximum number of semantic levels simultaneously active }
save_size: *integer*; { space for saving values outside of current group; must be at most *max_halfword* }

dvi_buf_size: *integer*; { size of the output buffer; must be a multiple of 8 }
expand_depth: *integer*; { limits recursive calls to the *expand* procedure }
parse_first_line_p: *cinttype*; { parse the first line for options }
file_line_error_style_p: *cinttype*; { format messages as file:line:error }
eight_bit_p: *cinttype*; { make all characters printable by default }
halt_on_error_p: *cinttype*; { stop at first error }
quoted_filename: *boolean*; { current filename is quoted }
  { Variables for source specials }
src_specials_p: *boolean*; { Whether *src_specials* are enabled at all }
insert_src_special_auto: *boolean*;
insert_src_special_every_par: *boolean*;
insert_src_special_every_parend: *boolean*;
insert_src_special_every_cr: *boolean*;
insert_src_special_every_math: *boolean*;
insert_src_special_every_hbox: *boolean*;
insert_src_special_every_vbox: *boolean*;
insert_src_special_every_display: *boolean*;

33* Here is how to open the terminal files. *t_open_out* does nothing. *t_open_in*, on the other hand, does the work of "rescanning," or getting any command line arguments the user has provided. It's defined in C.

define *t_open_out* ≡ { output already open for text output }

34* Sometimes it is necessary to synchronize the input/output mixture that happens on the user's terminal, and three system-dependent procedures are used for this purpose. The first of these, *update_terminal*, is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent. The second, *clear_terminal*, is called when we wish to cancel any input that the user may have typed ahead (since we are about to issue an unexpected error message). The third, *wake_up_terminal*, is supposed to revive the terminal if the user has disabled it by some instruction to the operating system. The following macros show how these operations can be specified with UNIX. *update_terminal* does an *fflush*. *clear_terminal* is redefined to do nothing, since the user should control the terminal.

define *update_terminal* ≡ *fflush*(*term_out*)
define *clear_terminal* ≡ *do_nothing*
define *wake_up_terminal* ≡ *do_nothing* { cancel the user's cancellation of output }

35* We need a special routine to read the first line of TEX input from the user's terminal. This line is different because it is read before we have opened the transcript file; there is sort of a "chicken and egg" problem here. If the user types '`\input paper`' on the first line, or if some macro invoked by that line does such an `\input`, the transcript file will be named '`paper.log`'; but if no `\input` commands are performed during the first line of terminal input, the transcript file will acquire its default name '`texput.log`'. (The transcript file will not contain error messages generated by the first line before the first `\input` command.)

The first line is even more special if we are lucky enough to have an operating system that treats TEX differently from a run-of-the-mill Pascal object program. It's nice to let the user start running a TEX job by typing a command line like '`tex paper`'; in such a case, TEX will operate as if the first line of input were '`paper`', i.e., the first line will consist of the remainder of the command line, after the part that invoked TEX.

The first line is special also because it may be read before TEX has input a format file. In such cases, normal error messages cannot yet be given. The following code uses concepts that will be explained later. (If the Pascal compiler does not support non-local **goto**, the statement '**goto** *final_end*' should be replaced by something that quietly terminates the program.)

Routine is implemented in C; part of module is, however, needed for e-TeX.

⟨ Report overflow of the input buffer, and abort 35* ⟩ ≡
begin *cur_input.loc_field* ← *first*; *cur_input.limit_field* ← *last* − 1; *overflow*("buffer␣size", *buf_size*);
end

This code is used in sections 31* and 1753.

37* The following program does the required initialization. Iff anything has been specified on the command line, then *t_open_in* will return with *last* > *first*.

```
function init_terminal: boolean;  { gets the terminal input started }
  label exit;
  begin t_open_in;
  if last > first then
    begin loc ← first;
    while (loc < last) ∧ (buffer[loc] = ´␣´) do incr(loc);
    if loc < last then
      begin init_terminal ← true; goto exit;
      end;
    end;
  loop begin wake_up_terminal; write(term_out, ´**´); update_terminal;
    if ¬input_ln(term_in, true) then  { this shouldn't happen }
      begin write_ln(term_out); write_ln(term_out, ´!␣End␣of␣file␣on␣the␣terminal...␣why?´);
      init_terminal ← false; return;
      end;
    loc ← first;
    while (loc < last) ∧ (buffer[loc] = "␣") do incr(loc);
    if loc < last then
      begin init_terminal ← true; return;  { return unless the line was all blank }
      end;
    write_ln(term_out, ´Please␣type␣the␣name␣of␣your␣input␣file.´);
    end;
exit: end;
```

38*. String handling. Control sequence names and diagnostic messages are variable-length strings of eight-bit characters. Since Pascal does not have a well-developed string mechanism, TEX does all of its string processing by homegrown methods.

Elaborate facilities for dynamic strings are not needed, so all of the necessary operations can be handled with a simple data structure. The array *str_pool* contains all of the (eight-bit) ASCII codes in all of the strings, and the array *str_start* contains indices of the starting points of each string. Strings are referred to by integer numbers, so that string number s comprises the characters $str\_pool[j]$ for $str\_start[s] \le j < str\_start[s+1]$. Additional integer variables *pool_ptr* and *str_ptr* indicate the number of entries used so far in *str_pool* and *str_start*, respectively; locations *str_pool*[*pool_ptr*] and *str_start*[*str_ptr*] are ready for the next string to be allocated.

String numbers 0 to 255 are reserved for strings that correspond to single ASCII characters. This is in accordance with the conventions of `WEB`, which converts single-character strings into the ASCII code number of the single character involved, while it converts other strings into integers and builds a string pool file. Thus, when the string constant `"."` appears in the program below, `WEB` converts it into the integer 46, which is the ASCII code for a period, while `WEB` will convert a string like `"hello"` into some integer greater than 255. String number 46 will presumably be the single character '`.`'; but some ASCII codes have no standard visible representation, and TEX sometimes needs to be able to print an arbitrary ASCII character, so the first 256 strings are used to specify exactly what should be printed for each of the 256 possibilities.

Elements of the *str_pool* array must be ASCII codes that can actually be printed; i.e., they must have an *xchr* equivalent in the local character set. (This restriction applies only to preloaded strings, not to those generated dynamically by the user.)

Some Pascal compilers won't pack integers into a single byte unless the integers lie in the range $-128\,..\,127$. To accommodate such systems we access the string pool only via macros that can easily be redefined.

define $si(\#) \equiv \#$ { convert from *ASCII_code* to *packed_ASCII_code* }
define $so(\#) \equiv \#$ { convert from *packed_ASCII_code* to *ASCII_code* }

⟨ Types in the outer block 18 ⟩ +≡
pool_pointer = *integer*; { for variables that point into *str_pool* }
str_number = 0 .. *ssup_max_strings*; { for variables that point into *str_start* }
packed_ASCII_code = 0 .. 255; { elements of *str_pool* array }

39*. ⟨ Global variables 13 ⟩ +≡
str_pool: ↑*packed_ASCII_code*; { the characters }
str_start: ↑*pool_pointer*; { the starting pointers }
pool_ptr: *pool_pointer*; { first unused position in *str_pool* }
str_ptr: *str_number*; { number of the current string being created }
init_pool_ptr: *pool_pointer*; { the starting value of *pool_ptr* }
init_str_ptr: *str_number*; { the starting value of *str_ptr* }

47*. The initial values of *str_pool*, *str_start*, *pool_ptr*, and *str_ptr* are computed by the `INITEX` program, based in part on the information that `WEB` has output while processing TEX.

⟨ Declare additional routines for string recycling 1873* ⟩
init function *get_strings_started*: *boolean*;
{ initializes the string pool, but returns *false* if something goes wrong }
label *done*, *exit*;
var *k*, *l*: 0 .. 255; { small indices or counters }
g: *str_number*; { garbage }
begin *pool_ptr* ← 0; *str_ptr* ← 0; *str_start*[0] ← 0; ⟨ Make the first 256 strings 48 ⟩;
⟨ Read the other strings from the `TEX.POOL` file and return *true*, or give an error message and return *false* 51* ⟩;
exit: **end**;
tini

49* The first 128 strings will contain 95 standard ASCII characters, and the other 33 characters will be printed in three-symbol form like '`^^A`' unless a system-dependent change is made here. Installations that have an extended character set, where for example $xchr[\text{'}32] =$ `'≠'`, would like string '32 to be printed as the single character '32 instead of the three characters '136, '136, '132 (`^^Z`). On the other hand, even people with an extended character set will want to represent string '15 by `^^M`, since '15 is *carriage_return*; the idea is to produce visible strings instead of tabs or line-feeds or carriage-returns or bell-rings or characters that are treated anomalously in text files.

Unprintable characters of codes 128–255 are, similarly, rendered `^^80`–`^^ff`.

The boolean expression defined here should be *true* unless TEX internal code number k corresponds to a non-troublesome visible symbol in the local character set. An appropriate formula for the extended character set recommended in *The TEXbook* would, for example, be '$k \in [0, \text{'}10 \mathrel{..} \text{'}12, \text{'}14, \text{'}15, \text{'}33, \text{'}177 \mathrel{..} \text{'}377]$'. If character k cannot be printed, and $k < \text{'}200$, then character $k + \text{'}100$ or $k - \text{'}100$ must be printable; moreover, ASCII codes $[\text{'}41 \mathrel{..} \text{'}46, \text{'}60 \mathrel{..} \text{'}71, \text{'}136, \text{'}141 \mathrel{..} \text{'}146, \text{'}160 \mathrel{..} \text{'}171]$ must be printable. Thus, at least 80 printable characters are needed.

⟨ Character k cannot be printed 49* ⟩ ≡
$(k <$ "␣"$) \lor (k >$ "~"$)$

This code is used in section 48.

51* ⟨ Read the other strings from the `TEX.POOL` file and return *true*, or give an error message and return *false* 51* ⟩ ≡
$g \leftarrow loadpoolstrings((pool\_size - string\_vacancies))$;
if $g = 0$ **then**
begin *wake_up_terminal*; *write_ln*(*term_out*, `'!␣You␣have␣to␣increase␣POOLSIZE.'`);
get_strings_started ← *false*; **return**;
end;
get_strings_started ← *true*;

This code is used in section 47*.

52* Empty module

53* Empty module

54* On-line and off-line printing. Messages that are sent to a user's terminal and to the transcript-log file are produced by several '*print*' procedures. These procedures will direct their output to a variety of places, based on the setting of the global variable *selector*, which has the following possible values:

term_and_log, the normal setting, prints on the terminal and on the transcript file.
log_only, prints only on the transcript file.
term_only, prints only on the terminal.
no_print, doesn't print at all. This is used only in rare cases before the transcript file is open.
pseudo, puts output into a cyclic buffer that is used by the *show_context* routine; when we get to that routine we shall discuss the reasoning behind this curious mode.
new_string, appends the output to the current string in the string pool.
0 to 15, prints on one of the sixteen files for `\write` output.

The symbolic names '*term_and_log*', etc., have been assigned numeric codes that satisfy the convenient relations $no\_print + 1 = term\_only$, $no\_print + 2 = log\_only$, $term\_only + 2 = log\_only + 1 = term\_and\_log$.

Three additional global variables, *tally* and *term_offset* and *file_offset*, record the number of characters that have been printed since they were most recently cleared to zero. We use *tally* to record the length of (possibly very long) stretches of printing; *term_offset* and *file_offset*, on the other hand, keep track of how many characters have appeared so far on the current line that has been output to the terminal or to the transcript file, respectively.

define *no_print* = 16 { *selector* setting that makes data disappear }
define *term_only* = 17 { printing is destined for the terminal only }
define *log_only* = 18 { printing is destined for the transcript file only }
define *term_and_log* = 19 { normal *selector* setting }
define *pseudo* = 20 { special *selector* setting for *show_context* }
define *new_string* = 21 { printing is deflected to the string pool }
define *max_selector* = 21 { highest selector setting }

⟨ Global variables 13 ⟩ +≡
log_file: *alpha_file*; { transcript of TEX session }
selector: 0 .. *max_selector*; { where to print a message }
dig: **array** [0 .. 22] **of** 0 .. 15; { digits in a number being output }
tally: *integer*; { the number of characters recently printed }
term_offset: 0 .. *max_print_line*; { the number of characters on the current terminal line }
file_offset: 0 .. *max_print_line*; { the number of characters on the current file line }
trick_buf: **array** [0 .. *ssup_error_line*] **of** *ASCII_code*; { circular buffer for pseudoprinting }
trick_count: *integer*; { threshold for pseudoprinting, explained later }
first_count: *integer*; { another variable for pseudoprinting }

59* An entire string is output by calling *print*. Note that if we are outputting the single standard ASCII character `c`, we could call *print*(`"c"`), since `"c"` = 99 is the number of a single-character string, as explained above. But *print_char*(`"c"`) is quicker, so TEX goes directly to the *print_char* routine when it knows that this is safe. (The present implementation assumes that it is always safe to print a visible ASCII character.)

⟨Basic printing procedures 57⟩ +≡

```
procedure print(s : integer);   { prints string s }
  label exit;
  var j: pool_pointer;   { current character code position }
    nl: integer;   { new-line character to restore }
  begin if s ≥ str_ptr then s ← "???"   { this can't happen }
  else if s < 256 then
      if s < 0 then s ← "???"   { can't happen }
      else begin if (selector > pseudo) ∧ (¬special_printing) ∧ (¬message_printing) then
          begin print_char(s); return;   { internal strings are not expanded }
          end;
        if (⟨Character s is the current new-line character 262⟩) then
          if selector < pseudo then
            begin print_ln; no_convert ← false; return;
            end
          else if message_printing then
              begin print_char(s); no_convert ← false; return;
              end;
        if (mubyte_log > 0) ∧ (¬no_convert) ∧ (mubyte_write[s] > 0) then s ← mubyte_write[s]
        else if xprn[s] ∨ special_printing then
            begin print_char(s); no_convert ← false; return;
            end;
        no_convert ← false; nl ← new_line_char; new_line_char ← −1;
            { temporarily disable new-line character }
        j ← str_start[s];
        while j < str_start[s + 1] do
          begin print_char(so(str_pool[j])); incr(j);
          end;
        new_line_char ← nl; return;
        end;
  j ← str_start[s];
  while j < str_start[s + 1] do
    begin print_char(so(str_pool[j])); incr(j);
    end;
exit: end;
```

61* Here is the very first thing that TEX prints: a headline that identifies the version number and format package. The *term_offset* variable is temporarily incorrect, but the discrepancy is not serious since we assume that this part of the program is system dependent.

⟨Initialize the output routines 55⟩ +≡
```
if src_specials_p ∨ file_line_error_style_p ∨ parse_first_line_p then wterm(banner_k)
else wterm(banner);
wterm(version_string);
if format_ident = 0 then wterm_ln(´␣(preloaded␣format=´, dump_name, ´)´)
else begin slow_print(format_ident); print_ln;
  end;
if shellenabledp then
  begin wterm(´␣´);
  if restrictedshell then
    begin wterm(´restricted␣´);
    end;
  wterm_ln(´\write18␣enabled.´);
  end;
if src_specials_p then
  begin wterm_ln(´␣Source␣specials␣enabled.´)
  end;
if translate_filename then
  begin wterm(´␣(´); fputs(translate_filename, stdout); wterm_ln(´)´);
  end;
update_terminal;
```

71* Here is a procedure that asks the user to type a line of input, assuming that the *selector* setting is either *term_only* or *term_and_log*. The input is placed into locations *first* through $last - 1$ of the *buffer* array, and echoed on the transcript file if appropriate.

This procedure is never called when $interaction < scroll\_mode$.

```
define prompt_input(#) ≡
        begin wake_up_terminal; print(#); term_input;
        end  { prints a string and gets a line of input }

procedure term_input;  { gets a line from the terminal }
  var k: 0 .. buf_size;  { index into buffer }
  begin update_terminal;  { now the user sees the prompt for sure }
  if ¬input_ln(term_in, true) then fatal_error("End␣of␣file␣on␣the␣terminal!");
  term_offset ← 0;  { the user's line ended with ⟨return⟩ }
  decr(selector);  { prepare to echo the input }
  k ← first;
  while k < last do
    begin print_buffer(k)
    end;
  print_ln; incr(selector);  { restore previous status }
  end;
```

73* The global variable *interaction* has four settings, representing increasing amounts of user interaction:

define *batch_mode* = 0 { omits all stops and omits terminal output }
define *nonstop_mode* = 1 { omits all stops }
define *scroll_mode* = 2 { omits error stops }
define *error_stop_mode* = 3 { stops at every opportunity to interact }
define *unspecified_mode* = 4 { extra value for command-line switch }
define *print_err*(#) ≡
  begin if *interaction* = *error_stop_mode* **then** *wake_up_terminal*;
  if *file_line_error_style_p* **then** *print_file_line*
  else *print_nl*("!␣");
  print(#);
  end

⟨ Global variables 13 ⟩ +≡
interaction: *batch_mode* .. *error_stop_mode*; { current level of interaction }
interaction_option: *batch_mode* .. *unspecified_mode*; { set from command line }

74* ⟨ Set initial values of key variables 21 ⟩ +≡
if *interaction_option* = *unspecified_mode* **then** *interaction* ← *error_stop_mode*
else *interaction* ← *interaction_option*;

81* The *jump_out* procedure just cuts across all active procedure levels. The body of *jump_out* simply calls '*close_files_and_terminate*;' followed by a call on some system procedure that quietly terminates the program.

format *noreturn* ≡ *procedure*
define *do_final_end* ≡
  begin *update_terminal*; *ready_already* ← 0;
  if (*history* ≠ *spotless*) ∧ (*history* ≠ *warning_issued*) **then** *uexit*(1)
  else *uexit*(0);
  end

⟨ Error handling procedures 78 ⟩ +≡
noreturn procedure *jump_out*;
  begin *close_files_and_terminate*; *do_final_end*;
  end;

82* Here now is the general *error* routine.

⟨Error handling procedures 78⟩ +≡
procedure *error*; { completes the job of error reporting }
  label *continue*, *exit*;
  var *c*: *ASCII_code*; { what the user types }
    s1, *s2*, *s3*, *s4*: *integer*; { used to save global variables when deleting tokens }
  begin if *history* < *error_message_issued* **then** *history* ← *error_message_issued*;
  print_char("."); *show_context*;
  if (*halt_on_error_p*) **then**
    begin *history* ← *fatal_error_stop*; *jump_out*;
    end;
  if *interaction* = *error_stop_mode* **then** ⟨Get user's advice and **return** 83⟩;
  incr(*error_count*);
  if *error_count* = 100 **then**
    begin *print_nl*("(That␣makes␣100␣errors;␣please␣try␣again.)"); *history* ← *fatal_error_stop*;
    jump_out;
    end;
  ⟨Put help message on the transcript file 90⟩;
exit: **end**;

84* It is desirable to provide an 'E' option here that gives the user an easy way to return from TEX to the system editor, with the offending line ready to be edited. We do this by calling the external procedure *call_edit* with a pointer to the filename, its length, and the line number. However, here we just set up the variables that will be used as arguments, since we don't want to do the switch-to-editor until after TeX has closed its files.

There is a secret 'D' option available when the debugging routines haven't been commented out.

  define *edit_file* ≡ *input_stack*[*base_ptr*]

⟨Interpret code *c* and **return** if done 84*⟩ ≡
  case *c* **of**
  "0", "1", "2", "3", "4", "5", "6", "7", "8", "9": **if** *deletions_allowed* **then**
      ⟨Delete *c* − "0" tokens and **goto** *continue* 88⟩;
 debug "D": **begin** *debug_help*; **goto** *continue*; **end**; **gubed**
  "E": **if** *base_ptr* > 0 **then**
      if *input_stack*[*base_ptr*].*name_field* ≥ 256 **then**
        begin *edit_name_start* ← *str_start*[*edit_file*.*name_field*];
        edit_name_length ← *str_start*[*edit_file*.*name_field* + 1] − *str_start*[*edit_file*.*name_field*];
        edit_line ← *line*; *jump_out*;
        end;
  "H": ⟨Print the help information and **goto** *continue* 89⟩;
  "I": ⟨Introduce new material from the terminal and **return** 87⟩;
  "Q", "R", "S": ⟨Change the interaction level and **return** 86⟩;
  "X": **begin** *interaction* ← *scroll_mode*; *jump_out*;
    end;
  othercases *do_nothing*
  endcases;
  ⟨Print the menu of available options 85⟩

This code is used in section 83.

93* The following procedure prints TEX's last words before dying.

define *succumb* ≡
  begin if *interaction* = *error_stop_mode* **then** *interaction* ← *scroll_mode*;
    { no more interaction }
  if *log_opened* **then** *error*;
  debug if *interaction* > *batch_mode* **then** *debug_help*;
  gubed
  history ← *fatal_error_stop*; *jump_out*; { irrecoverable error }
  end

⟨ Error handling procedures 78 ⟩ +≡
noreturn procedure *fatal_error*(*s* : *str_number*); { prints *s*, and that's it }
  begin *normalize_selector*;
  print_err("Emergency␣stop"); *help1*(*s*); *succumb*;
  end;

94* Here is the most dreaded error message.

⟨ Error handling procedures 78 ⟩ +≡
noreturn procedure *overflow*(*s* : *str_number*; *n* : *integer*); { stop due to finiteness }
  begin *normalize_selector*; *print_err*("TeX␣capacity␣exceeded,␣sorry␣["); *print*(*s*);
  print_char("="); *print_int*(*n*); *print_char*("]");
  help2("If␣you␣really␣absolutely␣need␣more␣capacity,")
  ("you␣can␣ask␣a␣wizard␣to␣enlarge␣me."); *succumb*;
  end;

95* The program might sometime run completely amok, at which point there is no choice but to stop. If no previous error has been detected, that's bad news; a message is printed that is really intended for the TEX maintenance person instead of the user (unless the user has been particularly diabolical). The index entries for 'this can't happen' may help to pinpoint the problem.

⟨ Error handling procedures 78 ⟩ +≡
noreturn procedure *confusion*(*s* : *str_number*); { consistency check violated; *s* tells where }
  begin *normalize_selector*;
  if *history* < *error_message_issued* **then**
    begin *print_err*("This␣can´t␣happen␣("); *print*(*s*); *print_char*(")");
    help1("I´m␣broken.␣Please␣show␣this␣to␣someone␣who␣can␣fix␣can␣fix");
    end
  else begin *print_err*("I␣can´t␣go␣on␣meeting␣you␣like␣this");
    help2("One␣of␣your␣faux␣pas␣seems␣to␣have␣wounded␣me␣deeply...")
    ("in␣fact,␣I´m␣barely␣conscious.␣Please␣fix␣it␣and␣try␣again.");
    end;
  succumb;
  end;

104* Physical sizes that a TEX user specifies for portions of documents are represented internally as scaled points. Thus, if we define an 'sp' (scaled point) as a unit equal to 2^{-16} printer's points, every dimension inside of TEX is an integer number of sp. There are exactly 4,736,286.72 sp per inch. Users are not allowed to specify dimensions larger than $2^{30} - 1$ sp, which is a distance of about 18.892 feet (5.7583 meters); two such quantities can be added without overflow on a 32-bit computer.

The present implementation of TEX does not check for overflow when dimensions are added or subtracted. This could be done by inserting a few dozen tests of the form '**if** $x \geq$ *´10000000000* **then** *report_overflow*', but the chance of overflow is so remote that such tests do not seem worthwhile.

TEX needs to do only a few arithmetic operations on scaled quantities, other than addition and subtraction, and the following subroutines do most of the work. A single computation might use several subroutine calls, and it is desirable to avoid producing multiple error messages in case of arithmetic overflow; so the routines set the global variable *arith_error* to *true* instead of reporting errors directly to the user. Another global variable, *remainder*, holds the remainder after a division.

define *remainder* ≡ *tex_remainder*

⟨ Global variables 13 ⟩ +≡
arith_error: *boolean*; { has arithmetic overflow occurred recently? }
remainder: *scaled*; { amount subtracted to get an exact division }

109* When TEX "packages" a list into a box, it needs to calculate the proportionality ratio by which the glue inside the box should stretch or shrink. This calculation does not affect TEX's decision making, so the precise details of rounding, etc., in the glue calculation are not of critical importance for the consistency of results on different computers.

We shall use the type *glue_ratio* for such proportionality ratios. A glue ratio should take the same amount of memory as an *integer* (usually 32 bits) if it is to blend smoothly with TEX's other data structures. Thus *glue_ratio* should be equivalent to *short_real* in some implementations of Pascal. Alternatively, it is possible to deal with glue ratios using nothing but fixed-point arithmetic; see *TUGboat* **3**,1 (March 1982), 10–27. (But the routines cited there must be modified to allow negative glue ratios.)

define *set_glue_ratio_zero*(#) ≡ # ← 0.0 { store the representation of zero ratio }
define *set_glue_ratio_one*(#) ≡ # ← 1.0 { store the representation of unit ratio }
define *float*(#) ≡ # { convert from *glue_ratio* to type *real* }
define *unfloat*(#) ≡ # { convert from *real* to type *glue_ratio* }
define *float_constant*(#) ≡ #.0 { convert *integer* constant to *real* }

⟨ Types in the outer block 18 ⟩ +≡

128*. Packed data. In order to make efficient use of storage space, TEX bases its major data structures on a *memory_word*, which contains either a (signed) integer, possibly scaled, or a (signed) *glue_ratio*, or a small number of fields that are one half or one quarter of the size used for storing integers.

If x is a variable of type *memory_word*, it contains up to four fields that can be referred to as follows:

$x.int$	(an *integer*)
$x.sc$	(a *scaled* integer)
$x.gr$	(a *glue_ratio*)
$x.hh.lh$, $x.hh.rh$	(two halfword fields)
$x.hh.b0$, $x.hh.b1$, $x.hh.rh$	(two quarterword fields, one halfword field)
$x.qqqq.b0$, $x.qqqq.b1$, $x.qqqq.b2$, $x.qqqq.b3$	(four quarterword fields)

This is somewhat cumbersome to write, and not very readable either, but macros will be used to make the notation shorter and more transparent. The Pascal code below gives a formal definition of *memory_word* and its subsidiary types, using packed variant records. TEX makes no assumptions about the relative positions of the fields within a word.

Since we are assuming 32-bit integers, a halfword must contain at least 16 bits, and a quarterword must contain at least 8 bits. But it doesn't hurt to have more bits; for example, with enough 36-bit words you might be able to have *mem_max* as large as 262142, which is eight times as much memory as anybody had during the first four years of TEX's existence.

N.B.: Valuable memory space will be dreadfully wasted unless TEX is compiled by a Pascal that packs all of the *memory_word* variants into the space of a single integer. This means, for example, that *glue_ratio* words should be *short_real* instead of *real* on some computers. Some Pascal compilers will pack an integer whose subrange is '0 .. 255' into an eight-bit field, but others insist on allocating space for an additional sign bit; on such systems you can get 256 values into a quarterword only if the subrange is '−128 .. 127'.

The present implementation tries to accommodate as many variations as possible, so it makes few assumptions. If integers having the subrange '*min_quarterword* .. *max_quarterword*' can be packed into a quarterword, and if integers having the subrange '*min_halfword* .. *max_halfword*' can be packed into a halfword, everything should work satisfactorily.

It is usually most efficient to have *min_quarterword* = *min_halfword* = 0, so one should try to achieve this unless it causes a severe problem. The values defined here are recommended for most 32-bit computers.

define *min_quarterword* = 0 { smallest allowable value in a *quarterword* }
define *max_quarterword* = 255 { largest allowable value in a *quarterword* }
define *min_halfword* ≡ −"FFFFFFF { smallest allowable value in a *halfword* }
define *max_halfword* ≡ "FFFFFFF { largest allowable value in a *halfword* }

129*. Here are the inequalities that the quarterword and halfword values must satisfy (or rather, the inequalities that they mustn't satisfy):

⟨ Check the "constant" values for consistency 14 ⟩ +≡
init if $(mem\_min \neq mem\_bot) \lor (mem\_max \neq mem\_top)$ **then** $bad \leftarrow 10$;
tini
if $(mem\_min > mem\_bot) \lor (mem\_max < mem\_top)$ **then** $bad \leftarrow 10$;
if $(min\_quarterword > 0) \lor (max\_quarterword < 127)$ **then** $bad \leftarrow 11$;
if $(min\_halfword > 0) \lor (max\_halfword < 32767)$ **then** $bad \leftarrow 12$;
if $(min\_quarterword < min\_halfword) \lor (max\_quarterword > max\_halfword)$ **then** $bad \leftarrow 13$;
if $(mem\_bot - sup\_main\_memory < min\_halfword) \lor (mem\_top + sup\_main\_memory \geq max\_halfword)$
  then $bad \leftarrow 14$;
if $(max\_font\_max < min\_halfword) \lor (max\_font\_max > max\_halfword)$ **then** $bad \leftarrow 15$;
if $font\_max > font\_base + max\_font\_max$ **then** $bad \leftarrow 16$;
if $(save\_size > max\_halfword) \lor (max\_strings > max\_halfword)$ **then** $bad \leftarrow 17$;
if $buf\_size > max\_halfword$ **then** $bad \leftarrow 18$;
if $max\_quarterword - min\_quarterword < 255$ **then** $bad \leftarrow 19$;

130* The operation of adding or subtracting *min_quarterword* occurs quite frequently in TEX, so it is convenient to abbreviate this operation by using the macros *qi* and *qo* for input and output to and from quarterword format.

The inner loop of TEX will run faster with respect to compilers that don't optimize expressions like '$x + 0$' and '$x - 0$', if these macros are simplified in the obvious way when *min_quarterword* $= 0$. So they have been simplified here in the obvious way.

The WEB source for TEX defines $hi(\#) \equiv \# + min\_halfword$ which can be simplified when $min\_halfword = 0$. The Web2C implementation of TEX can use $hi(\#) \equiv \#$ together with $min\_halfword < 0$ as long as *max_halfword* is sufficiently large.

define $qi(\#) \equiv \#$ { to put an *eight_bits* item into a quarterword }
define $qo(\#) \equiv \#$ { to take an *eight_bits* item from a quarterword }
define $hi(\#) \equiv \#$ { to put a sixteen-bit item into a halfword }
define $ho(\#) \equiv \#$ { to take a sixteen-bit item from a halfword }

131* The reader should study the following definitions closely:

define $sc \equiv int$ { *scaled* data is equivalent to *integer* }

⟨ Types in the outer block 18 ⟩ +≡
quarterword = *min_quarterword* .. *max_quarterword*; *halfword* = *min_halfword* .. *max_halfword*;
two_choices = 1 .. 2; { used when there are two variants in a record }
four_choices = 1 .. 4; { used when there are four variants in a record }
$\boxed{\texttt{\#include␣"texmfmem.h";}}$ *word_file* = *gzFile*;

134*. The *mem* array is divided into two regions that are allocated separately, but the dividing line between these two regions is not fixed; they grow together until finding their "natural" size in a particular job. Locations less than or equal to *lo_mem_max* are used for storing variable-length records consisting of two or more words each. This region is maintained using an algorithm similar to the one described in exercise 2.5–19 of *The Art of Computer Programming*. However, no size field appears in the allocated nodes; the program is responsible for knowing the relevant size when a node is freed. Locations greater than or equal to *hi_mem_min* are used for storing one-word records; a conventional `AVAIL` stack is used for allocation in this region.

Locations of *mem* between *mem_bot* and *mem_top* may be dumped as part of preloaded format files, by the `INITEX` preprocessor. Production versions of TEX may extend the memory at both ends in order to provide more space; locations between *mem_min* and *mem_bot* are always used for variable-size nodes, and locations between *mem_top* and *mem_max* are always used for single-word nodes.

The key pointers that govern *mem* allocation have a prescribed order:

$$null \le mem\_min \le mem\_bot < lo\_mem\_max < hi\_mem\_min < mem\_top \le mem\_end \le mem\_max .$$

Empirical tests show that the present implementation of TEX tends to spend about 9% of its running time allocating nodes, and about 6% deallocating them after their use.

⟨ Global variables 13 ⟩ +≡
yzmem: ↑*memory_word*; { the big dynamic storage area }
zmem: ↑*memory_word*; { the big dynamic storage area }
lo_mem_max: *pointer*; { the largest location of variable-size memory in use }
hi_mem_min: *pointer*; { the smallest location of one-word memory in use }

143*. A call to *get_node* with argument s returns a pointer to a new node of size s, which must be 2 or more. The *link* field of the first word of this new node is set to null. An overflow stop occurs if no suitable space exists.

If *get_node* is called with $s = 2^{30}$, it simply merges adjacent free areas and returns the value *max_halfword*.

function *get_node*(*s* : *integer*): *pointer*; { variable-size node allocation }
  label *found*, *exit*, *restart*;
  var *p*: *pointer*; { the node currently under inspection }
    q: *pointer*; { the node physically after node *p* }
    r: *integer*; { the newly allocated node, or a candidate for this honor }
    t: *integer*; { temporary register }
  begin *restart*: *p* ← *rover*; { start at some free node in the ring }
  repeat ⟨ Try to allocate within node *p* and its physical successors, and **goto** *found* if allocation was possible 145 ⟩;
    p ← *rlink*(*p*); { move to the next node in the ring }
  until *p* = *rover*; { repeat until the whole list has been traversed }
  if *s* = ´10000000000 **then**
    begin *get_node* ← *max_halfword*; **return**;
    end;
  if *lo_mem_max* + 2 < *hi_mem_min* **then**
    if *lo_mem_max* + 2 ≤ *mem_bot* + *max_halfword* **then**
      ⟨ Grow more variable-size memory and **goto** *restart* 144 ⟩;
  overflow("main␣memory␣size", *mem_max* + 1 − *mem_min*); { sorry, nothing satisfactory is left }
found: *link*(*r*) ← *null*; { this node is now nonempty }
  stat *var_used* ← *var_used* + *s*; { maintain usage statistics }
  tats
  ⟨ Initialize bigger nodes with *SyncTEX* information 1913* ⟩;
  get_node ← *r*;
exit: **end**;

153* An *hlist_node* stands for a box that was made from a horizontal list. Each *hlist_node* is seven words long, and contains the following fields (in addition to the mandatory *type* and *link*, which we shall not mention explicitly when discussing the other node types): The *height* and *width* and *depth* are scaled integers denoting the dimensions of the box. There is also a *shift_amount* field, a scaled integer indicating how much this box should be lowered (if it appears in a horizontal list), or how much it should be moved to the right (if it appears in a vertical list). There is a *list_ptr* field, which points to the beginning of the list from which this box was fabricated; if *list_ptr* is *null*, the box is empty. Finally, there are three fields that represent the setting of the glue: *glue_set*(*p*) is a word of type *glue_ratio* that represents the proportionality constant for glue setting; *glue_sign*(*p*) is *stretching* or *shrinking* or *normal* depending on whether or not the glue should stretch or shrink or remain rigid; and *glue_order*(*p*) specifies the order of infinity to which glue setting applies (*normal*, *fil*, *fill*, or *filll*). The *subtype* field is not used in TEX. In ε-TEX the *subtype* field records the box direction mode *box_lr*.

define *synctex_field_size* $= 2$ { Declare the *SyncTEX* field size to store the *SyncTEX* information: 2 integers for file tag and line }
define *sync_tag*(#) $\equiv$ *mem*[# $-$ *synctex_field_size*].*int* { The tag subfield }
define *sync_line*(#) $\equiv$ *mem*[# $-$ *synctex_field_size* $+ 1$].*int* { The line subfield }

define *hlist_node* $= 0$ { *type* of hlist nodes }
define *box_node_size* $= 7 +$ *synctex_field_size* { number of words to allocate for a box node }
define *width_offset* $= 1$ { position of *width* field in a box node }
define *depth_offset* $= 2$ { position of *depth* field in a box node }
define *height_offset* $= 3$ { position of *height* field in a box node }
define *width*(#) $\equiv$ *mem*[# + *width_offset*].*sc* { width of the box, in sp }
define *depth*(#) $\equiv$ *mem*[# + *depth_offset*].*sc* { depth of the box, in sp }
define *height*(#) $\equiv$ *mem*[# + *height_offset*].*sc* { height of the box, in sp }
define *shift_amount*(#) $\equiv$ *mem*[# + 4].*sc* { repositioning distance, in sp }
define *list_offset* $= 5$ { position of *list_ptr* field in a box node }
define *list_ptr*(#) $\equiv$ *link*(# + *list_offset*) { beginning of the list inside the box }
define *glue_order*(#) $\equiv$ *subtype*(# + *list_offset*) { applicable order of infinity }
define *glue_sign*(#) $\equiv$ *type*(# + *list_offset*) { stretching or shrinking }
define *normal* $= 0$ { the most common case when several cases are named }
define *stretching* $= 1$ { glue setting applies to the stretch components }
define *shrinking* $= 2$ { glue setting applies to the shrink components }
define *glue_offset* $= 6$ { position of *glue_set* in a box node }
define *glue_set*(#) $\equiv$ *mem*[# + *glue_offset*].*gr* { a word of type *glue_ratio* for glue setting }

156* A *rule_node* stands for a solid black rectangle; it has *width*, *depth*, and *height* fields just as in an *hlist_node*. However, if any of these dimensions is -2^{30}, the actual value will be determined by running the rule up to the boundary of the innermost enclosing box. This is called a "running dimension." The *width* is never running in an hlist; the *height* and *depth* are never running in a vlist.

define *rule_node* $= 2$ { *type* of rule nodes }
define *rule_node_size* $= 4 +$ *synctex_field_size* { number of words to allocate for a rule node }
define *null_flag* $\equiv -$'*10000000000* { -2^{30}, signifies a missing item }
define *is_running*(#) $\equiv$ (# = *null_flag*) { tests for a running dimension }

159*. A *mark_node* has a *mark_ptr* field that points to the reference count of a token list that contains the user's `\mark` text. In addition there is a *mark_class* field that contains the mark class.

define *mark_node* = 4 { *type* of a mark node }
define *small_node_size* = 2 { number of words to allocate for most node types }
define *medium_node_size* = *small_node_size* + *synctex_field_size* { number of words to allocate for synchronized node types like math, kern, glue and penalty nodes }
define *mark_ptr*(#) ≡ *link*(# + 1) { head of the token list for a mark }
define *mark_class*(#) ≡ *info*(# + 1) { the mark class }

162*. The *new_ligature* function creates a ligature node having given contents of the *font*, *character*, and *lig_ptr* fields. We also have a *new_lig_item* function, which returns a two-word node having a given *character* field. Such nodes are used for temporary processing as ligatures are being created.

function *new_ligature*(*f* : *internal_font_number*; *c* : *quarterword*; *q* : *pointer*): *pointer*;
  var *p*: *pointer*; { the new node }
  begin *p* ← *get_node*(*small_node_size*); *type*(*p*) ← *ligature_node*; *font*(*lig_char*(*p*)) ← *f*;
  character(*lig_char*(*p*)) ← *c*; *lig_ptr*(*p*) ← *q*; *subtype*(*p*) ← 0; *new_ligature* ← *p*;
  end;

function *new_lig_item*(*c* : *quarterword*): *pointer*;
  var *p*: *pointer*; { the new node }
  begin *p* ← *get_node*(*small_node_size*); *character*(*p*) ← *c*; *lig_ptr*(*p*) ← *null*; *new_lig_item* ← *p*;
  end;

165*. A *math_node*, which occurs only in horizontal lists, appears before and after mathematical formulas. The *subtype* field is *before* before the formula and *after* after it. There is a *width* field, which represents the amount of surrounding space inserted by `\mathsurround`.

In addition a *math_node* with *subtype* > *after* and *width* = 0 will be (ab)used to record a regular *math_node* reinserted after being discarded at a line break or one of the text direction primitives (`\beginL`, `\endL`, `\beginR`, and `\endR`).

define *math_node* = 9 { *type* of a math node }
define *before* = 0 { *subtype* for math node that introduces a formula }
define *after* = 1 { *subtype* for math node that winds up a formula }

define *M_code* = 2
define *begin_M_code* = *M_code* + *before* { *subtype* for `\beginM` node }
define *end_M_code* = *M_code* + *after* { *subtype* for `\endM` node }
define *L_code* = 4
define *begin_L_code* = *L_code* + *begin_M_code* { *subtype* for `\beginL` node }
define *end_L_code* = *L_code* + *end_M_code* { *subtype* for `\endL` node }
define *R_code* = *L_code* + *L_code*
define *begin_R_code* = *R_code* + *begin_M_code* { *subtype* for `\beginR` node }
define *end_R_code* = *R_code* + *end_M_code* { *subtype* for `\endR` node }

define *end_LR*(#) ≡ *odd*(*subtype*(#))
define *end_LR_type*(#) ≡ (*L_code* ∗ (*subtype*(#) **div** *L_code*) + *end_M_code*)
define *begin_LR_type*(#) ≡ (# − *after* + *before*)

function *new_math*(*w* : *scaled*; *s* : *small_number*): *pointer*;
  var *p*: *pointer*; { the new node }
  begin *p* ← *get_node*(*medium_node_size*); *type*(*p*) ← *math_node*; *subtype*(*p*) ← *s*; *width*(*p*) ← *w*;
  new_math ← *p*;
  end;

170* And here's a function that creates a glue node for a given parameter identified by its code number; for example, *new_param_glue*(*line_skip_code*) returns a pointer to a glue node for the current `\lineskip`.

function *new_param_glue*(n : *small_number*): *pointer*;
  var p: *pointer*; { the new node }
    q: *pointer*; { the glue specification }
  begin $p \leftarrow$ *get_node*(*medium_node_size*); *type*(p) $\leftarrow$ *glue_node*; *subtype*(p) $\leftarrow n+1$; *leader_ptr*(p) $\leftarrow$ *null*;
  $q \leftarrow$ ⟨ Current *mem* equivalent of glue parameter number n 242 ⟩; *glue_ptr*(p) $\leftarrow q$;
  incr(*glue_ref_count*(q)); *new_param_glue* $\leftarrow p$;
  end;

171* Glue nodes that are more or less anonymous are created by *new_glue*, whose argument points to a glue specification.

function *new_glue*(q : *pointer*): *pointer*;
  var p: *pointer*; { the new node }
  begin $p \leftarrow$ *get_node*(*medium_node_size*); *type*(p) $\leftarrow$ *glue_node*; *subtype*(p) $\leftarrow$ *normal*;
  leader_ptr(p) $\leftarrow$ *null*; *glue_ptr*(p) $\leftarrow q$; *incr*(*glue_ref_count*(q)); *new_glue* $\leftarrow p$;
  end;

174* The *new_kern* function creates a kern node having a given width.

function *new_kern*(w : *scaled*): *pointer*;
  var p: *pointer*; { the new node }
  begin $p \leftarrow$ *get_node*(*medium_node_size*); *type*(p) $\leftarrow$ *kern_node*; *subtype*(p) $\leftarrow$ *normal*; *width*(p) $\leftarrow w$;
  new_kern $\leftarrow p$;
  end;

176* Anyone who has been reading the last few sections of the program will be able to guess what comes next.

function *new_penalty*(m : *integer*): *pointer*;
  var p: *pointer*; { the new node }
  begin $p \leftarrow$ *get_node*(*medium_node_size*); *type*(p) $\leftarrow$ *penalty_node*; *subtype*(p) $\leftarrow 0$;
    { the *subtype* is not used }
  penalty(p) $\leftarrow m$; *new_penalty* $\leftarrow p$;
  end;

183* If TEX is extended improperly, the *mem* array might get screwed up. For example, some pointers might be wrong, or some "dead" nodes might not have been freed when the last reference to them disappeared. Procedures *check_mem* and *search_mem* are available to help diagnose such problems. These procedures make use of two arrays called *free* and *was_free* that are present only if TEX's debugging routines have been included. (You may want to decrease the size of *mem* while you are debugging.)

define *free* ≡ *free_arr*

⟨ Global variables 13 ⟩ +≡
{ The debug memory arrays have not been mallocated yet. }
debug *free*: **packed array** [0 . . 9] **of** *boolean*; { free cells }
was_free: **packed array** [0 . . 9] **of** *boolean*; { previously free cells }
was_mem_end, *was_lo_max*, *was_hi_min*: *pointer*; { previous *mem_end*, *lo_mem_max*, and *hi_mem_min* }
panicking: *boolean*; { do we want to check memory constantly? }
gubed

192* Boxes, rules, inserts, whatsits, marks, and things in general that are sort of "complicated" are indicated only by printing '`[]`'.

```
procedure print_font_identifier(f : internal_font_number);
  begin if pdf_font_blink[f] = null_font then print_esc(font_id_text(f))
  else print_esc(font_id_text(pdf_font_blink[f]));
  if pdf_tracing_fonts > 0 then
    begin print("␣("); print(font_name[f]);
    if font_size[f] ≠ font_dsize[f] then
      begin print("@"); print_scaled(font_size[f]); print("pt");
      end;
    print(")");
    end
  else if pdf_font_expand_ratio[f] ≠ 0 then
      begin print("␣(");
      if pdf_font_expand_ratio[f] > 0 then print("+");
      print_int(pdf_font_expand_ratio[f]); print(")");
      end;
  end;
procedure short_display(p : integer); { prints highlights of list p }
  var n: integer; { for replacement counts }
  begin while p > mem_min do
    begin if is_char_node(p) then
      begin if p ≤ mem_end then
        begin if font(p) ≠ font_in_short_display then
          begin if (font(p) > font_max) then print_char("*")
          else print_font_identifier(font(p));
          print_char("␣"); font_in_short_display ← font(p);
          end;
        print_ASCII(qo(character(p)));
        end;
      end
    else ⟨Print a short indication of the contents of node p 193⟩;
    p ← link(p);
    end;
  end;
```

194* The *show_node_list* routine requires some auxiliary subroutines: one to print a font-and-character combination, one to print a token list without its reference count, and one to print a rule dimension.

```
procedure print_font_and_char(p : integer); { prints char_node data }
  begin if p > mem_end then print_esc("CLOBBERED.")
  else begin if (font(p) > font_max) then print_char("*")
    else print_font_identifier(font(p));
    print_char("␣"); print_ASCII(qo(character(p)));
    end;
  end;
procedure print_mark(p : integer); { prints token list data in braces }
  begin print_char("{");
  if (p < hi_mem_min) ∨ (p > mem_end) then print_esc("CLOBBERED.")
  else show_token_list(link(p), null, max_print_line − 10);
  print_char("}");
  end;
procedure print_rule_dimen(d : scaled); { prints dimension in rule node }
  begin if is_running(d) then print_char("*")
  else print_scaled(d);
  end;
```

204* The code will have to change in this place if *glue_ratio* is a structured type instead of an ordinary *real*. Note that this routine should avoid arithmetic errors even if the *glue_set* field holds an arbitrary random value. The following code assumes that a properly formed nonzero *real* number has absolute value 2^{20} or more when it is regarded as an integer; this precaution was adequate to prevent floating point underflow on the author's computer.

```
⟨Display the value of glue_set(p) 204*⟩ ≡
  g ← float(glue_set(p));
  if (g ≠ float_constant(0)) ∧ (glue_sign(p) ≠ normal) then
    begin print(",␣glue␣set␣");
    if glue_sign(p) = shrinking then print("-␣"); { The Unix pc folks removed this restriction with a
          remark that invalid bit patterns were vanishingly improbable, so we follow their example without
          really understanding it. if abs(mem[p + glue_offset].int) < '4000000 then print('?.?') else }
    if fabs(g) > float_constant(20000) then
      begin if g > float_constant(0) then print_char(">")
      else print("<␣-");
      print_glue(20000 ∗ unity, glue_order(p), 0);
      end
    else print_glue(round(unity ∗ g), glue_order(p), 0);
    end
```

This code is used in section 202.

220* Now we are ready to delete any node list, recursively. In practice, the nodes deleted are usually charnodes (about 2/3 of the time), and they are glue nodes in about half of the remaining cases.

procedure *flush_node_list*(*p* : *pointer*); { erase list of nodes starting at *p* }
  label *done*; { go here when node *p* has been freed }
  var *q*: *pointer*; { successor to node *p* }
  begin while $p \neq null$ **do**
    begin $q \leftarrow link(p)$;
    if *is_char_node*(*p*) **then** *free_avail*(*p*)
    else begin case *type*(*p*) **of**
      hlist_node, *vlist_node*, *unset_node*: **begin** *flush_node_list*(*list_ptr*(*p*)); *free_node*(*p*, *box_node_size*);
        goto *done*;
        end;
      rule_node: **begin** *free_node*(*p*, *rule_node_size*); **goto** *done*;
        end;
      ins_node: **begin** *flush_node_list*(*ins_ptr*(*p*)); *delete_glue_ref*(*split_top_ptr*(*p*));
        free_node(*p*, *ins_node_size*); **goto** *done*;
        end;
      whatsit_node: ⟨ Wipe out the whatsit node *p* and **goto** *done* 1602 ⟩;
      glue_node: **begin** *fast_delete_glue_ref*(*glue_ptr*(*p*));
        if $leader\_ptr(p) \neq null$ **then** *flush_node_list*(*leader_ptr*(*p*));
        free_node(*p*, *medium_node_size*); **goto** *done*;
        end;
      kern_node, *math_node*, *penalty_node*: **begin** *free_node*(*p*, *medium_node_size*); **goto** *done*;
        end;
      margin_kern_node: **begin** *free_avail*(*margin_char*(*p*)); *free_node*(*p*, *margin_kern_node_size*);
        goto *done*;
        end;
      ligature_node: *flush_node_list*(*lig_ptr*(*p*));
      mark_node: *delete_token_ref*(*mark_ptr*(*p*));
      disc_node: **begin** *flush_node_list*(*pre_break*(*p*)); *flush_node_list*(*post_break*(*p*));
        end;
      adjust_node: *flush_node_list*(*adjust_ptr*(*p*));
      ⟨ Cases of *flush_node_list* that arise in mlists only 872 ⟩
      othercases *confusion*(`"flushing"`)
      endcases;
      free_node(*p*, *small_node_size*);
    done: **end**;
    $p \leftarrow q$;
    end;
  end;

224* ⟨ Case statement to copy different types and set *words* to the number of initial words not yet copied 224* ⟩ ≡

case *type*(*p*) **of**
hlist_node, *vlist_node*, *unset_node*: **begin** $r \leftarrow$ *get_node*(*box_node_size*);
⟨ Copy the box *SyncTEX* information 1932* ⟩;
mem[$r + 6$] $\leftarrow$ *mem*[$p + 6$]; *mem*[$r + 5$] $\leftarrow$ *mem*[$p + 5$]; { copy the last two words }
list_ptr(*r*) $\leftarrow$ *copy_node_list*(*list_ptr*(*p*)); { this affects *mem*[$r + 5$] }
words $\leftarrow$ 5;
end;
rule_node: **begin** $r \leftarrow$ *get_node*(*rule_node_size*); *words* $\leftarrow$ *rule_node_size* − *synctex_field_size*;
{ *SyncTEX*: do not let TEX copy the *SyncTEX* information }
⟨ Copy the rule *SyncTEX* information 1933* ⟩;
end;
ins_node: **begin** $r \leftarrow$ *get_node*(*ins_node_size*); *mem*[$r + 4$] $\leftarrow$ *mem*[$p + 4$]; *add_glue_ref*(*split_top_ptr*(*p*));
ins_ptr(*r*) $\leftarrow$ *copy_node_list*(*ins_ptr*(*p*)); { this affects *mem*[$r + 4$] }
words $\leftarrow$ *ins_node_size* − 1;
end;
whatsit_node: ⟨ Make a partial copy of the whatsit node *p* and make *r* point to it; set *words* to the number of initial words not yet copied 1601 ⟩;
glue_node: **begin** $r \leftarrow$ *get_node*(*medium_node_size*); *add_glue_ref*(*glue_ptr*(*p*));
⟨ Copy the medium sized node *SyncTEX* information 1934* ⟩;
glue_ptr(*r*) $\leftarrow$ *glue_ptr*(*p*); *leader_ptr*(*r*) $\leftarrow$ *copy_node_list*(*leader_ptr*(*p*));
end;
kern_node, *math_node*, *penalty_node*: **begin** $r \leftarrow$ *get_node*(*medium_node_size*);
words $\leftarrow$ *medium_node_size*;
end;
margin_kern_node: **begin** $r \leftarrow$ *get_node*(*margin_kern_node_size*); *fast_get_avail*(*margin_char*(*r*));
font(*margin_char*(*r*)) $\leftarrow$ *font*(*margin_char*(*p*));
character(*margin_char*(*r*)) $\leftarrow$ *character*(*margin_char*(*p*)); *words* $\leftarrow$ *small_node_size*;
end;
ligature_node: **begin** $r \leftarrow$ *get_node*(*small_node_size*); *mem*[*lig_char*(*r*)] $\leftarrow$ *mem*[*lig_char*(*p*)];
{ copy *font* and *character* }
lig_ptr(*r*) $\leftarrow$ *copy_node_list*(*lig_ptr*(*p*));
end;
disc_node: **begin** $r \leftarrow$ *get_node*(*small_node_size*); *pre_break*(*r*) $\leftarrow$ *copy_node_list*(*pre_break*(*p*));
post_break(*r*) $\leftarrow$ *copy_node_list*(*post_break*(*p*));
end;
mark_node: **begin** $r \leftarrow$ *get_node*(*small_node_size*); *add_token_ref*(*mark_ptr*(*p*));
words $\leftarrow$ *small_node_size*;
end;
adjust_node: **begin** $r \leftarrow$ *get_node*(*small_node_size*); *adjust_ptr*(*r*) $\leftarrow$ *copy_node_list*(*adjust_ptr*(*p*));
end; { *words* = 1 = *small_node_size* − 1 }
othercases *confusion*("copying")
endcases

This code is used in section 223.

227* The next codes are special; they all relate to mode-independent assignment of values to TEX's internal registers or tables. Codes that are *max_internal* or less represent internal quantities that might be expanded by '\the'.

define *toks_register* = 71 { token list register (\toks) }
define *assign_toks* = 72 { special token list (\output, \everypar, etc.) }
define *assign_int* = 73 { user-defined integer (\tolerance, \day, etc.) }
define *assign_dimen* = 74 { user-defined length (\hsize, etc.) }
define *assign_glue* = 75 { user-defined glue (\baselineskip, etc.) }
define *assign_mu_glue* = 76 { user-defined muglue (\thinmuskip, etc.) }
define *assign_font_dimen* = 77 { user-defined font dimension (\fontdimen) }
define *assign_font_int* = 78 { user-defined font integer (\hyphenchar, \skewchar) }
define *set_aux* = 79 { specify state info (\spacefactor, \prevdepth) }
define *set_prev_graf* = 80 { specify state info (\prevgraf) }
define *set_page_dimen* = 81 { specify state info (\pagegoal, etc.) }
define *set_page_int* = 82 { specify state info (\deadcycles, \insertpenalties) }
{ (or \interactionmode) }
define *set_box_dimen* = 83 { change dimension of box (\wd, \ht, \dp) }
define *set_shape* = 84 { specify fancy paragraph shape (\parshape) }
{ (or \interlinepenalties, etc.) }
define *def_code* = 85 { define a character code (\catcode, etc.) }
define *def_family* = 86 { declare math fonts (\textfont, etc.) }
define *set_font* = 87 { set current font (font identifiers) }
define *def_font* = 88 { define a font file (\font) }
define *register* = 89 { internal register (\count, \dimen, etc.) }
define *max_internal* = 89 { the largest code that can follow \the }
define *advance* = 90 { advance a register or parameter (\advance) }
define *multiply* = 91 { multiply a register or parameter (\multiply) }
define *divide* = 92 { divide a register or parameter (\divide) }
define *prefix* = 93 { qualify a definition (\global, \long, \outer) }
{ (or \protected) }
define *let* = 94 { assign a command code (\let, \futurelet) }
define *shorthand_def* = 95 { code definition (\chardef, \countdef, etc.) }
{ or \charsubdef }
define *read_to_cs* = 96 { read into a control sequence (\read) }
{ (or \readline) }
define *def* = 97 { macro definition (\def, \gdef, \xdef, \edef) }
define *set_box* = 98 { set a box (\setbox) }
define *hyph_data* = 99 { hyphenation data (\hyphenation, \patterns) }
define *set_interaction* = 100 { define level of interaction (\batchmode, etc.) }
define *letterspace_font* = 101 { letterspace a font (\letterspacefont) }
define *pdf_copy_font* = 102 { create a new font instance (\pdfcopyfont) }
define *max_command* = 102 { the largest command code seen at *big_switch* }

229* The semantic nest. TEX is typically in the midst of building many lists at once. For example, when a math formula is being processed, TEX is in math mode and working on an mlist; this formula has temporarily interrupted TEX from being in horizontal mode and building the hlist of a paragraph; and this paragraph has temporarily interrupted TEX from being in vertical mode and building the vlist for the next page of a document. Similarly, when a `\vbox` occurs inside of an `\hbox`, TEX is temporarily interrupted from working in restricted horizontal mode, and it enters internal vertical mode. The "semantic nest" is a stack that keeps track of what lists and modes are currently suspended.

At each level of processing we are in one of six modes:

vmode stands for vertical mode (the page builder);
hmode stands for horizontal mode (the paragraph builder);
mmode stands for displayed formula mode;
$-$*vmode* stands for internal vertical mode (e.g., in a `\vbox`);
$-$*hmode* stands for restricted horizontal mode (e.g., in an `\hbox`);
$-$*mmode* stands for math formula mode (not displayed).

The mode is temporarily set to zero while processing `\write` texts.

Numeric values are assigned to *vmode*, *hmode*, and *mmode* so that TEX's "big semantic switch" can select the appropriate thing to do by computing the value $abs(mode) + cur\_cmd$, where *mode* is the current mode and *cur_cmd* is the current command code.

```
  define vmode = 1  { vertical mode }
  define hmode = vmode + max_command + 1  { horizontal mode }
  define mmode = hmode + max_command + 1  { math mode }
procedure print_mode(m : integer);  { prints the mode represented by m }
  begin if m > 0 then
    case m div (max_command + 1) of
    0: print("vertical␣mode");
    1: print("horizontal␣mode");
    2: print("display␣math␣mode");
    end
  else if m = 0 then print("no␣mode")
    else case (−m) div (max_command + 1) of
      0: print("internal␣vertical␣mode");
      1: print("restricted␣horizontal␣mode");
      2: print("math␣mode");
      end;
  end;
procedure print_in_mode(m : integer);  { prints the mode represented by m }
  begin if m > 0 then
    case m div (max_command + 1) of
    0: print("´␣in␣vertical␣mode");
    1: print("´␣in␣horizontal␣mode");
    2: print("´␣in␣display␣math␣mode");
    end
  else if m = 0 then print("´␣in␣no␣mode")
    else case (−m) div (max_command + 1) of
      0: print("´␣in␣internal␣vertical␣mode");
      1: print("´␣in␣restricted␣horizontal␣mode");
      2: print("´␣in␣math␣mode");
      end;
  end;
```

231* **define** *mode* ≡ *cur_list.mode_field* { current mode }
define *head* ≡ *cur_list.head_field* { header node of current list }
define *tail* ≡ *cur_list.tail_field* { final node on current list }
define *eTeX_aux* ≡ *cur_list.eTeX_aux_field* { auxiliary data for ε-TEX }
define *LR_save* ≡ *eTeX_aux* { LR stack when a paragraph is interrupted }
define *LR_box* ≡ *eTeX_aux* { prototype box for display }
define *delim_ptr* ≡ *eTeX_aux* { most recent left or right noad of a math left group }
define *prev_graf* ≡ *cur_list.pg_field* { number of paragraph lines accumulated }
define *aux* ≡ *cur_list.aux_field* { auxiliary data about the current list }
define *prev_depth* ≡ *aux.sc* { the name of *aux* in vertical mode }
define *space_factor* ≡ *aux.hh.lh* { part of *aux* in horizontal mode }
define *clang* ≡ *aux.hh.rh* { the other part of *aux* in horizontal mode }
define *incompleat_noad* ≡ *aux.int* { the name of *aux* in math mode }
define *mode_line* ≡ *cur_list.ml_field* { source file line number at beginning of list }

⟨ Global variables 13 ⟩ +≡
nest: ↑*list_state_record*;
nest_ptr: 0 .. *nest_size*; { first unused location of *nest* }
max_nest_stack: 0 .. *nest_size*; { maximum of *nest_ptr* when pushing }
cur_list: *list_state_record*; { the "top" semantic state }
shown_mode: −*mmode* .. *mmode*; { most recent mode shown by `\tracingcommands` }
save_tail: *pointer*; { save *tail* so we can examine whether we have an auto kern before a glue }

233* We will see later that the vertical list at the bottom semantic level is split into two parts; the "current page" runs from *page_head* to *page_tail*, and the "contribution list" runs from *contrib_head* to *tail* of semantic level zero. The idea is that contributions are first formed in vertical mode, then "contributed" to the current page (during which time the page-breaking decisions are made). For now, we don't need to know any more details about the page-building process.

⟨ Set initial values of key variables 21 ⟩ +≡
nest_ptr ← 0; *max_nest_stack* ← 0; *mode* ← *vmode*; *head* ← *contrib_head*; *tail* ← *contrib_head*; *eTeX_aux* ← *null*; *save_tail* ← *null*; *prev_depth* ← *ignore_depth*; *mode_line* ← 0; *prev_graf* ← 0; *shown_mode* ← 0;
{ The following piece of code is a copy of module 991: }
page_contents ← *empty*; *page_tail* ← *page_head*; { *link*(*page_head*) ← *null*; }
last_glue ← *max_halfword*; *last_penalty* ← 0; *last_kern* ← 0; *last_node_type* ← −1; *page_depth* ← 0; *page_max_depth* ← 0;

237* ⟨Show the auxiliary field, a 237*⟩ ≡

```
case abs(m) div (max_command + 1) of
0: begin print_nl("prevdepth␣");
  if a.sc ≤ pdf_ignored_dimen then print("ignored")
  else print_scaled(a.sc);
  if nest[p].pg_field ≠ 0 then
    begin print(",␣prevgraf␣"); print_int(nest[p].pg_field);
    if nest[p].pg_field ≠ 1 then print("␣lines")
    else print("␣line");
    end;
  end;
1: begin print_nl("spacefactor␣"); print_int(a.hh.lh);
  if m > 0 then if a.hh.rh > 0 then
      begin print(",␣current␣language␣"); print_int(a.hh.rh); end;
  end;
2: if a.int ≠ null then
    begin print("this␣will␣begin␣denominator␣of:"); show_box(a.int); end;
end { there are no other cases }
```

This code is used in section 236.

238* The table of equivalents. Now that we have studied the data structures for TEX's semantic routines, we ought to consider the data structures used by its syntactic routines. In other words, our next concern will be the tables that TEX looks at when it is scanning what the user has written.

The biggest and most important such table is called *eqtb*. It holds the current "equivalents" of things; i.e., it explains what things mean or what their current values are, for all quantities that are subject to the nesting structure provided by TEX's grouping mechanism. There are six parts to *eqtb*:

1) $eqtb[active\_base \mathrel{.\,.} (hash\_base - 1)]$ holds the current equivalents of single-character control sequences.

2) $eqtb[hash\_base \mathrel{.\,.} (glue\_base - 1)]$ holds the current equivalents of multiletter control sequences.

3) $eqtb[glue\_base \mathrel{.\,.} (local\_base - 1)]$ holds the current equivalents of glue parameters like the current baselineskip.

4) $eqtb[local\_base \mathrel{.\,.} (int\_base - 1)]$ holds the current equivalents of local halfword quantities like the current box registers, the current "catcodes," the current font, and a pointer to the current paragraph shape. Additionally region 4 contains the table with MLTEX's character substitution definitions.

5) $eqtb[int\_base \mathrel{.\,.} (dimen\_base - 1)]$ holds the current equivalents of fullword integer parameters like the current hyphenation penalty.

6) $eqtb[dimen\_base \mathrel{.\,.} eqtb\_size]$ holds the current equivalents of fullword dimension parameters like the current hsize or amount of hanging indentation.

Note that, for example, the current amount of baselineskip glue is determined by the setting of a particular location in region 3 of *eqtb*, while the current meaning of the control sequence '`\baselineskip`' (which might have been changed by `\def` or `\let`) appears in region 2.

240* Many locations in *eqtb* have symbolic names. The purpose of the next paragraphs is to define these names, and to set up the initial values of the equivalents.

In the first region we have 256 equivalents for "active characters" that act as control sequences, followed by 256 equivalents for single-character control sequences.

Then comes region 2, which corresponds to the hash table that we will define later. The maximum address in this region is used for a dummy control sequence that is perpetually undefined. There also are several locations for control sequences that are perpetually defined (since they are used in error recovery).

define *active_base* = 1 { beginning of region 1, for active character equivalents }
define *single_base* = *active_base* + 256 { equivalents of one-character control sequences }
define *null_cs* = *single_base* + 256 { equivalent of `\csname\endcsname` }
define *hash_base* = *null_cs* + 1 { beginning of region 2, for the hash table }
define *frozen_control_sequence* = *hash_base* + *hash_size* { for error recovery }
define *frozen_protection* = *frozen_control_sequence* { inaccessible but definable }
define *frozen_cr* = *frozen_control_sequence* + 1 { permanent '`\cr`' }
define *frozen_end_group* = *frozen_control_sequence* + 2 { permanent '`\endgroup`' }
define *frozen_right* = *frozen_control_sequence* + 3 { permanent '`\right`' }
define *frozen_fi* = *frozen_control_sequence* + 4 { permanent '`\fi`' }
define *frozen_end_template* = *frozen_control_sequence* + 5 { permanent '`\endtemplate`' }
define *frozen_endv* = *frozen_control_sequence* + 6 { second permanent '`\endtemplate`' }
define *frozen_relax* = *frozen_control_sequence* + 7 { permanent '`\relax`' }
define *end_write* = *frozen_control_sequence* + 8 { permanent '`\endwrite`' }
define *frozen_dont_expand* = *frozen_control_sequence* + 9 { permanent '`\notexpanded:`' }
define *prim_size* = 2100 { maximum number of primitives }
define *frozen_special* = *frozen_control_sequence* + 10 { permanent '`\special`' }
define *frozen_null_font* = *frozen_control_sequence* + 12 + *prim_size* { permanent '`\nullfont`' }
define *frozen_primitive* = *frozen_control_sequence* + 11 { permanent '`\pdfprimitive`' }
define *prim_eqtb_base* = *frozen_primitive* + 1
define *font_id_base* = *frozen_null_font* − *font_base* { begins table of 257 permanent font identifiers }
define *undefined_control_sequence* = *frozen_null_font* + *max_font_max* + 1 { dummy location }
define *glue_base* = *undefined_control_sequence* + 1 { beginning of region 3 }

⟨ Initialize table entries (done by `INITEX` only) 182 ⟩ +≡
eq_type(*undefined_control_sequence*) ← *undefined_cs*; *equiv*(*undefined_control_sequence*) ← *null*;
eq_level(*undefined_control_sequence*) ← *level_zero*;
for *k* ← *active_base* **to** *eqtb_top* **do** *eqtb*[*k*] ← *eqtb*[*undefined_control_sequence*];

248* Region 4 of *eqtb* contains the local quantities defined here. The bulk of this region is taken up by five tables that are indexed by eight-bit characters; these tables are important to both the syntactic and semantic portions of TEX. There are also a bunch of special things like font and token parameters, as well as the tables of `\toks` and `\box` registers.

define *par_shape_loc* = *local_base* { specifies paragraph shape }
define *output_routine_loc* = *local_base* + 1 { points to token list for `\output` }
define *every_par_loc* = *local_base* + 2 { points to token list for `\everypar` }
define *every_math_loc* = *local_base* + 3 { points to token list for `\everymath` }
define *every_display_loc* = *local_base* + 4 { points to token list for `\everydisplay` }
define *every_hbox_loc* = *local_base* + 5 { points to token list for `\everyhbox` }
define *every_vbox_loc* = *local_base* + 6 { points to token list for `\everyvbox` }
define *every_job_loc* = *local_base* + 7 { points to token list for `\everyjob` }
define *every_cr_loc* = *local_base* + 8 { points to token list for `\everycr` }
define *err_help_loc* = *local_base* + 9 { points to token list for `\errhelp` }
define *tex_toks* = *local_base* + 10 { end of TEX's token list parameters }

define *pdftex_first_loc* = *tex_toks* { base for pdfTEX's token list parameters }
define *pdf_pages_attr_loc* = *pdftex_first_loc* + 0 { points to token list for `\pdfpagesattr` }
define *pdf_page_attr_loc* = *pdftex_first_loc* + 1 { points to token list for `\pdfpageattr` }
define *pdf_page_resources_loc* = *pdftex_first_loc* + 2 { points to token list for `\pdfpageresources` }
define *pdf_pk_mode_loc* = *pdftex_first_loc* + 3 { points to token list for `\pdfpkmode` }
define *pdf_toks* = *pdftex_first_loc* + 4 { end of pdfTEX's token list parameters }

define *etex_toks_base* = *pdf_toks* { base for ε-TEX's token list parameters }
define *every_eof_loc* = *etex_toks_base* { points to token list for `\everyeof` }
define *etex_toks* = *etex_toks_base* + 1 { end of ε-TEX's token list parameters }

define *toks_base* = *etex_toks* { table of 256 token list registers }

define *etex_pen_base* = *toks_base* + 256 { start of table of ε-TEX's penalties }
define *inter_line_penalties_loc* = *etex_pen_base* { additional penalties between lines }
define *club_penalties_loc* = *etex_pen_base* + 1 { penalties for creating club lines }
define *widow_penalties_loc* = *etex_pen_base* + 2 { penalties for creating widow lines }
define *display_widow_penalties_loc* = *etex_pen_base* + 3 { ditto, just before a display }
define *etex_pens* = *etex_pen_base* + 4 { end of table of ε-TEX's penalties }

define *box_base* = *etex_pens* { table of 256 box registers }
define *cur_font_loc* = *box_base* + 256 { internal font number outside math mode }
define *xord_code_base* = *cur_font_loc* + 1
define *xchr_code_base* = *xord_code_base* + 1
define *xprn_code_base* = *xchr_code_base* + 1
define *math_font_base* = *xprn_code_base* + 1
define *cat_code_base* = *math_font_base* + 48 { table of 256 command codes (the "catcodes") }
define *lc_code_base* = *cat_code_base* + 256 { table of 256 lowercase mappings }
define *uc_code_base* = *lc_code_base* + 256 { table of 256 uppercase mappings }
define *sf_code_base* = *uc_code_base* + 256 { table of 256 spacefactor mappings }
define *math_code_base* = *sf_code_base* + 256 { table of 256 math mode mappings }
define *char_sub_code_base* = *math_code_base* + 256 { table of character substitutions }
define *int_base* = *char_sub_code_base* + 256 { beginning of region 5 }

define *par_shape_ptr* ≡ *equiv*(*par_shape_loc*)
define *output_routine* ≡ *equiv*(*output_routine_loc*)
define *every_par* ≡ *equiv*(*every_par_loc*)
define *every_math* ≡ *equiv*(*every_math_loc*)
define *every_display* ≡ *equiv*(*every_display_loc*)
define *every_hbox* ≡ *equiv*(*every_hbox_loc*)
define *every_vbox* ≡ *equiv*(*every_vbox_loc*)

define *every_job* $\equiv$ *equiv*(*every_job_loc*)
define *every_cr* $\equiv$ *equiv*(*every_cr_loc*)
define *err_help* $\equiv$ *equiv*(*err_help_loc*)
define *pdf_pages_attr* $\equiv$ *equiv*(*pdf_pages_attr_loc*)
define *pdf_page_attr* $\equiv$ *equiv*(*pdf_page_attr_loc*)
define *pdf_page_resources* $\equiv$ *equiv*(*pdf_page_resources_loc*)
define *pdf_pk_mode* $\equiv$ *equiv*(*pdf_pk_mode_loc*)
define *toks*(#) $\equiv$ *equiv*(*toks_base* + #)
define *box*(#) $\equiv$ *equiv*(*box_base* + #)
define *cur_font* $\equiv$ *equiv*(*cur_font_loc*)
define *fam_fnt*(#) $\equiv$ *equiv*(*math_font_base* + #)
define *cat_code*(#) $\equiv$ *equiv*(*cat_code_base* + #)
define *lc_code*(#) $\equiv$ *equiv*(*lc_code_base* + #)
define *uc_code*(#) $\equiv$ *equiv*(*uc_code_base* + #)
define *sf_code*(#) $\equiv$ *equiv*(*sf_code_base* + #)
define *math_code*(#) $\equiv$ *equiv*(*math_code_base* + #)
{ Note: *math_code*(*c*) is the true math code plus *min_halfword* }
define *char_sub_code*(#) $\equiv$ *equiv*(*char_sub_code_base* + #)
{ Note: *char_sub_code*(*c*) is the true substitution info plus *min_halfword* }

⟨ Put each of TEX's primitives into the hash table 244 ⟩ +≡
primitive("output", *assign_toks*, *output_routine_loc*); *primitive*("everypar", *assign_toks*, *every_par_loc*);
primitive("everymath", *assign_toks*, *every_math_loc*);
primitive("everydisplay", *assign_toks*, *every_display_loc*);
primitive("everyhbox", *assign_toks*, *every_hbox_loc*); *primitive*("everyvbox", *assign_toks*, *every_vbox_loc*);
primitive("everyjob", *assign_toks*, *every_job_loc*); *primitive*("everycr", *assign_toks*, *every_cr_loc*);
primitive("errhelp", *assign_toks*, *err_help_loc*);
primitive("pdfpagesattr", *assign_toks*, *pdf_pages_attr_loc*);
primitive("pdfpageattr", *assign_toks*, *pdf_page_attr_loc*);
primitive("pdfpageresources", *assign_toks*, *pdf_page_resources_loc*);
primitive("pdfpkmode", *assign_toks*, *pdf_pk_mode_loc*);

254* Region 5 of *eqtb* contains the integer parameters and registers defined here, as well as the *del_code* table. The latter table differs from the *cat_code* .. *math_code* tables that precede it, since delimiter codes are fullword integers while the other kinds of codes occupy at most a halfword. This is what makes region 5 different from region 4. We will store the *eq_level* information in an auxiliary array of quarterwords that will be defined later.

define *pretolerance_code* = 0 { badness tolerance before hyphenation }
define *tolerance_code* = 1 { badness tolerance after hyphenation }
define *line_penalty_code* = 2 { added to the badness of every line }
define *hyphen_penalty_code* = 3 { penalty for break after discretionary hyphen }
define *ex_hyphen_penalty_code* = 4 { penalty for break after explicit hyphen }
define *club_penalty_code* = 5 { penalty for creating a club line }
define *widow_penalty_code* = 6 { penalty for creating a widow line }
define *display_widow_penalty_code* = 7 { ditto, just before a display }
define *broken_penalty_code* = 8 { penalty for breaking a page at a broken line }
define *bin_op_penalty_code* = 9 { penalty for breaking after a binary operation }
define *rel_penalty_code* = 10 { penalty for breaking after a relation }
define *pre_display_penalty_code* = 11 { penalty for breaking just before a displayed formula }
define *post_display_penalty_code* = 12 { penalty for breaking just after a displayed formula }
define *inter_line_penalty_code* = 13 { additional penalty between lines }
define *double_hyphen_demerits_code* = 14 { demerits for double hyphen break }
define *final_hyphen_demerits_code* = 15 { demerits for final hyphen break }
define *adj_demerits_code* = 16 { demerits for adjacent incompatible lines }
define *mag_code* = 17 { magnification ratio }
define *delimiter_factor_code* = 18 { ratio for variable-size delimiters }
define *looseness_code* = 19 { change in number of lines for a paragraph }
define *time_code* = 20 { current time of day }
define *day_code* = 21 { current day of the month }
define *month_code* = 22 { current month of the year }
define *year_code* = 23 { current year of our Lord }
define *show_box_breadth_code* = 24 { nodes per level in *show_box* }
define *show_box_depth_code* = 25 { maximum level in *show_box* }
define *hbadness_code* = 26 { hboxes exceeding this badness will be shown by *hpack* }
define *vbadness_code* = 27 { vboxes exceeding this badness will be shown by *vpack* }
define *pausing_code* = 28 { pause after each line is read from a file }
define *tracing_online_code* = 29 { show diagnostic output on terminal }
define *tracing_macros_code* = 30 { show macros as they are being expanded }
define *tracing_stats_code* = 31 { show memory usage if TEX knows it }
define *tracing_paragraphs_code* = 32 { show line-break calculations }
define *tracing_pages_code* = 33 { show page-break calculations }
define *tracing_output_code* = 34 { show boxes when they are shipped out }
define *tracing_lost_chars_code* = 35 { show characters that aren't in the font }
define *tracing_commands_code* = 36 { show command codes at *big_switch* }
define *tracing_restores_code* = 37 { show equivalents when they are restored }
define *uc_hyph_code* = 38 { hyphenate words beginning with a capital letter }
define *output_penalty_code* = 39 { penalty found at current page break }
define *max_dead_cycles_code* = 40 { bound on consecutive dead cycles of output }
define *hang_after_code* = 41 { hanging indentation changes after this many lines }
define *floating_penalty_code* = 42 { penalty for insertions held over after a split }
define *global_defs_code* = 43 { override `\global` specifications }
define *cur_fam_code* = 44 { current family }
define *escape_char_code* = 45 { escape character for token output }
define *default_hyphen_char_code* = 46 { value of `\hyphenchar` when a font is loaded }

define *default_skew_char_code* = 47 { value of `\skewchar` when a font is loaded }
define *end_line_char_code* = 48 { character placed at the right end of the buffer }
define *new_line_char_code* = 49 { character that prints as *print_ln* }
define *language_code* = 50 { current hyphenation table }
define *left_hyphen_min_code* = 51 { minimum left hyphenation fragment size }
define *right_hyphen_min_code* = 52 { minimum right hyphenation fragment size }
define *holding_inserts_code* = 53 { do not remove insertion nodes from `\box255` }
define *error_context_lines_code* = 54 { maximum intermediate line pairs shown }
define *tex_int_pars* = 55 { total number of TEX's integer parameters }

define *web2c_int_base* = *tex_int_pars* { base for web2c's integer parameters }
define *char_sub_def_min_code* = *web2c_int_base* { smallest value in the charsubdef list }
define *char_sub_def_max_code* = *web2c_int_base* + 1 { largest value in the charsubdef list }
define *tracing_char_sub_def_code* = *web2c_int_base* + 2 { traces changes to a charsubdef def }
define *tracing_stack_levels_code* = *web2c_int_base* + 3
{ tracing *input_stack* level if *tracingmacros* positive }
define *mubyte_in_code* = *web2c_int_base* + 4 { if positive then reading mubytes is active }
define *mubyte_out_code* = *web2c_int_base* + 5 { if positive then printing mubytes is active }
define *mubyte_log_code* = *web2c_int_base* + 6 { if positive then print mubytes to log and terminal }
define *spec_out_code* = *web2c_int_base* + 7 { if positive then print specials by mubytes }
define *web2c_int_pars* = *web2c_int_base* + 8 { total number of web2c's integer parameters }

define *pdftex_first_integer_code* = *web2c_int_pars* { base for pdfTEX's integer parameters }
define *pdf_output_code* = *pdftex_first_integer_code* + 0 { switch on PDF output if positive }
define *pdf_compress_level_code* = *pdftex_first_integer_code* + 1 { compress level of streams }
define *pdf_decimal_digits_code* = *pdftex_first_integer_code* + 2
{ digits after the decimal point of numbers }
define *pdf_move_chars_code* = *pdftex_first_integer_code* + 3 { move chars 0..31 to higher area if possible }
define *pdf_image_resolution_code* = *pdftex_first_integer_code* + 4 { default image resolution }
define *pdf_pk_resolution_code* = *pdftex_first_integer_code* + 5 { default resolution of PK font }
define *pdf_unique_resname_code* = *pdftex_first_integer_code* + 6 { generate unique names for resouces }
define *pdf_option_always_use_pdfpagebox_code* = *pdftex_first_integer_code* + 7
{ if the PDF inclusion should always use a specific PDF page box }
define *pdf_option_pdf_inclusion_errorlevel_code* = *pdftex_first_integer_code* + 8
{ if the PDF inclusion should treat pdfs newer than *pdf_minor_version* as an error }
define *pdf_major_version_code* = *pdftex_first_integer_code* + 9
{ integer part of the PDF version produced }
define *pdf_minor_version_code* = *pdftex_first_integer_code* + 10
{ fractional part of the PDF version produced }
define *pdf_force_pagebox_code* = *pdftex_first_integer_code* + 11
{ if the PDF inclusion should always use a specific PDF page box }
define *pdf_pagebox_code* = *pdftex_first_integer_code* + 12 { default pagebox to use for PDF inclusion }
define *pdf_inclusion_errorlevel_code* = *pdftex_first_integer_code* + 13
{ if the PDF inclusion should treat pdfs newer than *pdf_minor_version* as an error }
define *pdf_gamma_code* = *pdftex_first_integer_code* + 14
define *pdf_image_gamma_code* = *pdftex_first_integer_code* + 15
define *pdf_image_hicolor_code* = *pdftex_first_integer_code* + 16
define *pdf_image_apply_gamma_code* = *pdftex_first_integer_code* + 17
define *pdf_adjust_spacing_code* = *pdftex_first_integer_code* + 18 { level of spacing adjusting }
define *pdf_protrude_chars_code* = *pdftex_first_integer_code* + 19
{ protrude chars at left/right edge of paragraphs }
define *pdf_tracing_fonts_code* = *pdftex_first_integer_code* + 20 { level of font detail in log }
define *pdf_objcompresslevel_code* = *pdftex_first_integer_code* + 21 { activate object streams }
define *pdf_adjust_interword_glue_code* = *pdftex_first_integer_code* + 22 { adjust interword glue? }

define $pdf\_prepend\_kern\_code = pdftex\_first\_integer\_code + 23$
{ prepend kern before certain characters? }
define $pdf\_append\_kern\_code = pdftex\_first\_integer\_code + 24$ { append kern before certain characters? }
define $pdf\_gen\_tounicode\_code = pdftex\_first\_integer\_code + 25$ { generate ToUnicode for fonts? }
define $pdf\_draftmode\_code = pdftex\_first\_integer\_code + 26$ { switch on draftmode if positive }
define $pdf\_inclusion\_copy\_font\_code = pdftex\_first\_integer\_code + 27$ { generate ToUnicode for fonts? }
define $pdf\_suppress\_warning\_dup\_dest\_code = pdftex\_first\_integer\_code + 28$
{ suppress warning about duplicated destinations }
define $pdf\_suppress\_warning\_dup\_map\_code = pdftex\_first\_integer\_code + 29$
{ suppress warning about duplicated map lines }
define $pdf\_suppress\_warning\_page\_group\_code = pdftex\_first\_integer\_code + 30$
{ suppress warning about multiple pdfs with page group }
define $pdf\_info\_omit\_date\_code = pdftex\_first\_integer\_code + 31$
{ omit generating CreationDate and ModDate }
define $pdf\_suppress\_ptex\_info\_code = pdftex\_first\_integer\_code + 32$
{ suppress /PTEX.* entries in PDF dictionaries }
define $pdf\_omit\_charset\_code = pdftex\_first\_integer\_code + 33$
{ suppress /PTEX.* entries in PDF dictionaries }
define $pdf\_int\_pars = pdftex\_first\_integer\_code + 34$ { total number of pdfTEX's integer parameters }
define $etex\_int\_base = pdf\_int\_pars$ { base for ε-TEX's integer parameters }
define $tracing\_assigns\_code = etex\_int\_base$ { show assignments }
define $tracing\_groups\_code = etex\_int\_base + 1$ { show save/restore groups }
define $tracing\_ifs\_code = etex\_int\_base + 2$ { show conditionals }
define $tracing\_scan\_tokens\_code = etex\_int\_base + 3$ { show pseudo file open and close }
define $tracing\_nesting\_code = etex\_int\_base + 4$ { show incomplete groups and ifs within files }
define $pre\_display\_direction\_code = etex\_int\_base + 5$ { text direction preceding a display }
define $last\_line\_fit\_code = etex\_int\_base + 6$ { adjustment for last line of paragraph }
define $saving\_vdiscards\_code = etex\_int\_base + 7$ { save items discarded from vlists }
define $saving\_hyph\_codes\_code = etex\_int\_base + 8$ { save hyphenation codes for languages }
define $eTeX\_state\_code = etex\_int\_base + 9$ { ε-TEX state variables }
define $etex\_int\_pars = eTeX\_state\_code + eTeX\_states$ { total number of ε-TEX's integer parameters }
define $synctex\_code = etex\_int\_pars$
define $int\_pars = synctex\_code + 1$ { total number of integer parameters }
define $count\_base = int\_base + int\_pars$ { 256 user `\count` registers }
define $del\_code\_base = count\_base + 256$ { 256 delimiter code mappings }
define $dimen\_base = del\_code\_base + 256$ { beginning of region 6 }
define $del\_code(\#) \equiv eqtb[del\_code\_base + \#].int$
define $count(\#) \equiv eqtb[count\_base + \#].int$
define $int\_par(\#) \equiv eqtb[int\_base + \#].int$ { an integer parameter }
define $pretolerance \equiv int\_par(pretolerance\_code)$
define $tolerance \equiv int\_par(tolerance\_code)$
define $line\_penalty \equiv int\_par(line\_penalty\_code)$
define $hyphen\_penalty \equiv int\_par(hyphen\_penalty\_code)$
define $ex\_hyphen\_penalty \equiv int\_par(ex\_hyphen\_penalty\_code)$
define $club\_penalty \equiv int\_par(club\_penalty\_code)$
define $widow\_penalty \equiv int\_par(widow\_penalty\_code)$
define $display\_widow\_penalty \equiv int\_par(display\_widow\_penalty\_code)$
define $broken\_penalty \equiv int\_par(broken\_penalty\_code)$
define $bin\_op\_penalty \equiv int\_par(bin\_op\_penalty\_code)$
define $rel\_penalty \equiv int\_par(rel\_penalty\_code)$
define $pre\_display\_penalty \equiv int\_par(pre\_display\_penalty\_code)$
define $post\_display\_penalty \equiv int\_par(post\_display\_penalty\_code)$

define *inter_line_penalty* ≡ *int_par*(*inter_line_penalty_code*)
define *double_hyphen_demerits* ≡ *int_par*(*double_hyphen_demerits_code*)
define *final_hyphen_demerits* ≡ *int_par*(*final_hyphen_demerits_code*)
define *adj_demerits* ≡ *int_par*(*adj_demerits_code*)
define *mag* ≡ *int_par*(*mag_code*)
define *delimiter_factor* ≡ *int_par*(*delimiter_factor_code*)
define *looseness* ≡ *int_par*(*looseness_code*)
define *time* ≡ *int_par*(*time_code*)
define *day* ≡ *int_par*(*day_code*)
define *month* ≡ *int_par*(*month_code*)
define *year* ≡ *int_par*(*year_code*)
define *show_box_breadth* ≡ *int_par*(*show_box_breadth_code*)
define *show_box_depth* ≡ *int_par*(*show_box_depth_code*)
define *hbadness* ≡ *int_par*(*hbadness_code*)
define *vbadness* ≡ *int_par*(*vbadness_code*)
define *pausing* ≡ *int_par*(*pausing_code*)
define *tracing_online* ≡ *int_par*(*tracing_online_code*)
define *tracing_macros* ≡ *int_par*(*tracing_macros_code*)
define *tracing_stats* ≡ *int_par*(*tracing_stats_code*)
define *tracing_paragraphs* ≡ *int_par*(*tracing_paragraphs_code*)
define *tracing_pages* ≡ *int_par*(*tracing_pages_code*)
define *tracing_output* ≡ *int_par*(*tracing_output_code*)
define *tracing_lost_chars* ≡ *int_par*(*tracing_lost_chars_code*)
define *tracing_commands* ≡ *int_par*(*tracing_commands_code*)
define *tracing_restores* ≡ *int_par*(*tracing_restores_code*)
define *uc_hyph* ≡ *int_par*(*uc_hyph_code*)
define *output_penalty* ≡ *int_par*(*output_penalty_code*)
define *max_dead_cycles* ≡ *int_par*(*max_dead_cycles_code*)
define *hang_after* ≡ *int_par*(*hang_after_code*)
define *floating_penalty* ≡ *int_par*(*floating_penalty_code*)
define *global_defs* ≡ *int_par*(*global_defs_code*)
define *cur_fam* ≡ *int_par*(*cur_fam_code*)
define *escape_char* ≡ *int_par*(*escape_char_code*)
define *default_hyphen_char* ≡ *int_par*(*default_hyphen_char_code*)
define *default_skew_char* ≡ *int_par*(*default_skew_char_code*)
define *end_line_char* ≡ *int_par*(*end_line_char_code*)
define *new_line_char* ≡ *int_par*(*new_line_char_code*)
define *language* ≡ *int_par*(*language_code*)
define *left_hyphen_min* ≡ *int_par*(*left_hyphen_min_code*)
define *right_hyphen_min* ≡ *int_par*(*right_hyphen_min_code*)
define *holding_inserts* ≡ *int_par*(*holding_inserts_code*)
define *error_context_lines* ≡ *int_par*(*error_context_lines_code*)
define *synctex* ≡ *int_par*(*synctex_code*)
define *char_sub_def_min* ≡ *int_par*(*char_sub_def_min_code*)
define *char_sub_def_max* ≡ *int_par*(*char_sub_def_max_code*)
define *tracing_char_sub_def* ≡ *int_par*(*tracing_char_sub_def_code*)
define *mubyte_in* ≡ *int_par*(*mubyte_in_code*)
define *mubyte_out* ≡ *int_par*(*mubyte_out_code*)
define *mubyte_log* ≡ *int_par*(*mubyte_log_code*)
define *spec_out* ≡ *int_par*(*spec_out_code*)
define *tracing_stack_levels* ≡ *int_par*(*tracing_stack_levels_code*)
define *pdf_adjust_spacing* ≡ *int_par*(*pdf_adjust_spacing_code*)

define $pdf\_protrude\_chars \equiv int\_par(pdf\_protrude\_chars\_code)$
define $pdf\_tracing\_fonts \equiv int\_par(pdf\_tracing\_fonts\_code)$
define $pdf\_adjust\_interword\_glue \equiv int\_par(pdf\_adjust\_interword\_glue\_code)$
define $pdf\_prepend\_kern \equiv int\_par(pdf\_prepend\_kern\_code)$
define $pdf\_append\_kern \equiv int\_par(pdf\_append\_kern\_code)$
define $pdf\_gen\_tounicode \equiv int\_par(pdf\_gen\_tounicode\_code)$
define $pdf\_output \equiv int\_par(pdf\_output\_code)$
define $pdf\_compress\_level \equiv int\_par(pdf\_compress\_level\_code)$
define $pdf\_objcompresslevel \equiv int\_par(pdf\_objcompresslevel\_code)$
define $pdf\_decimal\_digits \equiv int\_par(pdf\_decimal\_digits\_code)$
define $pdf\_move\_chars \equiv int\_par(pdf\_move\_chars\_code)$
define $pdf\_image\_resolution \equiv int\_par(pdf\_image\_resolution\_code)$
define $pdf\_pk\_resolution \equiv int\_par(pdf\_pk\_resolution\_code)$
define $pdf\_unique\_resname \equiv int\_par(pdf\_unique\_resname\_code)$
define $pdf\_option\_always\_use\_pdfpagebox \equiv int\_par(pdf\_option\_always\_use\_pdfpagebox\_code)$
define $pdf\_option\_pdf\_inclusion\_errorlevel \equiv int\_par(pdf\_option\_pdf\_inclusion\_errorlevel\_code)$
define $pdf\_major\_version \equiv int\_par(pdf\_major\_version\_code)$
define $pdf\_minor\_version \equiv int\_par(pdf\_minor\_version\_code)$
define $pdf\_force\_pagebox \equiv int\_par(pdf\_force\_pagebox\_code)$
define $pdf\_pagebox \equiv int\_par(pdf\_pagebox\_code)$
define $pdf\_inclusion\_errorlevel \equiv int\_par(pdf\_inclusion\_errorlevel\_code)$
define $pdf\_gamma \equiv int\_par(pdf\_gamma\_code)$
define $pdf\_image\_gamma \equiv int\_par(pdf\_image\_gamma\_code)$
define $pdf\_image\_hicolor \equiv int\_par(pdf\_image\_hicolor\_code)$
define $pdf\_image\_apply\_gamma \equiv int\_par(pdf\_image\_apply\_gamma\_code)$
define $pdf\_draftmode \equiv int\_par(pdf\_draftmode\_code)$
define $pdf\_inclusion\_copy\_font \equiv int\_par(pdf\_inclusion\_copy\_font\_code)$
define $pdf\_suppress\_warning\_dup\_dest \equiv int\_par(pdf\_suppress\_warning\_dup\_dest\_code)$
define $pdf\_suppress\_warning\_dup\_map \equiv int\_par(pdf\_suppress\_warning\_dup\_map\_code)$
define $pdf\_suppress\_warning\_page\_group \equiv int\_par(pdf\_suppress\_warning\_page\_group\_code)$
define $pdf\_info\_omit\_date \equiv int\_par(pdf\_info\_omit\_date\_code)$
define $pdf\_suppress\_ptex\_info \equiv int\_par(pdf\_suppress\_ptex\_info\_code)$
define $pdf\_omit\_charset \equiv int\_par(pdf\_omit\_charset\_code)$

define $tracing\_assigns \equiv int\_par(tracing\_assigns\_code)$
define $tracing\_groups \equiv int\_par(tracing\_groups\_code)$
define $tracing\_ifs \equiv int\_par(tracing\_ifs\_code)$
define $tracing\_scan\_tokens \equiv int\_par(tracing\_scan\_tokens\_code)$
define $tracing\_nesting \equiv int\_par(tracing\_nesting\_code)$
define $pre\_display\_direction \equiv int\_par(pre\_display\_direction\_code)$
define $last\_line\_fit \equiv int\_par(last\_line\_fit\_code)$
define $saving\_vdiscards \equiv int\_par(saving\_vdiscards\_code)$
define $saving\_hyph\_codes \equiv int\_par(saving\_hyph\_codes\_code)$

⟨ Assign the values $depth\_threshold \leftarrow show\_box\_depth$ and $breadth\_max \leftarrow show\_box\_breadth$ 254* ⟩ ≡
$depth\_threshold \leftarrow show\_box\_depth;$ $breadth\_max \leftarrow show\_box\_breadth$

This code is used in section 216.

255* We can print the symbolic name of an integer parameter as follows.

procedure *print_param*(*n* : *integer*);
begin case *n* **of**
pretolerance_code: *print_esc*("pretolerance");
tolerance_code: *print_esc*("tolerance");
line_penalty_code: *print_esc*("linepenalty");
hyphen_penalty_code: *print_esc*("hyphenpenalty");
ex_hyphen_penalty_code: *print_esc*("exhyphenpenalty");
club_penalty_code: *print_esc*("clubpenalty");
widow_penalty_code: *print_esc*("widowpenalty");
display_widow_penalty_code: *print_esc*("displaywidowpenalty");
broken_penalty_code: *print_esc*("brokenpenalty");
bin_op_penalty_code: *print_esc*("binoppenalty");
rel_penalty_code: *print_esc*("relpenalty");
pre_display_penalty_code: *print_esc*("predisplaypenalty");
post_display_penalty_code: *print_esc*("postdisplaypenalty");
inter_line_penalty_code: *print_esc*("interlinepenalty");
double_hyphen_demerits_code: *print_esc*("doublehyphendemerits");
final_hyphen_demerits_code: *print_esc*("finalhyphendemerits");
adj_demerits_code: *print_esc*("adjdemerits");
mag_code: *print_esc*("mag");
delimiter_factor_code: *print_esc*("delimiterfactor");
looseness_code: *print_esc*("looseness");
time_code: *print_esc*("time");
day_code: *print_esc*("day");
month_code: *print_esc*("month");
year_code: *print_esc*("year");
show_box_breadth_code: *print_esc*("showboxbreadth");
show_box_depth_code: *print_esc*("showboxdepth");
hbadness_code: *print_esc*("hbadness");
vbadness_code: *print_esc*("vbadness");
pausing_code: *print_esc*("pausing");
tracing_online_code: *print_esc*("tracingonline");
tracing_macros_code: *print_esc*("tracingmacros");
tracing_stats_code: *print_esc*("tracingstats");
tracing_paragraphs_code: *print_esc*("tracingparagraphs");
tracing_pages_code: *print_esc*("tracingpages");
tracing_output_code: *print_esc*("tracingoutput");
tracing_lost_chars_code: *print_esc*("tracinglostchars");
tracing_commands_code: *print_esc*("tracingcommands");
tracing_restores_code: *print_esc*("tracingrestores");
uc_hyph_code: *print_esc*("uchyph");
output_penalty_code: *print_esc*("outputpenalty");
max_dead_cycles_code: *print_esc*("maxdeadcycles");
hang_after_code: *print_esc*("hangafter");
floating_penalty_code: *print_esc*("floatingpenalty");
global_defs_code: *print_esc*("globaldefs");
cur_fam_code: *print_esc*("fam");
escape_char_code: *print_esc*("escapechar");
default_hyphen_char_code: *print_esc*("defaulthyphenchar");
default_skew_char_code: *print_esc*("defaultskewchar");
end_line_char_code: *print_esc*("endlinechar");

new_line_char_code: *print_esc*("newlinechar");
language_code: *print_esc*("language");
left_hyphen_min_code: *print_esc*("lefthyphenmin");
right_hyphen_min_code: *print_esc*("righthyphenmin");
holding_inserts_code: *print_esc*("holdinginserts");
error_context_lines_code: *print_esc*("errorcontextlines");
char_sub_def_min_code: *print_esc*("charsubdefmin");
char_sub_def_max_code: *print_esc*("charsubdefmax");
tracing_char_sub_def_code: *print_esc*("tracingcharsubdef");
mubyte_in_code: *print_esc*("mubytein");
mubyte_out_code: *print_esc*("mubyteout");
mubyte_log_code: *print_esc*("mubytelog");
spec_out_code: *print_esc*("specialout");
tracing_stack_levels_code: *print_esc*("tracingstacklevels");

pdf_output_code: *print_esc*("pdfoutput");
pdf_compress_level_code: *print_esc*("pdfcompresslevel");
pdf_objcompresslevel_code: *print_esc*("pdfobjcompresslevel");
pdf_decimal_digits_code: *print_esc*("pdfdecimaldigits");
pdf_move_chars_code: *print_esc*("pdfmovechars");
pdf_image_resolution_code: *print_esc*("pdfimageresolution");
pdf_pk_resolution_code: *print_esc*("pdfpkresolution");
pdf_unique_resname_code: *print_esc*("pdfuniqueresname");
pdf_option_always_use_pdfpagebox_code: *print_esc*("pdfoptionalwaysusepdfpagebox");
pdf_option_pdf_inclusion_errorlevel_code: *print_esc*("pdfoptionpdfinclusionerrorlevel");
pdf_major_version_code: *print_esc*("pdfmajorversion");
pdf_minor_version_code: *print_esc*("pdfminorversion");
pdf_force_pagebox_code: *print_esc*("pdfforcepagebox");
pdf_pagebox_code: *print_esc*("pdfpagebox");
pdf_inclusion_errorlevel_code: *print_esc*("pdfinclusionerrorlevel");
pdf_gamma_code: *print_esc*("pdfgamma");
pdf_image_gamma_code: *print_esc*("pdfimagegamma");
pdf_image_hicolor_code: *print_esc*("pdfimagehicolor");
pdf_image_apply_gamma_code: *print_esc*("pdfimageapplygamma");
pdf_adjust_spacing_code: *print_esc*("pdfadjustspacing");
pdf_protrude_chars_code: *print_esc*("pdfprotrudechars");
pdf_tracing_fonts_code: *print_esc*("pdftracingfonts");
pdf_adjust_interword_glue_code: *print_esc*("pdfadjustinterwordglue");
pdf_prepend_kern_code: *print_esc*("pdfprependkern");
pdf_append_kern_code: *print_esc*("pdfappendkern");
pdf_gen_tounicode_code: *print_esc*("pdfgentounicode");
pdf_draftmode_code: *print_esc*("pdfdraftmode");
pdf_inclusion_copy_font_code: *print_esc*("pdfinclusioncopyfonts");
pdf_suppress_warning_dup_dest_code: *print_esc*("pdfsuppresswarningdupdest");
pdf_suppress_warning_dup_map_code: *print_esc*("pdfsuppresswarningdupmap");
pdf_suppress_warning_page_group_code: *print_esc*("pdfsuppresswarningpagegroup");
pdf_info_omit_date_code: *print_esc*("pdfinfoomitdate");
pdf_suppress_ptex_info_code: *print_esc*("pdfsuppressptexinfo");
pdf_omit_charset_code: *print_esc*("pdfomitcharset");
⟨ synctex case for *print_param* 1906* ⟩
⟨ Cases for *print_param* 1656 ⟩
othercases *print*("[unknown␣integer␣parameter!]")
endcases;

end;

256* The integer parameter names must be entered into the hash table.

⟨ Put each of TEX's primitives into the hash table 244 ⟩ +≡
primitive("pretolerance", *assign_int*, *int_base* + *pretolerance_code*);
primitive("tolerance", *assign_int*, *int_base* + *tolerance_code*);
primitive("linepenalty", *assign_int*, *int_base* + *line_penalty_code*);
primitive("hyphenpenalty", *assign_int*, *int_base* + *hyphen_penalty_code*);
primitive("exhyphenpenalty", *assign_int*, *int_base* + *ex_hyphen_penalty_code*);
primitive("clubpenalty", *assign_int*, *int_base* + *club_penalty_code*);
primitive("widowpenalty", *assign_int*, *int_base* + *widow_penalty_code*);
primitive("displaywidowpenalty", *assign_int*, *int_base* + *display_widow_penalty_code*);
primitive("brokenpenalty", *assign_int*, *int_base* + *broken_penalty_code*);
primitive("binoppenalty", *assign_int*, *int_base* + *bin_op_penalty_code*);
primitive("relpenalty", *assign_int*, *int_base* + *rel_penalty_code*);
primitive("predisplaypenalty", *assign_int*, *int_base* + *pre_display_penalty_code*);
primitive("postdisplaypenalty", *assign_int*, *int_base* + *post_display_penalty_code*);
primitive("interlinepenalty", *assign_int*, *int_base* + *inter_line_penalty_code*);
primitive("doublehyphendemerits", *assign_int*, *int_base* + *double_hyphen_demerits_code*);
primitive("finalhyphendemerits", *assign_int*, *int_base* + *final_hyphen_demerits_code*);
primitive("adjdemerits", *assign_int*, *int_base* + *adj_demerits_code*);
primitive("mag", *assign_int*, *int_base* + *mag_code*);
primitive("delimiterfactor", *assign_int*, *int_base* + *delimiter_factor_code*);
primitive("looseness", *assign_int*, *int_base* + *looseness_code*);
primitive("time", *assign_int*, *int_base* + *time_code*);
primitive("day", *assign_int*, *int_base* + *day_code*);
primitive("month", *assign_int*, *int_base* + *month_code*);
primitive("year", *assign_int*, *int_base* + *year_code*);
primitive("showboxbreadth", *assign_int*, *int_base* + *show_box_breadth_code*);
primitive("showboxdepth", *assign_int*, *int_base* + *show_box_depth_code*);
primitive("hbadness", *assign_int*, *int_base* + *hbadness_code*);
primitive("vbadness", *assign_int*, *int_base* + *vbadness_code*);
primitive("pausing", *assign_int*, *int_base* + *pausing_code*);
primitive("tracingonline", *assign_int*, *int_base* + *tracing_online_code*);
primitive("tracingmacros", *assign_int*, *int_base* + *tracing_macros_code*);
primitive("tracingstats", *assign_int*, *int_base* + *tracing_stats_code*);
primitive("tracingparagraphs", *assign_int*, *int_base* + *tracing_paragraphs_code*);
primitive("tracingpages", *assign_int*, *int_base* + *tracing_pages_code*);
primitive("tracingoutput", *assign_int*, *int_base* + *tracing_output_code*);
primitive("tracinglostchars", *assign_int*, *int_base* + *tracing_lost_chars_code*);
primitive("tracingcommands", *assign_int*, *int_base* + *tracing_commands_code*);
primitive("tracingrestores", *assign_int*, *int_base* + *tracing_restores_code*);
primitive("uchyph", *assign_int*, *int_base* + *uc_hyph_code*);
primitive("outputpenalty", *assign_int*, *int_base* + *output_penalty_code*);
primitive("maxdeadcycles", *assign_int*, *int_base* + *max_dead_cycles_code*);
primitive("hangafter", *assign_int*, *int_base* + *hang_after_code*);
primitive("floatingpenalty", *assign_int*, *int_base* + *floating_penalty_code*);
primitive("globaldefs", *assign_int*, *int_base* + *global_defs_code*);
primitive("fam", *assign_int*, *int_base* + *cur_fam_code*);
primitive("escapechar", *assign_int*, *int_base* + *escape_char_code*);
primitive("defaulthyphenchar", *assign_int*, *int_base* + *default_hyphen_char_code*);
primitive("defaultskewchar", *assign_int*, *int_base* + *default_skew_char_code*);
primitive("endlinechar", *assign_int*, *int_base* + *end_line_char_code*);
primitive("newlinechar", *assign_int*, *int_base* + *new_line_char_code*);

primitive("language", *assign_int*, *int_base* + *language_code*);
primitive("lefthyphenmin", *assign_int*, *int_base* + *left_hyphen_min_code*);
primitive("righthyphenmin", *assign_int*, *int_base* + *right_hyphen_min_code*);
primitive("holdinginserts", *assign_int*, *int_base* + *holding_inserts_code*);
primitive("errorcontextlines", *assign_int*, *int_base* + *error_context_lines_code*);
if *mltex_p* **then**
  begin *mltex_enabled_p* ← *true*; { enable character substitution }
  if *false* **then** { remove the if-clause to enable \charsubdefmin }
    primitive("charsubdefmin", *assign_int*, *int_base* + *char_sub_def_min_code*);
  primitive("charsubdefmax", *assign_int*, *int_base* + *char_sub_def_max_code*);
  primitive("tracingcharsubdef", *assign_int*, *int_base* + *tracing_char_sub_def_code*);
  end;
if *enctex_p* **then**
  begin *enctex_enabled_p* ← *true*; *primitive*("mubytein", *assign_int*, *int_base* + *mubyte_in_code*);
  primitive("mubyteout", *assign_int*, *int_base* + *mubyte_out_code*);
  primitive("mubytelog", *assign_int*, *int_base* + *mubyte_log_code*);
  primitive("specialout", *assign_int*, *int_base* + *spec_out_code*);
  end;
primitive("tracingstacklevels", *assign_int*, *int_base* + *tracing_stack_levels_code*);
primitive("pdfoutput", *assign_int*, *int_base* + *pdf_output_code*);
primitive("pdfcompresslevel", *assign_int*, *int_base* + *pdf_compress_level_code*);
primitive("pdfobjcompresslevel", *assign_int*, *int_base* + *pdf_objcompresslevel_code*);
primitive("pdfdecimaldigits", *assign_int*, *int_base* + *pdf_decimal_digits_code*);
primitive("pdfmovechars", *assign_int*, *int_base* + *pdf_move_chars_code*);
primitive("pdfimageresolution", *assign_int*, *int_base* + *pdf_image_resolution_code*);
primitive("pdfpkresolution", *assign_int*, *int_base* + *pdf_pk_resolution_code*);
primitive("pdfuniqueresname", *assign_int*, *int_base* + *pdf_unique_resname_code*);
primitive("pdfoptionpdfminorversion", *assign_int*, *int_base* + *pdf_minor_version_code*);
primitive("pdfoptionalwaysusepdfpagebox", *assign_int*,
    int_base + *pdf_option_always_use_pdfpagebox_code*);
primitive("pdfoptionpdfinclusionerrorlevel", *assign_int*,
    int_base + *pdf_option_pdf_inclusion_errorlevel_code*);
primitive("pdfmajorversion", *assign_int*, *int_base* + *pdf_major_version_code*);
primitive("pdfminorversion", *assign_int*, *int_base* + *pdf_minor_version_code*);
primitive("pdfforcepagebox", *assign_int*, *int_base* + *pdf_force_pagebox_code*);
primitive("pdfpagebox", *assign_int*, *int_base* + *pdf_pagebox_code*);
primitive("pdfinclusionerrorlevel", *assign_int*, *int_base* + *pdf_inclusion_errorlevel_code*);
primitive("pdfgamma", *assign_int*, *int_base* + *pdf_gamma_code*);
primitive("pdfimagegamma", *assign_int*, *int_base* + *pdf_image_gamma_code*);
primitive("pdfimagehicolor", *assign_int*, *int_base* + *pdf_image_hicolor_code*);
primitive("pdfimageapplygamma", *assign_int*, *int_base* + *pdf_image_apply_gamma_code*);
primitive("pdfadjustspacing", *assign_int*, *int_base* + *pdf_adjust_spacing_code*);
primitive("pdfprotrudechars", *assign_int*, *int_base* + *pdf_protrude_chars_code*);
primitive("pdftracingfonts", *assign_int*, *int_base* + *pdf_tracing_fonts_code*);
primitive("pdfadjustinterwordglue", *assign_int*, *int_base* + *pdf_adjust_interword_glue_code*);
primitive("pdfprependkern", *assign_int*, *int_base* + *pdf_prepend_kern_code*);
primitive("pdfappendkern", *assign_int*, *int_base* + *pdf_append_kern_code*);
primitive("pdfgentounicode", *assign_int*, *int_base* + *pdf_gen_tounicode_code*);
primitive("pdfdraftmode", *assign_int*, *int_base* + *pdf_draftmode_code*);
primitive("pdfinclusioncopyfonts", *assign_int*, *int_base* + *pdf_inclusion_copy_font_code*);
primitive("pdfsuppresswarningdupdest", *assign_int*, *int_base* + *pdf_suppress_warning_dup_dest_code*);
primitive("pdfsuppresswarningdupmap", *assign_int*, *int_base* + *pdf_suppress_warning_dup_map_code*);

primitive("pdfsuppresswarningpagegroup", *assign_int*, *int_base* + *pdf_suppress_warning_page_group_code*);
primitive("pdfinfoomitdate", *assign_int*, *int_base* + *pdf_info_omit_date_code*);
primitive("pdfsuppressptexinfo", *assign_int*, *int_base* + *pdf_suppress_ptex_info_code*);
primitive("pdfomitcharset", *assign_int*, *int_base* + *pdf_omit_charset_code*);

258* The integer parameters should really be initialized by a macro package; the following initialization does the minimum to keep TEX from complete failure.

⟨Initialize table entries (done by INITEX only) 182⟩ +≡
for $k \leftarrow$ *int_base* **to** *del_code_base* − 1 **do** *eqtb*[k].*int* ← 0;
char_sub_def_min ← 256; *char_sub_def_max* ← −1; { allow \charsubdef for char 0 }
{ *tracing_char_sub_def* ← 0 is already done }
mag ← 1000; *tolerance* ← 10000; *hang_after* ← 1; *max_dead_cycles* ← 25; *escape_char* ← "\";
end_line_char ← *carriage_return*;
for $k \leftarrow 0$ **to** 255 **do** *del_code*(k) ← −1;
del_code(".") ← 0; { this null delimiter is used in error recovery }

259* The following procedure, which is called just before TEX initializes its input and output, establishes the initial values of the date and time. It calls a *date_and_time* C macro (a.k.a. *dateandtime*), which calls the C function *get_date_and_time*, passing it the addresses of *sys_time*, etc., so they can be set by the routine. *get_date_and_time* also sets up interrupt catching if that is conditionally compiled in the C code.

We have to initialize the *sys_* variables because that is what gets output on the first line of the log file. (New in 2021.)

procedure *fix_date_and_time*;
begin *date_and_time*(*sys_time*, *sys_day*, *sys_month*, *sys_year*); *time* ← *sys_time*;
{ minutes since midnight }
day ← *sys_day*; { day of the month }
month ← *sys_month*; { month of the year }
year ← *sys_year*; { Anno Domini }
end;

270* Here is a procedure that displays the contents of *eqtb*[n] symbolically.

⟨Declare the procedure called *print_cmd_chr* 320⟩
stat procedure *show_eqtb*(n : *pointer*);
begin if $n <$ *active_base* **then** *print_char*("?") { this can't happen }
else if ($n <$ *glue_base*) ∨ (($n >$ *eqtb_size*) ∧ ($n \le$ *eqtb_top*)) **then** ⟨Show equivalent n, in region 1 or 2 241⟩
else if $n <$ *local_base* **then** ⟨Show equivalent n, in region 3 247⟩
else if $n <$ *int_base* **then** ⟨Show equivalent n, in region 4 251⟩
else if $n <$ *dimen_base* **then** ⟨Show equivalent n, in region 5 260⟩
else if $n \le$ *eqtb_size* **then** ⟨Show equivalent n, in region 6 269⟩
else *print_char*("?"); { this can't happen either }
end;
tats

271* The last two regions of *eqtb* have fullword values instead of the three fields *eq_level*, *eq_type*, and *equiv*. An *eq_type* is unnecessary, but TEX needs to store the *eq_level* information in another array called *xeq_level*.

⟨Global variables 13⟩ +≡
zeqtb: ↑*memory_word*;
xeq_level: **array** [*int_base* .. *eqtb_size*] **of** *quarterword*;

274*. The hash table. Control sequences are stored and retrieved by means of a fairly standard hash table algorithm called the method of "coalescing lists" (cf. Algorithm 6.4C in *The Art of Computer Programming*). Once a control sequence enters the table, it is never removed, because there are complicated situations involving `\gdef` where the removal of a control sequence at the end of a group would be a mistake preventable only by the introduction of a complicated reference-count mechanism.

The actual sequence of letters forming a control sequence identifier is stored in the *str_pool* array together with all the other strings. An auxiliary array *hash* consists of items with two halfword fields per word. The first of these, called $next(p)$, points to the next identifier belonging to the same coalesced list as the identifier corresponding to p; and the other, called $text(p)$, points to the *str_start* entry for p's identifier. If position p of the hash table is empty, we have $text(p) = 0$; if position p is either empty or the end of a coalesced hash list, we have $next(p) = 0$. An auxiliary pointer variable called *hash_used* is maintained in such a way that all locations $p \geq hash\_used$ are nonempty. The global variable *cs_count* tells how many multiletter control sequences have been defined, if statistics are being kept.

A global boolean variable called *no_new_control_sequence* is set to *true* during the time that new hash table entries are forbidden.

define $next(\#) \equiv hash[\#].lh$ { link for coalesced lists }
define $text(\#) \equiv hash[\#].rh$ { string number for control sequence name }
define $hash\_is\_full \equiv (hash\_used = hash\_base)$ { test if all positions are occupied }
define $font\_id\_text(\#) \equiv text(font\_id\_base + \#)$ { a frozen font identifier's name }

⟨ Global variables 13 ⟩ +≡
hash: ↑*two_halves*; { the hash table }
yhash: ↑*two_halves*; { auxiliary pointer for freeing hash }
hash_used: *pointer*; { allocation pointer for *hash* }
hash_extra: *pointer*; { *hash_extra* = *hash* above *eqtb_size* }
hash_top: *pointer*; { maximum of the hash array }
eqtb_top: *pointer*; { maximum of the *eqtb* }
hash_high: *pointer*; { pointer to next high hash location }
no_new_control_sequence: *boolean*; { are new identifiers legal? }
cs_count: *integer*; { total number of known identifiers }

276*. ⟨ Set initial values of key variables 21 ⟩ +≡
no_new_control_sequence ← *true*; { new identifiers are usually forbidden }
prim_next(0) ← 0; *prim_text*(0) ← 0;
for *k* ← 1 **to** *prim_size* **do** *prim*[*k*] ← *prim*[0];

277*. ⟨ Initialize table entries (done by `INITEX` only) 182 ⟩ +≡
prim_used ← *prim_size*; { nothing is used }
hash_used ← *frozen_control_sequence*; { nothing is used }
hash_high ← 0; *cs_count* ← 0; *eq_type*(*frozen_dont_expand*) ← *dont_expand*;
text(*frozen_dont_expand*) ← `"notexpanded:"`; *eq_type*(*frozen_primitive*) ← *ignore_spaces*;
equiv(*frozen_primitive*) ← 1; *eq_level*(*frozen_primitive*) ← *level_one*;
text(*frozen_primitive*) ← `"pdfprimitive"`;

279* ⟨Insert a new control sequence after p, then make p point to it 279*⟩ ≡

begin if $text(p) > 0$ **then**
  begin if $hash\_high < hash\_extra$ **then**
    begin $incr(hash\_high)$; $next(p) \leftarrow hash\_high + eqtb\_size$; $p \leftarrow hash\_high + eqtb\_size$;
    end
  else begin repeat if $hash\_is\_full$ **then** $overflow($"hash␣size"$, hash\_size + hash\_extra)$;
      $decr(hash\_used)$;
    until $text(hash\_used) = 0$; { search for an empty location in $hash$ }
    $next(p) \leftarrow hash\_used$; $p \leftarrow hash\_used$;
    end;
  end;
$str\_room(l)$; $d \leftarrow cur\_length$;
while $pool\_ptr > str\_start[str\_ptr]$ **do**
  begin $decr(pool\_ptr)$; $str\_pool[pool\_ptr + l] \leftarrow str\_pool[pool\_ptr]$;
  end; { move current string up to make room for another }
for $k \leftarrow j$ **to** $j + l - 1$ **do** $append\_char(buffer[k])$;
$text(p) \leftarrow make\_string$; $pool\_ptr \leftarrow pool\_ptr + d$;
stat $incr(cs\_count)$; **tats**
end

This code is used in section 278.

284* Single-character control sequences do not need to be looked up in a hash table, since we can use the character code itself as a direct address. The procedure *print_cs* prints the name of a control sequence, given a pointer to its address in *eqtb*. A space is printed after the name unless it is a single nonletter or an active character. This procedure might be invoked with invalid data, so it is "extra robust." The individual characters must be printed one at a time using *print*, since they may be unprintable.

The conversion from control sequence to byte sequence for encTEXis implemented here. Of course, the simplest way is to implement an array of string pointers with *hash_size* length, but we assume that only a few control sequences will need to be converted. So *mubyte_cswrite*, an array with only 128 items, is used. The items point to the token lists. First token includes a csname number and the second points the string to be output. The third token includes the number of another csname and fourth token its pointer to the string etc. We need to do the sequential searching in one of the 128 token lists.

```
⟨Basic printing procedures 57⟩ +≡
procedure print_cs(p : integer); { prints a purported control sequence }
  var q: pointer; s: str_number;
  begin if active_noconvert ∧ (¬no_convert) ∧ (eq_type(p) = let) ∧ (equiv(p) = normal + 11) then
        { noconvert }
    begin no_convert ← true; return;
    end;
  s ← 0;
  if cs_converting ∧ (¬no_convert) then
    begin q ← mubyte_cswrite[p mod 128];
    while q ≠ null do
      if info(q) = p then
        begin s ← info(link(q)); q ← null;
        end
      else q ← link(link(q));
    end;
  no_convert ← false;
  if s > 0 then print(s)
  else if p < hash_base then { single character }
      if p ≥ single_base then
        if p = null_cs then
          begin print_esc("csname"); print_esc("endcsname"); print_char("␣");
          end
        else begin print_esc(p − single_base);
          if cat_code(p − single_base) = letter then print_char("␣");
          end
      else if p < active_base then print_esc("IMPOSSIBLE.")
        else print(p − active_base)
    else if ((p ≥ undefined_control_sequence) ∧ (p ≤ eqtb_size)) ∨ (p > eqtb_top) then
        print_esc("IMPOSSIBLE.")
      else if (text(p) ≥ str_ptr) then print_esc("NONEXISTENT.")
        else begin if (p ≥ prim_eqtb_base) ∧ (p < frozen_null_font) then
              print_esc(prim_text(p − prim_eqtb_base) − 1)
          else print_esc(text(p));
          print_char("␣");
          end;
exit: end;
```

287* Many of TEX's primitives need no *equiv*, since they are identifiable by their *eq_type* alone. These primitives are loaded into the hash table as follows:

⟨Put each of TEX's primitives into the hash table 244⟩ +≡
primitive("␣", *ex_space*, 0);
primitive("/", *ital_corr*, 0);
primitive("accent", *accent*, 0);
primitive("advance", *advance*, 0);
primitive("afterassignment", *after_assignment*, 0);
primitive("aftergroup", *after_group*, 0);
primitive("begingroup", *begin_group*, 0);
primitive("char", *char_num*, 0);
primitive("csname", *cs_name*, 0);
primitive("delimiter", *delim_num*, 0);
primitive("divide", *divide*, 0);
primitive("endcsname", *end_cs_name*, 0);
if *enctex_p* **then**
begin *primitive*("endmubyte", *end_cs_name*, 10);
end;
primitive("endgroup", *end_group*, 0); *text*(*frozen_end_group*) ← "endgroup";
eqtb[*frozen_end_group*] ← *eqtb*[*cur_val*];
primitive("expandafter", *expand_after*, 0);
primitive("font", *def_font*, 0);
primitive("letterspacefont", *letterspace_font*, 0);
primitive("pdfcopyfont", *pdf_copy_font*, 0);
primitive("fontdimen", *assign_font_dimen*, 0);
primitive("halign", *halign*, 0);
primitive("hrule", *hrule*, 0);
primitive("ignorespaces", *ignore_spaces*, 0);
primitive("insert", *insert*, 0);
primitive("mark", *mark*, 0);
primitive("mathaccent", *math_accent*, 0);
primitive("mathchar", *math_char_num*, 0);
primitive("mathchoice", *math_choice*, 0);
primitive("multiply", *multiply*, 0);
primitive("noalign", *no_align*, 0);
primitive("noboundary", *no_boundary*, 0);
primitive("noexpand", *no_expand*, 0);
primitive("pdfprimitive", *no_expand*, 1);
primitive("nonscript", *non_script*, 0);
primitive("omit", *omit*, 0);
primitive("parshape", *set_shape*, *par_shape_loc*);
primitive("penalty", *break_penalty*, 0);
primitive("prevgraf", *set_prev_graf*, 0);
primitive("radical", *radical*, 0);
primitive("read", *read_to_cs*, 0);
primitive("relax", *relax*, 256); { cf. *scan_file_name* }
text(*frozen_relax*) ← "relax"; *eqtb*[*frozen_relax*] ← *eqtb*[*cur_val*];
primitive("setbox", *set_box*, 0);
primitive("the", *the*, 0);
primitive("toks", *toks_register*, *mem_bot*);
primitive("vadjust", *vadjust*, 0);
primitive("valign", *valign*, 0);

primitive("`vcenter`", *vcenter*, 0);
primitive("`vrule`", *vrule*, 0);

288* Each primitive has a corresponding inverse, so that it is possible to display the cryptic numeric contents of *eqtb* in symbolic form. Every call of *primitive* in this program is therefore accompanied by some straightforward code that forms part of the *print_cmd_chr* routine below.

⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡
accent: *print_esc*("accent");
advance: *print_esc*("advance");
after_assignment: *print_esc*("afterassignment");
after_group: *print_esc*("aftergroup");
assign_font_dimen: *print_esc*("fontdimen");
begin_group: *print_esc*("begingroup");
break_penalty: *print_esc*("penalty");
char_num: *print_esc*("char");
cs_name: *print_esc*("csname");
def_font: *print_esc*("font");
letterspace_font: *print_esc*("letterspacefont");
pdf_copy_font: *print_esc*("pdfcopyfont");
delim_num: *print_esc*("delimiter");
divide: *print_esc*("divide");
end_cs_name: **if** *chr_code* = 10 **then** *print_esc*("endmubyte")
  else *print_esc*("endcsname");
end_group: *print_esc*("endgroup");
ex_space: *print_esc*("␣");
expand_after: **if** *chr_code* = 0 **then** *print_esc*("expandafter")
    ⟨Cases of *expandafter* for *print_cmd_chr* 1760⟩;
halign: *print_esc*("halign");
hrule: *print_esc*("hrule");
ignore_spaces: **if** *chr_code* = 0 **then** *print_esc*("ignorespaces")
  else *print_esc*("pdfprimitive");
insert: *print_esc*("insert");
ital_corr: *print_esc*("/");
mark: **begin** *print_esc*("mark");
  if *chr_code* > 0 **then** *print_char*("s");
  end;
math_accent: *print_esc*("mathaccent");
math_char_num: *print_esc*("mathchar");
math_choice: *print_esc*("mathchoice");
multiply: *print_esc*("multiply");
no_align: *print_esc*("noalign");
no_boundary: *print_esc*("noboundary");
no_expand: **if** *chr_code* = 0 **then** *print_esc*("noexpand")
  else *print_esc*("pdfprimitive");
non_script: *print_esc*("nonscript");
omit: *print_esc*("omit");
radical: *print_esc*("radical");
read_to_cs: **if** *chr_code* = 0 **then** *print_esc*("read") ⟨Cases of *read* for *print_cmd_chr* 1757⟩;
relax: *print_esc*("relax");
set_box: *print_esc*("setbox");
set_prev_graf: *print_esc*("prevgraf");
set_shape: **case** *chr_code* **of**
  par_shape_loc: *print_esc*("parshape");
   ⟨Cases of *set_shape* for *print_cmd_chr* 1862⟩
  end; { there are no other cases }

the: **if** *chr_code* = 0 **then** *print_esc*("the") ⟨ Cases of *the* for *print_cmd_chr* 1684 ⟩;
toks_register: ⟨ Cases of *toks_register* for *print_cmd_chr* 1830 ⟩;
vadjust: *print_esc*("vadjust");
valign: **if** *chr_code* = 0 **then** *print_esc*("valign")
  ⟨ Cases of *valign* for *print_cmd_chr* 1699 ⟩;
vcenter: *print_esc*("vcenter");
vrule: *print_esc*("vrule");

293* ⟨ Global variables 13 ⟩ +≡
save_stack: ↑*memory_word*;
save_ptr: 0 . . *save_size*; { first unused entry on *save_stack* }
max_save_stack: 0 . . *save_size*; { maximum usage of save stack }
cur_level: *quarterword*; { current nesting level for groups }
cur_group: *group_code*; { current group type }
cur_boundary: 0 . . *save_size*; { where the current level begins }

305* A global definition, which sets the level to *level_one*, will not be undone by *unsave*. If at least one global definition of *eqtb*[*p*] has been carried out within the group that just ended, the last such definition will therefore survive.

⟨ Store *save_stack*[*save_ptr*] in *eqtb*[*p*], unless *eqtb*[*p*] holds a global value 305* ⟩ ≡
 if ($p < int\_base$) ∨ ($p > eqtb\_size$) **then**
  if *eq_level*(*p*) = *level_one* **then**
   begin *eq_destroy*(*save_stack*[*save_ptr*]); { destroy the saved value }
   stat if *tracing_restores* > 0 **then** *restore_trace*(*p*, "retaining");
   tats
   end
  else begin *eq_destroy*(*eqtb*[*p*]); { destroy the current value }
   eqtb[*p*] ← *save_stack*[*save_ptr*]; { restore the saved value }
   stat if *tracing_restores* > 0 **then** *restore_trace*(*p*, "restoring");
   tats
   end
 else if *xeq_level*[*p*] ≠ *level_one* **then**
   begin *eqtb*[*p*] ← *save_stack*[*save_ptr*]; *xeq_level*[*p*] ← *l*;
   stat if *tracing_restores* > 0 **then** *restore_trace*(*p*, "restoring");
   tats
   end
  else begin stat if *tracing_restores* > 0 **then** *restore_trace*(*p*, "retaining");
   tats
   end

This code is used in section 304.

312* ⟨ Check the "constant" values for consistency 14 ⟩ +≡
if $cs\_token\_flag + eqtb\_size + hash\_extra > max\_halfword$ **then** $bad \leftarrow 21$;
if $(hash\_offset < 0) \lor (hash\_offset > hash\_base)$ **then** $bad \leftarrow 42$;

322* Input stacks and states. This implementation of TEX uses two different conventions for representing sequential stacks.

1) If there is frequent access to the top entry, and if the stack is essentially never empty, then the top entry is kept in a global variable (even better would be a machine register), and the other entries appear in the array *stack*$[0 \mathinner{..} (ptr - 1)]$. For example, the semantic stack described above is handled this way, and so is the input stack that we are about to study.

2) If there is infrequent top access, the entire stack contents are in the array *stack*$[0 \mathinner{..} (ptr - 1)]$. For example, the *save_stack* is treated this way, as we have seen.

The state of TEX's input mechanism appears in the input stack, whose entries are records with six fields, called *state*, *index*, *start*, *loc*, *limit*, and *name*. This stack is maintained with convention (1), so it is declared in the following way:

⟨Types in the outer block 18⟩ +≡
in_state_record = **record** *state_field*, *index_field*: *quarterword*;
start_field, *loc_field*, *limit_field*, *name_field*: *halfword*;
synctex_tag_field: *integer*; { stack the tag of the current file }
end;

323* ⟨Global variables 13⟩ +≡
input_stack: ↑*in_state_record*;
input_ptr: 0 .. *stack_size*; { first unused location of *input_stack* }
max_in_stack: 0 .. *stack_size*; { largest value of *input_ptr* when pushing }
cur_input: *in_state_record*; { the "top" input state, according to convention (1) }

324* We've already defined the special variable *loc* ≡ *cur_input.loc_field* in our discussion of basic input-output routines. The other components of *cur_input* are defined in the same way:

define *state* ≡ *cur_input.state_field* { current scanner state }
define *index* ≡ *cur_input.index_field* { reference for buffer information }
define *start* ≡ *cur_input.start_field* { starting position in *buffer* }
define *limit* ≡ *cur_input.limit_field* { end of current line in *buffer* }
define *name* ≡ *cur_input.name_field* { name of the current file }
define *synctex_tag* ≡ *cur_input.synctex_tag_field* { *SyncTEX* tag of the current file }

326* Additional information about the current line is available via the *index* variable, which counts how many lines of characters are present in the buffer below the current level. We have $index = 0$ when reading from the terminal and prompting the user for each line; then if the user types, e.g., '`\input paper`', we will have $index = 1$ while reading the file `paper.tex`. However, it does not follow that *index* is the same as the input stack pointer, since many of the levels on the input stack may come from token lists. For example, the instruction '`\input paper`' might occur in a token list.

The global variable *in_open* is equal to the *index* value of the highest non-token-list level. Thus, the number of partially read lines in the buffer is $in\_open + 1$, and we have $in\_open = index$ when we are not reading a token list.

If we are not currently reading from the terminal, or from an input stream, we are reading from the file variable *input_file*[*index*]. We use the notation *terminal_input* as a convenient abbreviation for $name = 0$, and *cur_file* as an abbreviation for *input_file*[*index*].

The global variable *line* contains the line number in the topmost open file, for use in error messages. If we are not reading from the terminal, *line_stack*[*index*] holds the line number for the enclosing level, so that *line* can be restored when the current file has been read. Line numbers should never be negative, since the negative of the current line number is used to identify the user's output routine in the *mode_line* field of the semantic nest entries.

If more information about the input state is needed, it can be included in small arrays like those shown here. For example, the current page or segment number in the input file might be put into a variable *page*, maintained for enclosing levels in '*page_stack*: **array** [1 .. *max_in_open*] **of** *integer*' by analogy with *line_stack*.

define *terminal_input* ≡ (*name* = 0) { are we reading from the terminal? }
define *cur_file* ≡ *input_file*[*index*] { the current *alpha_file* variable }

⟨ Global variables 13 ⟩ +≡
in_open: 0 .. *max_in_open*; { the number of lines in the buffer, less one }
open_parens: 0 .. *max_in_open*; { the number of open text files }
input_file: ↑*alpha_file*;
line: *integer*; { current line number in the current source file }
line_stack: ↑*integer*;
source_filename_stack: ↑*str_number*;
full_source_filename_stack: ↑*str_number*;

328* Here is a procedure that uses *scanner_status* to print a warning message when a subfile has ended, and at certain other crucial times:

⟨Declare the procedure called *runaway* 328*⟩ ≡
procedure *runaway*;
  var *p*: *pointer*; { head of runaway list }
  begin if *scanner_status* > *skipping* **then**
    begin case *scanner_status* **of**
    defining: **begin** *print_nl*("Runaway␣definition"); *p* ← *def_ref*;
      end;
    matching: **begin** *print_nl*("Runaway␣argument"); *p* ← *temp_head*;
      end;
    aligning: **begin** *print_nl*("Runaway␣preamble"); *p* ← *hold_head*;
      end;
    absorbing: **begin** *print_nl*("Runaway␣text"); *p* ← *def_ref*;
      end;
    end; { there are no other cases }
    print_char("?"); *print_ln*; *show_token_list*(*link*(*p*), *null*, *error_line* − 10);
    end;
  end;

This code is used in section 137.

330* The *param_stack* is an auxiliary array used to hold pointers to the token lists for parameters at the current level and subsidiary levels of input. This stack is maintained with convention (2), and it grows at a different rate from the others.

⟨Global variables 13⟩ +≡
param_stack: ↑*pointer*; { token list pointers for parameters }
param_ptr: 0 .. *param_size*; { first unused entry in *param_stack* }
max_param_stack: *integer*; { largest value of *param_ptr*, will be ≤ *param_size* + 9 }

340* But the trick is distracting us from our current goal, which is to understand the input state. So let's concentrate on the data structures that are being pseudoprinted as we finish up the *show_context* procedure.

⟨Pseudoprint the line 340*⟩ ≡
  begin_pseudoprint;
  if *buffer*[*limit*] = *end_line_char* **then** *j* ← *limit*
  else *j* ← *limit* + 1; { determine the effective end of the line }
  i ← *start*; *mubyte_skeep* ← *mubyte_keep*; *mubyte_sstart* ← *mubyte_start*; *mubyte_start* ← *false*;
  if *j* > 0 **then**
    while *i* < *j* **do**
      begin if *i* = *loc* **then** *set_trick_count*;
      print_buffer(*i*);
      end;
  mubyte_keep ← *mubyte_skeep*; *mubyte_start* ← *mubyte_sstart*

This code is used in section 334.

350* The *begin_file_reading* procedure starts a new level of input for lines of characters to be read from a file, or as an insertion from the terminal. It does not take care of opening the file, nor does it set *loc* or *limit* or *line*.

procedure *begin_file_reading*;
  begin if *in_open* = *max_in_open* **then** *overflow*("text␣input␣levels", *max_in_open*);
  if *first* = *buf_size* **then** *overflow*("buffer␣size", *buf_size*);
  incr(*in_open*); *push_input*; *index* ← *in_open*; *source_filename_stack*[*index*] ← 0;
  full_source_filename_stack[*index*] ← 0; *eof_seen*[*index*] ← *false*; *grp_stack*[*index*] ← *cur_boundary*;
  if_stack[*index*] ← *cond_ptr*; *line_stack*[*index*] ← *line*; *start* ← *first*; *state* ← *mid_line*; *name* ← 0;
    { *terminal_input* is now *true* }
  ⟨ Prepare terminal input *SyncTEX* information 1916* ⟩;
  end;

353* To get TEX's whole input mechanism going, we perform the following actions.

⟨ Initialize the input routines 353* ⟩ ≡
  begin *input_ptr* ← 0; *max_in_stack* ← 0; *source_filename_stack*[0] ← 0;
  full_source_filename_stack[0] ← 0; *in_open* ← 0; *open_parens* ← 0; *max_buf_stack* ← 0; *grp_stack*[0] ← 0;
  if_stack[0] ← *null*; *param_ptr* ← 0; *max_param_stack* ← 0; *first* ← *buf_size*;
  repeat *buffer*[*first*] ← 0; *decr*(*first*);
  until *first* = 0;
  scanner_status ← *normal*; *warning_index* ← *null*; *first* ← 1; *state* ← *new_line*; *start* ← 1; *index* ← 0;
  line ← 0; *name* ← 0; *force_eof* ← *false*; *align_state* ← 1000000;
  if ¬*init_terminal* **then goto** *final_end*;
  limit ← *last*; *first* ← *last* + 1; { *init_terminal* has set *loc* and *last* }
  end

This code is used in section 1515*.

354*. Getting the next token. The heart of TEX's input mechanism is the *get_next* procedure, which we shall develop in the next few sections of the program. Perhaps we shouldn't actually call it the "heart," however, because it really acts as TEX's eyes and mouth, reading the source files and gobbling them up. And it also helps TEX to regurgitate stored token lists that are to be processed again.

The main duty of *get_next* is to input one token and to set *cur_cmd* and *cur_chr* to that token's command code and modifier. Furthermore, if the input token is a control sequence, the *eqtb* location of that control sequence is stored in *cur_cs*; otherwise *cur_cs* is set to zero.

Underlying this simple description is a certain amount of complexity because of all the cases that need to be handled. However, the inner loop of *get_next* is reasonably short and fast.

When *get_next* is asked to get the next token of a `\read` line, it sets *cur_cmd* = *cur_chr* = *cur_cs* = 0 in the case that no more tokens appear on that line. (There might not be any tokens at all, if the *end_line_char* has *ignore* as its catcode.)

Some additional routines used by the encTEXextension have to be declared at this point.

⟨Declare additional routines for encTEX 1897*⟩

360*. ⟨Tell the user what has run away and try to recover 360*⟩ ≡

```
begin runaway; { print a definition, argument, or preamble }
if cur_cs = 0 then print_err("File␣ended")
else begin cur_cs ← 0; print_err("Forbidden␣control␣sequence␣found");
  end;
⟨Print either ‘definition’ or ‘use’ or ‘preamble’ or ‘text’, and insert tokens that should lead to
    recovery 361*⟩;
print("␣of␣"); sprint_cs(warning_index);
help4("I␣suspect␣you␣have␣forgotten␣a␣`}´,␣causing␣me")
("to␣read␣past␣where␣you␣wanted␣me␣to␣stop.")
("I´ll␣try␣to␣recover;␣but␣if␣the␣error␣is␣serious,")
("you´d␣better␣type␣`E´␣or␣`X´␣now␣and␣fix␣your␣file.");
error;
end
```

This code is used in section 358.

361*. The recovery procedure can't be fully understood without knowing more about the TEX routines that should be aborted, but we can sketch the ideas here: For a runaway definition or a runaway balanced text we will insert a right brace; for a runaway preamble, we will insert a special `\cr` token and a right brace; and for a runaway argument, we will set *long_state* to *outer_call* and insert `\par`.

⟨Print either ‘definition’ or ‘use’ or ‘preamble’ or ‘text’, and insert tokens that should lead to recovery 361*⟩ ≡

```
p ← get_avail;
case scanner_status of
defining: begin print("␣while␣scanning␣definition"); info(p) ← right_brace_token + "}";
  end;
matching: begin print("␣while␣scanning␣use"); info(p) ← par_token; long_state ← outer_call;
  end;
aligning: begin print("␣while␣scanning␣preamble"); info(p) ← right_brace_token + "}"; q ← p;
  p ← get_avail; link(p) ← q; info(p) ← cs_token_flag + frozen_cr; align_state ← −1000000;
  end;
absorbing: begin print("␣while␣scanning␣text"); info(p) ← right_brace_token + "}";
  end;
end; { there are no other cases }
ins_list(p)
```

This code is used in section 360*.

363*. Now we're ready to take the plunge into *get_next* itself. Parts of this routine are executed more often than any other instructions of TEX.

define *switch* = 25 { a label in *get_next* }
define *start_cs* = 26 { another }

procedure *get_next*; { sets *cur_cmd*, *cur_chr*, *cur_cs* to next token }
label *restart*, { go here to get the next input token }
switch, { go here to eat the next character from a file }
reswitch, { go here to digest it again }
start_cs, { go here to start looking for a control sequence }
found, { go here when a control sequence has been found }
exit; { go here when the next input token has been got }
var *k*: 0 .. *buf_size*; { an index into *buffer* }
t: *halfword*; { a token }
i, *j*: 0 .. *buf_size*; { more indexes for encTeX }
mubyte_incs: *boolean*; { control sequence is converted by mubyte }
p: *pointer*; { for encTeX test if noexpanding }
cat: 0 .. *max_char_code*; { *cat_code*(*cur_chr*), usually }
c, *cc*: *ASCII_code*; { constituents of a possible expanded code }
d: 2 .. 3; { number of excess characters in an expanded code }
begin *restart*: *cur_cs* ← 0;
if *state* ≠ *token_list* **then** ⟨Input from external file, **goto** *restart* if no input found 365*⟩
else ⟨Input from token list, **goto** *restart* if end of list or if a parameter needs to be expanded 379*⟩;
⟨If an alignment entry has just ended, take appropriate action 364⟩;
exit: **end**;

365*. ⟨Input from external file, **goto** *restart* if no input found 365*⟩ ≡
begin *switch*: **if** *loc* ≤ *limit* **then** { current line not yet finished }
begin { Use *k* instead of *loc* for type correctness. }
k ← *loc*; *cur_chr* ← *read_buffer*(*k*); *loc* ← *k*; *incr*(*loc*);
if (*mubyte_token* > 0) **then**
begin *state* ← *mid_line*; *cur_cs* ← *mubyte_token* − *cs_token_flag*; **goto** *found*;
end;
reswitch: *cur_cmd* ← *cat_code*(*cur_chr*); ⟨Change state if necessary, and **goto** *switch* if the current character should be ignored, or **goto** *reswitch* if the current character changes to another 366⟩;
end
else begin *state* ← *new_line*;
⟨Move to next line of file, or **goto** *restart* if there is no next line, or **return** if a `\read` line has finished 382⟩;
check_interrupt; **goto** *switch*;
end;
end

This code is used in section 363*.

376* Control sequence names are scanned only when they appear in some line of a file; once they have been scanned the first time, their *eqtb* location serves as a unique identification, so TEX doesn't need to refer to the original name any more except when it prints the equivalent in symbolic form.

The program that scans a control sequence has been written carefully in order to avoid the blowups that might otherwise occur if a malicious user tried something like '`\catcode´15=0`'. The algorithm might look at *buffer*[*limit* + 1], but it never looks at *buffer*[*limit* + 2].

If expanded characters like '`^^A`' or '`^^df`' appear in or just following a control sequence name, they are converted to single characters in the buffer and the process is repeated, slowly but surely.

```
⟨Scan a control sequence and set state ← skip_blanks or mid_line 376*⟩ ≡
  begin if loc > limit then cur_cs ← null_cs   { state is irrelevant in this case }
  else begin start_cs: mubyte_incs ← false; k ← loc; mubyte_skeep ← mubyte_keep;
    cur_chr ← read_buffer(k); cat ← cat_code(cur_chr);
    if (mubyte_in > 0) ∧ (¬mubyte_incs) ∧ ((mubyte_skip > 0) ∨ (cur_chr ≠ buffer[k])) then
      mubyte_incs ← true;
    incr(k);
    if mubyte_token > 0 then
      begin state ← mid_line; cur_cs ← mubyte_token − cs_token_flag; goto found;
      end;
    if cat = letter then state ← skip_blanks
    else if cat = spacer then state ← skip_blanks
      else state ← mid_line;
    if (cat = letter) ∧ (k ≤ limit) then ⟨Scan ahead in the buffer until finding a nonletter; if an expanded
          code is encountered, reduce it and goto start_cs; otherwise if a multiletter control sequence is
          found, adjust cur_cs and loc, and goto found 378*⟩
    else ⟨If an expanded code is present, reduce it and goto start_cs 377*⟩;
    mubyte_keep ← mubyte_skeep; cur_cs ← single_base + read_buffer(loc); incr(loc);
    end;
found: cur_cmd ← eq_type(cur_cs); cur_chr ← equiv(cur_cs);
  if cur_cmd ≥ outer_call then check_outer_validity;
  if write_noexpanding then
    begin p ← mubyte_cswrite[cur_cs mod 128];
    while p ≠ null do
      if info(p) = cur_cs then
        begin cur_cmd ← relax; cur_chr ← 256; p ← null;
        end
      else p ← link(link(p));
    end;
  end
```

This code is used in section 366.

377* Whenever we reach the following piece of code, we will have $cur\_chr = buffer[k-1]$ and $k \le limit + 1$ and $cat = cat\_code(cur\_chr)$. If an expanded code like `^^A` or `^^df` appears in $buffer[(k-1) \mathinner{\ldotp\ldotp} (k+1)]$ or $buffer[(k-1) \mathinner{\ldotp\ldotp} (k+2)]$, we will store the corresponding code in $buffer[k-1]$ and shift the rest of the buffer left two or three places.

⟨If an expanded code is present, reduce it and **goto** *start_cs* 377*⟩ ≡

```
begin if buffer[k] = cur_chr then if cat = sup_mark then if k < limit then
      begin c ← buffer[k + 1]; if c < ´200 then   { yes, one is indeed present }
        begin d ← 2;
        if is_hex(c) then if k + 2 ≤ limit then
            begin cc ← buffer[k + 2]; if is_hex(cc) then incr(d);
            end;
        if d > 2 then
          begin hex_to_cur_chr; buffer[k − 1] ← cur_chr;
          end
        else if c < ´100 then buffer[k − 1] ← c + ´100
          else buffer[k − 1] ← c − ´100;
        limit ← limit − d; first ← first − d;
        if mubyte_in > 0 then mubyte_keep ← k − loc;
        while k ≤ limit do
          begin buffer[k] ← buffer[k + d]; incr(k);
          end;
        goto start_cs;
        end;
      end;
  end
```

This code is used in sections 376* and 378*.

378* ⟨ Scan ahead in the buffer until finding a nonletter; if an expanded code is encountered, reduce it and **goto** *start_cs*; otherwise if a multiletter control sequence is found, adjust *cur_cs* and *loc*, and **goto** *found* 378* ⟩ ≡

```
begin repeat cur_chr ← read_buffer(k); cat ← cat_code(cur_chr);
  if mubyte_token > 0 then cat ← escape;
  if (mubyte_in > 0) ∧ (¬mubyte_incs) ∧ (cat = letter) ∧ ((mubyte_skip > 0) ∨ (cur_chr ≠ buffer[k]))
       then mubyte_incs ← true;
  incr(k);
until (cat ≠ letter) ∨ (k > limit);
⟨ If an expanded code is present, reduce it and goto start_cs 377* ⟩;
if cat ≠ letter then
  begin decr(k); k ← k − mubyte_skip;
  end;
if k > loc + 1 then { multiletter control sequence has been scanned }
  begin if mubyte_incs then { multibyte in csname occurrs }
    begin i ← loc; j ← first; mubyte_keep ← mubyte_skeep;
    if j − loc + k > max_buf_stack then
      begin max_buf_stack ← j − loc + k;
      if max_buf_stack ≥ buf_size then
        begin max_buf_stack ← buf_size; overflow("buffer␣size", buf_size);
        end;
      end;
    while i < k do
      begin buffer[j] ← read_buffer(i); incr(i); incr(j);
      end;
    if j = first + 1 then cur_cs ← single_base + buffer[first]
    else cur_cs ← id_lookup(first, j − first);
    end
  else cur_cs ← id_lookup(loc, k − loc);
  loc ← k; goto found;
  end;
end
```

This code is used in section 376*.

379* Let's consider now what happens when *get_next* is looking at a token list.

⟨Input from token list, **goto** *restart* if end of list or if a parameter needs to be expanded 379*⟩ ≡
if $loc \neq null$ **then** { list not exhausted }
  begin $t \leftarrow info(loc)$; $loc \leftarrow link(loc)$; { move to next }
  if $t \geq cs\_token\_flag$ **then** { a control sequence token }
    begin $cur\_cs \leftarrow t - cs\_token\_flag$; $cur\_cmd \leftarrow eq\_type(cur\_cs)$; $cur\_chr \leftarrow equiv(cur\_cs)$;
    if $cur\_cmd \geq outer\_call$ **then**
      if $cur\_cmd = dont\_expand$ **then** ⟨Get the next token, suppressing expansion 380⟩
      else *check_outer_validity*;
    if *write_noexpanding* **then**
      begin $p \leftarrow mubyte\_cswrite[cur\_cs$ **mod** $128]$;
      while $p \neq null$ **do**
        if $info(p) = cur\_cs$ **then**
          begin $cur\_cmd \leftarrow relax$; $cur\_chr \leftarrow 256$; $p \leftarrow null$;
          end
        else $p \leftarrow link(link(p))$;
      end;
    end
  else begin $cur\_cmd \leftarrow t$ **div** '400; $cur\_chr \leftarrow t$ **mod** '400;
    case *cur_cmd* **of**
    left_brace: *incr*(*align_state*);
    right_brace: *decr*(*align_state*);
    out_param: ⟨Insert macro parameter and **goto** *restart* 381⟩;
    othercases *do_nothing*
    endcases;
    end;
  end
else begin { we are done with this token list }
  end_token_list; **goto** *restart*; { resume previous level }
  end

This code is used in section 363*.

385* If the user has set the *pausing* parameter to some positive value, and if nonstop mode has not been selected, each line of input is displayed on the terminal and the transcript file, followed by '=>'. TEX waits for a response. If the response is simply *carriage_return*, the line is accepted as it stands, otherwise the line typed is used instead of the line in the file.

```
procedure firm_up_the_line;
  var k: 0 .. buf_size;   { an index into buffer }
  begin limit ← last;
  if pausing > 0 then
    if interaction > nonstop_mode then
      begin wake_up_terminal; print_ln; k ← start;
      while k < limit do
        begin print_buffer(k)
        end;
      first ← limit; prompt_input("=>");   { wait for user response }
      if last > first then
        begin for k ← first to last − 1 do   { move line down in buffer }
          buffer[k + start − first] ← buffer[k];
        limit ← start + last − first;
        end;
      end;
  end;
```

388* Expanding the next token. Only a dozen or so command codes > *max_command* can possibly be returned by *get_next*; in increasing order, they are *undefined_cs*, *expand_after*, *no_expand*, *input*, *if_test*, *fi_or_else*, *cs_name*, *convert*, *the*, *top_bot_mark*, *call*, *long_call*, *outer_call*, *long_outer_call*, and *end_template*.

The *expand* subroutine is used when *cur_cmd* > *max_command*. It removes a "call" or a conditional or one of the other special operations just listed. It follows that *expand* might invoke itself recursively. In all cases, *expand* destroys the current token, but it sets things up so that the next *get_next* will deliver the appropriate next token. The value of *cur_tok* need not be known when *expand* is called.

Since several of the basic scanning routines communicate via global variables, their values are saved as local variables of *expand* so that recursive calls don't invalidate them.

⟨Declare the procedure called *macro_call* 415⟩
⟨Declare the procedure called *insert_relax* 405⟩
⟨Declare ε-TEX procedures for expanding 1749⟩
procedure *pass_text*; *forward*;
procedure *start_input*; *forward*;
procedure *conditional*; *forward*;
procedure *get_x_token*; *forward*;
procedure *conv_toks*; *forward*;
procedure *ins_the_toks*; *forward*;
procedure *expand*;
  label *reswitch*;
  var *t*: *halfword*; { token that is being "expanded after" }
    b: *boolean*; { keep track of nested csnames }
    p, *q*, *r*: *pointer*; { for list manipulation }
    j: 0 . . *buf_size*; { index into *buffer* }
    cv_backup: *integer*; { to save the global quantity *cur_val* }
    cvl_backup, *radix_backup*, *co_backup*: *small_number*; { to save *cur_val_level*, etc. }
    backup_backup: *pointer*; { to save *link*(*backup_head*) }
    save_scanner_status: *small_number*; { temporary storage of *scanner_status* }
  begin *incr*(*expand_depth_count*);
  if *expand_depth_count* ≥ *expand_depth* **then** *overflow*("expansion␣depth", *expand_depth*);
  cv_backup ← *cur_val*; *cvl_backup* ← *cur_val_level*; *radix_backup* ← *radix*; *co_backup* ← *cur_order*;
  backup_backup ← *link*(*backup_head*);
reswitch: **if** *cur_cmd* < *call* **then** ⟨Expand a nonmacro 391⟩
  else if *cur_cmd* < *end_template* **then** *macro_call*
    else ⟨Insert a token containing *frozen_endv* 401⟩;
  cur_val ← *cv_backup*; *cur_val_level* ← *cvl_backup*; *radix* ← *radix_backup*; *cur_order* ← *co_backup*;
  link(*backup_head*) ← *backup_backup*; *decr*(*expand_depth_count*);
  end;

398* ⟨ Manufacture a control sequence name 398* ⟩ ≡
begin $r \leftarrow get\_avail$; $p \leftarrow r$; { head of the list of characters }
$b \leftarrow is\_in\_csname$; $is\_in\_csname \leftarrow true$;
repeat *get_x_token*;
  if $cur\_cs = 0$ **then** $store\_new\_token(cur\_tok)$;
until $cur\_cs \neq 0$;
if $(cur\_cmd \neq end\_cs\_name) \lor (cur\_chr \neq 0)$ **then** ⟨ Complain about missing `\endcsname` 399 ⟩;
$is\_in\_csname \leftarrow b$; ⟨ Look up the characters of list r in the hash table, and set *cur_cs* 400 ⟩;
$flush\_list(r)$;
if $eq\_type(cur\_cs) = undefined\_cs$ **then**
  begin $eq\_define(cur\_cs, relax, 256)$; { N.B.: The *save_stack* might change }
  end; { the control sequence will now match '`\relax`' }
$cur\_tok \leftarrow cur\_cs + cs\_token\_flag$; *back_input*;
end

This code is used in section 391.

426* If the parameter consists of a single group enclosed in braces, we must strip off the enclosing braces. That's why *rbrace_ptr* was introduced.

⟨ Tidy up the parameter just scanned, and tuck it away 426* ⟩ ≡
begin if $(m = 1) \land (info(p) < right\_brace\_limit)$ **then**
  begin $link(rbrace\_ptr) \leftarrow null$; $free\_avail(p)$; $p \leftarrow link(temp\_head)$; $pstack[n] \leftarrow link(p)$; $free\_avail(p)$;
  end
else $pstack[n] \leftarrow link(temp\_head)$;
$incr(n)$;
if $tracing\_macros > 0$ **then**
  if $(tracing\_stack\_levels = 0) \lor (input\_ptr < tracing\_stack\_levels)$ **then**
    begin *begin_diagnostic*; $print\_nl(match\_chr)$; $print\_int(n)$; *print*("<-");
    $show\_token\_list(pstack[n-1], null, 1000)$; $end\_diagnostic(false)$;
    end;
end

This code is used in section 418.

427* ⟨ Show the text of the macro being expanded 427* ⟩ ≡
begin *begin_diagnostic*;
if $tracing\_stack\_levels > 0$ **then**
  if $input\_ptr < tracing\_stack\_levels$ **then**
    begin $v \leftarrow input\_ptr$; *print_ln*; *print_char*("~");
    while $v > 0$ **do**
      begin *print_char*("."); $decr(v)$;
      end;
    $print\_cs(warning\_index)$; $token\_show(ref\_count)$;
    end
  else begin *print_char*("~"); *print_char*("~"); $print\_cs(warning\_index)$;
    end
else begin *print_ln*; $print\_cs(warning\_index)$; $token\_show(ref\_count)$;
  end;
$end\_diagnostic(false)$;
end

This code is used in section 415.

440* ⟨Fetch a character code from some table 440*⟩ ≡
begin *scan_char_num*;
if $m = xord\_code\_base$ **then** $scanned\_result(xord[cur\_val])(int\_val)$
else if $m = xchr\_code\_base$ **then** $scanned\_result(xchr[cur\_val])(int\_val)$
else if $m = xprn\_code\_base$ **then** $scanned\_result(xprn[cur\_val])(int\_val)$
else if $m = math\_code\_base$ **then** $scanned\_result(ho(math\_code(cur\_val)))(int\_val)$
else if $m < math\_code\_base$ **then** $scanned\_result(equiv(m + cur\_val))(int\_val)$
else $scanned\_result(eqtb[m + cur\_val].int)(int\_val)$;
end

This code is used in section 439.

527* ⟨Either process `\ifcase` or set b to the value of a boolean condition 527*⟩ ≡
case *this_if* **of**
if_char_code, *if_cat_code*: ⟨Test if two characters match 532⟩;
if_int_code, *if_dim_code*: ⟨Test relation between integers or dimensions 529⟩;
if_odd_code: ⟨Test if an integer is odd 530⟩;
if_vmode_code: $b \leftarrow (abs(mode) = vmode)$;
if_hmode_code: $b \leftarrow (abs(mode) = hmode)$;
if_mmode_code: $b \leftarrow (abs(mode) = mmode)$;
if_inner_code: $b \leftarrow (mode < 0)$;
if_void_code, *if_hbox_code*, *if_vbox_code*: ⟨Test box register status 531⟩;
ifx_code: ⟨Test if two tokens match 533⟩;
if_eof_code: **begin** *scan_four_bit_int_or_18*;
  if $cur\_val = 18$ **then** $b \leftarrow \neg shellenabledp$
  else $b \leftarrow (read\_open[cur\_val] = closed)$;
  end;
if_true_code: $b \leftarrow true$;
if_false_code: $b \leftarrow false$;
  ⟨Cases for *conditional* 1763⟩
if_case_code: ⟨Select the appropriate case and **return** or **goto** *common_ending* 535⟩;
if_pdfprimitive_code: **begin** $save\_scanner\_status \leftarrow scanner\_status$; $scanner\_status \leftarrow normal$; *get_next*;
  $scanner\_status \leftarrow save\_scanner\_status$;
  if $cur\_cs < hash\_base$ **then** $m \leftarrow prim\_lookup(cur\_cs - single\_base)$
  else $m \leftarrow prim\_lookup(text(cur\_cs))$;
  $b \leftarrow ((cur\_cmd \neq undefined\_cs) \land (m \neq undefined\_primitive) \land (cur\_cmd = prim\_eq\_type(m)) \land (cur\_chr = prim\_equiv(m)))$;
  end;
end {there are no other cases}

This code is used in section 524.

539*. The file names we shall deal with have the following structure: If the name contains '`/`' or '`:`' (for Amiga only), the file area consists of all characters up to and including the final such character; otherwise the file area is null. If the remaining file name contains '`.`', the file extension consists of all such characters from the last '`.`' to the end, otherwise the file extension is null.

We can scan such file names easily by using two global variables that keep track of the occurrences of area and extension delimiters:

⟨ Global variables 13 ⟩ +≡
area_delimiter: *pool_pointer*; { the most recent '`/`', if any }
ext_delimiter: *pool_pointer*; { the most recent '`.`', if any }

540*. Input files that can't be found in the user's area may appear in a standard system area called *TEX_area*. Font metric files whose areas are not given explicitly are assumed to appear in a standard system area called *TEX_font_area*. These system area names will, of course, vary from place to place.

In C, the default paths are specified separately.

541*. Here now is the first of the system-dependent routines for file name scanning.

procedure *begin_name*;
  begin *area_delimiter* ← 0; *ext_delimiter* ← 0; *quoted_filename* ← *false*;
  end;

542*. And here's the second. The string pool might change as the file name is being scanned, since a new `\csname` might be entered; therefore we keep *area_delimiter* and *ext_delimiter* relative to the beginning of the current string, instead of assigning an absolute address like *pool_ptr* to them.

function *more_name*(*c* : *ASCII_code*): *boolean*;
  begin if ($c =$ "␣") ∧ *stop_at_space* ∧ (¬*quoted_filename*) **then** *more_name* ← *false*
  else if $c =$ """" **then**
      begin *quoted_filename* ← ¬*quoted_filename*; *more_name* ← *true*;
      end
    else begin *str_room*(1); *append_char*(*c*); { contribute *c* to the current string }
      if *IS_DIR_SEP*(*c*) **then**
        begin *area_delimiter* ← *cur_length*; *ext_delimiter* ← 0;
        end
      else if $c =$ "." **then** *ext_delimiter* ← *cur_length*;
      more_name ← *true*;
      end;
  end;

543* The third. If a string is already in the string pool, the function *slow_make_string* does not create a new string but returns this string number, thus saving string space. Because of this new property of the returned string number it is not possible to apply *flush_string* to these strings.

```
procedure end_name;
  var temp_str: str_number; { result of file name cache lookups }
    j, s, t: pool_pointer; { running indices }
    must_quote: boolean; { whether we need to quote a string }
  begin if str_ptr + 3 > max_strings then overflow("number␣of␣strings", max_strings − init_str_ptr);
  str_room(6); { Room for quotes, if needed. }
    { add quotes if needed }
  if area_delimiter ≠ 0 then
    begin { maybe quote cur_area }
    must_quote ← false; s ← str_start[str_ptr]; t ← str_start[str_ptr] + area_delimiter; j ← s;
    while (¬must_quote) ∧ (j < t) do
      begin must_quote ← str_pool[j] = "␣"; incr(j);
      end;
    if must_quote then
      begin for j ← pool_ptr − 1 downto t do str_pool[j + 2] ← str_pool[j];
      str_pool[t + 1] ← """";
      for j ← t − 1 downto s do str_pool[j + 1] ← str_pool[j];
      str_pool[s] ← """";
      if ext_delimiter ≠ 0 then ext_delimiter ← ext_delimiter + 2;
      area_delimiter ← area_delimiter + 2; pool_ptr ← pool_ptr + 2;
      end;
    end; { maybe quote cur_name }
  s ← str_start[str_ptr] + area_delimiter;
  if ext_delimiter = 0 then t ← pool_ptr
  else t ← str_start[str_ptr] + ext_delimiter − 1;
  must_quote ← false; j ← s;
  while (¬must_quote) ∧ (j < t) do
    begin must_quote ← str_pool[j] = "␣"; incr(j);
    end;
  if must_quote then
    begin for j ← pool_ptr − 1 downto t do str_pool[j + 2] ← str_pool[j];
    str_pool[t + 1] ← """";
    for j ← t − 1 downto s do str_pool[j + 1] ← str_pool[j];
    str_pool[s] ← """";
    if ext_delimiter ≠ 0 then ext_delimiter ← ext_delimiter + 2;
    pool_ptr ← pool_ptr + 2;
    end;
  if ext_delimiter ≠ 0 then
    begin { maybe quote cur_ext }
    s ← str_start[str_ptr] + ext_delimiter − 1; t ← pool_ptr; must_quote ← false; j ← s;
    while (¬must_quote) ∧ (j < t) do
      begin must_quote ← str_pool[j] = "␣"; incr(j);
      end;
    if must_quote then
      begin str_pool[t + 1] ← """";
      for j ← t − 1 downto s do str_pool[j + 1] ← str_pool[j];
      str_pool[s] ← """"; pool_ptr ← pool_ptr + 2;
      end;
    end;
```

if *area_delimiter* $= 0$ **then** *cur_area* $\leftarrow$ `""`
else begin *cur_area* $\leftarrow$ *str_ptr*; *str_start*[*str_ptr* $+ 1$] $\leftarrow$ *str_start*[*str_ptr*] $+$ *area_delimiter*; *incr*(*str_ptr*);
  temp_str $\leftarrow$ *search_string*(*cur_area*);
  if *temp_str* > 0 **then**
    begin *cur_area* $\leftarrow$ *temp_str*; *decr*(*str_ptr*); { no *flush_string*, *pool_ptr* will be wrong! }
    for $j \leftarrow$ *str_start*[*str_ptr* $+ 1$] **to** *pool_ptr* $- 1$ **do**
      begin *str_pool*[$j -$ *area_delimiter*] $\leftarrow$ *str_pool*[j];
      end;
    pool_ptr $\leftarrow$ *pool_ptr* $-$ *area_delimiter*; { update *pool_ptr* }
    end;
  end;
if *ext_delimiter* $= 0$ **then**
  begin *cur_ext* $\leftarrow$ `""`; *cur_name* $\leftarrow$ *slow_make_string*;
  end
else begin *cur_name* $\leftarrow$ *str_ptr*;
  str_start[*str_ptr* $+ 1$] $\leftarrow$ *str_start*[*str_ptr*] $+$ *ext_delimiter* $-$ *area_delimiter* $- 1$; *incr*(*str_ptr*);
  cur_ext $\leftarrow$ *make_string*; *decr*(*str_ptr*); { undo extension string to look at name part }
  temp_str $\leftarrow$ *search_string*(*cur_name*);
  if *temp_str* > 0 **then**
    begin *cur_name* $\leftarrow$ *temp_str*; *decr*(*str_ptr*); { no *flush_string*, *pool_ptr* will be wrong! }
    for $j \leftarrow$ *str_start*[*str_ptr* $+ 1$] **to** *pool_ptr* $- 1$ **do**
      begin *str_pool*[$j -$ *ext_delimiter* $+$ *area_delimiter* $+ 1$] $\leftarrow$ *str_pool*[j];
      end;
    pool_ptr $\leftarrow$ *pool_ptr* $-$ *ext_delimiter* $+$ *area_delimiter* $+ 1$; { update *pool_ptr* }
    end;
  cur_ext $\leftarrow$ *slow_make_string*; { remake extension string }
  end;
end;

544* Conversely, here is a routine that takes three strings and prints a file name that might have produced them. (The routine is system dependent, because some operating systems put the file area last instead of first.)

```
define check_quoted(#) ≡ { check if string # needs quoting }
    if # ≠ 0 then
      begin j ← str_start[#];
      while (¬must_quote) ∧ (j < str_start[# + 1]) do
        begin must_quote ← str_pool[j] = "␣"; incr(j);
        end;
      end
define print_quoted(#) ≡ { print string #, omitting quotes }
    if # ≠ 0 then
      for j ← str_start[#] to str_start[# + 1] − 1 do
        if so(str_pool[j]) ≠ """" then print(so(str_pool[j]))
⟨Basic printing procedures 57⟩ +≡
procedure print_file_name(n, a, e : integer);
  var must_quote: boolean; { whether to quote the filename }
    j: pool_pointer; { index into str_pool }
  begin must_quote ← false; check_quoted(a); check_quoted(n);
  check_quoted(e); { FIXME: Alternative is to assume that any filename that has to be quoted has at least
      one quoted component...if we pick this, a number of insertions of print_file_name should go away.
      must_quote:=((a¡¿0)and(str_pool[str_start[a]]=””””))or ((n¡¿0)and(str_pool[str_start[n]]=””””))or
      ((e¡¿0)and(str_pool[str_start[e]]=””””)); }
  if must_quote then print_char("""");
  print_quoted(a); print_quoted(n); print_quoted(e);
  if must_quote then print_char("""");
  end;
```

545* Another system-dependent routine is needed to convert three internal TEX strings into the *name_of_file* value that is used to open files. The present code allows both lowercase and uppercase letters in the file name.

```
define append_to_name(#) ≡
      begin c ← #;
      if ¬(c = """") then
        begin incr(k);
        if k ≤ file_name_size then name_of_file[k] ← xchr[c];
        end
      end
procedure pack_file_name(n, a, e : str_number);
  var k: integer; { number of positions filled in name_of_file }
    c: ASCII_code; { character being packed }
    j: pool_pointer; { index into str_pool }
  begin k ← 0;
  if name_of_file then libc_free(name_of_file);
  name_of_file ← xmalloc_array(ASCII_code, length(a) + length(n) + length(e) + 1);
  for j ← str_start[a] to str_start[a + 1] − 1 do append_to_name(so(str_pool[j]));
  for j ← str_start[n] to str_start[n + 1] − 1 do append_to_name(so(str_pool[j]));
  for j ← str_start[e] to str_start[e + 1] − 1 do append_to_name(so(str_pool[j]));
  if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
  name_of_file[name_length + 1] ← 0;
  end;
```

546* A messier routine is also needed, since format file names must be scanned before TEX's string mechanism has been initialized. We shall use the global variable *TEX_format_default* to supply the text for default system areas and extensions related to format files.

Under UNIX we don't give the area part, instead depending on the path searching that will happen during file opening. Also, the length will be set in the main program.

define *format_area_length* $= 0$ { length of its area part }
define *format_ext_length* $= 4$ { length of its '`.fmt`' part }
define *format_extension* = `".fmt"` { the extension, as a `WEB` constant }

⟨ Global variables 13 ⟩ +≡
format_default_length: *integer*;
TEX_format_default: *cstring*;

547* We set the name of the default format file and the length of that name in C, instead of Pascal, since we want them to depend on the name of the program.

549* Here is the messy routine that was just mentioned. It sets *name_of_file* from the first n characters of *TEX_format_default*, followed by *buffer*$[a \mathrel{..} b]$, followed by the last *format_ext_length* characters of *TEX_format_default*.

We dare not give error messages here, since TEX calls this routine before the *error* routine is ready to roll. Instead, we simply drop excess characters, since the error will be detected in another way when a strange file name isn't found.

procedure *pack_buffered_name*(n : *small_number*; a, b : *integer*);
  var k: *integer*; { number of positions filled in *name_of_file* }
    c: *ASCII_code*; { character being packed }
    j: *integer*; { index into *buffer* or *TEX_format_default* }
  begin if $n + b - a + 1 +$ *format_ext_length* $>$ *file_name_size* **then**
    $b \leftarrow a +$ *file_name_size* $- n - 1 -$ *format_ext_length*;
  $k \leftarrow 0$;
  if *name_of_file* **then** *libc_free*(*name_of_file*);
  name_of_file $\leftarrow$ *xmalloc_array*(*ASCII_code*, $n + (b - a + 1) +$ *format_ext_length* $+ 1$);
  for $j \leftarrow 1$ **to** n **do** *append_to_name*(*xord*[*ucharcast*(*TEX_format_default*[j])]);
  for $j \leftarrow a$ **to** b **do** *append_to_name*(*buffer*[j]);
  for $j \leftarrow$ *format_default_length* $-$ *format_ext_length* $+ 1$ **to** *format_default_length* **do**
    append_to_name(*xord*[*ucharcast*(*TEX_format_default*[j])]);
  if $k \le$ *file_name_size* **then** *name_length* $\leftarrow k$ **else** *name_length* $\leftarrow$ *file_name_size*;
  name_of_file[*name_length* $+ 1$] $\leftarrow 0$;
  end;

550* Here is the only place we use *pack_buffered_name*. This part of the program becomes active when a "virgin" TEX is trying to get going, just after the preliminary initialization, or when the user is substituting another format file by typing '&' after the initial '**' prompt. The buffer contains the first line of input in *buffer*[*loc* .. (*last* − 1)], where *loc* < *last* and *buffer*[*loc*] ≠ "␣".

⟨ Declare the function called *open_fmt_file* 550* ⟩ ≡
function *open_fmt_file*: *boolean*;
  label *found*, *exit*;
  var *j*: 0 .. *buf_size*; { the first space after the format file name }
  begin *j* ← *loc*;
  if *buffer*[*loc*] = "&" **then**
    begin *incr*(*loc*); *j* ← *loc*; *buffer*[*last*] ← "␣";
    while *buffer*[*j*] ≠ "␣" **do** *incr*(*j*);
    pack_buffered_name(0, *loc*, *j* − 1); { Kpathsea does everything }
    if *w_open_in*(*fmt_file*) **then goto** *found*;
    wake_up_terminal; *wterm*(´Sorry,␣I␣can´´t␣find␣the␣format␣`´);
    fputs(*stringcast*(*name_of_file* + 1), *stdout*); *wterm*(´´´;␣will␣try␣`´);
    fputs(*TEX_format_default* + 1, *stdout*); *wterm_ln*(´´´.´); *update_terminal*;
    end; { now pull out all the stops: try for the system `plain` file }
  pack_buffered_name(*format_default_length* − *format_ext_length*, 1, 0);
  if ¬*w_open_in*(*fmt_file*) **then**
    begin *wake_up_terminal*; *wterm*(´I␣can´´t␣find␣the␣format␣file␣`´);
    fputs(*TEX_format_default* + 1, *stdout*); *wterm_ln*(´´´!´); *open_fmt_file* ← *false*; **return**;
    end;
found: *loc* ← *j*; *open_fmt_file* ← *true*;
exit: **end**;

This code is used in section 1479*.

551* Operating systems often make it possible to determine the exact name (and possible version number) of a file that has been opened. The following routine, which simply makes a TEX string from the value of *name_of_file*, should ideally be changed to deduce the full name of file *f*, which is the file most recently opened, if it is possible to do this in a Pascal program.

This routine might be called after string memory has overflowed, hence we dare not use '*str_room*'.

function *make_name_string*: *str_number*;
  var *k*: 1 .. *file_name_size*; { index into *name_of_file* }
    save_area_delimiter, *save_ext_delimiter*: *pool_pointer*;
    save_name_in_progress, *save_stop_at_space*: *boolean*;
  begin if $(pool\_ptr + name\_length > pool\_size) \lor (str\_ptr = max\_strings) \lor (cur\_length > 0)$ **then**
    make_name_string ← "?"
  else begin for $k \leftarrow 1$ **to** *name_length* **do** *append_char*(*xord*[*name_of_file*[*k*]]);
    make_name_string ← *make_string*; { At this point we also set *cur_name*, *cur_ext*, and *cur_area* to match the contents of *name_of_file*. }
    save_area_delimiter ← *area_delimiter*; *save_ext_delimiter* ← *ext_delimiter*;
    save_name_in_progress ← *name_in_progress*; *save_stop_at_space* ← *stop_at_space*;
    name_in_progress ← *true*; *begin_name*; *stop_at_space* ← *false*; $k \leftarrow 1$;
    while $(k \le name\_length) \land (more\_name(name\_of\_file[k]))$ **do** *incr*(*k*);
    stop_at_space ← *save_stop_at_space*; *end_name*; *name_in_progress* ← *save_name_in_progress*;
    area_delimiter ← *save_area_delimiter*; *ext_delimiter* ← *save_ext_delimiter*;
    end;
  end;
function *a_make_name_string*(**var** *f* : *alpha_file*): *str_number*;
  begin *a_make_name_string* ← *make_name_string*;
  end;
function *b_make_name_string*(**var** *f* : *byte_file*): *str_number*;
  begin *b_make_name_string* ← *make_name_string*;
  end;
function *w_make_name_string*(**var** *f* : *word_file*): *str_number*;
  begin *w_make_name_string* ← *make_name_string*;
  end;

552* Now let's consider the "driver" routines by which TEX deals with file names in a system-independent manner. First comes a procedure that looks for a file name in the input by calling *get_x_token* for the information.

procedure *scan_file_name*;
  label *done*;
  var *save_warning_index*: *pointer*;
  begin *save_warning_index* ← *warning_index*; *warning_index* ← *cur_cs*;
    { store *cur_cs* here to remember until later }
  ⟨ Get the next non-blank non-relax non-call token 430 ⟩;
    { here the program expands tokens and removes spaces and `\relax`es from the input. The `\relax` removal follows LuaTeX"s implementation, and other cases of balanced text scanning. }
  back_input; { return the last token to be read by either code path }
  if *cur_cmd* = *left_brace* **then** *scan_file_name_braced*
  else begin *name_in_progress* ← *true*; *begin_name*; ⟨ Get the next non-blank non-call token 432 ⟩;
    loop begin if (*cur_cmd* > *other_char*) ∨ (*cur_chr* > 255) **then** { not a character }
        begin *back_input*; **goto** *done*;
        end; { If *cur_chr* is a space and we're not scanning a token list, check whether we're at the end of the buffer. Otherwise we end up adding spurious spaces to file names in some cases. }
      if (*cur_chr* = "␣") ∧ (*state* ≠ *token_list*) ∧ (*loc* > *limit*) **then goto** *done*;
      if ¬*more_name*(*cur_chr*) **then goto** *done*;
      get_x_token;
      end;
    end;
done: *end_name*; *name_in_progress* ← *false*; *warning_index* ← *save_warning_index*;
    { restore *warning_index* }
  end;

556* If some trouble arises when TEX tries to open a file, the following routine calls upon the user to supply another file name. Parameter *s* is used in the error message to identify the type of file; parameter *e* is the default extension if none is given. Upon exit from the routine, variables *cur_name*, *cur_area*, *cur_ext*, and *name_of_file* are ready for another attempt at file opening.

```
procedure prompt_file_name(s, e : str_number);
  label done;
  var k: 0 .. buf_size;  { index into buffer }
    saved_cur_name: str_number;  { to catch empty terminal input }
    saved_cur_ext: str_number;  { to catch empty terminal input }
    saved_cur_area: str_number;  { to catch empty terminal input }
  begin if interaction = scroll_mode then wake_up_terminal;
  if s = "input␣file␣name" then print_err("I␣can´t␣find␣file␣`")
  else print_err("I␣can´t␣write␣on␣file␣`");
  print_file_name(cur_name, cur_area, cur_ext); print("´.");
  if (e = ".tex") ∨ (e = "") then show_context;
  print_ln; print_c_string(prompt_file_name_help_msg);
  if (e ≠ "") then
    begin print(";␣default␣file␣extension␣is␣`"); print(e); print("´");
    end;
  print(")"); print_ln; print_nl("Please␣type␣another␣"); print(s);
  if interaction < scroll_mode then fatal_error("***␣(job␣aborted,␣file␣error␣in␣nonstop␣mode)");
  saved_cur_name ← cur_name; saved_cur_ext ← cur_ext; saved_cur_area ← cur_area; clear_terminal;
  prompt_input(":␣"); ⟨Scan file name in the buffer 557⟩;
  if (length(cur_name) = 0) ∧ (cur_ext = "") ∧ (cur_area = "") then
    begin cur_name ← saved_cur_name; cur_ext ← saved_cur_ext; cur_area ← saved_cur_area;
    end
  else if cur_ext = "" then cur_ext ← e;
  pack_cur_name;
  end;
```

558* Here's an example of how these conventions are used. Whenever it is time to ship out a box of stuff, we shall use the macro *ensure_dvi_open*.

```
  define log_name ≡ texmf_log_name
  define ensure_dvi_open ≡
           if output_file_name = 0 then
             begin if job_name = 0 then open_log_file;
             pack_job_name(".dvi");
             while ¬b_open_out(dvi_file) do prompt_file_name("file␣name␣for␣output", ".dvi");
             output_file_name ← b_make_name_string(dvi_file);
             end

⟨Global variables 13⟩ +≡
dvi_file: byte_file;  { the device-independent output goes here }
output_file_name: str_number;  { full name of the output file }
log_name: str_number;  { full name of the log file }
```

560* The *open_log_file* routine is used to open the transcript file and to help it catch up to what has previously been printed on the terminal.

procedure *open_log_file*;
  var *old_setting*: 0 .. *max_selector*; { previous *selector* setting }
    k: 0 .. *buf_size*; { index into *months* and *buffer* }
    l: 0 .. *buf_size*; { end of first input line }
    months: *const_cstring*;
  begin *old_setting* ← *selector*;
  if *job_name* = 0 **then** *job_name* ← *get_job_name*(`"texput"`);
  pack_job_name(`".fls"`); *recorder_change_filename*(*stringcast*(*name_of_file* + 1)); *pack_job_name*(`".log"`);
  while ¬*a_open_out*(*log_file*) **do** ⟨ Try to get a different log file name 561 ⟩;
  log_name ← *a_make_name_string*(*log_file*); *selector* ← *log_only*; *log_opened* ← *true*;
  ⟨ Print the banner line, including the date and time 562* ⟩;
  if *mltex_enabled_p* **then**
    begin *wlog_cr*; *wlog*(`´MLTeX␣v2.2␣enabled´`);
    end;
  if *enctex_enabled_p* **then**
    begin *wlog_cr*; *wlog*(*encTeX_banner*); *wlog*(`´,␣reencoding␣enabled´`);
    if *translate_filename* **then**
      begin *wlog_cr*; *wlog*(`´␣(\xordcode,␣\xchrcode,␣\xprncode␣overridden␣by␣TCX)´`);
      end;
    end;
  input_stack[*input_ptr*] ← *cur_input*; { make sure bottom level is in memory }
  print_nl(`"**"`); *l* ← *input_stack*[0].*limit_field*; { last position of first line }
  if *buffer*[*l*] = *end_line_char* **then** *decr*(*l*);
  for *k* ← 1 **to** *l* **do** *print*(*buffer*[*k*]);
  print_ln; { now the transcript file contains the first line of input }
  selector ← *old_setting* + 2; { *log_only* or *term_and_log* }
  end;

562* ⟨Print the banner line, including the date and time 562*⟩ ≡

```
begin if src_specials_p ∨ file_line_error_style_p ∨ parse_first_line_p then wlog(banner_k)
else wlog(banner);
wlog(version_string); slow_print(format_ident); print("␣␣"); print_int(sys_day); print_char("␣");
months ← ´␣JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC´;
for k ← 3 ∗ sys_month − 2 to 3 ∗ sys_month do wlog(months[k]);
print_char("␣"); print_int(sys_year); print_char("␣"); print_two(sys_time div 60); print_char(":");
print_two(sys_time mod 60);
if eTeX_ex then
  begin ; wlog_cr; wlog(´entering␣extended␣mode´);
  end;
if shellenabledp then
  begin wlog_cr; wlog(´␣´);
  if restrictedshell then
    begin wlog(´restricted␣´);
    end;
  wlog(´\write18␣enabled.´)
  end;
if src_specials_p then
  begin wlog_cr; wlog(´␣Source␣specials␣enabled.´)
  end;
if file_line_error_style_p then
  begin wlog_cr; wlog(´␣file:line:error␣style␣messages␣enabled.´)
  end;
if parse_first_line_p then
  begin wlog_cr; wlog(´␣%&-line␣parsing␣enabled.´);
  end;
if translate_filename then
  begin wlog_cr; wlog(´␣(´); fputs(translate_filename, log_file); wlog(´)´);
  end;
end
```

This code is used in section 560*.

563* Let's turn now to the procedure that is used to initiate file reading when an '`\input`' command is being processed. Beware: For historic reasons, this code foolishly conserves a tiny bit of string pool space; but that can confuse the interactive '`E`' option.

procedure *start_input*; { TEX will `\input` something }
  label *done*;
  var *temp_str*: *str_number*; *v*: *pointer*;
  begin *scan_file_name*; { set *cur_name* to desired file name }
  pack_cur_name;
  loop begin *begin_file_reading*; { set up *cur_file* and new level of input }
    tex_input_type ← 1; { Tell *open_input* we are `\input`. }
      { Kpathsea tries all the various ways to get the file. }
    if *kpse_in_name_ok*(*stringcast*(*name_of_file* + 1)) ∧ *a_open_in*(*cur_file*, *kpse_tex_format*) **then**
      goto *done*;
    end_file_reading; { remove the level that didn't work }
    prompt_file_name("`input␣file␣name`", "");
    end;
done: *name* ← *a_make_name_string*(*cur_file*); *source_filename_stack*[*in_open*] ← *name*;
  full_source_filename_stack[*in_open*] ← *make_full_name_string*;
  if *name* = *str_ptr* − 1 **then** { we can try to conserve string pool space now }
    begin *temp_str* ← *search_string*(*name*);
    if *temp_str* > 0 **then**
      begin *name* ← *temp_str*; *flush_string*;
      end;
    end;
  if *job_name* = 0 **then**
    begin *job_name* ← *get_job_name*(*cur_name*); *open_log_file*;
    end; { *open_log_file* doesn't *show_context*, so *limit* and *loc* needn't be set to meaningful values yet }
  if *term_offset* + *length*(*full_source_filename_stack*[*in_open*]) > *max_print_line* − 2 **then** *print_ln*
  else if (*term_offset* > 0) ∨ (*file_offset* > 0) **then** *print_char*("␣");
  print_char("("); *incr*(*open_parens*); *slow_print*(*full_source_filename_stack*[*in_open*]); *update_terminal*;
  if *tracing_stack_levels* > 0 **then**
    begin *begin_diagnostic*; *print_ln*; *print_char*("~"); *v* ← *input_ptr* − 1;
    if *v* < *tracing_stack_levels* **then**
      while *v* > 0 **do**
        begin *print_char*("."); *decr*(*v*);
        end
    else *print_char*("~");
    slow_print("`INPUT␣`"); *slow_print*(*cur_name*); *slow_print*(*cur_ext*); *print_ln*; *end_diagnostic*(*false*);
    end;
  state ← *new_line*; ⟨ Prepare new file *SyncTEX* information 1915* ⟩;
  ⟨ Read the first line of the new file 564 ⟩;
  end;

574* So that is what `TFM` files hold. Since TEX has to absorb such information about lots of fonts, it stores most of the data in a large array called *font_info*. Each item of *font_info* is a *memory_word*; the *fix_word* data gets converted into *scaled* entries, while everything else goes into words of type *four_quarters*.

When the user defines `\font\f`, say, TEX assigns an internal number to the user's font `\f`. Adding this number to *font_id_base* gives the *eqtb* location of a "frozen" control sequence that will always select the font.

⟨Types in the outer block 18⟩ +≡
internal_font_number = *integer*; { *font* in a *char_node* }
font_index = *integer*; { index into *font_info* }
nine_bits = *min_quarterword* .. *non_char*;

575* Here now is the (rather formidable) array of font arrays.

define *non_char* ≡ *qi*(256) { a *halfword* code that can't match a real character }
define *non_address* = 0 { a spurious *bchar_label* }

⟨Global variables 13⟩ +≡
font_info: ↑*fmemory_word*; { the big collection of font data }
fmem_ptr: *font_index*; { first unused word of *font_info* }
font_ptr: *internal_font_number*; { largest internal font number in use }
font_check: ↑*four_quarters*; { check sum }
font_size: ↑*scaled*; { "at" size }
font_dsize: ↑*scaled*; { "design" size }
font_params: ↑*font_index*; { how many font parameters are present }
font_name: ↑*str_number*; { name of the font }
font_area: ↑*str_number*; { area of the font }
font_bc: ↑*eight_bits*; { beginning (smallest) character code }
font_ec: ↑*eight_bits*; { ending (largest) character code }
font_glue: ↑*pointer*; { glue specification for interword space, *null* if not allocated }
font_used: ↑*boolean*; { has a character from this font actually appeared in the output? }
hyphen_char: ↑*integer*; { current `\hyphenchar` values }
skew_char: ↑*integer*; { current `\skewchar` values }
bchar_label: ↑*font_index*;
{ start of *lig_kern* program for left boundary character, *non_address* if there is none }
font_bchar: ↑*nine_bits*; { boundary character, *non_char* if there is none }
font_false_bchar: ↑*nine_bits*; { *font_bchar* if it doesn't exist in the font, otherwise *non_char* }

576* Besides the arrays just enumerated, we have directory arrays that make it easy to get at the individual entries in *font_info*. For example, the *char_info* data for character c in font f will be in *font_info*[*char_base*[f] + c].*qqqq*; and if w is the *width_index* part of this word (the *b0* field), the width of the character is *font_info*[*width_base*[f] + w].*sc*. (These formulas assume that *min_quarterword* has already been added to c and to w, since TEX stores its quarterwords that way.)

⟨Global variables 13⟩ +≡
char_base: ↑*integer*; { base addresses for *char_info* }
width_base: ↑*integer*; { base addresses for widths }
height_base: ↑*integer*; { base addresses for heights }
depth_base: ↑*integer*; { base addresses for depths }
italic_base: ↑*integer*; { base addresses for italic corrections }
lig_kern_base: ↑*integer*; { base addresses for ligature/kerning programs }
kern_base: ↑*integer*; { base addresses for kerns }
exten_base: ↑*integer*; { base addresses for extensible recipes }
param_base: ↑*integer*; { base addresses for font parameters }

577* ⟨Set initial values of key variables 21⟩ +≡

578.* TEX always knows at least one font, namely the null font. It has no characters, and its seven parameters are all equal to zero.

⟨ Initialize table entries (done by INITEX only) 182 ⟩ +≡

580* Of course we want to define macros that suppress the detail of how font information is actually packed, so that we don't have to write things like

$$font\_info[width\_base[f] + font\_info[char\_base[f] + c].qqqq.b0].sc$$

too often. The WEB definitions here make *char_info*$(f)(c)$ the *four_quarters* word of font information corresponding to character c of font f. If q is such a word, *char_width*$(f)(q)$ will be the character's width; hence the long formula above is at least abbreviated to

$$char\_width(f)(char\_info(f)(c)).$$

Usually, of course, we will fetch q first and look at several of its fields at the same time.

The italic correction of a character will be denoted by *char_italic*$(f)(q)$, so it is analogous to *char_width*. But we will get at the height and depth in a slightly different way, since we usually want to compute both height and depth if we want either one. The value of *height_depth*(q) will be the 8-bit quantity

$$b = height\_index \times 16 + depth\_index,$$

and if b is such a byte we will write *char_height*$(f)(b)$ and *char_depth*$(f)(b)$ for the height and depth of the character c for which $q = char\_info(f)(c)$. Got that?

The tag field will be called *char_tag*(q); the remainder byte will be called *rem_byte*(q), using a macro that we have already defined above.

Access to a character's *width*, *height*, *depth*, and *tag* fields is part of TEX's inner loop, so we want these macros to produce code that is as fast as possible under the circumstances.

MLTEX will assume that a character c exists iff either exists in the current font or a character substitution definition for this character was defined using `\charsubdef`. To avoid the distinction between these two cases, MLTEX introduces the notion "effective character" of an input character c. If c exists in the current font, the effective character of c is the character c itself. If it doesn't exist but a character substitution is defined, the effective character of c is the base character defined in the character substitution. If there is an effective character for a non-existing character c, the "virtual character" c will get appended to the horizontal lists.

The effective character is used within *char_info* to access appropriate character descriptions in the font. For example, when calculating the width of a box, MLTEX will use the metrics of the effective characters. For the case of a substitution, MLTEX uses the metrics of the base character, ignoring the metrics of the accent character.

If character substitutions are changed, it will be possible that a character c neither exists in a font nor there is a valid character substitution for c. To handle these cases *effective_char* should be called with its first argument set to *true* to ensure that it will still return an existing character in the font. If neither c nor the substituted base character in the current character substitution exists, *effective_char* will output a warning and return the character *font_bc*$[f]$ (which is incorrect, but can not be changed within the current framework).

Sometimes character substitutions are unwanted, therefore the original definition of *char_info* can be used using the macro *orig_char_info*. Operations in which character substitutions should be avoided are, for example, loading a new font and checking the font metric information in this font, and character accesses in math mode.

define *char_list_exists*(#) $\equiv$ (*char_sub_code*(#) $>$ *hi*(0))
define *char_list_accent*(#) $\equiv$ (*ho*(*char_sub_code*(#)) **div** 256)
define *char_list_char*(#) $\equiv$ (*ho*(*char_sub_code*(#)) **mod** 256)
define *char_info_end*(#) $\equiv$ # $\boxed{)}$] .*qqqq*
define *char_info*(#) $\equiv$ *font_info* [*char_base*[#] + *effective_char* $\boxed{(}$ *true*, #, *char_info_end*
define *orig_char_info_end*(#) $\equiv$ #] .*qqqq*
define *orig_char_info*(#) $\equiv$ *font_info* [*char_base*[#] + *orig_char_info_end*

define *char_width_end*(#) ≡ #.*b0*] .*sc*
define *char_width*(#) ≡ *font_info* [*width_base*[#] + *char_width_end*
define *char_exists*(#) ≡ (#.*b0* > *min_quarterword*)
define *char_italic_end*(#) ≡ (*qo*(#.*b2*)) **div** 4] .*sc*
define *char_italic*(#) ≡ *font_info* [*italic_base*[#] + *char_italic_end*
define *height_depth*(#) ≡ *qo*(#.*b1*)
define *char_height_end*(#) ≡ (#) **div** 16] .*sc*
define *char_height*(#) ≡ *font_info* [*height_base*[#] + *char_height_end*
define *char_depth_end*(#) ≡ (#) **mod** 16] .*sc*
define *char_depth*(#) ≡ *font_info* [*depth_base*[#] + *char_depth_end*
define *char_tag*(#) ≡ ((*qo*(#.*b2*)) **mod** 4)

586*. TEX checks the information of a `TFM` file for validity as the file is being read in, so that no further checks will be needed when typesetting is going on. The somewhat tedious subroutine that does this is called *read_font_info*. It has four parameters: the user font identifier *u*, the file name and area strings *nom* and *aire*, and the "at" size *s*. If *s* is negative, it's the negative of a scale factor to be applied to the design size; $s = -1000$ is the normal case. Otherwise *s* will be substituted for the design size; in this case, *s* must be positive and less than 2048 pt (i.e., it must be less than 2^{27} when considered as an integer).

The subroutine opens and closes a global file variable called *tfm_file*. It returns the value of the internal font number that was just loaded. If an error is detected, an error message is issued and no font information is stored; *null_font* is returned in this case.

define *bad_tfm* = 11 { label for *read_font_info* }
define *abort* ≡ **goto** *bad_tfm* { do this when the `TFM` data is wrong }

⟨ Declare additional functions for MLTEX 1882* ⟩
function *read_font_info*(*u* : *pointer*; *nom*, *aire* : *str_number*; *s* : *scaled*): *internal_font_number*;
{ input a `TFM` file }
label *done*, *bad_tfm*, *not_found*;
var *k*: *font_index*; { index into *font_info* }
name_too_long: *boolean*; { *nom* or *aire* exceeds 255 bytes? }
file_opened: *boolean*; { was *tfm_file* successfully opened? }
lf, *lh*, *bc*, *ec*, *nw*, *nh*, *nd*, *ni*, *nl*, *nk*, *ne*, *np*: *halfword*; { sizes of subfiles }
f: *internal_font_number*; { the new font's number }
g: *internal_font_number*; { the number to return }
a, *b*, *c*, *d*: *eight_bits*; { byte variables }
qw: *four_quarters*; *sw*: *scaled*; { accumulators }
bch_label: *integer*; { left boundary start location, or infinity }
bchar: 0 . . 256; { boundary character, or 256 }
z: *scaled*; { the design size or the "at" size }
alpha: *integer*; *beta*: 1 . . 16; { auxiliary quantities used in fixed-point multiplication }
begin *g* ← *null_font*;
⟨ Read and check the font data; *abort* if the `TFM` file is malformed; if there's no room for this font, say so and **goto** *done*; otherwise *incr*(*font_ptr*) and **goto** *done* 588 ⟩;
bad_tfm: ⟨ Report that the font won't be loaded 587* ⟩;
done: **if** *file_opened* **then** *b_close*(*tfm_file*);
read_font_info ← *g*;
end;

587* There are programs called `TFtoPL` and `PLtoTF` that convert between the `TFM` format and a symbolic property-list format that can be easily edited. These programs contain extensive diagnostic information, so TEX does not have to bother giving precise details about why it rejects a particular `TFM` file.

define *start_font_error_message* ≡ *print_err*("Font␣"); *sprint_cs*(*u*); *print_char*("=");
print_file_name(*nom*, *aire*, "");
if $s \ge 0$ **then**
begin *print*("␣at␣"); *print_scaled*(*s*); *print*("pt");
end
else if $s \ne -1000$ **then**
begin *print*("␣scaled␣"); *print_int*($-s$);
end

⟨ Report that the font won't be loaded 587* ⟩ ≡
start_font_error_message;
if *file_opened* **then** *print*("␣not␣loadable:␣Bad␣metric␣(TFM)␣file")
else if *name_too_long* **then** *print*("␣not␣loadable:␣Metric␣(TFM)␣file␣name␣too␣long")
else *print*("␣not␣loadable:␣Metric␣(TFM)␣file␣not␣found");
help5("I␣wasn´t␣able␣to␣read␣the␣size␣data␣for␣this␣font,")
("so␣I␣will␣ignore␣the␣font␣specification.")
("[Wizards␣can␣fix␣TFM␣files␣using␣TFtoPL/PLtoTF.]")
("You␣might␣try␣inserting␣a␣different␣font␣spec;")
("e.g.,␣type␣`I\font<same␣font␣id>=<substitute␣font␣name>´."); *error*

This code is used in section 586*.

589* ⟨ Open *tfm_file* for input 589* ⟩ ≡
file_opened ← *false*; *name_too_long* ← (*length*(*nom*) > 255) ∨ (*length*(*aire*) > 255);
if *name_too_long* **then** *abort*; { *kpse_find_file* will append the ".tfm", and avoid searching the disk before the font alias files as well. }
pack_file_name(*nom*, *aire*, "");
if ¬*b_open_in*(*tfm_file*) **then** *abort*;
file_opened ← *true*

This code is used in section 588.

590* Note: A malformed `TFM` file might be shorter than it claims to be; thus *eof*(*tfm_file*) might be true when *read_font_info* refers to *tfm_file*↑ or when it says *get*(*tfm_file*). If such circumstances cause system error messages, you will have to defeat them somehow, for example by defining *fget* to be '**begin** *get*(*tfm_file*); **if** *eof*(*tfm_file*) **then** *abort*; **end**'.

define *fget* ≡ *tfm_temp* ← *getc*(*tfm_file*)
define *fbyte* ≡ *tfm_temp*
define *read_sixteen*(#) ≡
begin # ← *fbyte*;
if # > 127 **then** *abort*;
fget; # ← # ∗ ´400 + *fbyte*;
end
define *store_four_quarters*(#) ≡
begin *fget*; *a* ← *fbyte*; *qw.b0* ← *qi*(*a*); *fget*; *b* ← *fbyte*; *qw.b1* ← *qi*(*b*); *fget*; *c* ← *fbyte*;
qw.b2 ← *qi*(*c*); *fget*; *d* ← *fbyte*; *qw.b3* ← *qi*(*d*); # ← *qw*;
end

596* We want to make sure that there is no cycle of characters linked together by *list_tag* entries, since such a cycle would get TEX into an endless loop. If such a cycle exists, the routine here detects it when processing the largest character code in the cycle.

define *check_byte_range*(#) ≡
  begin if $(\# < bc) \lor (\# > ec)$ **then** *abort*
  end
define *current_character_being_worked_on* ≡ $k + bc - fmem\_ptr$

⟨Check for charlist cycle 596*⟩ ≡
begin *check_byte_range*(*d*);
while *d* < *current_character_being_worked_on* **do**
  begin *qw* ← *orig_char_info*(*f*)(*d*); { N.B.: not *qi*(*d*), since *char_base*[*f*] hasn't been adjusted yet }
  if *char_tag*(*qw*) ≠ *list_tag* **then goto** *not_found*;
  d ← *qo*(*rem_byte*(*qw*)); { next character on the list }
  end;
if *d* = *current_character_being_worked_on* **then** *abort*; { yes, there's a cycle }
not_found: **end**

This code is used in section 595.

600* **define** *check_existence*(#) ≡
  begin *check_byte_range*(#); *qw* ← *orig_char_info*(*f*)(#); { N.B.: not *qi*(#) }
  if ¬*char_exists*(*qw*) **then** *abort*;
  end

⟨Read ligature/kern program 600*⟩ ≡
bch_label ← ´77777; *bchar* ← 256;
if *nl* > 0 **then**
  begin for *k* ← *lig_kern_base*[*f*] **to** *kern_base*[*f*] + *kern_base_offset* − 1 **do**
    begin *store_four_quarters*(*font_info*[*k*].*qqqq*);
    if *a* > 128 **then**
      begin if $256 * c + d \ge nl$ **then** *abort*;
      if *a* = 255 **then**
        if *k* = *lig_kern_base*[*f*] **then** *bchar* ← *b*;
      end
    else begin if *b* ≠ *bchar* **then** *check_existence*(*b*);
      if *c* < 128 **then** *check_existence*(*d*) { check ligature }
      else if $256 * (c - 128) + d \ge nk$ **then** *abort*; { check kern }
      if *a* < 128 **then**
        if $k - lig\_kern\_base[f] + a + 1 \ge nl$ **then** *abort*;
      end;
    end;
  if *a* = 255 **then** *bch_label* ← $256 * c + d$;
  end;
for *k* ← *kern_base*[*f*] + *kern_base_offset* **to** *exten_base*[*f*] − 1 **do** *store_scaled*(*font_info*[*k*].*sc*);

This code is used in section 588.

602* We check to see that the TFM file doesn't end prematurely; but no error message is given for files having more than *lf* words.

⟨Read font parameters 602*⟩ ≡
begin for $k \leftarrow 1$ **to** *np* **do**
  if $k = 1$ **then** {the *slant* parameter is a pure number}
    begin *fget*; $sw \leftarrow fbyte$;
    if $sw > 127$ **then** $sw \leftarrow sw - 256$;
    fget; $sw \leftarrow sw * '400 + fbyte$; *fget*; $sw \leftarrow sw * '400 + fbyte$; *fget*;
    $font\_info[param\_base[f]].sc \leftarrow (sw * '20) + (fbyte \ \mathbf{div}\ '20)$;
    end
  else $store\_scaled(font\_info[param\_base[f] + k - 1].sc)$;
if $feof(tfm\_file)$ **then** *abort*;
for $k \leftarrow np + 1$ **to** 7 **do** $font\_info[param\_base[f] + k - 1].sc \leftarrow 0$;
end

This code is used in section 588.

603* Now to wrap it up, we have checked all the necessary things about the TFM file, and all we need to do is put the finishing touches on the data for the new font.

define $adjust(\#) \equiv \#[f] \leftarrow qo(\#[f])$ {correct for the excess *min_quarterword* that was added}

⟨Make final adjustments and **goto** *done* 603*⟩ ≡
if $np \ge 7$ **then** $font\_params[f] \leftarrow np$ **else** $font\_params[f] \leftarrow 7$;
$hyphen\_char[f] \leftarrow default\_hyphen\_char$; $skew\_char[f] \leftarrow default\_skew\_char$;
if $bch\_label < nl$ **then** $bchar\_label[f] \leftarrow bch\_label + lig\_kern\_base[f]$
else $bchar\_label[f] \leftarrow non\_address$;
$font\_bchar[f] \leftarrow qi(bchar)$; $font\_false\_bchar[f] \leftarrow qi(bchar)$;
if $bchar \le ec$ **then**
  if $bchar \ge bc$ **then**
    begin $qw \leftarrow orig\_char\_info(f)(bchar)$; {N.B.: not $qi(bchar)$}
    if $char\_exists(qw)$ **then** $font\_false\_bchar[f] \leftarrow non\_char$;
    end;
$font\_name[f] \leftarrow nom$; $font\_area[f] \leftarrow aire$; $font\_bc[f] \leftarrow bc$; $font\_ec[f] \leftarrow ec$; $font\_glue[f] \leftarrow null$;
$adjust(char\_base)$; $adjust(width\_base)$; $adjust(lig\_kern\_base)$; $adjust(kern\_base)$; $adjust(exten\_base)$;
$decr(param\_base[f])$; $fmem\_ptr \leftarrow fmem\_ptr + lf$; $font\_ptr \leftarrow f$; $g \leftarrow f$; **goto** *done*

This code is used in section 588.

604* Before we forget about the format of these tables, let's deal with two of TEX's basic scanning routines related to font information.

⟨Declare procedures that scan font-related stuff 604*⟩ ≡
function *test_no_ligatures*(*f* : *internal_font_number*): *integer*;
  label *exit*;
  var *c*: *integer*;
  begin *test_no_ligatures* ← 1;
  for *c* ← *font_bc*[*f*] **to** *font_ec*[*f*] **do**
    if *char_exists*(*orig_char_info*(*f*)(*c*)) **then**
      if *odd*(*char_tag*(*orig_char_info*(*f*)(*c*))) **then**
        begin *test_no_ligatures* ← 0; **return**;
        end;
exit: **end**;
function *get_tag_code*(*f* : *internal_font_number*; *c* : *eight_bits*): *integer*;
  var *i*: *small_number*;
  begin if *is_valid_char*(*c*) **then**
    begin *i* ← *char_tag*(*orig_char_info*(*f*)(*c*));
    if *i* = *lig_tag* **then** *get_tag_code* ← 1
    else if *i* = *list_tag* **then** *get_tag_code* ← 2
      else if *i* = *ext_tag* **then** *get_tag_code* ← 4
        else *get_tag_code* ← 0;
    end
  else *get_tag_code* ← −1;
  end;
procedure *scan_font_ident*;
  var *f*: *internal_font_number*; *m*: *halfword*;
  begin ⟨Get the next non-blank non-call token 432⟩;
  if (*cur_cmd* = *def_font*) ∨ (*cur_cmd* = *letterspace_font*) ∨ (*cur_cmd* = *pdf_copy_font*) **then** *f* ← *cur_font*
  else if *cur_cmd* = *set_font* **then** *f* ← *cur_chr*
    else if *cur_cmd* = *def_family* **then**
        begin *m* ← *cur_chr*; *scan_four_bit_int*; *f* ← *equiv*(*m* + *cur_val*);
        end
      else begin *print_err*("Missing␣font␣identifier");
        help2("I␣was␣looking␣for␣a␣control␣sequence␣whose")
        ("current␣meaning␣has␣been␣defined␣by␣\font."); *back_error*; *f* ← *null_font*;
        end;
  cur_val ← *f*;
  end;

See also section 605.

This code is used in section 435.

608* When TEX wants to typeset a character that doesn't exist, the character node is not created; thus the output routine can assume that characters exist when it sees them. The following procedure prints a warning message unless the user has suppressed it.

procedure *char_warning*(*f* : *internal_font_number*; *c* : *eight_bits*);
  var *old_setting*: *integer*; { saved value of *tracing_online* }
  begin if *tracing_lost_chars* > 0 **then**
    begin *old_setting* ← *tracing_online*;
    if *eTeX_ex* ∧ (*tracing_lost_chars* > 1) **then** *tracing_online* ← 1;
    if *tracing_lost_chars* > 2 **then** *print_err*("Missing␣character:␣There␣is␣no␣")
    else begin *begin_diagnostic*; *print_nl*("Missing␣character:␣There␣is␣no␣")
      end;
    print_ASCII(*c*);
    if *tracing_lost_chars* > 2 **then**
      begin *print*("␣("); *print_hex*(*c*); *print*(")");
      end;
    print("␣in␣font␣"); *slow_print*(*font_name*[*f*]);
    if *tracing_lost_chars* < 3 **then** *print_char*("!");
    tracing_online ← *old_setting*;
    if *tracing_lost_chars* > 2 **then**
      begin *help0*; *error*;
      end
    else *end_diagnostic*(*false*);
    end; { of *tracing_lost_chars* > 0 }
  end; { of procedure }

609* Here is a function that returns a pointer to a character node for a given character in a given font. If that character doesn't exist, *null* is returned instead.

This allows a character node to be used if there is an equivalent in the *char_sub_code* list.

function *new_character*(*f* : *internal_font_number*; *c* : *eight_bits*): *pointer*;
  label *exit*;
  var *p*: *pointer*; { newly allocated node }
    ec: *quarterword*; { effective character of *c* }
  begin *ec* ← *effective_char*(*false*, *f*, *qi*(*c*));
  if *font_bc*[*f*] ≤ *qo*(*ec*) **then**
    if *font_ec*[*f*] ≥ *qo*(*ec*) **then**
      if *char_exists*(*orig_char_info*(*f*)(*ec*)) **then** { N.B.: not *char_info* }
        begin *p* ← *get_avail*; *font*(*p*) ← *f*; *character*(*p*) ← *qi*(*c*); *new_character* ← *p*; **return**;
        end;
  char_warning(*f*, *c*); *new_character* ← *null*;
exit: **end**;

619*. Shipping pages out. After considering TEX's eyes and stomach, we come now to the bowels.

The *ship_out* procedure is given a pointer to a box; its mission is to describe that box in DVI form, outputting a "page" to *dvi_file*. The DVI coordinates $(h, v) = (0, 0)$ should correspond to the upper left corner of the box being shipped.

Since boxes can be inside of boxes inside of boxes, the main work of *ship_out* is done by two mutually recursive routines, *hlist_out* and *vlist_out*, which traverse the hlists and vlists inside of horizontal and vertical boxes.

As individual pages are being processed, we need to accumulate information about the entire set of pages, since such statistics must be reported in the postamble. The global variables *total_pages*, *max_v*, *max_h*, *max_push*, and *last_bop* are used to record this information.

The variable *doing_leaders* is *true* while leaders are being output. The variable *dead_cycles* contains the number of times an output routine has been initiated since the last *ship_out*.

A few additional global variables are also defined here for use in *vlist_out* and *hlist_out*. They could have been local variables, but that would waste stack space when boxes are deeply nested, since the values of these variables are not needed during recursive calls.

⟨ Global variables 13 ⟩ +≡
total_pages: *integer*; { the number of pages that have been shipped out }
max_v: *scaled*; { maximum height-plus-depth of pages shipped so far }
max_h: *scaled*; { maximum width of pages shipped so far }
max_push: *integer*; { deepest nesting of *push* commands encountered so far }
last_bop: *integer*; { location of previous *bop* in the DVI output }
dead_cycles: *integer*; { recent outputs that didn't ship anything out }
doing_leaders: *boolean*; { are we inside a leader box? }
{ character and font in current *char_node* }
c: *quarterword*;
f: *internal_font_number*;
rule_ht, *rule_dp*, *rule_wd*: *scaled*; { size of current rule being output }
g: *pointer*; { current glue specification }
lq, *lr*: *integer*; { quantities used in calculations for leaders }

622*. Some systems may find it more efficient to make *dvi_buf* a **packed** array, since output of four bytes at once may be facilitated.

⟨ Global variables 13 ⟩ +≡
dvi_buf: ↑*eight_bits*; { buffer for DVI output }
half_buf: *integer*; { half of *dvi_buf_size* }
dvi_limit: *integer*; { end of the current half buffer }
dvi_ptr: *integer*; { the next available buffer address }
dvi_offset: *integer*; { *dvi_buf_size* times the number of times the output buffer has been fully emptied }
dvi_gone: *integer*; { the number of bytes already output to *dvi_file* }

624*. The actual output of *dvi_buf* $[a \mathinner{\ldotp\ldotp} b]$ to *dvi_file* is performed by calling *write_dvi*(a, b). For best results, this procedure should be optimized to run as fast as possible on each particular system, since it is part of TEX's inner loop. It is safe to assume that a and $b + 1$ will both be multiples of 4 when *write_dvi*(a, b) is called; therefore it is possible on many machines to use efficient methods to pack four bytes per word and to output an array of words with one system call.

In C, we use a macro to call *fwrite* or *write* directly, writing all the bytes in one shot. Much better even than writing four bytes at a time.

625* To put a byte in the buffer without paying the cost of invoking a procedure each time, we use the macro *dvi_out*.

The length of *dvi_file* should not exceed "7FFFFFFF; we set $cur\_s \leftarrow -2$ to prevent further DVI output causing infinite recursion.

define $dvi\_out(\#) \equiv$ **begin** $dvi\_buf[dvi\_ptr] \leftarrow \#$; $incr(dvi\_ptr)$;
  if $dvi\_ptr = dvi\_limit$ **then** dvi_swap;
  end

procedure *dvi_swap*; { outputs half of the buffer }
begin if $dvi\_ptr > ("7FFFFFFF - dvi\_offset)$ **then**
  begin $cur\_s \leftarrow -2$; *fatal_error*("dvi␣length␣exceeds␣""7FFFFFFF");
  end;
if $dvi\_limit = dvi\_buf\_size$ **then**
  begin $write\_dvi(0, half\_buf - 1)$; $dvi\_limit \leftarrow half\_buf$; $dvi\_offset \leftarrow dvi\_offset + dvi\_buf\_size$;
  $dvi\_ptr \leftarrow 0$;
  end
else begin $write\_dvi(half\_buf, dvi\_buf\_size - 1)$; $dvi\_limit \leftarrow dvi\_buf\_size$;
  end;
$dvi\_gone \leftarrow dvi\_gone + half\_buf$;
end;

626* Here is how we clean out the buffer when TEX is all through; *dvi_ptr* will be a multiple of 4.

⟨Empty the last bytes out of *dvi_buf* 626*⟩ ≡
if $dvi\_limit = half\_buf$ **then** $write\_dvi(half\_buf, dvi\_buf\_size - 1)$;
if $dvi\_ptr > ("7FFFFFFF - dvi\_offset)$ **then**
  begin $cur\_s \leftarrow -2$; *fatal_error*("dvi␣length␣exceeds␣""7FFFFFFF");
  end;
if $dvi\_ptr > 0$ **then** $write\_dvi(0, dvi\_ptr - 1)$

This code is used in section 670*.

629* Here's a procedure that outputs a font definition. Since TEX82 uses at most 256 different fonts per job, *fnt_def1* is always used as the command code.

procedure *dvi_font_def*(*f* : *internal_font_number*);
var *k*: *pool_pointer*; { index into *str_pool* }
begin if $f \le 256 + font\_base$ **then**
  begin $dvi\_out(fnt\_def1)$; $dvi\_out(f - font\_base - 1)$;
  end
else begin $dvi\_out(fnt\_def1 + 1)$; $dvi\_out((f - font\_base - 1)$ **div** '400);
  $dvi\_out((f - font\_base - 1)$ **mod** '400);
  end;
$dvi\_out(qo(font\_check[f].b0))$; $dvi\_out(qo(font\_check[f].b1))$; $dvi\_out(qo(font\_check[f].b2))$;
$dvi\_out(qo(font\_check[f].b3))$;
$dvi\_four(font\_size[f])$; $dvi\_four(font\_dsize[f])$;
$dvi\_out(length(font\_area[f]))$; $dvi\_out(length(font\_name[f]))$;
⟨Output the font name whose internal number is *f* 630⟩;
end;

645* ⟨Initialize variables as *ship_out* begins 645*⟩ ≡

```
dvi_h ← 0; dvi_v ← 0; cur_h ← h_offset; dvi_f ← null_font;
⟨Calculate DVI page dimensions and margins 644⟩;
ensure_dvi_open;
if total_pages = 0 then
  begin dvi_out(pre); dvi_out(id_byte); { output the preamble }
  dvi_four(25400000); dvi_four(473628672); { conversion ratio for sp }
  prepare_mag; dvi_four(mag); { magnification factor is frozen }
  if output_comment then
    begin l ← strlen(output_comment); dvi_out(l);
    for s ← 0 to l − 1 do dvi_out(output_comment[s]);
    end
  else begin { the default code is unchanged }
    old_setting ← selector; selector ← new_string; print("␣TeX␣output␣"); print_int(year);
    print_char("."); print_two(month); print_char("."); print_two(day); print_char(":");
    print_two(time div 60); print_two(time mod 60); selector ← old_setting; dvi_out(cur_length);
    for s ← str_start[str_ptr] to pool_ptr − 1 do dvi_out(so(str_pool[s]));
    pool_ptr ← str_start[str_ptr]; { flush the current string }
    end;
  end
```

This code is used in section 668*.

647* The recursive procedures *hlist_out* and *vlist_out* each have local variables *save_h* and *save_v* to hold the values of *dvi_h* and *dvi_v* just before entering a new level of recursion. In effect, the values of *save_h* and *save_v* on TEX's run-time stack correspond to the values of *h* and *v* that a DVI-reading program will push onto its coordinate stack.

define *move_past* = 13 { go to this label when advancing past glue or a rule }
define *fin_rule* = 14 { go to this label to finish processing a rule }
define *next_p* = 15 { go to this label when finished with node *p* }

⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1612* ⟩
procedure *hlist_out*; { output an *hlist_node* box }
label *reswitch*, *move_past*, *fin_rule*, *next_p*, *continue*, *found*;
var *base_line*: *scaled*; { the baseline coordinate for this box }
left_edge: *scaled*; { the left coordinate for this box }
save_h, *save_v*: *scaled*; { what *dvi_h* and *dvi_v* should pop to }
this_box: *pointer*; { pointer to containing box }
g_order: *glue_ord*; { applicable order of infinity for glue }
g_sign: *normal* .. *shrinking*; { selects type of glue }
p: *pointer*; { current position in the hlist }
save_loc: *integer*; { DVI byte location upon entry }
leader_box: *pointer*; { the leader box being replicated }
leader_wd: *scaled*; { width of leader box being replicated }
lx: *scaled*; { extra space between leader boxes }
outer_doing_leaders: *boolean*; { were we doing leaders? }
edge: *scaled*; { right edge of sub-box or leader space }
prev_p: *pointer*; { one step behind *p* }
glue_temp: *real*; { glue value before rounding }
cur_glue: *real*; { glue seen so far }
cur_g: *scaled*; { rounded equivalent of *cur_glue* times the glue ratio }
begin *cur_g* ← 0; *cur_glue* ← *float_constant*(0); *this_box* ← *temp_ptr*; *g_order* ← *glue_order*(*this_box*);
g_sign ← *glue_sign*(*this_box*); *p* ← *list_ptr*(*this_box*); *incr*(*cur_s*);
if *cur_s* > 0 **then** *dvi_out*(*push*);
if *cur_s* > *max_push* **then** *max_push* ← *cur_s*;
save_loc ← *dvi_offset* + *dvi_ptr*; *base_line* ← *cur_v*; *prev_p* ← *this_box* + *list_offset*;
⟨ Initialize *hlist_out* for mixed direction typesetting 1711 ⟩;
left_edge ← *cur_h*; ⟨ Start hlist *SyncTEX* information record 1924* ⟩;
while *p* ≠ *null* **do** ⟨ Output node *p* for *hlist_out* and move to the next node, maintaining the condition *cur_v* = *base_line* 648* ⟩;
⟨ Finish hlist *SyncTEX* information record 1925* ⟩;
⟨ Finish *hlist_out* for mixed direction typesetting 1712 ⟩;
prune_movements(*save_loc*);
if *cur_s* > 0 **then** *dvi_pop*(*save_loc*);
decr(*cur_s*);
end;

648* We ought to give special care to the efficiency of one part of *hlist_out*, since it belongs to TEX's inner loop. When a *char_node* is encountered, we save a little time by processing several nodes in succession until reaching a non-*char_node*. The program uses the fact that $set\_char\_0 = 0$.

In MLTEX this part looks for the existence of a substitution definition for a character c, if c does not exist in the font, and create appropriate `DVI` commands. Former versions of MLTEX have spliced appropriate character, kern, and box nodes into the horizontal list. Because the user can change character substitutions or `\charsubdefmax` on the fly, we have to test a again for valid substitutions. (Additional it is necessary to be careful—if leaders are used the current hlist is normally traversed more than once!)

⟨Output node p for *hlist_out* and move to the next node, maintaining the condition $cur\_v = base\_line$ 648*⟩ ≡
reswitch: **if** *is_char_node*(p) **then**
  begin *synch_h*; *synch_v*;
  repeat $f \leftarrow font(p)$; $c \leftarrow character(p)$;
    if $f \neq dvi\_f$ **then** ⟨Change font *dvi_f* to f 649*⟩;
    if $font\_ec[f] \geq qo(c)$ **then**
      if $font\_bc[f] \leq qo(c)$ **then**
        if *char_exists*(*orig_char_info*(f)(c)) **then** { N.B.: not *char_info* }
          begin if $c \geq qi(128)$ **then** *dvi_out*(*set1*);
          dvi_out(*qo*(c));
          $cur\_h \leftarrow cur\_h + char\_width(f)(orig\_char\_info(f)(c))$; **goto** *continue*;
          end;
    if *mltex_enabled_p* **then** ⟨Output a substitution, **goto** *continue* if not possible 1883*⟩;
  continue: $prev\_p \leftarrow link(prev\_p)$; { N.B.: not $prev\_p \leftarrow p$, p might be *lig_trick* }
    $p \leftarrow link(p)$;
  until $\neg is\_char\_node(p)$;
  ⟨Record current point *SyncTEX* information 1927*⟩;
  $dvi\_h \leftarrow cur\_h$;
  end
else ⟨Output the non-*char_node* p for *hlist_out* and move to the next node 650*⟩

This code is used in section 647*.

649* ⟨Change font *dvi_f* to f 649*⟩ ≡
begin if $\neg font\_used[f]$ **then**
  begin *dvi_font_def*(f); $font\_used[f] \leftarrow true$;
  end;
if $f \leq 64 + font\_base$ **then** $dvi\_out(f - font\_base - 1 + fnt\_num\_0)$
else if $f \leq 256 + font\_base$ **then**
    begin *dvi_out*(*fnt1*); $dvi\_out(f - font\_base - 1)$;
    end
  else begin $dvi\_out(fnt1 + 1)$; $dvi\_out((f - font\_base - 1)$ **div** ´400);
    $dvi\_out((f - font\_base - 1)$ **mod** ´400);
    end;
$dvi\_f \leftarrow f$;
end

This code is used in section 648*.

650*. ⟨ Output the non-*char_node* p for *hlist_out* and move to the next node 650* ⟩ ≡
begin case *type*(p) **of**
hlist_node, *vlist_node*: ⟨ Output a box in an hlist 651* ⟩;
rule_node: **begin** *rule_ht* ← *height*(p); *rule_dp* ← *depth*(p); *rule_wd* ← *width*(p); **goto** *fin_rule*;
end;
whatsit_node: ⟨ Output the whatsit node p in an hlist 1611 ⟩;
glue_node: ⟨ Move right or output leaders 653 ⟩;
margin_kern_node: *cur_h* ← *cur_h* + *width*(p); { separate the *margin_kern_node* case in *hlist_out* }
kern_node: **begin** ⟨ Record *kern_node* *SyncTEX* information 1929* ⟩;
cur_h ← *cur_h* + *width*(p);
end;
math_node: **begin** ⟨ Record *math_node* *SyncTEX* information 1930* ⟩;
⟨ Handle a math node in *hlist_out* 1713 ⟩;
end;
ligature_node: ⟨ Make node p look like a *char_node* and **goto** *reswitch* 824 ⟩;
⟨ Cases of *hlist_out* that arise in mixed direction text only 1717 ⟩
othercases *do_nothing*
endcases;
goto *next_p*;
fin_rule: ⟨ Output a rule in an hlist 652 ⟩;
move_past: **begin** *cur_h* ← *cur_h* + *rule_wd*;
⟨ Record horizontal *rule_node* or *glue_node* *SyncTEX* information 1928* ⟩;
end;
next_p: *prev_p* ← p; p ← *link*(p);
end

This code is used in section 648*.

651*. ⟨ Output a box in an hlist 651* ⟩ ≡
if *list_ptr*(p) = *null* **then**
begin ⟨ Record void list *SyncTEX* information 1926* ⟩;
cur_h ← *cur_h* + *width*(p);
end
else begin *save_h* ← *dvi_h*; *save_v* ← *dvi_v*; *cur_v* ← *base_line* + *shift_amount*(p);
{ shift the box down }
temp_ptr ← p; *edge* ← *cur_h* + *width*(p);
if *cur_dir* = *right_to_left* **then** *cur_h* ← *edge*;
if *type*(p) = *vlist_node* **then** *vlist_out* **else** *hlist_out*;
dvi_h ← *save_h*; *dvi_v* ← *save_v*; *cur_h* ← *edge*; *cur_v* ← *base_line*;
end

This code is used in section 650*.

657* The *vlist_out* routine is similar to *hlist_out*, but a bit simpler.

procedure *vlist_out*; { output a *vlist_node* box }
label *move_past*, *fin_rule*, *next_p*;
var *left_edge*: *scaled*; { the left coordinate for this box }
top_edge: *scaled*; { the top coordinate for this box }
save_h, *save_v*: *scaled*; { what *dvi_h* and *dvi_v* should pop to }
this_box: *pointer*; { pointer to containing box }
g_order: *glue_ord*; { applicable order of infinity for glue }
g_sign: *normal* .. *shrinking*; { selects type of glue }
p: *pointer*; { current position in the vlist }
save_loc: *integer*; { DVI byte location upon entry }
leader_box: *pointer*; { the leader box being replicated }
leader_ht: *scaled*; { height of leader box being replicated }
lx: *scaled*; { extra space between leader boxes }
outer_doing_leaders: *boolean*; { were we doing leaders? }
edge: *scaled*; { bottom boundary of leader space }
glue_temp: *real*; { glue value before rounding }
cur_glue: *real*; { glue seen so far }
cur_g: *scaled*; { rounded equivalent of *cur_glue* times the glue ratio }
begin $cur\_g \leftarrow 0$; $cur\_glue \leftarrow float\_constant(0)$; $this\_box \leftarrow temp\_ptr$; $g\_order \leftarrow glue\_order(this\_box)$;
$g\_sign \leftarrow glue\_sign(this\_box)$; $p \leftarrow list\_ptr(this\_box)$; *incr*(*cur_s*);
if $cur\_s > 0$ **then** *dvi_out*(*push*);
if $cur\_s > max\_push$ **then** $max\_push \leftarrow cur\_s$;
$save\_loc \leftarrow dvi\_offset + dvi\_ptr$; $left\_edge \leftarrow cur\_h$; ⟨Start vlist *SyncTEX* information record 1922*⟩;
$cur\_v \leftarrow cur\_v - height(this\_box)$; $top\_edge \leftarrow cur\_v$;
while $p \neq null$ **do** ⟨Output node *p* for *vlist_out* and move to the next node, maintaining the condition $cur\_h = left\_edge$ 658⟩;
⟨Finish vlist *SyncTEX* information record 1923*⟩;
prune_movements(*save_loc*);
if $cur\_s > 0$ **then** *dvi_pop*(*save_loc*);
decr(*cur_s*);
end;

660* The *synch_v* here allows the DVI output to use one-byte commands for adjusting *v* in most cases, since the baselineskip distance will usually be constant.

⟨Output a box in a vlist 660*⟩ ≡
if $list\_ptr(p) = null$ **then**
begin $cur\_v \leftarrow cur\_v + height(p)$; ⟨Record void list *SyncTEX* information 1926*⟩;
$cur\_v \leftarrow cur\_v + depth(p)$;
end
else begin $cur\_v \leftarrow cur\_v + height(p)$; *synch_v*; $save\_h \leftarrow dvi\_h$; $save\_v \leftarrow dvi\_v$;
if $cur\_dir = right\_to\_left$ **then** $cur\_h \leftarrow left\_edge - shift\_amount(p)$
else $cur\_h \leftarrow left\_edge + shift\_amount(p)$; { shift the box right }
$temp\_ptr \leftarrow p$;
if $type(p) = vlist\_node$ **then** *vlist_out* **else** *hlist_out*;
$dvi\_h \leftarrow save\_h$; $dvi\_v \leftarrow save\_v$; $cur\_v \leftarrow save\_v + depth(p)$; $cur\_h \leftarrow left\_edge$;
end

This code is used in section 659.

666* The *hlist_out* and *vlist_out* procedures are now complete, so we are ready for the *dvi_ship_out* routine that gets them started in the first place.

```
procedure dvi_ship_out(p : pointer); { output the box p }
  label done;
  var page_loc: integer; { location of the current bop }
    j, k: 0 .. 9; { indices to first ten count registers }
    s: pool_pointer; { index into str_pool }
    old_setting: 0 .. max_selector; { saved selector setting }
  begin ⟨Start sheet SyncTEX information record 1920*⟩;
  begin if tracing_output > 0 then
    begin print_nl(""); print_ln; print("Completed␣box␣being␣shipped␣out");
    end;
  if term_offset > max_print_line − 9 then print_ln
  else if (term_offset > 0) ∨ (file_offset > 0) then print_char("␣");
  print_char("["); j ← 9;
  while (count(j) = 0) ∧ (j > 0) do decr(j);
  for k ← 0 to j do
    begin print_int(count(k));
    if k < j then print_char(".");
    end;
  update_terminal;
  if tracing_output > 0 then
    begin print_char("]"); begin_diagnostic; show_box(p); end_diagnostic(true);
    end;
  ⟨Ship box p out 668*⟩;
  if eTeX_ex then ⟨Check for LR anomalies at the end of ship_out 1727⟩;
  if tracing_output ≤ 0 then print_char("]");
  dead_cycles ← 0; update_terminal; { progress report }
  ⟨Flush the box from memory, showing statistics if requested 667⟩;
  end; ⟨Finish sheet SyncTEX information record 1921*⟩;
  end;
```

668* ⟨ Ship box p out 668* ⟩ ≡
⟨ Update the values of *max_h* and *max_v*; but if the page is too large, **goto** *done* 669 ⟩;
⟨ Initialize variables as *ship_out* begins 645* ⟩;
page_loc ← *dvi_offset* + *dvi_ptr*; *dvi_out*(*bop*);
for $k \leftarrow 0$ **to** 9 **do** *dvi_four*(*count*(k));
dvi_four(*last_bop*); *last_bop* ← *page_loc*; *cur_v* ← *height*(p) + *v_offset*; *temp_ptr* ← p;
if *type*(p) = *vlist_node* **then** *vlist_out* **else** *hlist_out*;
dvi_out(*eop*); *incr*(*total_pages*); *cur_s* ← −1; *ifdef*(´IPC´)
  if *ipc_on* > 0 **then**
    begin if *dvi_limit* = *half_buf* **then**
      begin *write_dvi*(*half_buf*, *dvi_buf_size* − 1); *flush_dvi*; *dvi_gone* ← *dvi_gone* + *half_buf*;
      end;
    if *dvi_ptr* > (˝7FFFFFFF − *dvi_offset*) **then**
      begin *cur_s* ← −2; *fatal_error*("dvi␣length␣exceeds␣""7FFFFFFF");
      end;
    if *dvi_ptr* > 0 **then**
      begin *write_dvi*(0, *dvi_ptr* − 1); *flush_dvi*; *dvi_offset* ← *dvi_offset* + *dvi_ptr*;
      dvi_gone ← *dvi_gone* + *dvi_ptr*;
      end;
    dvi_ptr ← 0; *dvi_limit* ← *dvi_buf_size*; *ipc_page*(*dvi_gone*);
    end;
endif(´IPC´);
done:

This code is used in section 666*.

670* At the end of the program, we must finish things off by writing the postamble. If *total_pages* = 0, the DVI file was never opened. If *total_pages* ≥ 65536, the DVI file will lie. And if *max_push* ≥ 65536, the user deserves whatever chaos might ensue.

An integer variable *k* will be declared for use by this routine.

```
⟨Finish the DVI file 670*⟩ ≡
  while cur_s > −1 do
    begin if cur_s > 0 then dvi_out(pop)
    else begin dvi_out(eop); incr(total_pages);
      end;
    decr(cur_s);
    end;
  if total_pages = 0 then print_nl("No␣pages␣of␣output.")
  else if cur_s ≠ −2 then
      begin dvi_out(post); { beginning of the postamble }
      dvi_four(last_bop); last_bop ← dvi_offset + dvi_ptr − 5; { post location }
      dvi_four(25400000); dvi_four(473628672); { conversion ratio for sp }
      prepare_mag; dvi_four(mag); { magnification factor }
      dvi_four(max_v); dvi_four(max_h);
      dvi_out(max_push div 256); dvi_out(max_push mod 256);
      dvi_out((total_pages div 256) mod 256); dvi_out(total_pages mod 256);
      ⟨Output the font definitions for all fonts that were used 671⟩;
      dvi_out(post_post); dvi_four(last_bop); dvi_out(id_byte);
      ifdef(´IPC´)k ← 7 − ((3 + dvi_offset + dvi_ptr) mod 4); { the number of 223's }
      endif(´IPC´)ifndef(´IPC´)k ← 4 + ((dvi_buf_size − dvi_ptr) mod 4); { the number of 223's }
      endifn(´IPC´)
        while k > 0 do
          begin dvi_out(223); decr(k);
          end;
      ⟨Empty the last bytes out of dvi_buf 626*⟩;
      print_nl("Output␣written␣on␣"); print_file_name(0, output_file_name, 0); print("␣(");
      print_int(total_pages);
      if total_pages ≠ 1 then print("␣pages")
      else print("␣page");
      print(",␣"); print_int(dvi_offset + dvi_ptr); print("␣bytes)."); b_close(dvi_file);
      end
```

This code is used in section 1511*.

673* The subroutines define the corresponding macros so we can use them in C.

define *flushable*(#) $\equiv$ (# = *str_ptr* − 1)
define *is_valid_char*(#) $\equiv$ ((*font_bc*[*f*] $\le$ #) $\land$ (# $\le$ *font_ec*[*f*]) $\land$ *char_exists*(*orig_char_info*(*f*)(#)))

function *get_pdf_compress_level*: *integer*;
begin *get_pdf_compress_level* ← *pdf_compress_level*;
end;
function *get_pdf_suppress_warning_dup_map*: *integer*;
begin *get_pdf_suppress_warning_dup_map* ← *pdf_suppress_warning_dup_map*;
end;
function *get_pdf_suppress_warning_page_group*: *integer*;
begin *get_pdf_suppress_warning_page_group* ← *pdf_suppress_warning_page_group*;
end;
function *get_pdf_suppress_ptex_info*: *integer*;
begin *get_pdf_suppress_ptex_info* ← *pdf_suppress_ptex_info*;
end;
function *get_pdf_omit_charset*: *integer*;
begin *get_pdf_omit_charset* ← *pdf_omit_charset*;
end;
function *get_nullfont*: *internal_font_number*;
begin *get_nullfont* ← *null_font*;
end;
function *get_fontbase*: *internal_font_number*;
begin *get_fontbase* ← *font_base*;
end;
function *get_nullcs*: *pointer*;
begin *get_nullcs* ← *null_cs*;
end;
function *get_nullptr*: *pointer*;
begin *get_nullptr* ← *null*;
end;
function *get_tex_int*(*code* : *integer*): *integer*;
begin *get_tex_int* ← *int_par*(*code*);
end;
function *get_tex_dimen*(*code* : *integer*): *scaled*;
begin *get_tex_dimen* ← *dimen_par*(*code*);
end;
function *get_x_height*(*f* : *internal_font_number*): *scaled*;
begin *get_x_height* ← *x_height*(*f*);
end;
function *get_charwidth*(*f* : *internal_font_number*; *c* : *eight_bits*): *scaled*;
begin if *is_valid_char*(*c*) **then** *get_charwidth* ← *char_width*(*f*)(*orig_char_info*(*f*)(*c*))
else *get_charwidth* ← 0;
end;
function *get_charheight*(*f* : *internal_font_number*; *c* : *eight_bits*): *scaled*;
begin if *is_valid_char*(*c*) **then** *get_charheight* ← *char_height*(*f*)(*height_depth*(*orig_char_info*(*f*)(*c*)))
else *get_charheight* ← 0;
end;
function *get_chardepth*(*f* : *internal_font_number*; *c* : *eight_bits*): *scaled*;
begin if *is_valid_char*(*c*) **then** *get_chardepth* ← *char_depth*(*f*)(*height_depth*(*orig_char_info*(*f*)(*c*)))
else *get_chardepth* ← 0;
end;
function *get_quad*(*f* : *internal_font_number*): *scaled*;

begin *get_quad* ← *quad*(*f*);
end;
function *get_slant*(*f* : *internal_font_number*): *scaled*;
begin *get_slant* ← *slant*(*f*);
end;
function *new_dummy_font*: *internal_font_number*;
begin *new_dummy_font* ← *read_font_info*(*null_cs*, "dummy", "", −1000);
end;

729* The implementation of procedure *pdf_hlist_out* is similar to *hlist_out*.

⟨Declare procedures needed in *pdf_hlist_out*, *pdf_vlist_out* 727⟩
procedure *pdf_hlist_out*; { output an *hlist_node* box }
  label *reswitch*, *move_past*, *fin_rule*, *next_p*, *found*, *continue*;
  var *base_line*: *scaled*; { the baseline coordinate for this box }
    left_edge: *scaled*; { the left coordinate for this box }
    save_h: *scaled*; { what *cur_h* should pop to }
    this_box: *pointer*; { pointer to containing box }
    g_order: *glue_ord*; { applicable order of infinity for glue }
    g_sign: *normal* .. *shrinking*; { selects type of glue }
    p: *pointer*; { current position in the hlist }
    leader_box: *pointer*; { the leader box being replicated }
    leader_wd: *scaled*; { width of leader box being replicated }
    lx: *scaled*; { extra space between leader boxes }
    outer_doing_leaders: *boolean*; { were we doing leaders? }
    edge: *scaled*; { right edge of sub-box or leader space }
    prev_p: *pointer*; { one step behind *p* }
    glue_temp: *real*; { glue value before rounding }
    cur_glue: *real*; { glue seen so far }
    cur_g: *scaled*; { rounded equivalent of *cur_glue* times the glue ratio }
    i: *small_number*; { index to scan *pdf_link_stack* }
  begin *cur_g* ← 0; *cur_glue* ← *float_constant*(0); *this_box* ← *temp_ptr*; *g_order* ← *glue_order*(*this_box*);
  g_sign ← *glue_sign*(*this_box*); *p* ← *list_ptr*(*this_box*); *incr*(*cur_s*); *base_line* ← *cur_v*;
  prev_p ← *this_box* + *list_offset*; ⟨Initialize *hlist_out* for mixed direction typesetting 1711⟩;
  left_edge ← *cur_h*; ⟨Create link annotations for the current hbox if needed 730⟩;
  ⟨Start hlist *SyncTEX* information record 1924*⟩;
  while *p* ≠ *null* **do** ⟨Output node *p* for *pdf_hlist_out* and move to the next node, maintaining the condition *cur_v* = *base_line* 731*⟩;
  ⟨Finish hlist *SyncTEX* information record 1925*⟩;
  ⟨Finish *hlist_out* for mixed direction typesetting 1712⟩;
  decr(*cur_s*);
  end;

731* ⟨Output node *p* for *pdf_hlist_out* and move to the next node, maintaining the condition *cur_v* = *base_line* 731*⟩ ≡
reswitch: **if** *is_char_node*(*p*) **then**
    begin repeat *f* ← *font*(*p*); *c* ← *character*(*p*);
      if *is_valid_char*(*c*) **then**
        begin *output_one_char*(*c*); *cur_h* ← *cur_h* + *char_width*(*f*)(*char_info*(*f*)(*c*)); **goto** *continue*;
        end;
      if *mltex_enabled_p* **then** ⟨(pdfTEX) Output a substitution, **goto** *continue* if not possible 1884*⟩;
    continue: *prev_p* ← *link*(*prev_p*); { N.B.: not *prev_p* ← *p*, *p* might be *lig_trick* }
      p ← *link*(*p*);
    until ¬*is_char_node*(*p*);
    ⟨Record current point *SyncTEX* information 1927*⟩;
    end
  else ⟨Output the non-*char_node* *p* for *pdf_hlist_out* and move to the next node 732*⟩

This code is used in section 729*.

732* ⟨ Output the non-*char_node* p for *pdf_hlist_out* and move to the next node 732* ⟩ ≡
begin case *type*(p) **of**
hlist_node, *vlist_node*: ⟨ (pdfTEX) Output a box in an hlist 733* ⟩;
rule_node: **begin** *rule_ht* ← *height*(p); *rule_dp* ← *depth*(p); *rule_wd* ← *width*(p); **goto** *fin_rule*;
end;
whatsit_node: ⟨ Output the whatsit node p in *pdf_hlist_out* 1642 ⟩;
glue_node: ⟨ (pdfTEX) Move right or output leaders 735 ⟩;
margin_kern_node: *cur_h* ← *cur_h* + *width*(p);
kern_node: **begin** ⟨ Record *kern_node* *SyncTEX* information 1929* ⟩;
cur_h ← *cur_h* + *width*(p);
end;
math_node: **begin** ⟨ Record *math_node* *SyncTEX* information 1930* ⟩;
⟨ Handle a math node in *hlist_out* 1713 ⟩;
end;
ligature_node: ⟨ Make node p look like a *char_node* and **goto** *reswitch* 824 ⟩;
⟨ Cases of *hlist_out* that arise in mixed direction text only 1717 ⟩
othercases *do_nothing*
endcases;
goto *next_p*;
fin_rule: ⟨ (pdfTEX) Output a rule in an hlist 734 ⟩;
move_past: **begin** *cur_h* ← *cur_h* + *rule_wd*;
⟨ Record horizontal *rule_node* or *glue_node* *SyncTEX* information 1928* ⟩;
end;
next_p: *prev_p* ← p; p ← *link*(p);
end

This code is used in section 731*.

733* ⟨ (pdfTEX) Output a box in an hlist 733* ⟩ ≡
if *list_ptr*(p) = *null* **then**
begin ⟨ Record void list *SyncTEX* information 1926* ⟩;
cur_h ← *cur_h* + *width*(p);
end
else begin *cur_v* ← *base_line* + *shift_amount*(p); { shift the box down }
temp_ptr ← p; *edge* ← *cur_h* + *width*(p);
if *cur_dir* = *right_to_left* **then** *cur_h* ← *edge*;
if *type*(p) = *vlist_node* **then** *pdf_vlist_out* **else** *pdf_hlist_out*;
cur_h ← *edge*; *cur_v* ← *base_line*;
end

This code is used in section 732*.

738* The *pdf_vlist_out* routine is similar to *pdf_hlist_out*, but a bit simpler.

procedure *pdf_vlist_out*; { output a *pdf_vlist_node* box }
  label *move_past*, *fin_rule*, *next_p*;
  var *left_edge*: *scaled*; { the left coordinate for this box }
    top_edge: *scaled*; { the top coordinate for this box }
    save_v: *scaled*; { what *cur_v* should pop to }
    this_box: *pointer*; { pointer to containing box }
    g_order: *glue_ord*; { applicable order of infinity for glue }
    g_sign: *normal* .. *shrinking*; { selects type of glue }
    p: *pointer*; { current position in the vlist }
    leader_box: *pointer*; { the leader box being replicated }
    leader_ht: *scaled*; { height of leader box being replicated }
    lx: *scaled*; { extra space between leader boxes }
    outer_doing_leaders: *boolean*; { were we doing leaders? }
    edge: *scaled*; { bottom boundary of leader space }
    glue_temp: *real*; { glue value before rounding }
    cur_glue: *real*; { glue seen so far }
    cur_g: *scaled*; { rounded equivalent of *cur_glue* times the glue ratio }
  begin *cur_g* ← 0; *cur_glue* ← *float_constant*(0); *this_box* ← *temp_ptr*; *g_order* ← *glue_order*(*this_box*);
  g_sign ← *glue_sign*(*this_box*); *p* ← *list_ptr*(*this_box*); *incr*(*cur_s*); *left_edge* ← *cur_h*;
  ⟨ Start vlist *SyncTEX* information record 1922* ⟩;
  cur_v ← *cur_v* − *height*(*this_box*); *top_edge* ← *cur_v*; ⟨ Create thread for the current vbox if needed 739 ⟩;
  while *p* ≠ *null* **do** ⟨ Output node *p* for *pdf_vlist_out* and move to the next node, maintaining the condition *cur_h* = *left_edge* 740 ⟩;
  ⟨ Finish vlist *SyncTEX* information record 1923* ⟩;
  decr(*cur_s*);
  end;

742* ⟨ (pdfTEX) Output a box in a vlist 742* ⟩ ≡
  if *list_ptr*(*p*) = *null* **then**
    begin *cur_v* ← *cur_v* + *height*(*p*); ⟨ Record void list *SyncTEX* information 1926* ⟩;
    cur_v ← *cur_v* + *depth*(*p*);
    end
  else begin *cur_v* ← *cur_v* + *height*(*p*); *save_v* ← *cur_v*;
    if *cur_dir* = *right_to_left* **then** *cur_h* ← *left_edge* − *shift_amount*(*p*)
    else *cur_h* ← *left_edge* + *shift_amount*(*p*); { shift the box right }
    temp_ptr ← *p*;
    if *type*(*p*) = *vlist_node* **then** *pdf_vlist_out* **else** *pdf_hlist_out*;
    cur_v ← *save_v* + *depth*(*p*); *cur_h* ← *left_edge*;
    end

This code is used in section 741.

750* *pdf_ship_out* is used instead of *ship_out* to shipout a box to PDF output. If *shipping_page* is not set then the output will be a Form object, otherwise it will be a Page object.

```
procedure pdf_ship_out(p : pointer; shipping_page : boolean);   { output the box p }
  label done, done1;
  var i, j, k: integer;   { general purpose accumulators }
    s: pool_pointer;   { index into str_pool }
    mediabox_given: boolean; save_font_list: pointer;
        { to save pdf_font_list during flushing pending forms }
    save_obj_list: pointer;   { to save pdf_obj_list }
    save_ximage_list: pointer;   { to save pdf_ximage_list }
    save_xform_list: pointer;   { to save pdf_xform_list }
    save_image_procset: integer;   { to save pdf_image_procset }
    save_text_procset: integer;   { to save pdf_text_procset }
    pdf_last_resources: integer;   { pointer to most recently generated Resources object }
  begin if tracing_output > 0 then
    begin print_nl(""); print_ln; print("Completed␣box␣being␣shipped␣out");
    end;
  if ¬init_pdf_output then
    begin ⟨Initialize variables for PDF output 792⟩;
    init_pdf_output ← true;
    end;
  is_shipping_page ← shipping_page;
  if shipping_page then
    begin if term_offset > max_print_line − 9 then print_ln
    else if (term_offset > 0) ∨ (file_offset > 0) then print_char("␣");
    print_char("["); j ← 9;
    while (count(j) = 0) ∧ (j > 0) do decr(j);
    for k ← 0 to j do
      begin print_int(count(k));
      if k < j then print_char(".");
      end;
    update_terminal;
    end;
  if tracing_output > 0 then
    begin if shipping_page then print_char("]");
    begin_diagnostic; show_box(p); end_diagnostic(true);
    end;
  pdf_output_value ← pdf_output;   { SyncTEX: we assume that pdf_output is properly set up }
  if shipping_page then synctex_sheet(mag) else synctex_pdfxform(p);
  ⟨(pdfTEX) Ship box p out 751⟩;
  if shipping_page then synctex_teehs else synctex_mrofxfdp;
  if eTeX_ex then ⟨Check for LR anomalies at the end of ship_out 1727⟩;
  if (tracing_output ≤ 0) ∧ shipping_page then print_char("]");
  dead_cycles ← 0; update_terminal;   { progress report }
  ⟨Flush the box from memory, showing statistics if requested 667⟩;
  end;
```

882* ⟨Look at the list of characters starting with x in font g; set f and c whenever a better character is found; **goto** *found* as soon as a large enough variant is encountered 882*⟩ ≡

begin $y \leftarrow x$;
if $(qo(y) \geq font\_bc[g]) \land (qo(y) \leq font\_ec[g])$ **then**
  begin *continue*: $q \leftarrow orig\_char\_info(g)(y)$;
  if $char\_exists(q)$ **then**
    begin if $char\_tag(q) = ext\_tag$ **then**
      begin $f \leftarrow g$; $c \leftarrow y$; **goto** *found*;
      end;
    $hd \leftarrow height\_depth(q)$; $u \leftarrow char\_height(g)(hd) + char\_depth(g)(hd)$;
    if $u > w$ **then**
      begin $f \leftarrow g$; $c \leftarrow y$; $w \leftarrow u$;
      if $u \geq v$ **then goto** *found*;
      end;
    if $char\_tag(q) = list\_tag$ **then**
      begin $y \leftarrow rem\_byte(q)$; **goto** *continue*;
      end;
    end;
  end;
end

This code is used in section 881.

895* Here we save memory space in a common case.

⟨Simplify a trivial box 895*⟩ ≡
  $q \leftarrow list\_ptr(x)$;
  if $is\_char\_node(q)$ **then**
    begin $r \leftarrow link(q)$;
    if $r \neq null$ **then**
      if $link(r) = null$ **then**
        if $\neg is\_char\_node(r)$ **then**
          if $type(r) = kern\_node$ **then** { unneeded italic correction }
            begin $free\_node(r, medium\_node\_size)$; $link(q) \leftarrow null$;
            end;
    end

This code is used in section 894.

896* It is convenient to have a procedure that converts a *math_char* field to an "unpacked" form. The *fetch* routine sets *cur_f*, *cur_c*, and *cur_i* to the font code, character code, and character information bytes of a given noad field. It also takes care of issuing error messages for nonexistent characters; in such cases, *char_exists*(*cur_i*) will be *false* after *fetch* has acted, and the field will also have been reset to *empty*.

procedure $fetch(a : pointer)$; { unpack the *math_char* field a }
  begin $cur\_c \leftarrow character(a)$; $cur\_f \leftarrow fam\_fnt(fam(a) + cur\_size)$;
  if $cur\_f = null\_font$ **then** ⟨Complain about an undefined family and set *cur_i* null 897⟩
  else begin if $(qo(cur\_c) \geq font\_bc[cur\_f]) \land (qo(cur\_c) \leq font\_ec[cur\_f])$ **then**
      $cur\_i \leftarrow orig\_char\_info(cur\_f)(cur\_c)$
    else $cur\_i \leftarrow null\_character$;
    if $\neg(char\_exists(cur\_i))$ **then**
      begin $char\_warning(cur\_f, qo(cur\_c))$; $math\_type(a) \leftarrow empty$; $cur\_i \leftarrow null\_character$;
      end;
    end;
  end;

914* ⟨Switch to a larger accent if available and appropriate 914*⟩ ≡
  loop begin if $char\_tag(i) \neq list\_tag$ **then goto** *done*;
    $y \leftarrow rem\_byte(i)$; $i \leftarrow orig\_char\_info(f)(y)$;
    if $\neg char\_exists(i)$ **then goto** *done*;
    if $char\_width(f)(i) > w$ **then goto** *done*;
    $c \leftarrow y$;
    end;
done:

This code is used in section 912.

923* If the nucleus of an *op_noad* is a single character, it is to be centered vertically with respect to the axis, after first being enlarged (via a character list in the font) if we are in display style. The normal convention for placing displayed limits is to put them above and below the operator in display style.

The italic correction is removed from the character if there is a subscript and the limits are not being displayed. The *make_op* routine returns the value that should be used as an offset between subscript and superscript.

After *make_op* has acted, *subtype*(q) will be *limits* if and only if the limits have been set above and below the operator. In that case, *new_hlist*(q) will already contain the desired final box.

⟨Declare math construction procedures 908⟩ +≡
function *make_op*(q : *pointer*): *scaled*;
  var *delta*: *scaled*; { offset between subscript and superscript }
    p, v, x, y, z: *pointer*; { temporary registers for box construction }
    c: *quarterword*; i: *four_quarters*; { registers for character examination }
    shift_up, *shift_down*: *scaled*; { dimensions for box calculation }
  begin if (*subtype*(q) = *normal*) ∧ (*cur_style* < *text_style*) **then** *subtype*(q) ← *limits*;
  if *math_type*(*nucleus*(q)) = *math_char* **then**
    begin *fetch*(*nucleus*(q));
    if (*cur_style* < *text_style*) ∧ (*char_tag*(*cur_i*) = *list_tag*) **then** { make it larger }
      begin c ← *rem_byte*(*cur_i*); i ← *orig_char_info*(*cur_f*)(c);
      if *char_exists*(i) **then**
        begin *cur_c* ← c; *cur_i* ← i; *character*(*nucleus*(q)) ← c;
        end;
      end;
    delta ← *char_italic*(*cur_f*)(*cur_i*); x ← *clean_box*(*nucleus*(q), *cur_style*);
    if (*math_type*(*subscr*(q)) ≠ *empty*) ∧ (*subtype*(q) ≠ *limits*) **then** *width*(x) ← *width*(x) − *delta*;
        { remove italic correction }
    shift_amount(x) ← *half*(*height*(x) − *depth*(x)) − *axis_height*(*cur_size*); { center vertically }
    math_type(*nucleus*(q)) ← *sub_box*; *info*(*nucleus*(q)) ← x;
    end
  else *delta* ← 0;
  if *subtype*(q) = *limits* **then** ⟨Construct a box with limits above and below it, skewed by *delta* 924⟩;
  make_op ← *delta*;
  end;

1085* **define** $wrap\_lig(\#) \equiv$
if *ligature_present* **then**
begin $p \leftarrow new\_ligature(hf, cur\_l, link(cur\_q))$;
if *lft_hit* **then**
begin $subtype(p) \leftarrow 2$; $lft\_hit \leftarrow false$;
end;
if # **then**
if $lig\_stack = null$ **then**
begin $incr(subtype(p))$; $rt\_hit \leftarrow false$;
end;
$link(cur\_q) \leftarrow p$; $t \leftarrow p$; $ligature\_present \leftarrow false$;
end
define $pop\_lig\_stack \equiv$
begin if $lig\_ptr(lig\_stack) > null$ **then**
begin $link(t) \leftarrow lig\_ptr(lig\_stack)$; { this is a charnode for $hu[j+1]$ }
$t \leftarrow link(t)$; $incr(j)$;
end;
$p \leftarrow lig\_stack$; $lig\_stack \leftarrow link(p)$; $free\_node(p, small\_node\_size)$;
if $lig\_stack = null$ **then** *set_cur_r* **else** $cur\_r \leftarrow character(lig\_stack)$;
end { if *lig_stack* isn't *null* we have $cur\_rh = non\_char$ }

⟨ Append a ligature and/or kern to the translation; **goto** *continue* if the stack of inserted ligatures is nonempty 1085* ⟩ ≡
$wrap\_lig(rt\_hit)$;
if $w \neq 0$ **then**
begin $link(t) \leftarrow new\_kern(w)$; $t \leftarrow link(t)$; $w \leftarrow 0$; $sync\_tag(t + medium\_node\_size) \leftarrow 0$;
{ *SyncTEX*: do nothing, it is too late }
end;
if $lig\_stack > null$ **then**
begin $cur\_q \leftarrow t$; $cur\_l \leftarrow character(lig\_stack)$; $ligature\_present \leftarrow true$; *pop_lig_stack*;
goto *continue*;
end

This code is used in section 1081.

1095*. The patterns are stored in a compact table that is also efficient for retrieval, using a variant of "trie memory" [cf. *The Art of Computer Programming* **3** (1973), 481–505]. We can find each pattern $p_1 \ldots p_k$ by letting z_0 be one greater than the relevant language index and then, for $1 \le i \le k$, setting $z_i \leftarrow trie\_link(z_{i-1}) + p_i$; the pattern will be identified by the number z_k. Since all the pattern information is packed together into a single *trie_link* array, it is necessary to prevent confusion between the data from inequivalent patterns, so another table is provided such that $trie\_char(z_i) = p_i$ for all i. There is also a table $trie\_op(z_k)$ to identify the numbers $n_0 \ldots n_k$ associated with $p_1 \ldots p_k$.

The theory that comparatively few different number sequences $n_0 \ldots n_k$ actually occur, since most of the n's are generally zero, seems to fail at least for the large German hyphenation patterns. Therefore the number sequences cannot any longer be encoded in such a way that $trie\_op(z_k)$ is only one byte long. We have introduced a new constant *max_trie_op* for the maximum allowable hyphenation operation code value; *max_trie_op* might be different for TEX and `INITEX` and must not exceed *max_halfword*. An opcode will occupy a halfword if *max_trie_op* exceeds *max_quarterword* or a quarterword otherwise. If $trie\_op(z_k) \ne min\_trie\_op$, when $p_1 \ldots p_k$ has matched the letters in $hc[(l-k+1) \mathrel{..} l]$ of language t, we perform all of the required operations for this pattern by carrying out the following little program: Set $v \leftarrow trie\_op(z_k)$. Then set $v \leftarrow v + op\_start[t]$, $hyf[l - hyf\_distance[v]] \leftarrow \max(hyf[l - hyf\_distance[v]], hyf\_num[v])$, and $v \leftarrow hyf\_next[v]$; repeat, if necessary, until $v = min\_trie\_op$.

⟨ Types in the outer block 18 ⟩ +≡
trie_pointer = 0 .. *ssup_trie_size*; { an index into *trie* }
trie_opcode = 0 .. *ssup_trie_opcode*; { a trie opcode }

1096*. For more than 255 trie op codes, the three fields *trie_link*, *trie_char*, and *trie_op* will no longer fit into one memory word; thus using web2c we define *trie* as three array instead of an array of records. The variant will be implemented by reusing the opcode field later on with another macro.

define *trie_link*(#) ≡ *trie_trl*[#] { "downward" link in a trie }
define *trie_char*(#) ≡ *trie_trc*[#] { character matched at this trie location }
define *trie_op*(#) ≡ *trie_tro*[#] { program for hyphenation at this trie location }

⟨ Global variables 13 ⟩ +≡
{ We will dynamically allocate these arrays. }
trie_trl: ↑*trie_pointer*; { *trie_link* }
trie_tro: ↑*trie_pointer*; { *trie_op* }
trie_trc: ↑*quarterword*; { *trie_char* }
hyf_distance: **array** [1 .. *trie_op_size*] **of** *small_number*; { position $k - j$ of n_j }
hyf_num: **array** [1 .. *trie_op_size*] **of** *small_number*; { value of n_j }
hyf_next: **array** [1 .. *trie_op_size*] **of** *trie_opcode*; { continuation code }
op_start: **array** [*ASCII_code*] **of** 0 .. *trie_op_size*; { offset for current language }

1098* Assuming that these auxiliary tables have been set up properly, the hyphenation algorithm is quite short. In the following code we set $hc[hn+2]$ to the impossible value 256, in order to guarantee that $hc[hn+3]$ will never be fetched.

⟨ Find hyphen locations for the word in *hc*, or **return** 1098* ⟩ ≡
  for $j \leftarrow 0$ **to** hn **do** $hyf[j] \leftarrow 0$;
  ⟨ Look for the word $hc[1 \mathrel{..} hn]$ in the exception table, and **goto** *found* (with *hyf* containing the hyphens) if an entry is found 1105* ⟩;
  if $trie\_char(cur\_lang + 1) \neq qi(cur\_lang)$ **then return**; { no patterns for *cur_lang* }
  $hc[0] \leftarrow 0$; $hc[hn+1] \leftarrow 0$; $hc[hn+2] \leftarrow 256$; { insert delimiters }
  for $j \leftarrow 0$ **to** $hn - r\_hyf + 1$ **do**
    begin $z \leftarrow trie\_link(cur\_lang + 1) + hc[j]$; $l \leftarrow j$;
    while $hc[l] = qo(trie\_char(z))$ **do**
      begin if $trie\_op(z) \neq min\_trie\_op$ **then** ⟨ Store maximum values in the *hyf* table 1099* ⟩;
      $incr(l)$; $z \leftarrow trie\_link(z) + hc[l]$;
      end;
    end;
found: **for** $j \leftarrow 0$ **to** $l\_hyf - 1$ **do** $hyf[j] \leftarrow 0$;
  for $j \leftarrow 0$ **to** $r\_hyf - 1$ **do** $hyf[hn - j] \leftarrow 0$

This code is used in section 1070.

1099* ⟨ Store maximum values in the *hyf* table 1099* ⟩ ≡
  begin $v \leftarrow trie\_op(z)$;
  repeat $v \leftarrow v + op\_start[cur\_lang]$; $i \leftarrow l - hyf\_distance[v]$;
    if $hyf\_num[v] > hyf[i]$ **then** $hyf[i] \leftarrow hyf\_num[v]$;
    $v \leftarrow hyf\_next[v]$;
  until $v = min\_trie\_op$;
  end

This code is used in section 1098*.

1100* The exception table that is built by TEX's `\hyphenation` primitive is organized as an ordered hash table [cf. Amble and Knuth, *The Computer Journal* **17** (1974), 135–142] using linear probing. If α and β are words, we will say that $\alpha < \beta$ if $|\alpha| < |\beta|$ or if $|\alpha| = |\beta|$ and α is lexicographically smaller than β. (The notation $|\alpha|$ stands for the length of α.) The idea of ordered hashing is to arrange the table so that a given word α can be sought by computing a hash address $h = h(\alpha)$ and then looking in table positions h, $h - 1$, ..., until encountering the first word $\leq \alpha$. If this word is different from α, we can conclude that α is not in the table. This is a clever scheme which saves the need for a hash link array. However, it is difficult to increase the size of the hyphen exception arrays. To make this easier, the ordered hash has been replaced by a simple hash, using an additional array *hyph_link*. The value 0 in $hyph\_link[k]$ means that there are no more entries corresponding to the specific hash chain. When $hyph\_link[k] > 0$, the next entry in the hash chain is $hyph\_link[k] - 1$. This value is used because the arrays start at 0.

The words in the table point to lists in *mem* that specify hyphen positions in their *info* fields. The list for $c_1 \ldots c_n$ contains the number k if the word $c_1 \ldots c_n$ has a discretionary hyphen between c_k and c_{k+1}.

⟨ Types in the outer block 18 ⟩ +≡
  $hyph\_pointer = 0 \mathrel{..} ssup\_hyph\_size$;
    { index into hyphen exceptions hash table; enlarging this requires changing (un)dump code }

1101* ⟨Global variables 13⟩ +≡
hyph_word: ↑*str_number*; { exception words }
hyph_list: ↑*pointer*; { lists of hyphen positions }
hyph_link: ↑*hyph_pointer*; { link array for hyphen exceptions hash table }
hyph_count: *integer*; { the number of words in the exception dictionary }
hyph_next: *integer*; { next free slot in hyphen exceptions hash table }

1103* ⟨Set initial values of key variables 21⟩ +≡
for $z \gets 0$ **to** *hyph_size* **do**
begin *hyph_word*$[z] \gets 0$; *hyph_list*$[z] \gets$ *null*; *hyph_link*$[z] \gets 0$;
end;
hyph_count $\gets 0$; *hyph_next* $\gets$ *hyph_prime* $+ 1$;
if *hyph_next* $>$ *hyph_size* **then** *hyph_next* $\gets$ *hyph_prime*;

1105* First we compute the hash code h, then we search until we either find the word or we don't. Words from different languages are kept separate by appending the language code to the string.

⟨Look for the word $hc[1\,..\,hn]$ in the exception table, and **goto** *found* (with *hyf* containing the hyphens) if an entry is found 1105*⟩ ≡
$h \gets hc[1]$; *incr*(hn); $hc[hn] \gets$ *cur_lang*;
for $j \gets 2$ **to** *hn* **do** $h \gets (h + h + hc[j])$ **mod** *hyph_prime*;
loop begin ⟨If the string *hyph_word*$[h]$ is less than $hc[1\,..\,hn]$, **goto** *not_found*; but if the two strings are equal, set *hyf* to the hyphen positions and **goto** *found* 1106*⟩;
$h \gets$ *hyph_link*$[h]$;
if $h = 0$ **then goto** *not_found*;
decr(h);
end;
not_found: *decr*(hn)

This code is used in section 1098*.

1106* ⟨If the string *hyph_word*$[h]$ is less than $hc[1\,..\,hn]$, **goto** *not_found*; but if the two strings are equal, set *hyf* to the hyphen positions and **goto** *found* 1106*⟩ ≡
{ This is now a simple hash list, not an ordered one, so the module title is no longer descriptive. }
$k \gets$ *hyph_word*$[h]$;
if $k = 0$ **then goto** *not_found*;
if *length*$(k) = hn$ **then**
begin $j \gets 1$; $u \gets$ *str_start*$[k]$;
repeat if *so*(*str_pool*$[u]$) $\neq hc[j]$ **then goto** *done*;
incr(j); *incr*(u);
until $j > hn$;
⟨Insert hyphens as specified in *hyph_list*$[h]$ 1107⟩;
decr(hn); **goto** *found*;
end;
done:

This code is used in section 1105*.

1109* We have now completed the hyphenation routine, so the *line_break* procedure is finished at last. Since the hyphenation exception table is fresh in our minds, it's a good time to deal with the routine that adds new entries to it.

When TEX has scanned '`\hyphenation`', it calls on a procedure named *new_hyph_exceptions* to do the right thing.

define *set_cur_lang* ≡
  if *language* ≤ 0 **then** *cur_lang* ← 0
  else if *language* > 255 **then** *cur_lang* ← 0
   else *cur_lang* ← *language*

procedure *new_hyph_exceptions*; { enters new exceptions }
 label *reswitch*, *exit*, *found*, *not_found*, *not_found1*;
 var *n*: 0 . . 64; { length of current word; not always a *small_number* }
  j: 0 . . 64; { an index into *hc* }
  h: *hyph_pointer*; { an index into *hyph_word* and *hyph_list* }
  k: *str_number*; { an index into *str_start* }
  p: *pointer*; { head of a list of hyphen positions }
  q: *pointer*; { used when creating a new node for list *p* }
  s: *str_number*; { strings being compared or stored }
  u, *v*: *pool_pointer*; { indices into *str_pool* }
 begin *scan_left_brace*; { a left brace must follow `\hyphenation` }
 set_cur_lang;
 init if *trie_not_ready* **then**
  begin *hyph_index* ← 0; **goto** *not_found1*;
  end;
 tini
 set_hyph_index;
not_found1: ⟨ Enter as many hyphenation exceptions as are listed, until coming to a right brace; then **return** 1110 ⟩;
exit: **end**;

1114* ⟨ Enter a hyphenation exception 1114* ⟩ ≡
 begin *incr*(*n*); *hc*[*n*] ← *cur_lang*; *str_room*(*n*); *h* ← 0;
 for *j* ← 1 **to** *n* **do**
  begin *h* ← (*h* + *h* + *hc*[*j*]) **mod** *hyph_prime*; *append_char*(*hc*[*j*]);
  end;
 s ← *make_string*; ⟨ Insert the pair (*s*, *p*) into the exception table 1115* ⟩;
 end

This code is used in section 1110.

1115* ⟨ Insert the pair (s, p) into the exception table 1115* ⟩ ≡
  if *hyph_next* $\leq$ *hyph_prime* **then**
    while $(hyph\_next > 0) \land (hyph\_word[hyph\_next - 1] > 0)$ **do** *decr*(*hyph_next*);
  if $(hyph\_count = hyph\_size) \lor (hyph\_next = 0)$ **then** *overflow*(`"exception␣dictionary"`, *hyph_size*);
  incr(*hyph_count*);
  while *hyph_word*[h] $\neq 0$ **do**
    begin ⟨ If the string *hyph_word*[h] is less than or equal to s, interchange (*hyph_word*[h], *hyph_list*[h]) with (s, p) 1116* ⟩;
    if *hyph_link*[h] $= 0$ **then**
      begin *hyph_link*[h] ← *hyph_next*;
      if *hyph_next* $\geq$ *hyph_size* **then** *hyph_next* ← *hyph_prime*;
      if *hyph_next* $>$ *hyph_prime* **then** *incr*(*hyph_next*);
      end;
    h ← *hyph_link*[h] $- 1$;
    end;
found: *hyph_word*[h] ← s; *hyph_list*[h] ← p

This code is used in section 1114*.

1116* ⟨ If the string *hyph_word*[h] is less than or equal to s, interchange (*hyph_word*[h], *hyph_list*[h]) with (s, p) 1116* ⟩ ≡
  { This is now a simple hash list, not an ordered one, so the module title is no longer descriptive. }
  k ← *hyph_word*[h];
  if *length*(k) $\neq$ *length*(s) **then goto** *not_found*;
  u ← *str_start*[k]; v ← *str_start*[s];
  repeat if *str_pool*[u] $\neq$ *str_pool*[v] **then goto** *not_found*;
    incr(u); *incr*(v);
  until $u =$ *str_start*[$k + 1$]; { repeat hyphenation exception; flushing old data }
  flush_string; s ← *hyph_word*[h]; { avoid *slow_make_string*! }
  decr(*hyph_count*); { We could also *flush_list*(*hyph_list*[h]);, but it interferes with `trip.log`. }
  goto *found*;
not_found:

This code is used in section 1115*.

1118* Before we discuss trie building in detail, let's consider the simpler problem of creating the *hyf_distance*, *hyf_num*, and *hyf_next* arrays.

Suppose, for example, that TEX reads the pattern '`ab2cde1`'. This is a pattern of length 5, with $n_0 \ldots n_5 =$ 0 0 2 0 0 1 in the notation above. We want the corresponding *trie_op* code v to have $hyf\_distance[v] = 3$, $hyf\_num[v] = 2$, and $hyf\_next[v] = v'$, where the auxiliary *trie_op* code v' has $hyf\_distance[v'] = 0$, $hyf\_num[v'] = 1$, and $hyf\_next[v'] = min\_trie\_op$.

TEX computes an appropriate value v with the *new_trie_op* subroutine below, by setting

$$v' \leftarrow new\_trie\_op(0, 1, min\_trie\_op), \qquad v \leftarrow new\_trie\_op(3, 2, v').$$

This subroutine looks up its three parameters in a special hash table, assigning a new value only if these three have not appeared before for the current language.

The hash table is called *trie_op_hash*, and the number of entries it contains is *trie_op_ptr*.

⟨ Global variables 13 ⟩ +≡
init *trie_op_hash*: **array** [*neg_trie_op_size* .. *trie_op_size*] **of** 0 .. *trie_op_size*;
{ trie op codes for quadruples }
trie_used: **array** [*ASCII_code*] **of** *trie_opcode*; { largest opcode used so far for this language }
trie_op_lang: **array** [1 .. *trie_op_size*] **of** *ASCII_code*; { language part of a hashed quadruple }
trie_op_val: **array** [1 .. *trie_op_size*] **of** *trie_opcode*; { opcode corresponding to a hashed quadruple }
trie_op_ptr: 0 .. *trie_op_size*; { number of stored ops so far }
tini
max_op_used: *trie_opcode*; { largest opcode used for any language }
small_op: *boolean*; { flag used while dumping or undumping }

1119* It's tempting to remove the *overflow* stops in the following procedure; *new_trie_op* could return *min_trie_op* (thereby simply ignoring part of a hyphenation pattern) instead of aborting the job. However, that would lead to different hyphenation results on different installations of TEX using the same patterns. The *overflow* stops are necessary for portability of patterns.

⟨Declare procedures for preprocessing hyphenation patterns 1119*⟩ ≡
function *new_trie_op*(*d*, *n* : *small_number*; *v* : *trie_opcode*): *trie_opcode*;
  label *exit*;
  var *h*: *neg_trie_op_size* .. *trie_op_size*; { trial hash location }
    u: *trie_opcode*; { trial op code }
    l: 0 .. *trie_op_size*; { pointer to stored data }
  begin *h* ← *abs*(*n*+313∗*d*+361∗*v*+1009∗*cur_lang*) **mod** (*trie_op_size* − *neg_trie_op_size*) + *neg_trie_op_size*;
  loop begin *l* ← *trie_op_hash*[*h*];
    if *l* = 0 **then** { empty position found for a new op }
      begin if *trie_op_ptr* = *trie_op_size* **then** *overflow*("pattern␣memory␣ops", *trie_op_size*);
      u ← *trie_used*[*cur_lang*];
      if *u* = *max_trie_op* **then**
        overflow("pattern␣memory␣ops␣per␣language", *max_trie_op* − *min_trie_op*);
      incr(*trie_op_ptr*); *incr*(*u*); *trie_used*[*cur_lang*] ← *u*;
      if *u* > *max_op_used* **then** *max_op_used* ← *u*;
      hyf_distance[*trie_op_ptr*] ← *d*; *hyf_num*[*trie_op_ptr*] ← *n*; *hyf_next*[*trie_op_ptr*] ← *v*;
      trie_op_lang[*trie_op_ptr*] ← *cur_lang*; *trie_op_hash*[*h*] ← *trie_op_ptr*; *trie_op_val*[*trie_op_ptr*] ← *u*;
      new_trie_op ← *u*; **return**;
      end;
    if (*hyf_distance*[*l*] = *d*) ∧ (*hyf_num*[*l*] = *n*) ∧ (*hyf_next*[*l*] = *v*) ∧ (*trie_op_lang*[*l*] = *cur_lang*) **then**
      begin *new_trie_op* ← *trie_op_val*[*l*]; **return**;
      end;
    if *h* > −*trie_op_size* **then** *decr*(*h*) **else** *h* ← *trie_op_size*;
    end;
exit: **end**;

See also sections 1123, 1124, 1128, 1132, 1134, 1135*, and 1141*.

This code is used in section 1117.

1120* After *new_trie_op* has compressed the necessary opcode information, plenty of information is available to unscramble the data into the final form needed by our hyphenation algorithm.

⟨Sort the hyphenation op tables into proper order 1120*⟩ ≡
  op_start[0] ← −*min_trie_op*;
  for *j* ← 1 **to** 255 **do** *op_start*[*j*] ← *op_start*[*j* − 1] + *qo*(*trie_used*[*j* − 1]);
  for *j* ← 1 **to** *trie_op_ptr* **do** *trie_op_hash*[*j*] ← *op_start*[*trie_op_lang*[*j*]] + *trie_op_val*[*j*]; { destination }
  for *j* ← 1 **to** *trie_op_ptr* **do**
    while *trie_op_hash*[*j*] > *j* **do**
      begin *k* ← *trie_op_hash*[*j*];
      t ← *hyf_distance*[*k*]; *hyf_distance*[*k*] ← *hyf_distance*[*j*]; *hyf_distance*[*j*] ← *t*;
      t ← *hyf_num*[*k*]; *hyf_num*[*k*] ← *hyf_num*[*j*]; *hyf_num*[*j*] ← *t*;
      t ← *hyf_next*[*k*]; *hyf_next*[*k*] ← *hyf_next*[*j*]; *hyf_next*[*j*] ← *t*;
      trie_op_hash[*j*] ← *trie_op_hash*[*k*]; *trie_op_hash*[*k*] ← *k*;
      end

This code is used in section 1127.

1121* Before we forget how to initialize the data structures that have been mentioned so far, let's write down the code that gets them started.

⟨ Initialize table entries (done by INITEX only) 182 ⟩ +≡
for $k \leftarrow -trie\_op\_size$ **to** $trie\_op\_size$ **do** $trie\_op\_hash[k] \leftarrow 0$;
for $k \leftarrow 0$ **to** 255 **do** $trie\_used[k] \leftarrow min\_trie\_op$;
$max\_op\_used \leftarrow min\_trie\_op$; $trie\_op\_ptr \leftarrow 0$;

1122* The linked trie that is used to preprocess hyphenation patterns appears in several global arrays. Each node represents an instruction of the form "if you see character c, then perform operation o, move to the next character, and go to node l; otherwise go to node r." The four quantities c, o, l, and r are stored in four arrays *trie_c*, *trie_o*, *trie_l*, and *trie_r*. The root of the trie is $trie\_l[0]$, and the number of nodes is *trie_ptr*. Null trie pointers are represented by zero. To initialize the trie, we simply set $trie\_l[0]$ and *trie_ptr* to zero. We also set $trie\_c[0]$ to some arbitrary value, since the algorithm may access it.

The algorithms maintain the condition

$$trie\_c[trie\_r[z]] > trie\_c[z] \qquad \text{whenever } z \neq 0 \text{ and } trie\_r[z] \neq 0;$$

in other words, sibling nodes are ordered by their c fields.

define $trie\_root \equiv trie\_l[0]$ { root of the linked trie }

⟨ Global variables 13 ⟩ +≡
init *trie_c*: ↑*packed_ASCII_code*; { characters to match }
trie_o: ↑*trie_opcode*; { operations to perform }
trie_l: ↑*trie_pointer*; { left subtrie links }
trie_r: ↑*trie_pointer*; { right subtrie links }
trie_ptr: *trie_pointer*; { the number of nodes in the trie }
trie_hash: ↑*trie_pointer*; { used to identify equivalent subtries }
tini

1125* The compressed trie will be packed into the *trie* array using a "top-down first-fit" procedure. This is a little tricky, so the reader should pay close attention: The *trie_hash* array is cleared to zero again and renamed *trie_ref* for this phase of the operation; later on, $trie\_ref[p]$ will be nonzero only if the linked trie node p is the smallest character in a family and if the characters c of that family have been allocated to locations $trie\_ref[p] + c$ in the *trie* array. Locations of *trie* that are in use will have $trie\_link = 0$, while the unused holes in *trie* will be doubly linked with *trie_link* pointing to the next larger vacant location and *trie_back* pointing to the next smaller one. This double linking will have been carried out only as far as *trie_max*, where *trie_max* is the largest index of *trie* that will be needed. To save time at the low end of the trie, we maintain array entries $trie\_min[c]$ pointing to the smallest hole that is greater than c. Another array *trie_taken* tells whether or not a given location is equal to $trie\_ref[p]$ for some p; this array is used to ensure that distinct nodes in the compressed trie will have distinct *trie_ref* entries.

define $trie\_ref \equiv trie\_hash$ { where linked trie families go into *trie* }
define $trie\_back(\#) \equiv trie\_tro[\#]$ { use the opcode field now for backward links }

⟨ Global variables 13 ⟩ +≡
init *trie_taken*: ↑*boolean*; { does a family start here? }
trie_min: **array** [*ASCII_code*] **of** *trie_pointer*; { the first possible slot for each character }
trie_max: *trie_pointer*; { largest location used in *trie* }
trie_not_ready: *boolean*; { is the trie still in linked form? }
tini

1126* Each time `\patterns` appears, it contributes further patterns to the future trie, which will be built only when hyphenation is attempted or when a format file is dumped. The boolean variable *trie_not_ready* will change to *false* when the trie is compressed; this will disable further patterns.

⟨ Initialize table entries (done by INITEX only) 182 ⟩ +≡
trie_not_ready ← *true*;

1133* When the whole trie has been allocated into the sequential table, we must go through it once again so that *trie* contains the correct information. Null pointers in the linked trie will be represented by the value 0, which properly implements an "empty" family.

```
define clear_trie ≡ { clear trie[r] }
    begin trie_link(r) ← 0; trie_op(r) ← min_trie_op; trie_char(r) ← min_quarterword;
        { trie_char ← qi(0) }
    end
```

⟨ Move the data into *trie* 1133* ⟩ ≡

```
if trie_max = 0 then { no patterns were given }
  begin for r ← 0 to 256 do clear_trie;
  trie_max ← 256;
  end
else begin if hyph_root > 0 then trie_fix(hyph_root);
  if trie_root > 0 then trie_fix(trie_root); { this fixes the non-holes in trie }
  r ← 0; { now we will zero out all the holes }
  repeat s ← trie_link(r); clear_trie; r ← s;
  until r > trie_max;
  end;
trie_char(0) ← qi("?"); { make trie_char(c) ≠ c for all c }
```

This code is used in section 1141*.

1135* Now let's go back to the easier problem, of building the linked trie. When INITEX has scanned the '`\patterns`' control sequence, it calls on *new_patterns* to do the right thing.

⟨ Declare procedures for preprocessing hyphenation patterns 1119* ⟩ +≡

```
procedure new_patterns; { initializes the hyphenation pattern data }
  label done, done1;
  var k, l: 0 .. 64; { indices into hc and hyf; not always in small_number range }
    digit_sensed: boolean; { should the next digit be treated as a letter? }
    v: trie_opcode; { trie op code }
    p, q: trie_pointer; { nodes of trie traversed during insertion }
    first_child: boolean; { is p = trie_l[q]? }
    c: ASCII_code; { character being inserted }
  begin if trie_not_ready then
    begin set_cur_lang; scan_left_brace; { a left brace must follow \patterns }
    ⟨ Enter all of the patterns into a linked trie, until coming to a right brace 1136 ⟩;
    if saving_hyph_codes > 0 then ⟨ Store hyphenation codes for current language 1852 ⟩;
    end
  else begin print_err("Too␣late␣for␣"); print_esc("patterns");
    help1("All␣patterns␣must␣be␣given␣before␣typesetting␣begins."); error;
    link(garbage) ← scan_toks(false, false); flush_list(def_ref);
    end;
  end;
```

1138* When the following code comes into play, the pattern $p_1 \ldots p_k$ appears in $hc[1 \mathinner{..} k]$, and the corresponding sequence of numbers $n_0 \ldots n_k$ appears in $hyf[0 \mathinner{..} k]$.

⟨Insert a new pattern into the linked trie 1138*⟩ ≡
begin ⟨Compute the trie op code, v, and set $l \leftarrow 0$ 1140*⟩;
$q \leftarrow 0$; $hc[0] \leftarrow cur\_lang$;
while $l \le k$ **do**
begin $c \leftarrow hc[l]$; $incr(l)$; $p \leftarrow trie\_l[q]$; $first\_child \leftarrow true$;
while $(p > 0) \land (c > so(trie\_c[p]))$ **do**
begin $q \leftarrow p$; $p \leftarrow trie\_r[q]$; $first\_child \leftarrow false$;
end;
if $(p = 0) \lor (c < so(trie\_c[p]))$ **then**
⟨Insert a new trie node between q and p, and make p point to it 1139*⟩;
$q \leftarrow p$; { now node q represents $p_1 \ldots p_{l-1}$ }
end;
if $trie\_o[q] \neq min\_trie\_op$ **then**
begin *print_err*("Duplicate␣pattern"); *help1*("(See␣Appendix␣H.)"); *error*;
end;
$trie\_o[q] \leftarrow v$;
end

This code is used in section 1136.

1139* ⟨Insert a new trie node between q and p, and make p point to it 1139*⟩ ≡
begin if $trie\_ptr = trie\_size$ **then** *overflow*("pattern␣memory", *trie_size*);
$incr(trie\_ptr)$; $trie\_r[trie\_ptr] \leftarrow p$; $p \leftarrow trie\_ptr$; $trie\_l[p] \leftarrow 0$;
if *first_child* **then** $trie\_l[q] \leftarrow p$ **else** $trie\_r[q] \leftarrow p$;
$trie\_c[p] \leftarrow si(c)$; $trie\_o[p] \leftarrow min\_trie\_op$;
end

This code is used in sections 1138*, 1852, and 1853.

1140* ⟨Compute the trie op code, v, and set $l \leftarrow 0$ 1140*⟩ ≡
if $hc[1] = 0$ **then** $hyf[0] \leftarrow 0$;
if $hc[k] = 0$ **then** $hyf[k] \leftarrow 0$;
$l \leftarrow k$; $v \leftarrow min\_trie\_op$;
loop begin if $hyf[l] \neq 0$ **then** $v \leftarrow new\_trie\_op(k - l, hyf[l], v)$;
if $l > 0$ **then** $decr(l)$ **else goto** *done1*;
end;
done1:

This code is used in section 1138*.

1141* Finally we put everything together: Here is how the trie gets to its final, efficient form. The following packing routine is rigged so that the root of the linked tree gets mapped into location 1 of *trie*, as required by the hyphenation algorithm. This happens because the first call of *first_fit* will "take" location 1.

⟨Declare procedures for preprocessing hyphenation patterns 1119*⟩ +≡
procedure *init_trie*;
  var *p*: *trie_pointer*; { pointer for initialization }
    j, *k*, *t*: *integer*; { all-purpose registers for initialization }
    r, *s*: *trie_pointer*; { used to clean up the packed *trie* }
  begin ⟨Get ready to compress the trie 1127⟩;
  if *trie_root* $\neq 0$ **then**
    begin *first_fit*(*trie_root*); *trie_pack*(*trie_root*);
    end;
  if *hyph_root* $\neq 0$ **then** ⟨Pack all stored *hyph_codes* 1854⟩;
  ⟨Move the data into *trie* 1133*⟩;
  trie_not_ready ← *false*;
  end;

1163*. Pages are built by appending nodes to the current list in TEX's vertical mode, which is at the outermost level of the semantic nest. This vlist is split into two parts; the "current page" that we have been talking so much about already, and the "contribution list" that receives new nodes as they are created. The current page contains everything that the page builder has accounted for in its data structures, as described above, while the contribution list contains other things that have been generated by other parts of TEX but have not yet been seen by the page builder. The contribution list starts at *link*(*contrib_head*), and it ends at the current node in TEX's vertical mode.

When TEX has appended new material in vertical mode, it calls the procedure *build_page*, which tries to catch up by moving nodes from the contribution list to the current page. This procedure will succeed in its goal of emptying the contribution list, unless a page break is discovered, i.e., unless the current page has grown to the point where the optimum next page break has been determined. In the latter case, the nodes after the optimum break will go back onto the contribution list, and control will effectively pass to the user's output routine.

We make *type*(*page_head*) = *glue_node*, so that an initial glue node on the current page will not be considered a valid breakpoint.

⟨Initialize the special list heads and constant nodes 964⟩ +≡
type(*page_head*) ← *glue_node*; *subtype*(*page_head*) ← *normal*;
{ *SyncTEX* watch point: box(*page_head*) size ¿= *glue_node* size }

1209*. We leave the *space_factor* unchanged if *sf_code*(*cur_chr*) = 0; otherwise we set it equal to *sf_code*(*cur_chr*), except that it should never change from a value less than 1000 to a value exceeding 1000. The most common case is *sf_code*(*cur_chr*) = 1000, so we want that case to be fast.

The overall structure of the main loop is presented here. Some program labels are inside the individual sections.

define *adjust_space_factor* ≡
  main_s ← *sf_code*(*cur_chr*);
  if *main_s* = 1000 **then** *space_factor* ← 1000
  else if *main_s* < 1000 **then**
    begin if *main_s* > 0 **then** *space_factor* ← *main_s*;
    end
   else if *space_factor* < 1000 **then** *space_factor* ← 1000
    else *space_factor* ← *main_s*

⟨Append character *cur_chr* and the following characters (if any) to the current hlist in the current font; **goto** *reswitch* when a non-character has been fetched 1209*⟩ ≡
if ((*head* = *tail*) ∧ (*mode* > 0)) **then**
  begin if (*insert_src_special_auto*) **then** *append_src_special*;
  end;
adjust_space_factor;
save_tail ← *null*; *main_f* ← *cur_font*; *bchar* ← *font_bchar*[*main_f*];
false_bchar ← *font_false_bchar*[*main_f*];
if *mode* > 0 **then**
  if *language* ≠ *clang* **then** *fix_language*;
fast_get_avail(*lig_stack*); *font*(*lig_stack*) ← *main_f*; *cur_l* ← *qi*(*cur_chr*); *character*(*lig_stack*) ← *cur_l*;
cur_q ← *tail*; *tmp_k1* ← *get_auto_kern*(*main_f*, *non_char*, *cur_l*);
⟨If *tmp_k1* is not null then append that kern 1215⟩;
if *cancel_boundary* **then**
  begin *cancel_boundary* ← *false*; *main_k* ← *non_address*;
  end
else *main_k* ← *bchar_label*[*main_f*];
if *main_k* = *non_address* **then goto** *main_loop_move* + 2; { no left boundary processing }
cur_r ← *cur_l*; *cur_l* ← *non_char*; **goto** *main_lig_loop* + 1; { begin with cursor after left boundary }
main_loop_wrapup: ⟨Make a ligature node, if *ligature_present*; insert a null discretionary, if appropriate 1210⟩;
main_loop_move: ⟨If the cursor is immediately followed by the right boundary, **goto** *reswitch*; if it's followed by an invalid character, **goto** *big_switch*; otherwise move the cursor one step to the right and **goto** *main_lig_loop* 1211*⟩;
main_loop_lookahead: ⟨Look ahead for another character, or leave *lig_stack* empty if there's none there 1213⟩;
main_lig_loop: ⟨If there's a ligature/kern command relevant to *cur_l* and *cur_r*, adjust the text appropriately; exit to *main_loop_wrapup* 1214⟩;
main_loop_move_lig: ⟨Move the cursor past a pseudo-ligature, then **goto** *main_loop_lookahead* or *main_lig_loop* 1212*⟩

This code is used in section 1205.

1211* ⟨If the cursor is immediately followed by the right boundary, **goto** *reswitch*; if it's followed by an invalid character, **goto** *big_switch*; otherwise move the cursor one step to the right and **goto** *main_lig_loop* 1211*⟩ ≡

```
if lig_stack = null then goto reswitch;
cur_q ← tail; cur_l ← character(lig_stack);
main_loop_move + 1: if ¬is_char_node(lig_stack) then goto main_loop_move_lig;
main_loop_move + 2: if (qo(effective_char(false, main_f,
      qi(cur_chr))) > font_ec[main_f]) ∨ (qo(effective_char(false, main_f, qi(cur_chr))) < font_bc[main_f])
      then
    begin char_warning(main_f, cur_chr); free_avail(lig_stack); goto big_switch;
    end;
  main_i ← effective_char_info(main_f, cur_l);
  if ¬char_exists(main_i) then
    begin char_warning(main_f, cur_chr); free_avail(lig_stack); goto big_switch;
    end;
  link(tail) ← lig_stack; tail ← lig_stack   { main_loop_lookahead is next }
```

This code is used in section 1209*.

1212* Here we are at *main_loop_move_lig*. When we begin this code we have $cur\_q = tail$ and $cur\_l = character(lig\_stack)$.

⟨Move the cursor past a pseudo-ligature, then **goto** *main_loop_lookahead* or *main_lig_loop* 1212*⟩ ≡

```
  main_p ← lig_ptr(lig_stack);
  if main_p > null then tail_append(main_p);   { append a single character }
  temp_ptr ← lig_stack; lig_stack ← link(temp_ptr); free_node(temp_ptr, small_node_size);
      { SyncTEX watch point: proper size! }
  main_i ← char_info(main_f)(cur_l); ligature_present ← true;
  if lig_stack = null then
    if main_p > null then goto main_loop_lookahead
    else cur_r ← bchar
  else cur_r ← character(lig_stack);
  goto main_lig_loop
```

This code is used in section 1209*.

1225* The '*you_cant*' procedure prints a line saying that the current command is illegal in the current mode; it identifies these things symbolically.

⟨Declare action procedures for use by *main_control* 1219⟩ +≡

```
procedure you_cant;
  begin print_err("You␣can´t␣use␣`"); print_cmd_chr(cur_cmd, cur_chr); print_in_mode(mode);
  end;
```

1267* ⟨Declare action procedures for use by *main_control* 1219⟩ +≡

```
function norm_min(h : integer): small_number;
  begin if h ≤ 0 then norm_min ← 1 else if h ≥ 63 then norm_min ← 63 else norm_min ← h;
  end;

procedure new_graf(indented : boolean);
  begin prev_graf ← 0;
  if (mode = vmode) ∨ (head ≠ tail) then tail_append(new_param_glue(par_skip_code));
  push_nest; mode ← hmode; space_factor ← 1000; set_cur_lang; clang ← cur_lang;
  prev_graf ← (norm_min(left_hyphen_min) ∗ ´100 + norm_min(right_hyphen_min)) ∗ ´200000 + cur_lang;
  if indented then
    begin tail ← new_null_box; link(head) ← tail; width(tail) ← par_indent;
    if (insert_src_special_every_par) then insert_src_special;
    end;
  if every_par ≠ null then begin_token_list(every_par, every_par_text);
  if nest_ptr = 1 then build_page; { put par_skip glue on current page }
  end;
```

1311* ⟨Declare action procedures for use by *main_control* 1219⟩ +≡

```
procedure cs_error;
  begin if cur_chr = 10 then
    begin print_err("Extra␣"); print_esc("endmubyte");
    help1("I´m␣ignoring␣this,␣since␣I␣wasn´t␣doing␣a␣\mubyte.");
    end
  else begin print_err("Extra␣"); print_esc("endcsname");
    help1("I´m␣ignoring␣this,␣since␣I␣wasn´t␣doing␣a␣\csname.");
    end;
  error;
  end;
```

1315* ⟨ Go into ordinary math mode 1315* ⟩ ≡
begin *push_math*(*math_shift_group*); *eq_word_define*(*int_base* + *cur_fam_code*, −1);
if (*insert_src_special_every_math*) **then** *insert_src_special*;
if *every_math* ≠ *null* **then** *begin_token_list*(*every_math*, *every_math_text*);
end

This code is used in sections 1314 and 1318.

1343* ⟨ Cases of *main_control* that build boxes and lists 1232 ⟩ +≡
mmode + *vcenter*: **begin** *scan_spec*(*vcenter_group*, *false*); *normal_paragraph*; *push_nest*; *mode* ← −*vmode*;
prev_depth ← *pdf_ignored_dimen*;
if (*insert_src_special_every_vbox*) **then** *insert_src_special*;
if *every_vbox* ≠ *null* **then** *begin_token_list*(*every_vbox*, *every_vbox_text*);
end;

1387* If the user says, e.g., '\global\global', the redundancy is silently accepted.

⟨Declare action procedures for use by *main_control* 1219⟩ +≡
⟨Declare subprocedures for *prefixed_command* 1391*⟩
procedure *prefixed_command*;
  label *done*, *exit*;
  var *a*: *small_number*; { accumulated prefix codes so far }
    f: *internal_font_number*; { identifies a font }
    j: *halfword*; { index into a \parshape specification }
    k: *font_index*; { index into *font_info* }
    p, *q*, *r*: *pointer*; { for temporary short-term use }
    n: *integer*; { ditto }
    e: *boolean*; { should a definition be expanded? or was \let not done? }
  begin $a \leftarrow 0$;
  while $cur\_cmd = prefix$ **do**
    begin if $\neg odd(a \,\mathbf{div}\, cur\_chr)$ **then** $a \leftarrow a + cur\_chr$;
    ⟨Get the next non-blank non-relax non-call token 430⟩;
    if $cur\_cmd \le max\_non\_prefixed\_command$ **then** ⟨Discard erroneous prefixes and **return** 1388⟩;
    if $tracing\_commands > 2$ **then**
      if *eTeX_ex* **then** *show_cur_cmd_chr*;
    end;
  ⟨Discard the prefixes \long and \outer if they are irrelevant 1389⟩;
  ⟨Adjust for the setting of \globaldefs 1390⟩;
  case *cur_cmd* **of**
  ⟨Assignments 1393⟩
  othercases *confusion*("prefix")
  endcases;
done: ⟨Insert a token saved by \afterassignment, if any 1445⟩;
exit: **end**;

1391* When a control sequence is to be defined, by \def or \let or something similar, the *get_r_token* routine will substitute a special control sequence for a token that is not redefinable.

⟨Declare subprocedures for *prefixed_command* 1391*⟩ ≡
procedure *get_r_token*;
  label *restart*;
  begin *restart*: **repeat** *get_token*;
  until $cur\_tok \ne space\_token$;
  if $(cur\_cs = 0) \lor (cur\_cs > eqtb\_top) \lor ((cur\_cs > frozen\_control\_sequence) \land (cur\_cs \le eqtb\_size))$ **then**
    begin *print_err*("Missing␣control␣sequence␣inserted");
    help5("Please␣don´t␣say␣`\def␣cs{...}´,␣say␣`\def\cs{...}´.")
    ("I´ve␣inserted␣an␣inaccessible␣control␣sequence␣so␣that␣your")
    ("definition␣will␣be␣completed␣without␣mixing␣me␣up␣too␣badly.")
    ("You␣can␣recover␣graciously␣from␣this␣error,␣if␣you´re")
    ("careful;␣see␣exercise␣27.2␣in␣The␣TeXbook.");
    if $cur\_cs = 0$ **then** *back_input*;
    $cur\_tok \leftarrow cs\_token\_flag + frozen\_protection$; *ins_error*; **goto** *restart*;
    end;
  end;

See also sections 1405, 1412, 1419, 1420, 1421, 1422, 1423, 1433*, and 1441*.

This code is used in section 1387*.

1395.* Both `\let` and `\futurelet` share the command code *let*.

⟨ Put each of TEX's primitives into the hash table 244 ⟩ +≡
  primitive("let", *let*, *normal*);
  primitive("futurelet", *let*, *normal* + 1);
  if *enctex_p* **then**
    begin *primitive*("mubyte", *let*, *normal* + 10);
    primitive("noconvert", *let*, *normal* + 11);
    end;

1396.* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 245 ⟩ +≡
let: **if** *chr_code* ≠ *normal* **then**
    if *chr_code* = *normal* + 10 **then** *print_esc*("mubyte")
    else if *chr_code* = *normal* + 11 **then** *print_esc*("noconvert")
      else *print_esc*("futurelet")
  else *print_esc*("let");

1397.* ⟨ Assignments 1393 ⟩ +≡

```
let: if cur_chr = normal + 11 then do_nothing  { noconvert primitive }
  else if cur_chr = normal + 10 then  { mubyte primitive }
      begin selector ← term_and_log; get_token; mubyte_stoken ← cur_tok;
      if cur_tok ≤ cs_token_flag then mubyte_stoken ← cur_tok mod 256;
      mubyte_prefix ← 60; mubyte_relax ← false; mubyte_tablein ← true; mubyte_tableout ← true;
      get_x_token;
      if cur_cmd = spacer then get_x_token;
      if cur_cmd = sub_mark then
        begin mubyte_tableout ← false; get_x_token;
        if cur_cmd = sub_mark then
          begin mubyte_tableout ← true; mubyte_tablein ← false; get_x_token;
          end;
        end
      else if (mubyte_stoken > cs_token_flag) ∧ (cur_cmd = mac_param) then
          begin mubyte_tableout ← false; scan_int; mubyte_prefix ← cur_val; get_x_token;
          if mubyte_prefix > 50 then mubyte_prefix ← 52;
          if mubyte_prefix ≤ 0 then mubyte_prefix ← 51;
          end
        else if (mubyte_stoken > cs_token_flag) ∧ (cur_cmd = relax) then
            begin mubyte_tableout ← true; mubyte_tablein ← false; mubyte_relax ← true; get_x_token;
            end;
      r ← get_avail; p ← r;
      while cur_cs = 0 do
        begin store_new_token(cur_tok); get_x_token;
        end;
      if (cur_cmd ≠ end_cs_name) ∨ (cur_chr ≠ 10) then
        begin print_err("Missing␣"); print_esc("endmubyte"); print("␣inserted");
        help2("The␣control␣sequence␣marked␣<to␣be␣read␣again>␣should")
        ("not␣appear␣in␣<byte␣sequence>␣between␣\mubyte␣and␣\endmubyte."); back_error;
        end;
      p ← link(r);
      if (p = null) ∧ mubyte_tablein then
        begin print_err("The␣empty␣<byte␣sequence>,␣"); print_esc("mubyte"); print("␣ignored");
        help2("The␣<byte␣sequence>␣in")
        ("\mubyte␣<token>␣<byte␣sequence>\endmubyte␣should␣not␣be␣empty."); error;
        end
      else begin while p ≠ null do
          begin append_char(info(p) mod 256); p ← link(p);
          end;
        flush_list(r);
        if (str_start[str_ptr] + 1 = pool_ptr) ∧ (str_pool[pool_ptr − 1] = mubyte_stoken) then
          begin if mubyte_read[mubyte_stoken] ≠ null ∧ mubyte_tablein then  { clearing data }
            dispose_munode(mubyte_read[mubyte_stoken]);
          if mubyte_tablein then mubyte_read[mubyte_stoken] ← null;
          if mubyte_tableout then mubyte_write[mubyte_stoken] ← 0;
          pool_ptr ← str_start[str_ptr];
          end
        else begin if mubyte_tablein then mubyte_update;  { updating input side }
          if mubyte_tableout then  { updating output side }
            begin if mubyte_stoken > cs_token_flag then  { control sequence }
              begin dispose_mutableout(mubyte_stoken − cs_token_flag);
```

```
            if (str_start[str_ptr] < pool_ptr) ∨ mubyte_relax then
              begin { store data }
              r ← mubyte_cswrite[(mubyte_stoken − cs_token_flag) mod 128]; p ← get_avail;
              mubyte_cswrite[(mubyte_stoken − cs_token_flag) mod 128] ← p;
              info(p) ← mubyte_stoken − cs_token_flag; link(p) ← get_avail; p ← link(p);
              if mubyte_relax then
                begin info(p) ← 0; pool_ptr ← str_start[str_ptr];
                end
              else info(p) ← slow_make_string;
              link(p) ← r;
              end;
            end
          else begin { single character }
            if str_start[str_ptr] = pool_ptr then mubyte_write[mubyte_stoken] ← 0
            else mubyte_write[mubyte_stoken] ← slow_make_string;
            end;
          end
        else pool_ptr ← str_start[str_ptr];
        end;
      end;
    end
  else begin { let primitive }
    n ← cur_chr; get_r_token; p ← cur_cs;
    if n = normal then
      begin repeat get_token;
      until cur_cmd ≠ spacer;
      if cur_tok = other_token + "=" then
        begin get_token;
        if cur_cmd = spacer then get_token;
        end;
      end
    else begin get_token; q ← cur_tok; get_token; back_input; cur_tok ← q; back_input;
          { look ahead, then back up }
      end; { note that back_input doesn't affect cur_cmd, cur_chr }
    if cur_cmd ≥ call then add_token_ref(cur_chr)
    else if (cur_cmd = register) ∨ (cur_cmd = toks_register) then
        if (cur_chr < mem_bot) ∨ (cur_chr > lo_mem_stat_max) then add_sa_ref(cur_chr);
    define(p, cur_cmd, cur_chr);
    end;
```

1398*. A `\chardef` creates a control sequence whose *cmd* is *char_given*; a `\mathchardef` creates a control sequence whose *cmd* is *math_given*; and the corresponding *chr* is the character code or math code. A `\countdef` or `\dimendef` or `\skipdef` or `\muskipdef` creates a control sequence whose *cmd* is *assign_int* or ... or *assign_mu_glue*, and the corresponding *chr* is the *eqtb* location of the internal register in question.

define *char_def_code* = 0 { *shorthand_def* for `\chardef` }
define *math_char_def_code* = 1 { *shorthand_def* for `\mathchardef` }
define *count_def_code* = 2 { *shorthand_def* for `\countdef` }
define *dimen_def_code* = 3 { *shorthand_def* for `\dimendef` }
define *skip_def_code* = 4 { *shorthand_def* for `\skipdef` }
define *mu_skip_def_code* = 5 { *shorthand_def* for `\muskipdef` }
define *toks_def_code* = 6 { *shorthand_def* for `\toksdef` }
define *char_sub_def_code* = 7 { *shorthand_def* for `\charsubdef` }

⟨Put each of TEX's primitives into the hash table 244⟩ +≡
primitive("`chardef`", *shorthand_def*, *char_def_code*);
primitive("`mathchardef`", *shorthand_def*, *math_char_def_code*);
primitive("`countdef`", *shorthand_def*, *count_def_code*);
primitive("`dimendef`", *shorthand_def*, *dimen_def_code*);
primitive("`skipdef`", *shorthand_def*, *skip_def_code*);
primitive("`muskipdef`", *shorthand_def*, *mu_skip_def_code*);
primitive("`toksdef`", *shorthand_def*, *toks_def_code*);
if *mltex_p* **then**
  begin *primitive*("`charsubdef`", *shorthand_def*, *char_sub_def_code*);
  end;

1399*. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡
shorthand_def: **case** *chr_code* **of**
  char_def_code: *print_esc*("`chardef`");
  math_char_def_code: *print_esc*("`mathchardef`");
  count_def_code: *print_esc*("`countdef`");
  dimen_def_code: *print_esc*("`dimendef`");
  skip_def_code: *print_esc*("`skipdef`");
  mu_skip_def_code: *print_esc*("`muskipdef`");
  char_sub_def_code: *print_esc*("`charsubdef`");
  othercases *print_esc*("`toksdef`")
  endcases;
char_given: **begin** *print_esc*("`char`"); *print_hex*(*chr_code*);
  end;
math_given: **begin** *print_esc*("`mathchar`"); *print_hex*(*chr_code*);
  end;

1400*. We temporarily define p to be *relax*, so that an occurrence of p while scanning the definition will simply stop the scanning instead of producing an "undefined control sequence" error or expanding the previous meaning. This allows, for instance, '`\chardef\foo=123\foo`'.

⟨ Assignments 1393 ⟩ +≡
shorthand_def: **if** *cur_chr* = *char_sub_def_code* **then**
  begin *scan_char_num*; $p \leftarrow$ *char_sub_code_base* + *cur_val*; *scan_optional_equals*; *scan_char_num*;
  $n \leftarrow$ *cur_val*; { accent character in substitution }
  scan_char_num;
  if (*tracing_char_sub_def* > 0) **then**
    begin *begin_diagnostic*; *print_nl*("New␣character␣substitution:␣");
    print_ASCII(p − *char_sub_code_base*); *print*("␣=␣"); *print_ASCII*(n); *print_char*("␣");
    print_ASCII(*cur_val*); *end_diagnostic*(*false*);
    end;
  $n \leftarrow n * 256$ + *cur_val*; *define*(p, *data*, *hi*(n));
  if (p − *char_sub_code_base*) < *char_sub_def_min* **then**
    word_define(*int_base* + *char_sub_def_min_code*, p − *char_sub_code_base*);
  if (p − *char_sub_code_base*) > *char_sub_def_max* **then**
    word_define(*int_base* + *char_sub_def_max_code*, p − *char_sub_code_base*);
  end
 else begin $n \leftarrow$ *cur_chr*; *get_r_token*; $p \leftarrow$ *cur_cs*; *define*(p, *relax*, 256); *scan_optional_equals*;
  case n **of**
  char_def_code: **begin** *scan_char_num*; *define*(p, *char_given*, *cur_val*);
    end;
  math_char_def_code: **begin** *scan_fifteen_bit_int*; *define*(p, *math_given*, *cur_val*);
    end;
  othercases begin *scan_register_num*;
    if *cur_val* > 255 **then**
      begin $j \leftarrow n$ − *count_def_code*; { *int_val* .. *box_val* }
      if j > *mu_val* **then** $j \leftarrow$ *tok_val*; { *int_val* .. *mu_val* or *tok_val* }
      find_sa_element(j, *cur_val*, *true*); *add_sa_ref*(*cur_ptr*);
      if j = *tok_val* **then** $j \leftarrow$ *toks_register* **else** $j \leftarrow$ *register*;
      define(p, j, *cur_ptr*);
      end
    else case n **of**
      count_def_code: *define*(p, *assign_int*, *count_base* + *cur_val*);
      dimen_def_code: *define*(p, *assign_dimen*, *scaled_base* + *cur_val*);
      skip_def_code: *define*(p, *assign_glue*, *skip_base* + *cur_val*);
      mu_skip_def_code: *define*(p, *assign_mu_glue*, *mu_skip_base* + *cur_val*);
      toks_def_code: *define*(p, *assign_toks*, *toks_base* + *cur_val*);
      end; { there are no other cases }
    end
  endcases;
  end;

1406* The various character code tables are changed by the *def_code* commands, and the font families are declared by *def_family*.

⟨Put each of TEX's primitives into the hash table 244⟩ +≡
primitive("catcode", *def_code*, *cat_code_base*);
if *enctex_p* **then**
begin *primitive*("xordcode", *def_code*, *xord_code_base*);
primitive("xchrcode", *def_code*, *xchr_code_base*); *primitive*("xprncode", *def_code*, *xprn_code_base*);
end;
primitive("mathcode", *def_code*, *math_code_base*); *primitive*("lccode", *def_code*, *lc_code_base*);
primitive("uccode", *def_code*, *uc_code_base*); *primitive*("sfcode", *def_code*, *sf_code_base*);
primitive("delcode", *def_code*, *del_code_base*); *primitive*("textfont", *def_family*, *math_font_base*);
primitive("scriptfont", *def_family*, *math_font_base* + *script_size*);
primitive("scriptscriptfont", *def_family*, *math_font_base* + *script_script_size*);

1407* ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡
def_code: **if** *chr_code* = *xord_code_base* **then** *print_esc*("xordcode")
else if *chr_code* = *xchr_code_base* **then** *print_esc*("xchrcode")
else if *chr_code* = *xprn_code_base* **then** *print_esc*("xprncode")
else if *chr_code* = *cat_code_base* **then** *print_esc*("catcode")
else if *chr_code* = *math_code_base* **then** *print_esc*("mathcode")
else if *chr_code* = *lc_code_base* **then** *print_esc*("lccode")
else if *chr_code* = *uc_code_base* **then** *print_esc*("uccode")
else if *chr_code* = *sf_code_base* **then** *print_esc*("sfcode")
else *print_esc*("delcode");
def_family: *print_size*(*chr_code* − *math_font_base*);

1408* The different types of code values have different legal ranges; the following program is careful to check each case properly.

⟨Assignments 1393⟩ +≡
def_code: **begin** ⟨Let *n* be the largest legal code value, based on *cur_chr* 1409⟩;
p ← *cur_chr*; *scan_char_num*;
if *p* = *xord_code_base* **then** *p* ← *cur_val*
else if *p* = *xchr_code_base* **then** *p* ← *cur_val* + 256
else if *p* = *xprn_code_base* **then** *p* ← *cur_val* + 512
else *p* ← *p* + *cur_val*;
scan_optional_equals; *scan_int*;
if ((*cur_val* < 0) ∧ (*p* < *del_code_base*)) ∨ (*cur_val* > *n*) **then**
begin *print_err*("Invalid␣code␣("); *print_int*(*cur_val*);
if *p* < *del_code_base* **then** *print*("),␣should␣be␣in␣the␣range␣0..")
else *print*("),␣should␣be␣at␣most␣");
print_int(*n*); *help1*("I´m␣going␣to␣use␣0␣instead␣of␣that␣illegal␣code␣value.");
error; *cur_val* ← 0;
end;
if *p* < 256 **then** *xord*[*p*] ← *cur_val*
else if *p* < 512 **then** *xchr*[*p* − 256] ← *cur_val*
else if *p* < 768 **then** *xprn*[*p* − 512] ← *cur_val*
else if *p* < *math_code_base* **then** *define*(*p*, *data*, *cur_val*)
else if *p* < *del_code_base* **then** *define*(*p*, *data*, *hi*(*cur_val*))
else *word_define*(*p*, *cur_val*);
end;

1428* ⟨ Assignments 1393 ⟩ +≡
hyph_data: **if** *cur_chr* = 1 **then**
  begin Init *new_patterns*; **goto** *done*; **Tini**
  print_err("Patterns␣can␣be␣loaded␣only␣by␣INITEX"); *help0*; *error*;
  repeat *get_token*;
  until *cur_cmd* = *right_brace*; { flush the patterns }
  return;
  end
 else begin *new_hyph_exceptions*; **goto** *done*;
  end;

1433* ⟨ Declare subprocedures for *prefixed_command* 1391* ⟩ +≡
procedure *new_font*(*a* : *small_number*);
 label *common_ending*;
 var *u*: *pointer*; { user's font identifier }
  s: *scaled*; { stated "at" size, or negative of scaled magnification }
  f: *internal_font_number*; { runs through existing fonts }
  t: *str_number*; { name for the frozen font identifier }
  old_setting: 0 . . *max_selector*; { holds *selector* setting }
 begin if *job_name* = 0 **then** *open_log_file*; { avoid confusing `texput` with the font name }
 get_r_token; *u* ← *cur_cs*;
 if *u* ≥ *hash_base* **then** *t* ← *text*(*u*)
 else if *u* ≥ *single_base* **then**
   if *u* = *null_cs* **then** *t* ← "FONT" **else** *t* ← *u* − *single_base*
  else begin *old_setting* ← *selector*; *selector* ← *new_string*; *print*("FONT"); *print*(*u* − *active_base*);
   selector ← *old_setting*; *str_room*(1); *t* ← *make_string*;
   end;
 define(*u*, *set_font*, *null_font*); *scan_optional_equals*; *scan_file_name*;
 ⟨ Scan the font size specification 1434 ⟩;
 ⟨ If this font has already been loaded, set *f* to the internal font number and **goto** *common_ending* 1436* ⟩;
 f ← *read_font_info*(*u*, *cur_name*, *cur_area*, *s*);
common_ending: *define*(*u*, *set_font*, *f*); *eqtb*[*font_id_base* + *f*] ← *eqtb*[*u*]; *font_id_text*(*f*) ← *t*;
 end;

1436* When the user gives a new identifier to a font that was previously loaded, the new name becomes the font identifier of record. Font names '`xyz`' and '`XYZ`' are considered to be different.

⟨ If this font has already been loaded, set *f* to the internal font number and **goto** *common_ending* 1436* ⟩ ≡
 for *f* ← *font_base* + 1 **to** *font_ptr* **do**
  if *str_eq_str*(*font_name*[*f*], *cur_name*) ∧ *str_eq_str*(*font_area*[*f*], *cur_area*) **then**
   begin if *pdf_font_step*[*f*] = 0 **then**
    begin if *s* > 0 **then**
     begin if *s* = *font_size*[*f*] **then goto** *common_ending*;
     end
    else if *font_size*[*f*] = *xn_over_d*(*font_dsize*[*f*], −*s*, 1000) **then goto** *common_ending*;
    end
   end

This code is used in section 1433*.

1441* ⟨ Declare subprocedures for *prefixed_command* 1391* ⟩ +≡
procedure *new_interaction*;
  begin *print_ln*; *interaction* ← *cur_chr*;
  if *interaction* = *batch_mode* **then** *kpse_make_tex_discard_errors* ← 1
  else *kpse_make_tex_discard_errors* ← 0;
  ⟨ Initialize the print *selector* based on *interaction* 75 ⟩;
  if *log_opened* **then** *selector* ← *selector* + 2;
  end;

1451* ⟨ Declare action procedures for use by *main_control* 1219 ⟩ +≡
procedure *open_or_close_in*;
  var *c*: 0 . . 1; { 1 for `\openin`, 0 for `\closein` }
    n: 0 . . 15; { stream number }
  begin *c* ← *cur_chr*; *scan_four_bit_int*; *n* ← *cur_val*;
  if *read_open*[*n*] ≠ *closed* **then**
    begin *a_close*(*read_file*[*n*]); *read_open*[*n*] ← *closed*;
    end;
  if *c* ≠ 0 **then**
    begin *scan_optional_equals*; *scan_file_name*; *pack_cur_name*; *tex_input_type* ← 0;
        { Tell *open_input* we are `\openin`. }
    if *kpse_in_name_ok*(*stringcast*(*name_of_file* + 1)) ∧ *a_open_in*(*read_file*[*n*], *kpse_tex_format*) **then**
      read_open[*n*] ← *just_open*;
    end;
  end;

1455* ⟨ Declare action procedures for use by *main_control* 1219 ⟩ +≡
procedure *issue_message*;
  var *old_setting*: 0 . . *max_selector*; { holds *selector* setting }
    c: 0 . . 1; { identifies `\message` and `\errmessage` }
    s: *str_number*; { the message }
  begin *c* ← *cur_chr*; *link*(*garbage*) ← *scan_toks*(*false*, *true*); *old_setting* ← *selector*;
  selector ← *new_string*; *message_printing* ← *true*; *active_noconvert* ← *true*; *token_show*(*def_ref*);
  message_printing ← *false*; *active_noconvert* ← *false*; *selector* ← *old_setting*; *flush_list*(*def_ref*);
  str_room(1); *s* ← *make_string*;
  if *c* = 0 **then** ⟨ Print string *s* on the terminal 1456* ⟩
  else ⟨ Print string *s* as an error message 1459* ⟩;
  flush_string;
  end;

1456* ⟨ Print string *s* on the terminal 1456* ⟩ ≡
  begin if *term_offset* + *length*(*s*) > *max_print_line* − 2 **then** *print_ln*
  else if (*term_offset* > 0) ∨ (*file_offset* > 0) **then** *print_char*("␣");
  print(*s*); *update_terminal*;
  end

This code is used in section 1455*.

1459*. ⟨ Print string *s* as an error message 1459* ⟩ ≡

```
begin print_err(""); print(s);
if err_help ≠ null then use_err_help ← true
else if long_help_seen then help1("(That␣was␣another␣\errmessage.)")
  else begin if interaction < error_stop_mode then long_help_seen ← true;
    help4("This␣error␣message␣was␣generated␣by␣an␣\errmessage")
    ("command,␣so␣I␣can´t␣give␣any␣explicit␣help.")
    ("Pretend␣that␣you´re␣Hercule␣Poirot:␣Examine␣all␣clues,")
    ("and␣deduce␣the␣truth␣by␣order␣and␣method.");
    end;
error; use_err_help ← false;
end
```

This code is used in section 1455*.

1477* ⟨ Initialize table entries (done by `INITEX` only) 182 ⟩ +≡
if *ini_version* **then** *format_ident* ← "␣(INITEX)";

1478* ⟨ Declare action procedures for use by *main_control* 1219 ⟩ +≡
init procedure *store_fmt_file*;
label *found1* , *found2* , *done1* , *done2* ;
var j, k, l: *integer*; { all-purpose indices }
p, q: *pointer*; { all-purpose pointers }
x: *integer*; { something to dump }
format_engine: ↑*text_char*;
begin ⟨ If dumping is not allowed, abort 1480 ⟩;
⟨ Create the *format_ident*, open the format file, and inform the user that dumping has begun 1506 ⟩;
⟨ Dump constants for consistency check 1483* ⟩;
⟨ Dump MLTEX-specific data 1890* ⟩;
⟨ Dump encTEX-specific data 1899* ⟩;
⟨ Dump the string pool 1485* ⟩;
⟨ Dump the dynamic memory 1487* ⟩;
⟨ Dump the table of equivalents 1489 ⟩;
⟨ Dump the font information 1496* ⟩;
⟨ Dump the hyphenation tables 1500* ⟩;
⟨ Dump pdftex data 1502 ⟩;
⟨ Dump a couple more things and the closing check word 1504 ⟩;
⟨ Close the format file 1507 ⟩;
end;
tini

1479*. Corresponding to the procedure that dumps a format file, we have a function that reads one in. The function returns *false* if the dumped format is incompatible with the present TEX table sizes, etc.

define *bad_fmt* = 6666 { go here if the format file is unacceptable }
define *too_small*(#) ≡
  begin *wake_up_terminal*; *wterm_ln*(´---!␣Must␣increase␣the␣´, #); **goto** *bad_fmt*;
  end

⟨ Declare the function called *open_fmt_file* 550* ⟩
function *load_fmt_file*: *boolean*;
  label *bad_fmt*, *exit*;
  var *j*, *k*: *integer*; { all-purpose indices }
    p, *q*: *pointer*; { all-purpose pointers }
    x: *integer*; { something undumped }
    format_engine: ↑*text_char*; *dummy_xord*: *ASCII_code*; *dummy_xchr*: *text_char*;
    dummy_xprn: *ASCII_code*;
  begin ⟨ Undump constants for consistency check 1484* ⟩;
  ⟨ Undump MLTEX-specific data 1891* ⟩;
  ⟨ Undump encTEX-specific data 1900* ⟩;
  ⟨ Undump the string pool 1486* ⟩;
  ⟨ Undump the dynamic memory 1488* ⟩;
  ⟨ Undump the table of equivalents 1490* ⟩;
  ⟨ Undump the font information 1497* ⟩;
  ⟨ Undump the hyphenation tables 1501* ⟩;
  ⟨ Undump pdftex data 1503 ⟩;
  ⟨ Undump a couple more things and the closing check word 1505* ⟩;
  prev_depth ← *pdf_ignored_dimen*; *load_fmt_file* ← *true*; **return**; { it worked! }
bad_fmt: *wake_up_terminal*; *wterm_ln*(´(Fatal␣format␣file␣error;␣I´´m␣stymied)´);
  load_fmt_file ← *false*;
exit: **end**;

1481*. Format files consist of *memory_word* items, and we use the following macros to dump words of different types:

⟨ Global variables 13 ⟩ +≡
fmt_file: *word_file*; { for input or output of format information }

1482* The inverse macros are slightly more complicated, since we need to check the range of the values we are reading in. We say '*undump*(a)(b)(x)' to read an integer value x that is supposed to be in the range $a \le x \le b$. System error messages should be suppressed when undumping.

define *undump_end_end*(#) ≡ # ← x; **end**
define *undump_end*(#) ≡ (x > #) **then goto** *bad_fmt* **else** *undump_end_end*
define *undump*(#) ≡
  begin *undump_int*(x);
  if (x < #) ∨ *undump_end*
define *format_debug_end*(#) ≡ *write_ln*(*stderr*, ´␣=␣´, #);
  end ;
define *format_debug*(#) ≡
  if *debug_format_file* **then**
   begin *write*(*stderr*, ´fmtdebug:´, #); *format_debug_end*
define *undump_size_end_end*(#) ≡ *too_small*(#) **else** *format_debug*(#)(x); *undump_end_end*
define *undump_size_end*(#) ≡
   if x > # **then** *undump_size_end_end*
define *undump_size*(#) ≡
  begin *undump_int*(x);
  if x < # **then goto** *bad_fmt*;
  undump_size_end

1483* The next few sections of the program should make it clear how we use the dump/undump macros.

⟨ Dump constants for consistency check 1483* ⟩ ≡
dump_int("57325458); { Web2C TEX's magic constant: "W2TX" }
 { Align engine to 4 bytes with one or more trailing NUL }
x ← *strlen*(*engine_name*); *format_engine* ← *xmalloc_array*(*text_char*, $x + 4$);
strcpy(*stringcast*(*format_engine*), *engine_name*);
for k ← x **to** $x + 3$ **do** *format_engine*[k] ← 0;
x ← $x + 4 - (x \bmod 4)$; *dump_int*(x); *dump_things*(*format_engine*[0], x); *libc_free*(*format_engine*);
dump_int(@$);
⟨ Dump *xord*, *xchr*, and *xprn* 1871* ⟩;
dump_int(*max_halfword*);
dump_int(*hash_high*); ⟨ Dump the ε-TEX state 1651 ⟩
dump_int(*mem_bot*);
dump_int(*mem_top*);
dump_int(*eqtb_size*);
dump_int(*hash_prime*);
dump_int(*hyph_prime*)

This code is used in section 1478*.

1484* Sections of a WEB program that are "commented out" still contribute strings to the string pool; therefore INITEX and TEX will have the same strings. (And it is, of course, a good thing that they do.)

⟨Undump constants for consistency check 1484*⟩ ≡ **Init** *libc_free*(*font_info*); *libc_free*(*str_pool*);
  libc_free(*str_start*); *libc_free*(*yhash*); *libc_free*(*zeqtb*); *libc_free*(*yzmem*); **Tini** *undump_int*(x);
  format_debug(´format␣magic␣number´)(x);
  if $x \neq$ "57325458 **then goto** *bad_fmt*; { not a format file }
  undump_int(x); *format_debug*(´engine␣name␣size´)(x);
  if $(x < 0) \lor (x > 256)$ **then goto** *bad_fmt*; { corrupted format file }
  format_engine ← *xmalloc_array*(*text_char*, x); *undump_things*(*format_engine*[0], x);
  format_engine[$x - 1$] ← 0; { force string termination, just in case }
  if *strcmp*(*engine_name*, *stringcast*(*format_engine*)) **then**
    begin *wake_up_terminal*;
    wterm_ln(´---!␣´, *stringcast*(*name_of_file* + 1), ´␣was␣written␣by␣´, *format_engine*);
    libc_free(*format_engine*); **goto** *bad_fmt*;
    end;
  libc_free(*format_engine*); *undump_int*(x); *format_debug*(´string␣pool␣checksum´)(x);
  if $x \neq$ @$ **then**
    begin { check that strings are the same }
    wake_up_terminal;
    wterm_ln(´---!␣´, *stringcast*(*name_of_file* + 1), ´␣made␣by␣different␣executable␣version´);
    goto *bad_fmt*;
    end;
  ⟨Undump *xord*, *xchr*, and *xprn* 1872*⟩;
  undump_int(x);
  if $x \neq$ *max_halfword* **then goto** *bad_fmt*; { check *max_halfword* }
  undump_int(*hash_high*);
  if (*hash_high* < 0) ∨ (*hash_high* > *sup_hash_extra*) **then goto** *bad_fmt*;
  if *hash_extra* < *hash_high* **then** *hash_extra* ← *hash_high*;
  eqtb_top ← *eqtb_size* + *hash_extra*;
  if *hash_extra* = 0 **then** *hash_top* ← *undefined_control_sequence*
  else *hash_top* ← *eqtb_top*;
  yhash ← *xmalloc_array*(*two_halves*, 1 + *hash_top* − *hash_offset*); *hash* ← *yhash* − *hash_offset*;
  next(*hash_base*) ← 0; *text*(*hash_base*) ← 0;
  for x ← *hash_base* + 1 **to** *hash_top* **do** *hash*[x] ← *hash*[*hash_base*];
  zeqtb ← *xmalloc_array*(*memory_word*, *eqtb_top* + 1); *eqtb* ← *zeqtb*;
  eq_type(*undefined_control_sequence*) ← *undefined_cs*; *equiv*(*undefined_control_sequence*) ← *null*;
  eq_level(*undefined_control_sequence*) ← *level_zero*;
  for x ← *eqtb_size* + 1 **to** *eqtb_top* **do** *eqtb*[x] ← *eqtb*[*undefined_control_sequence*];
  ⟨Undump the ε-TEX state 1652⟩
  undump_int(x); *format_debug*(´mem_bot´)(x);
  if $x \neq$ *mem_bot* **then goto** *bad_fmt*;
  undump_int(*mem_top*); *format_debug*(´mem_top´)(*mem_top*);
  if *mem_bot* + 1100 > *mem_top* **then goto** *bad_fmt*;
  head ← *contrib_head*; *tail* ← *contrib_head*; *page_tail* ← *page_head*; { page initialization }
  mem_min ← *mem_bot* − *extra_mem_bot*; *mem_max* ← *mem_top* + *extra_mem_top*;
  yzmem ← *xmalloc_array*(*memory_word*, *mem_max* − *mem_min* + 1); *zmem* ← *yzmem* − *mem_min*;
    { this pointer arithmetic fails with some compilers }
  mem ← *zmem*; *undump_int*(x);
  if $x \neq$ *eqtb_size* **then goto** *bad_fmt*;
  undump_int(x);
  if $x \neq$ *hash_prime* **then goto** *bad_fmt*;
  undump_int(x);

if $x \neq hyph\_prime$ **then goto** bad_fmt

This code is used in section 1479*.

1485*. **define** $dump\_four\_ASCII \equiv w.b0 \leftarrow qi(so(str\_pool[k]))$; $w.b1 \leftarrow qi(so(str\_pool[k+1]))$; $w.b2 \leftarrow qi(so(str\_pool[k+2]))$; $w.b3 \leftarrow qi(so(str\_pool[k+3]))$; $dump\_qqqq(w)$

⟨Dump the string pool 1485*⟩ ≡
$dump\_int(pool\_ptr)$; $dump\_int(str\_ptr)$; $dump\_things(str\_start[0], str\_ptr+1)$;
$dump\_things(str\_pool[0], pool\_ptr)$; $print\_ln$; $print\_int(str\_ptr)$; $print($"␣strings␣of␣total␣length␣"$)$;
$print\_int(pool\_ptr)$

This code is used in section 1478*.

1486*. **define** $undump\_four\_ASCII \equiv undump\_qqqq(w)$; $str\_pool[k] \leftarrow si(qo(w.b0))$; $str\_pool[k+1] \leftarrow si(qo(w.b1))$; $str\_pool[k+2] \leftarrow si(qo(w.b2))$; $str\_pool[k+3] \leftarrow si(qo(w.b3))$

⟨Undump the string pool 1486*⟩ ≡
$undump\_size(0)(sup\_pool\_size - pool\_free)$('string␣pool␣size'$)(pool\_ptr)$;
if $pool\_size < pool\_ptr + pool\_free$ **then** $pool\_size \leftarrow pool\_ptr + pool\_free$;
$undump\_size(0)(sup\_max\_strings - strings\_free)$('sup␣strings'$)(str\_ptr)$;
if $max\_strings < str\_ptr + strings\_free$ **then** $max\_strings \leftarrow str\_ptr + strings\_free$;
$str\_start \leftarrow xmalloc\_array(pool\_pointer, max\_strings)$;
$undump\_checked\_things(0, pool\_ptr, str\_start[0], str\_ptr+1)$;
$str\_pool \leftarrow xmalloc\_array(packed\_ASCII\_code, pool\_size)$; $undump\_things(str\_pool[0], pool\_ptr)$;
$init\_str\_ptr \leftarrow str\_ptr$; $init\_pool\_ptr \leftarrow pool\_ptr$

This code is used in section 1479*.

1487*. By sorting the list of available spaces in the variable-size portion of *mem*, we are usually able to get by without having to dump very much of the dynamic memory.

We recompute *var_used* and *dyn_used*, so that INITEX dumps valid information even when it has not been gathering statistics.

⟨Dump the dynamic memory 1487*⟩ ≡
$sort\_avail$; $var\_used \leftarrow 0$; $dump\_int(lo\_mem\_max)$; $dump\_int(rover)$;
if $eTeX\_ex$ **then**
  for $k \leftarrow int\_val$ **to** tok_val **do** $dump\_int(sa\_root[k])$;
$p \leftarrow mem\_bot$; $q \leftarrow rover$; $x \leftarrow 0$;
repeat $dump\_things(mem[p], q+2-p)$; $x \leftarrow x+q+2-p$; $var\_used \leftarrow var\_used + q - p$;
  $p \leftarrow q + node\_size(q)$; $q \leftarrow rlink(q)$;
until $q = rover$;
$var\_used \leftarrow var\_used + lo\_mem\_max - p$; $dyn\_used \leftarrow mem\_end + 1 - hi\_mem\_min$;
$dump\_things(mem[p], lo\_mem\_max + 1 - p)$; $x \leftarrow x + lo\_mem\_max + 1 - p$; $dump\_int(hi\_mem\_min)$;
$dump\_int(avail)$; $dump\_things(mem[hi\_mem\_min], mem\_end + 1 - hi\_mem\_min)$;
$x \leftarrow x + mem\_end + 1 - hi\_mem\_min$; $p \leftarrow avail$;
while $p \neq null$ **do**
  begin $decr(dyn\_used)$; $p \leftarrow link(p)$;
  end;
$dump\_int(var\_used)$; $dump\_int(dyn\_used)$; $print\_ln$; $print\_int(x)$;
$print($"␣memory␣locations␣dumped;␣current␣usage␣is␣"$)$; $print\_int(var\_used)$; $print\_char($"&"$)$;
$print\_int(dyn\_used)$

This code is used in section 1478*.

1488* ⟨Undump the dynamic memory 1488*⟩ ≡
undump(*lo_mem_stat_max* + 1000)(*hi_mem_stat_min* − 1)(*lo_mem_max*);
undump(*lo_mem_stat_max* + 1)(*lo_mem_max*)(*rover*);
if *eTeX_ex* **then**
  for $k \leftarrow$ *int_val* **to** *tok_val* **do** *undump*(*null*)(*lo_mem_max*)(*sa_root*[k]);
$p \leftarrow$ *mem_bot*; $q \leftarrow$ *rover*;
repeat *undump_things*(*mem*[p], $q + 2 - p$); $p \leftarrow q +$ *node_size*(q);
  if ($p >$ *lo_mem_max*) ∨ (($q \geq$ *rlink*(q)) ∧ (*rlink*(q) ≠ *rover*)) **then goto** *bad_fmt*;
  $q \leftarrow$ *rlink*(q);
until $q =$ *rover*;
undump_things(*mem*[p], *lo_mem_max* + 1 − p);
if *mem_min* < *mem_bot* − 2 **then** { make more low memory available }
  begin $p \leftarrow$ *llink*(*rover*); $q \leftarrow$ *mem_min* + 1; *link*(*mem_min*) ← *null*; *info*(*mem_min*) ← *null*;
    { we don't use the bottom word }
  rlink(p) ← q; *llink*(*rover*) ← q;
  rlink(q) ← *rover*; *llink*(q) ← p; *link*(q) ← *empty_flag*; *node_size*(q) ← *mem_bot* − q;
  end;
undump(*lo_mem_max* + 1)(*hi_mem_stat_min*)(*hi_mem_min*); *undump*(*null*)(*mem_top*)(*avail*);
mem_end ← *mem_top*; *undump_things*(*mem*[*hi_mem_min*], *mem_end* + 1 − *hi_mem_min*);
undump_int(*var_used*); *undump_int*(*dyn_used*)

This code is used in section 1479*.

1490* ⟨Undump the table of equivalents 1490*⟩ ≡
⟨Undump regions 1 to 6 of *eqtb* 1493*⟩;
undump(*hash_base*)(*hash_top*)(*par_loc*); *par_token* ← *cs_token_flag* + *par_loc*;
undump(*hash_base*)(*hash_top*)(*write_loc*);
⟨Undump the hash table 1495*⟩

This code is used in section 1479*.

1491* The table of equivalents usually contains repeated information, so we dump it in compressed form: The sequence of $n+2$ values $(n, x_1, \ldots, x_n, m)$ in the format file represents $n+m$ consecutive entries of *eqtb*, with m extra copies of x_n, namely $(x_1, \ldots, x_n, x_n, \ldots, x_n)$.

⟨Dump regions 1 to 4 of *eqtb* 1491*⟩ ≡
$k \leftarrow$ *active_base*;
repeat $j \leftarrow k$;
  while $j <$ *int_base* − 1 **do**
    begin if (*equiv*(j) = *equiv*($j + 1$)) ∧ (*eq_type*(j) = *eq_type*($j + 1$)) ∧ (*eq_level*(j) = *eq_level*($j + 1$))
      then goto *found1*;
    incr(j);
    end;
  $l \leftarrow$ *int_base*; **goto** *done1*; { $j =$ *int_base* − 1 }
found1: *incr*(j); $l \leftarrow j$;
  while $j <$ *int_base* − 1 **do**
    begin if (*equiv*(j) ≠ *equiv*($j + 1$)) ∨ (*eq_type*(j) ≠ *eq_type*($j + 1$)) ∨ (*eq_level*(j) ≠ *eq_level*($j + 1$))
      then goto *done1*;
    incr(j);
    end;
done1: *dump_int*($l - k$); *dump_things*(*eqtb*[k], $l - k$); $k \leftarrow j + 1$; *dump_int*($k - l$);
until $k =$ *int_base*

This code is used in section 1489.

1492*. ⟨ Dump regions 5 and 6 of *eqtb* 1492* ⟩ ≡

repeat $j \leftarrow k$;
  while $j < eqtb\_size$ **do**
    begin if $eqtb[j].int = eqtb[j+1].int$ **then goto** *found2*;
    $incr(j)$;
    end;
  $l \leftarrow eqtb\_size + 1$; **goto** *done2*; { $j = eqtb\_size$ }
found2: $incr(j)$; $l \leftarrow j$;
  while $j < eqtb\_size$ **do**
    begin if $eqtb[j].int \neq eqtb[j+1].int$ **then goto** *done2*;
    $incr(j)$;
    end;
done2: $dump\_int(l - k)$; $dump\_things(eqtb[k], l - k)$; $k \leftarrow j + 1$; $dump\_int(k - l)$;
until $k > eqtb\_size$;
if $hash\_high > 0$ **then** $dump\_things(eqtb[eqtb\_size + 1], hash\_high)$; { dump *hash_extra* part }

This code is used in section 1489.

1493*. ⟨ Undump regions 1 to 6 of *eqtb* 1493* ⟩ ≡

$k \leftarrow active\_base$;
repeat $undump\_int(x)$;
  if $(x < 1) \lor (k + x > eqtb\_size + 1)$ **then goto** *bad_fmt*;
  $undump\_things(eqtb[k], x)$; $k \leftarrow k + x$; $undump\_int(x)$;
  if $(x < 0) \lor (k + x > eqtb\_size + 1)$ **then goto** *bad_fmt*;
  for $j \leftarrow k$ **to** $k + x - 1$ **do** $eqtb[j] \leftarrow eqtb[k - 1]$;
  $k \leftarrow k + x$;
until $k > eqtb\_size$;
if $hash\_high > 0$ **then** $undump\_things(eqtb[eqtb\_size + 1], hash\_high)$; { undump *hash_extra* part }

This code is used in section 1490*.

1494*. A different scheme is used to compress the hash table, since its lower region is usually sparse. When $text(p) \neq 0$ for $p \leq hash\_used$, we output two words, p and $hash[p]$. The hash table is, of course, densely packed for $p \geq hash\_used$, so the remaining entries are output in a block.

⟨ Dump the hash table 1494* ⟩ ≡

for $p \leftarrow 0$ **to** $prim\_size$ **do** $dump\_hh(prim[p])$;
$dump\_int(hash\_used)$; $cs\_count \leftarrow frozen\_control\_sequence - 1 - hash\_used + hash\_high$;
for $p \leftarrow hash\_base$ **to** $hash\_used$ **do**
  if $text(p) \neq 0$ **then**
    begin $dump\_int(p)$; $dump\_hh(hash[p])$; $incr(cs\_count)$;
    end;
$dump\_things(hash[hash\_used + 1], undefined\_control\_sequence - 1 - hash\_used)$;
if $hash\_high > 0$ **then** $dump\_things(hash[eqtb\_size + 1], hash\_high)$;
$dump\_int(cs\_count)$;
print_ln; $print\_int(cs\_count)$; *print*("␣multiletter␣control␣sequences")

This code is used in section 1489.

1495* ⟨Undump the hash table 1495*⟩ ≡
for $p \leftarrow 0$ **to** *prim_size* **do** *undump_hh*(*prim*[p]);
undump(*hash_base*)(*frozen_control_sequence*)(*hash_used*); $p \leftarrow hash\_base - 1$;
repeat *undump*($p + 1$)(*hash_used*)(p); *undump_hh*(*hash*[p]);
until $p = hash\_used$;
undump_things(*hash*[*hash_used* + 1], *undefined_control_sequence* − 1 − *hash_used*);
if *debug_format_file* **then**
begin *print_csnames*(*hash_base*, *undefined_control_sequence* − 1);
end;
if *hash_high* > 0 **then**
begin *undump_things*(*hash*[*eqtb_size* + 1], *hash_high*);
if *debug_format_file* **then**
begin *print_csnames*(*eqtb_size* + 1, *hash_high* − (*eqtb_size* + 1));
end;
end;
undump_int(*cs_count*)

This code is used in section 1490*.

1496* ⟨Dump the font information 1496*⟩ ≡
dump_int(*fmem_ptr*); *dump_things*(*font_info*[0], *fmem_ptr*); *dump_int*(*font_ptr*);
⟨Dump the array info for internal font number k 1498*⟩;
print_ln; *print_int*(*fmem_ptr* − 7); *print*("␣words␣of␣font␣info␣for␣");
print_int(*font_ptr* − *font_base*);
if *font_ptr* ≠ *font_base* + 1 **then** *print*("␣preloaded␣fonts")
else *print*("␣preloaded␣font")

This code is used in section 1478*.

1497* ⟨Undump the font information 1497*⟩ ≡
undump_size(7)(*sup_font_mem_size*)('font␣mem␣size')(*fmem_ptr*);
if *fmem_ptr* > *font_mem_size* **then** *font_mem_size* ← *fmem_ptr*;
font_info ← *xmalloc_array*(*fmemory_word*, *font_mem_size*); *undump_things*(*font_info*[0], *fmem_ptr*);
undump_size(*font_base*)(*font_base* + *max_font_max*)('font␣max')(*font_ptr*);
{ This undumps all of the font info, despite the name. }
⟨Undump the array info for internal font number k 1499*⟩;

This code is used in section 1479*.

1498* ⟨ Dump the array info for internal font number k 1498* ⟩ ≡

begin *dump_things*(*font_check*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_size*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_dsize*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_params*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*hyphen_char*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*skew_char*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_name*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_area*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_bc*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_ec*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*char_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*width_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*height_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*depth_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*italic_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*lig_kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*exten_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*param_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_glue*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*bchar_label*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_false_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
for k ← *null_font* **to** *font_ptr* **do**
  begin *print_nl*("\font"); *print_esc*(*font_id_text*(k)); *print_char*("=");
  print_file_name(*font_name*[k], *font_area*[k], "");
  if *font_size*[k] ≠ *font_dsize*[k] **then**
    begin *print*("␣at␣"); *print_scaled*(*font_size*[k]); *print*("pt");
    end;
  end;
end

This code is used in section 1496*.

1499*. This module should now be named 'Undump all the font arrays'.

⟨ Undump the array info for internal font number k 1499* ⟩ ≡
begin { Allocate the font arrays }
font_check ← *xmalloc_array*(*four_quarters*, *font_max*); *font_size* ← *xmalloc_array*(*scaled*, *font_max*);
font_dsize ← *xmalloc_array*(*scaled*, *font_max*); *font_params* ← *xmalloc_array*(*font_index*, *font_max*);
font_name ← *xmalloc_array*(*str_number*, *font_max*); *font_area* ← *xmalloc_array*(*str_number*, *font_max*);
font_bc ← *xmalloc_array*(*eight_bits*, *font_max*); *font_ec* ← *xmalloc_array*(*eight_bits*, *font_max*);
font_glue ← *xmalloc_array*(*halfword*, *font_max*); *hyphen_char* ← *xmalloc_array*(*integer*, *font_max*);
skew_char ← *xmalloc_array*(*integer*, *font_max*); *bchar_label* ← *xmalloc_array*(*font_index*, *font_max*);
font_bchar ← *xmalloc_array*(*nine_bits*, *font_max*); *font_false_bchar* ← *xmalloc_array*(*nine_bits*, *font_max*);
char_base ← *xmalloc_array*(*integer*, *font_max*); *width_base* ← *xmalloc_array*(*integer*, *font_max*);
height_base ← *xmalloc_array*(*integer*, *font_max*); *depth_base* ← *xmalloc_array*(*integer*, *font_max*);
italic_base ← *xmalloc_array*(*integer*, *font_max*); *lig_kern_base* ← *xmalloc_array*(*integer*, *font_max*);
kern_base ← *xmalloc_array*(*integer*, *font_max*); *exten_base* ← *xmalloc_array*(*integer*, *font_max*);
param_base ← *xmalloc_array*(*integer*, *font_max*);
pdf_char_used ← *xmalloc_array*(*char_used_array*, *font_max*);
pdf_font_size ← *xmalloc_array*(*scaled*, *font_max*); *pdf_font_num* ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_map ← *xmalloc_array*(*fm_entry_ptr*, *font_max*);
pdf_font_type ← *xmalloc_array*(*eight_bits*, *font_max*);
pdf_font_attr ← *xmalloc_array*(*str_number*, *font_max*);
pdf_font_blink ← *xmalloc_array*(*internal_font_number*, *font_max*);
pdf_font_elink ← *xmalloc_array*(*internal_font_number*, *font_max*);
pdf_font_stretch ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_shrink ← *xmalloc_array*(*integer*, *font_max*); *pdf_font_step* ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_expand_ratio ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_auto_expand ← *xmalloc_array*(*boolean*, *font_max*);
pdf_font_lp_base ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_rp_base ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_ef_base ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_kn_bs_base ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_st_bs_base ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_sh_bs_base ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_kn_bc_base ← *xmalloc_array*(*integer*, *font_max*);
pdf_font_kn_ac_base ← *xmalloc_array*(*integer*, *font_max*);
vf_packet_base ← *xmalloc_array*(*integer*, *font_max*);
vf_default_font ← *xmalloc_array*(*internal_font_number*, *font_max*);
vf_local_font_num ← *xmalloc_array*(*internal_font_number*, *font_max*);
vf_e_fnts ← *xmalloc_array*(*integer*, *font_max*);
vf_i_fnts ← *xmalloc_array*(*internal_font_number*, *font_max*);
pdf_font_nobuiltin_tounicode ← *xmalloc_array*(*boolean*, *font_max*);
for *font_k* ← *font_base* **to** *font_max* **do**
  begin for $k \leftarrow 0$ **to** 31 **do** *pdf_char_used*[*font_k*, *k*] ← 0;
  pdf_font_size[*font_k*] ← 0; *pdf_font_num*[*font_k*] ← 0; *pdf_font_map*[*font_k*] ← 0;
  pdf_font_type[*font_k*] ← *new_font_type*; *pdf_font_attr*[*font_k*] ← ""; *pdf_font_blink*[*font_k*] ← *null_font*;
  pdf_font_elink[*font_k*] ← *null_font*; *pdf_font_stretch*[*font_k*] ← *null_font*;
  pdf_font_shrink[*font_k*] ← *null_font*; *pdf_font_step*[*font_k*] ← 0; *pdf_font_expand_ratio*[*font_k*] ← 0;
  pdf_font_auto_expand[*font_k*] ← *false*; *pdf_font_lp_base*[*font_k*] ← 0; *pdf_font_rp_base*[*font_k*] ← 0;
  pdf_font_ef_base[*font_k*] ← 0; *pdf_font_kn_bs_base*[*font_k*] ← 0; *pdf_font_st_bs_base*[*font_k*] ← 0;
  pdf_font_sh_bs_base[*font_k*] ← 0; *pdf_font_kn_bc_base*[*font_k*] ← 0; *pdf_font_kn_ac_base*[*font_k*] ← 0;
  pdf_font_nobuiltin_tounicode[*font_k*] ← *false*;
  end;
make_pdftex_banner; *undump_things*(*font_check*[*null_font*], *font_ptr* + 1 − *null_font*);

undump_things(*font_size*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*font_dsize*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(*min_halfword*, *max_halfword*, *font_params*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*hyphen_char*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*skew_char*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_upper_check_things(*str_ptr*, *font_name*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_upper_check_things(*str_ptr*, *font_area*[*null_font*], *font_ptr* + 1 − *null_font*); { There's no point in checking these values against the range [0, 255], since the data type is *unsigned char*, and all values of that type are in that range by definition. }
undump_things(*font_bc*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*font_ec*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*char_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*width_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*height_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*depth_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*italic_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*lig_kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*exten_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*param_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(*min_halfword*, *lo_mem_max*, *font_glue*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(0, *fmem_ptr* − 1, *bchar_label*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(*min_quarterword*, *non_char*, *font_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(*min_quarterword*, *non_char*, *font_false_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
end

This code is used in section 1497*.

1500* ⟨ Dump the hyphenation tables 1500* ⟩ ≡

```
dump_int(hyph_count);
if hyph_next ≤ hyph_prime then hyph_next ← hyph_size;
dump_int(hyph_next); { minimum value of hyphen_size needed }
for k ← 0 to hyph_size do
  if hyph_word[k] ≠ 0 then
    begin dump_int(k + 65536 ∗ hyph_link[k]);
        { assumes number of hyphen exceptions does not exceed 65535 }
    dump_int(hyph_word[k]); dump_int(hyph_list[k]);
    end;
print_ln; print_int(hyph_count);
if hyph_count ≠ 1 then print("␣hyphenation␣exceptions")
else print("␣hyphenation␣exception");
if trie_not_ready then init_trie;
dump_int(trie_max); dump_int(hyph_start); dump_things(trie_trl[0], trie_max + 1);
dump_things(trie_tro[0], trie_max + 1); dump_things(trie_trc[0], trie_max + 1); dump_int(trie_op_ptr);
dump_things(hyf_distance[1], trie_op_ptr); dump_things(hyf_num[1], trie_op_ptr);
dump_things(hyf_next[1], trie_op_ptr); print_nl("Hyphenation␣trie␣of␣length␣"); print_int(trie_max);
print("␣has␣"); print_int(trie_op_ptr);
if trie_op_ptr ≠ 1 then print("␣ops")
else print("␣op");
print("␣out␣of␣"); print_int(trie_op_size);
for k ← 255 downto 0 do
  if trie_used[k] > min_quarterword then
    begin print_nl("␣␣"); print_int(qo(trie_used[k])); print("␣for␣language␣"); print_int(k);
    dump_int(k); dump_int(qo(trie_used[k]));
    end
```

This code is used in section 1478*.

1501* Only "nonempty" parts of *op_start* need to be restored.

⟨Undump the hyphenation tables 1501*⟩ ≡
undump_size(0)(*hyph_size*)(´hyph_size´)(*hyph_count*);
undump_size(*hyph_prime*)(*hyph_size*)(´hyph_size´)(*hyph_next*); $j \leftarrow 0$;
for $k \leftarrow 1$ **to** *hyph_count* **do**
  begin *undump_int*(j);
  if $j < 0$ **then goto** *bad_fmt*;
  if $j > 65535$ **then**
    begin *hyph_next* ← j **div** 65536; $j \leftarrow j -$ *hyph_next* ∗ 65536;
    end
  else *hyph_next* ← 0;
  if ($j \geq$ *hyph_size*) ∨ (*hyph_next* > *hyph_size*) **then goto** *bad_fmt*;
  hyph_link[j] ← *hyph_next*; *undump*(0)(*str_ptr*)(*hyph_word*[j]);
  undump(*min_halfword*)(*max_halfword*)(*hyph_list*[j]);
  end; { j is now the largest occupied location in *hyph_word* }
incr(j);
if $j <$ *hyph_prime* **then** $j \leftarrow$ *hyph_prime*;
hyph_next ← j;
if *hyph_next* ≥ *hyph_size* **then** *hyph_next* ← *hyph_prime*
else if *hyph_next* ≥ *hyph_prime* **then** *incr*(*hyph_next*);
undump_size(0)(*trie_size*)(´trie␣size´)(j); **init** *trie_max* ← j; **tini** *undump*(0)(j)(*hyph_start*);
  { These first three haven't been allocated yet unless we're INITEX; we do that precisely so we don't allocate more space than necessary. }
if ¬*trie_trl* **then** *trie_trl* ← *xmalloc_array*(*trie_pointer*, $j + 1$);
undump_things(*trie_trl*[0], $j + 1$);
if ¬*trie_tro* **then** *trie_tro* ← *xmalloc_array*(*trie_pointer*, $j + 1$);
undump_things(*trie_tro*[0], $j + 1$);
if ¬*trie_trc* **then** *trie_trc* ← *xmalloc_array*(*quarterword*, $j + 1$);
undump_things(*trie_trc*[0], $j + 1$);
undump_size(0)(*trie_op_size*)(´trie␣op␣size´)(j); **init** *trie_op_ptr* ← j; **tini**
  { I'm not sure we have such a strict limitation (64) on these values, so let's leave them unchecked. }
undump_things(*hyf_distance*[1], j); *undump_things*(*hyf_num*[1], j);
undump_upper_check_things(*max_trie_op*, *hyf_next*[1], j);
init for $k \leftarrow 0$ **to** 255 **do** *trie_used*[k] ← *min_quarterword*;
tini
$k \leftarrow 256$;
while $j > 0$ **do**
  begin *undump*(0)($k - 1$)(k); *undump*(1)(j)(x); **init** *trie_used*[k] ← *qi*(x); **tini**
  $j \leftarrow j - x$; *op_start*[k] ← *qo*(j);
  end;
init *trie_not_ready* ← *false* **tini**

This code is used in section 1479*.

1505* ⟨Undump a couple more things and the closing check word 1505*⟩ ≡
undump(*batch_mode*)(*error_stop_mode*)(*interaction*);
if *interaction_option* ≠ *unspecified_mode* **then** *interaction* ← *interaction_option*;
undump(0)(*str_ptr*)(*format_ident*); *undump_int*(x);
if $x \neq 69069$ **then goto** *bad_fmt*

This code is used in section 1479*.

1510.* Now this is really it: TEX starts and ends here.

The initial test involving *ready_already* should be deleted if the Pascal runtime system is smart enough to detect such a "mistake."

```
define const_chk(#) ≡
        begin if # < inf@&# then # ← inf@&#
        else if # > sup@&# then # ← sup@&#
        end  { setup_bound_var stuff duplicated in mf.ch. }
define setup_bound_var(#) ≡ bound_default ← #; setup_bound_var_end
define setup_bound_var_end(#) ≡ bound_name ← #; setup_bound_var_end_end
define setup_bound_var_end_end(#) ≡ setup_bound_variable(addressof(#), bound_name, bound_default)

procedure main_body;
  begin  { start_here }
    { Bounds that may be set from the configuration file. We want the user to be able to specify the names
      with underscores, but TANGLE removes underscores, so we're stuck giving the names twice, once as a
      string, once as the identifier. How ugly. }
  setup_bound_var(0)('mem_bot')(mem_bot); setup_bound_var(250000)('main_memory')(main_memory);
      { memory_words for mem in INITEX }
  setup_bound_var(0)('extra_mem_top')(extra_mem_top);  { increase high mem in VIRTEX }
  setup_bound_var(0)('extra_mem_bot')(extra_mem_bot);  { increase low mem in VIRTEX }
  setup_bound_var(200000)('pool_size')(pool_size);
  setup_bound_var(75000)('string_vacancies')(string_vacancies);
  setup_bound_var(5000)('pool_free')(pool_free);  { min pool avail after fmt }
  setup_bound_var(15000)('max_strings')(max_strings);
  setup_bound_var(100)('strings_free')(strings_free);
  setup_bound_var(100000)('font_mem_size')(font_mem_size);
  setup_bound_var(500)('font_max')(font_max); setup_bound_var(20000)('trie_size')(trie_size);
      { if ssup_trie_size increases, recompile }
  setup_bound_var(659)('hyph_size')(hyph_size); setup_bound_var(3000)('buf_size')(buf_size);
  setup_bound_var(50)('nest_size')(nest_size); setup_bound_var(15)('max_in_open')(max_in_open);
  setup_bound_var(60)('param_size')(param_size); setup_bound_var(4000)('save_size')(save_size);
  setup_bound_var(300)('stack_size')(stack_size);
  setup_bound_var(16384)('dvi_buf_size')(dvi_buf_size); setup_bound_var(79)('error_line')(error_line);
  setup_bound_var(50)('half_error_line')(half_error_line);
  setup_bound_var(79)('max_print_line')(max_print_line);
  setup_bound_var(0)('hash_extra')(hash_extra);
  setup_bound_var(10000)('expand_depth')(expand_depth); setup_bound_var(72)('pk_dpi')(pk_dpi);
  const_chk(mem_bot); const_chk(main_memory); Init extra_mem_top ← 0; extra_mem_bot ← 0; Tini
  if extra_mem_bot > sup_main_memory then extra_mem_bot ← sup_main_memory;
  if extra_mem_top > sup_main_memory then extra_mem_top ← sup_main_memory;
        { mem_top is an index, main_memory a size }
  mem_top ← mem_bot + main_memory − 1; mem_min ← mem_bot; mem_max ← mem_top;
      { Check other constants against their sup and inf. }
  const_chk(trie_size); const_chk(hyph_size); const_chk(buf_size); const_chk(nest_size);
  const_chk(max_in_open); const_chk(param_size); const_chk(save_size); const_chk(stack_size);
  const_chk(dvi_buf_size); const_chk(pool_size); const_chk(string_vacancies); const_chk(pool_free);
  const_chk(max_strings); const_chk(strings_free); const_chk(font_mem_size); const_chk(font_max);
  const_chk(hash_extra); const_chk(obj_tab_size); const_chk(pdf_mem_size); const_chk(dest_names_size);
  const_chk(pk_dpi);
  if error_line > ssup_error_line then error_line ← ssup_error_line;  { array memory allocation }
  buffer ← xmalloc_array(ASCII_code, buf_size); nest ← xmalloc_array(list_state_record, nest_size);
  save_stack ← xmalloc_array(memory_word, save_size);
  input_stack ← xmalloc_array(in_state_record, stack_size);
```

```
input_file ← xmalloc_array(alpha_file, max_in_open); line_stack ← xmalloc_array(integer, max_in_open);
eof_seen ← xmalloc_array(boolean, max_in_open); grp_stack ← xmalloc_array(save_pointer, max_in_open);
if_stack ← xmalloc_array(pointer, max_in_open);
source_filename_stack ← xmalloc_array(str_number, max_in_open);
full_source_filename_stack ← xmalloc_array(str_number, max_in_open);
param_stack ← xmalloc_array(halfword, param_size); dvi_buf ← xmalloc_array(eight_bits, dvi_buf_size);
hyph_word ← xmalloc_array(str_number, hyph_size); hyph_list ← xmalloc_array(halfword, hyph_size);
hyph_link ← xmalloc_array(hyph_pointer, hyph_size);
obj_tab ← xmalloc_array(obj_entry, inf_obj_tab_size); { will grow dynamically }
pdf_mem ← xmalloc_array(integer, inf_pdf_mem_size); { will grow dynamically }
dest_names ← xmalloc_array(dest_name_entry, inf_dest_names_size); { will grow dynamically }
pdf_op_buf ← xmalloc_array(eight_bits, pdf_op_buf_size);
pdf_os_buf ← xmalloc_array(eight_bits, inf_pdf_os_buf_size); { will grow dynamically }
pdf_os_objnum ← xmalloc_array(integer, pdf_os_max_objs);
pdf_os_objoff ← xmalloc_array(integer, pdf_os_max_objs); Init
    yzmem ← xmalloc_array(memory_word, mem_top − mem_bot + 1); zmem ← yzmem − mem_bot;
    { Some compilers require mem_bot = 0 }
eqtb_top ← eqtb_size + hash_extra;
if hash_extra = 0 then hash_top ← undefined_control_sequence
else hash_top ← eqtb_top;
yhash ← xmalloc_array(two_halves, 1 + hash_top − hash_offset); hash ← yhash − hash_offset;
    { Some compilers require hash_offset = 0 }
next(hash_base) ← 0; text(hash_base) ← 0;
for hash_used ← hash_base + 1 to hash_top do hash[hash_used] ← hash[hash_base];
zeqtb ← xmalloc_array(memory_word, eqtb_top); eqtb ← zeqtb;
str_start ← xmalloc_array(pool_pointer, max_strings);
str_pool ← xmalloc_array(packed_ASCII_code, pool_size);
font_info ← xmalloc_array(fmemory_word, font_mem_size); Tini history ← fatal_error_stop;
    { in case we quit during initialization }
t_open_out; { open the terminal for output }
if ready_already = 314159 then goto start_of_TEX;
⟨ Check the "constant" values for consistency 14 ⟩
if bad > 0 then
  begin wterm_ln('Ouch---my␣internal␣constants␣have␣been␣clobbered!', '---case␣', bad : 1);
  goto final_end;
  end;
initialize; { set global variables to their starting values }
Init if ¬get_strings_started then goto final_end;
init_prim; { call primitive for each primitive }
init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr; fix_date_and_time;
Tini
ready_already ← 314159;
start_of_TEX: ⟨ Initialize the output routines 55 ⟩;
⟨ Get the first line of input and prepare to start 1515* ⟩;
history ← spotless; { ready to go! }
⟨ Initialize synctex primitive 1910* ⟩ main_control; { come to life }
final_cleanup; { prepare for death }
close_files_and_terminate;
final_end: do_final_end;
end { main_body }
;
```

1511* Here we do whatever is needed to complete TEX's job gracefully on the local operating system. The code here might come into play after a fatal error; it must therefore consist entirely of "safe" operations that cannot produce error messages. For example, it would be a mistake to call *str_room* or *make_string* at this time, because a call on *overflow* might lead to an infinite loop. (Actually there's one way to get error messages, via *prepare_mag*; but that can't cause infinite recursion.)

If *final_cleanup* is bypassed, this program doesn't bother to close the input files that may still be open.

```
⟨Last-minute procedures 1511*⟩ ≡
procedure close_files_and_terminate;
  label done, done1;
  var a, b, c, i, j, k, l: integer; { all-purpose index }
    is_root: boolean; { pdf_last_pages is root of Pages tree? }
    is_names: boolean; { flag for name tree output: is it Names or Kids? }
    root, outlines, threads, names_tree, dests: integer; xref_offset_width, names_head, names_tail: integer;
  begin ⟨Finish the extensions 1623⟩;
  new_line_char ← −1;
  stat if tracing_stats > 0 then ⟨Output statistics about this job 1512*⟩; tats
  wake_up_terminal;
  if ¬fixed_pdfoutput_set then fix_pdfoutput;
  if fixed_pdfoutput > 0 then
    begin if history = fatal_error_stop then
      begin remove_pdffile; synctex_abort(log_opened);
      print_err("␣==>␣Fatal␣error␣occurred,␣no␣output␣PDF␣file␣produced!")
      end
    else begin ⟨Finish the PDF file 794⟩;
      if log_opened then
        begin wlog_cr; wlog_ln(´PDF␣statistics:´); wlog_ln(´␣´, obj_ptr : 1,
            ´␣PDF␣objects␣out␣of␣´, obj_tab_size : 1, ´␣(max.␣´, sup_obj_tab_size : 1, ´)´);
        if pdf_os_cntr > 0 then
          begin wlog(´␣´, ((pdf_os_cntr − 1) ∗ pdf_os_max_objs + pdf_os_objidx + 1) : 1,
              ´␣compressed␣objects␣within␣´, pdf_os_cntr : 1, ´␣object␣stream´);
          if pdf_os_cntr > 1 then wlog(´s´);
          wlog_cr;
          end;
        wlog_ln(´␣´, pdf_dest_names_ptr : 1, ´␣named␣destinations␣out␣of␣´, dest_names_size : 1,
            ´␣(max.␣´, sup_dest_names_size : 1, ´)´);
        wlog_ln(´␣´, pdf_mem_ptr : 1, ´␣words␣of␣extra␣memory␣for␣PDF␣output␣out␣of␣´,
            pdf_mem_size : 1, ´␣(max.␣´, sup_pdf_mem_size : 1, ´)´);
        end;
      end;
    end
  else begin ⟨Finish the DVI file 670*⟩;
    end;
  ⟨Close SyncTEX file and write status 1918*⟩;
  if log_opened then
    begin wlog_cr; a_close(log_file); selector ← selector − 2;
    if selector = term_only then
      begin print_nl("Transcript␣written␣on␣"); print_file_name(0, log_name, 0); print_char(".");
      end;
    end;
  print_ln;
  if (edit_name_start ≠ 0) ∧ (interaction > batch_mode) then
    call_edit(str_pool, edit_name_start, edit_name_length, edit_line);
```

end;

See also sections 1513*, 1514, and 1516*.

This code is used in section 1508.

1512* The present section goes directly to the log file instead of using *print* commands, because there's no need for these strings to take up *str_pool* memory when a non-**stat** version of TeX is being used.

⟨ Output statistics about this job 1512* ⟩ ≡
  if *log_opened* **then**
    begin *wlog_ln*(´␣´); *wlog_ln*(´Here␣is␣how␣much␣of␣TeX´´s␣memory´, ´␣you␣used:´);
    wlog(´␣´, *str_ptr* − *init_str_ptr* : 1, ´␣string´);
    if *str_ptr* ≠ *init_str_ptr* + 1 **then** *wlog*(´s´);
    wlog_ln(´␣out␣of␣´, *max_strings* − *init_str_ptr* : 1);
    wlog_ln(´␣´, *pool_ptr* − *init_pool_ptr* : 1, ´␣string␣characters␣out␣of␣´, *pool_size* − *init_pool_ptr* : 1);
    wlog_ln(´␣´, *lo_mem_max* − *mem_min* + *mem_end* − *hi_mem_min* + 2 : 1,
        ´␣words␣of␣memory␣out␣of␣´, *mem_end* + 1 − *mem_min* : 1);
    wlog_ln(´␣´, *cs_count* : 1, ´␣multiletter␣control␣sequences␣out␣of␣´, *hash_size* : 1, ´+´,
        hash_extra : 1);
    wlog(´␣´, *fmem_ptr* : 1, ´␣words␣of␣font␣info␣for␣´, *font_ptr* − *font_base* : 1, ´␣font´);
    if *font_ptr* ≠ *font_base* + 1 **then** *wlog*(´s´);
    wlog_ln(´,␣out␣of␣´, *font_mem_size* : 1, ´␣for␣´, *font_max* − *font_base* : 1);
    wlog(´␣´, *hyph_count* : 1, ´␣hyphenation␣exception´);
    if *hyph_count* ≠ 1 **then** *wlog*(´s´);
    wlog_ln(´␣out␣of␣´, *hyph_size* : 1);
    wlog_ln(´␣´, *max_in_stack* : 1, ´i,´, *max_nest_stack* : 1, ´n,´, *max_param_stack* : 1, ´p,´,
        max_buf_stack + 1 : 1, ´b,´, *max_save_stack* + 6 : 1, ´s␣stack␣positions␣out␣of␣´,
        stack_size : 1, ´i,´, *nest_size* : 1, ´n,´, *param_size* : 1, ´p,´, *buf_size* : 1, ´b,´, *save_size* : 1, ´s´);
    end

This code is used in section 1511*.

1513* We get to the *final_cleanup* routine when `\end` or `\dump` has been scanned and *its_all_over*.

⟨ Last-minute procedures 1511* ⟩ +≡
procedure *final_cleanup*;
  label *exit*;
  var *c*: *small_number*; { 0 for `\end`, 1 for `\dump` }
  begin $c \leftarrow cur\_chr$;
  if $c \neq 1$ **then** $new\_line\_char \leftarrow -1$;
  if $job\_name = 0$ **then** *open_log_file*;
  while $input\_ptr > 0$ **do**
    if $state = token\_list$ **then** *end_token_list* **else** *end_file_reading*;
  while $open\_parens > 0$ **do**
    begin *print*("␣)"); *decr*(*open_parens*);
    end;
  if $cur\_level > level\_one$ **then**
    begin *print_nl*("("); *print_esc*("end␣occurred␣"); *print*("inside␣a␣group␣at␣level␣");
    print_int($cur\_level - level\_one$); *print_char*(")");
    if *eTeX_ex* **then** *show_save_groups*;
    end;
  while $cond\_ptr \neq null$ **do**
    begin *print_nl*("("); *print_esc*("end␣occurred␣"); *print*("when␣"); *print_cmd_chr*(*if_test*, *cur_if*);
    if $if\_line \neq 0$ **then**
      begin *print*("␣on␣line␣"); *print_int*(*if_line*);
      end;
    print("␣was␣incomplete)"); $if\_line \leftarrow if\_line\_field(cond\_ptr)$; $cur\_if \leftarrow subtype(cond\_ptr)$;
    $temp\_ptr \leftarrow cond\_ptr$; $cond\_ptr \leftarrow link(cond\_ptr)$; *free_node*(*temp_ptr*, *if_node_size*);
    end;
  if $history \neq spotless$ **then**
    if $((history = warning\_issued) \lor (interaction < error\_stop\_mode))$ **then**
      if $selector = term\_and\_log$ **then**
        begin $selector \leftarrow term\_only$;
        print_nl("(see␣the␣transcript␣file␣for␣additional␣information)");
        $selector \leftarrow term\_and\_log$;
        end;
  if $c = 1$ **then**
    begin Init for $c \leftarrow top\_mark\_code$ **to** *split_bot_mark_code* **do**
      if $cur\_mark[c] \neq null$ **then** *delete_token_ref*($cur\_mark[c]$);
    if $sa\_mark \neq null$ **then**
      if *do_marks*(*destroy_marks*, 0, *sa_mark*) **then** $sa\_mark \leftarrow null$;
    for $c \leftarrow last\_box\_code$ **to** *vsplit_code* **do** *flush_node_list*($disc\_ptr[c]$);
    if $last\_glue \neq max\_halfword$ **then** *delete_glue_ref*(*last_glue*);
    store_fmt_file; **return**; **Tini**
    print_nl("(\dump␣is␣performed␣only␣by␣INITEX)"); **return**;
    end;
exit: **end**;

1515* When we begin the following code, TEX's tables may still contain garbage; the strings might not even be present. Thus we must proceed cautiously to get bootstrapped in.

But when we finish this part of the program, TEX is ready to call on the *main_control* routine to do its work.

⟨ Get the first line of input and prepare to start 1515* ⟩ ≡
 begin ⟨ Initialize the input routines 353* ⟩;
 ⟨ Enable ε-TEX, if requested 1645* ⟩
 if (*format_ident* = 0) ∨ (*buffer*[*loc*] = "&") ∨ *dump_line* **then**
  begin if *format_ident* ≠ 0 **then** *initialize*; { erase preloaded format }
  if ¬*open_fmt_file* **then goto** *final_end*;
  if ¬*load_fmt_file* **then**
   begin *w_close*(*fmt_file*); **goto** *final_end*;
   end;
  w_close(*fmt_file*); *eqtb* ← *zeqtb*;
  while (*loc* < *limit*) ∧ (*buffer*[*loc*] = "␣") **do** *incr*(*loc*);
  end;
 if (*pdf_output_option* ≠ 0) **then** *pdf_output* ← *pdf_output_value*;
 if (*pdf_draftmode_option* ≠ 0) **then** *pdf_draftmode* ← *pdf_draftmode_value*;
 pdf_init_map_file('pdftex.map');
 if *eTeX_ex* **then** *wterm_ln*('entering␣extended␣mode');
 if *end_line_char_inactive* **then** *decr*(*limit*)
 else *buffer*[*limit*] ← *end_line_char*;
 if *mltex_enabled_p* **then**
  begin *wterm_ln*('MLTeX␣v2.2␣enabled');
  end;
 if *enctex_enabled_p* **then**
  begin *wterm*(*encTeX_banner*); *wterm_ln*(',␣reencoding␣enabled.');
  if *translate_filename* **then**
   begin *wterm_ln*('␣(\xordcode,␣\xchrcode,␣\xprncode␣overridden␣by␣TCX)');
   end;
  end;
 fix_date_and_time;
 init if *trie_not_ready* **then**
  begin { initex without format loaded }
  trie_trl ← *xmalloc_array*(*trie_pointer*, *trie_size*); *trie_tro* ← *xmalloc_array*(*trie_pointer*, *trie_size*);
  trie_trc ← *xmalloc_array*(*quarterword*, *trie_size*); *trie_c* ← *xmalloc_array*(*packed_ASCII_code*, *trie_size*);
  trie_o ← *xmalloc_array*(*trie_opcode*, *trie_size*); *trie_l* ← *xmalloc_array*(*trie_pointer*, *trie_size*);
  trie_r ← *xmalloc_array*(*trie_pointer*, *trie_size*); *trie_hash* ← *xmalloc_array*(*trie_pointer*, *trie_size*);
  trie_taken ← *xmalloc_array*(*boolean*, *trie_size*); *trie_root* ← 0; *trie_c*[0] ← *si*(0); *trie_ptr* ← 0;
  hyph_root ← 0; *hyph_start* ← 0; { Allocate and initialize font arrays }
  font_check ← *xmalloc_array*(*four_quarters*, *font_max*); *font_size* ← *xmalloc_array*(*scaled*, *font_max*);
  font_dsize ← *xmalloc_array*(*scaled*, *font_max*); *font_params* ← *xmalloc_array*(*font_index*, *font_max*);
  font_name ← *xmalloc_array*(*str_number*, *font_max*);
  font_area ← *xmalloc_array*(*str_number*, *font_max*); *font_bc* ← *xmalloc_array*(*eight_bits*, *font_max*);
  font_ec ← *xmalloc_array*(*eight_bits*, *font_max*); *font_glue* ← *xmalloc_array*(*halfword*, *font_max*);
  hyphen_char ← *xmalloc_array*(*integer*, *font_max*); *skew_char* ← *xmalloc_array*(*integer*, *font_max*);
  bchar_label ← *xmalloc_array*(*font_index*, *font_max*); *font_bchar* ← *xmalloc_array*(*nine_bits*, *font_max*);
  font_false_bchar ← *xmalloc_array*(*nine_bits*, *font_max*); *char_base* ← *xmalloc_array*(*integer*, *font_max*);
  width_base ← *xmalloc_array*(*integer*, *font_max*); *height_base* ← *xmalloc_array*(*integer*, *font_max*);
  depth_base ← *xmalloc_array*(*integer*, *font_max*); *italic_base* ← *xmalloc_array*(*integer*, *font_max*);
  lig_kern_base ← *xmalloc_array*(*integer*, *font_max*); *kern_base* ← *xmalloc_array*(*integer*, *font_max*);
  exten_base ← *xmalloc_array*(*integer*, *font_max*); *param_base* ← *xmalloc_array*(*integer*, *font_max*);

$pdf\_char\_used \gets xmalloc\_array(char\_used\_array, font\_max)$;
$pdf\_font\_size \gets xmalloc\_array(scaled, font\_max)$; $pdf\_font\_num \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_map \gets xmalloc\_array(fm\_entry\_ptr, font\_max)$;
$pdf\_font\_type \gets xmalloc\_array(eight\_bits, font\_max)$;
$pdf\_font\_attr \gets xmalloc\_array(str\_number, font\_max)$;
$pdf\_font\_blink \gets xmalloc\_array(internal\_font\_number, font\_max)$;
$pdf\_font\_elink \gets xmalloc\_array(internal\_font\_number, font\_max)$;
$pdf\_font\_stretch \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_shrink \gets xmalloc\_array(integer, font\_max)$; $pdf\_font\_step \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_expand\_ratio \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_auto\_expand \gets xmalloc\_array(boolean, font\_max)$;
$pdf\_font\_lp\_base \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_rp\_base \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_ef\_base \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_kn\_bs\_base \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_st\_bs\_base \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_sh\_bs\_base \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_kn\_bc\_base \gets xmalloc\_array(integer, font\_max)$;
$pdf\_font\_kn\_ac\_base \gets xmalloc\_array(integer, font\_max)$;
$vf\_packet\_base \gets xmalloc\_array(integer, font\_max)$;
$vf\_default\_font \gets xmalloc\_array(internal\_font\_number, font\_max)$;
$vf\_local\_font\_num \gets xmalloc\_array(internal\_font\_number, font\_max)$;
$vf\_e\_fnts \gets xmalloc\_array(integer, font\_max)$;
$vf\_i\_fnts \gets xmalloc\_array(internal\_font\_number, font\_max)$;
$pdf\_font\_nobuiltin\_tounicode \gets xmalloc\_array(boolean, font\_max)$;
for $font\_k \gets font\_base$ **to** $font\_max$ **do**
  begin for $k \gets 0$ **to** 31 **do** $pdf\_char\_used[font\_k, k] \gets 0$;
  $pdf\_font\_size[font\_k] \gets 0$; $pdf\_font\_num[font\_k] \gets 0$; $pdf\_font\_map[font\_k] \gets 0$;
  $pdf\_font\_type[font\_k] \gets new\_font\_type$; $pdf\_font\_attr[font\_k] \gets$ ""; $pdf\_font\_blink[font\_k] \gets null\_font$;
  $pdf\_font\_elink[font\_k] \gets null\_font$; $pdf\_font\_stretch[font\_k] \gets null\_font$;
  $pdf\_font\_shrink[font\_k] \gets null\_font$; $pdf\_font\_step[font\_k] \gets 0$; $pdf\_font\_expand\_ratio[font\_k] \gets 0$;
  $pdf\_font\_auto\_expand[font\_k] \gets false$; $pdf\_font\_lp\_base[font\_k] \gets 0$; $pdf\_font\_rp\_base[font\_k] \gets 0$;
  $pdf\_font\_ef\_base[font\_k] \gets 0$; $pdf\_font\_kn\_bs\_base[font\_k] \gets 0$; $pdf\_font\_st\_bs\_base[font\_k] \gets 0$;
  $pdf\_font\_sh\_bs\_base[font\_k] \gets 0$; $pdf\_font\_kn\_bc\_base[font\_k] \gets 0$; $pdf\_font\_kn\_ac\_base[font\_k] \gets 0$;
  $pdf\_font\_nobuiltin\_tounicode[font\_k] \gets false$;
  end;
$font\_ptr \gets null\_font$; $fmem\_ptr \gets 7$; $make\_pdftex\_banner$; $font\_name[null\_font] \gets$ "nullfont";
$font\_area[null\_font] \gets$ ""; $hyphen\_char[null\_font] \gets$ "-"; $skew\_char[null\_font] \gets -1$;
$bchar\_label[null\_font] \gets non\_address$; $font\_bchar[null\_font] \gets non\_char$;
$font\_false\_bchar[null\_font] \gets non\_char$; $font\_bc[null\_font] \gets 1$; $font\_ec[null\_font] \gets 0$;
$font\_size[null\_font] \gets 0$; $font\_dsize[null\_font] \gets 0$; $char\_base[null\_font] \gets 0$;
$width\_base[null\_font] \gets 0$; $height\_base[null\_font] \gets 0$; $depth\_base[null\_font] \gets 0$;
$italic\_base[null\_font] \gets 0$; $lig\_kern\_base[null\_font] \gets 0$; $kern\_base[null\_font] \gets 0$;
$exten\_base[null\_font] \gets 0$; $font\_glue[null\_font] \gets null$; $font\_params[null\_font] \gets 7$;
$param\_base[null\_font] \gets -1$;
for $font\_k \gets 0$ **to** 6 **do** $font\_info[font\_k].sc \gets 0$;
end;
tini
$font\_used \gets xmalloc\_array(boolean, font\_max)$;
for $font\_k \gets font\_base$ **to** $font\_max$ **do** $font\_used[font\_k] \gets false$;
$random\_seed \gets (microseconds * 1000) + (epochseconds$ **mod** $1000000)$;
$init\_randoms(random\_seed)$;

⟨ Compute the magic offset 939 ⟩;
⟨ Initialize the print *selector* based on *interaction* 75 ⟩;
if $(loc < limit) \land (cat\_code(buffer[loc]) \neq escape)$ **then** *start_input*; { `\input` assumed }
end

This code is used in section 1510*.

1516* Debugging. Once TEX is working, you should be able to diagnose most errors with the `\show` commands and other diagnostic features. But for the initial stages of debugging, and for the revelation of really deep mysteries, you can compile TEX with a few more aids, including the Pascal runtime checks and its debugger. An additional routine called *debug_help* will also come into play when you type '`D`' after an error message; *debug_help* also occurs just before a fatal error causes TEX to succumb.

The interface to *debug_help* is primitive, but it is good enough when used with a Pascal debugger that allows you to set breakpoints and to read variables and change their values. After getting the prompt '`debug #`', you type either a negative number (this exits *debug_help*), or zero (this goes to a location where you can set a breakpoint, thereby entering into dialog with the Pascal debugger), or a positive number m followed by an argument n. The meaning of m and n will be clear from the program below. (If $m = 13$, there is an additional argument, l.)

define *breakpoint* = 888 { place where a breakpoint is desirable }

⟨ Last-minute procedures 1511* ⟩ +≡
debug procedure *debug_help*; { routine to display various things }
label *breakpoint*, *exit*;
var k, l, m, n: *integer*;
begin *clear_terminal*;
loop
  begin *wake_up_terminal*; *print_nl*("`debug␣#␣(-1␣to␣exit):`"); *update_terminal*; *read*(*term_in*, m);
  if $m < 0$ **then return**
  else if $m = 0$ **then** *dump_core* { do something to cause a core dump }
    else begin *read*(*term_in*, n);
      case m **of**
      ⟨ Numbered cases for *debug_help* 1517* ⟩
      othercases *print*("`?`")
      endcases;
      end;
  end;
exit: **end**;
gubed

1517* ⟨ Numbered cases for *debug_help* 1517* ⟩ ≡
1: *print_word*(*mem*[*n*]); { display *mem*[*n*] in all forms }
2: *print_int*(*info*(*n*));
3: *print_int*(*link*(*n*));
4: *print_word*(*eqtb*[*n*]);
5: **begin** *print_scaled*(*font_info*[*n*].*sc*); *print_char*("␣");
print_int(*font_info*[*n*].*qqqq*.*b0*); *print_char*(":");
print_int(*font_info*[*n*].*qqqq*.*b1*); *print_char*(":");
print_int(*font_info*[*n*].*qqqq*.*b2*); *print_char*(":");
print_int(*font_info*[*n*].*qqqq*.*b3*);
end;
6: *print_word*(*save_stack*[*n*]);
7: *show_box*(*n*); { show a box, abbreviated by *show_box_depth* and *show_box_breadth* }
8: **begin** *breadth_max* ← 10000; *depth_threshold* ← *pool_size* − *pool_ptr* − 10; *show_node_list*(*n*);
{ show a box in its entirety }
end;
9: *show_token_list*(*n*, *null*, 1000);
10: *slow_print*(*n*);
11: *check_mem*($n > 0$); { check wellformedness; print new busy locations if $n > 0$ }
12: *search_mem*(*n*); { look for pointers to *n* }
13: **begin** *read*(*term_in*, *l*); *print_cmd_chr*(*n*, *l*);
end;
14: **for** *k* ← 0 **to** *n* **do** *print*(*buffer*[*k*]);
15: **begin** *font_in_short_display* ← *null_font*; *short_display*(*n*);
end;
16: *panicking* ← ¬*panicking*;

This code is used in section 1516*.

1519* First let's consider the format of whatsit nodes that are used to represent the data associated with `\write` and its relatives. Recall that a whatsit has *type* = *whatsit_node*, and the *subtype* is supposed to distinguish different kinds of whatsits. Each node occupies two or more words; the exact number is immaterial, as long as it is readily determined from the *subtype* or other data.

We shall introduce five *subtype* values here, corresponding to the control sequences `\openout`, `\write`, `\closeout`, `\special`, and `\setlanguage`. The second word of I/O whatsits has a *write_stream* field that identifies the write-stream number (0 to 15, or 16 for out-of-range and positive, or 17 for out-of-range and negative). In the case of `\write` and `\special`, there is also a field that points to the reference count of a token list that should be sent. In the case of `\openout`, we need three words and three auxiliary subfields to hold the string numbers for name, area, and extension.

define *write_node_size* = 2 { number of words in a write/whatsit node }
define *open_node_size* = 3 { number of words in an open/whatsit node }
define *open_node* = 0 { *subtype* in whatsits that represent files to `\openout` }
define *write_node* = 1 { *subtype* in whatsits that represent things to `\write` }
define *close_node* = 2 { *subtype* in whatsits that represent streams to `\closeout` }
define *special_node* = 3 { *subtype* in whatsits that represent `\special` things }
define *language_node* = 4 { *subtype* in whatsits that change the current language }
define *what_lang*(#) ≡ *link*(# + 1) { language number, in the range 0 . . 255 }
define *what_lhm*(#) ≡ *type*(# + 1) { minimum left fragment, in the range 1 . . 63 }
define *what_rhm*(#) ≡ *subtype*(# + 1) { minimum right fragment, in the range 1 . . 63 }
define *write_tokens*(#) ≡ *link*(# + 1) { reference count of token list to write }
define *write_stream*(#) ≡ *type*(# + 1) { stream number (0 to 17) }
define *mubyte_zero* ≡ 64
define *write_mubyte*(#) ≡ *subtype*(# + 1) { mubyte value + *mubyte_zero* }
define *open_name*(#) ≡ *link*(# + 1) { string number of file name to open }
define *open_area*(#) ≡ *info*(# + 2) { string number of file area for *open_name* }
define *open_ext*(#) ≡ *link*(# + 2) { string number of file extension for *open_name* }

1522* Extensions might introduce new command codes; but it's best to use *extension* with a modifier, whenever possible, so that *main_control* stays the same.

define *immediate_code* = 4 { command modifier for `\immediate` }
define *set_language_code* = 5 { command modifier for `\setlanguage` }
define *pdftex_first_extension_code* = 6
define *pdf_literal_node* ≡ *pdftex_first_extension_code* + 0
define *pdf_obj_code* ≡ *pdftex_first_extension_code* + 1
define *pdf_refobj_node* ≡ *pdftex_first_extension_code* + 2
define *pdf_xform_code* ≡ *pdftex_first_extension_code* + 3
define *pdf_refxform_node* ≡ *pdftex_first_extension_code* + 4
define *pdf_ximage_code* ≡ *pdftex_first_extension_code* + 5
define *pdf_refximage_node* ≡ *pdftex_first_extension_code* + 6
define *pdf_annot_node* ≡ *pdftex_first_extension_code* + 7
define *pdf_start_link_node* ≡ *pdftex_first_extension_code* + 8
define *pdf_end_link_node* ≡ *pdftex_first_extension_code* + 9
define *pdf_outline_code* ≡ *pdftex_first_extension_code* + 10
define *pdf_dest_node* ≡ *pdftex_first_extension_code* + 11
define *pdf_thread_node* ≡ *pdftex_first_extension_code* + 12
define *pdf_start_thread_node* ≡ *pdftex_first_extension_code* + 13
define *pdf_end_thread_node* ≡ *pdftex_first_extension_code* + 14
define *pdf_save_pos_node* ≡ *pdftex_first_extension_code* + 15
define *pdf_info_code* ≡ *pdftex_first_extension_code* + 16
define *pdf_catalog_code* ≡ *pdftex_first_extension_code* + 17
define *pdf_names_code* ≡ *pdftex_first_extension_code* + 18
define *pdf_font_attr_code* ≡ *pdftex_first_extension_code* + 19
define *pdf_include_chars_code* ≡ *pdftex_first_extension_code* + 20
define *pdf_map_file_code* ≡ *pdftex_first_extension_code* + 21
define *pdf_map_line_code* ≡ *pdftex_first_extension_code* + 22
define *pdf_trailer_code* ≡ *pdftex_first_extension_code* + 23
define *pdf_trailer_id_code* ≡ *pdftex_first_extension_code* + 24
define *reset_timer_code* ≡ *pdftex_first_extension_code* + 25
define *pdf_font_expand_code* ≡ *pdftex_first_extension_code* + 26
define *set_random_seed_code* ≡ *pdftex_first_extension_code* + 27
define *pdf_snap_ref_point_node* ≡ *pdftex_first_extension_code* + 28
define *pdf_snapy_node* ≡ *pdftex_first_extension_code* + 29
define *pdf_snapy_comp_node* ≡ *pdftex_first_extension_code* + 30
define *pdf_glyph_to_unicode_code* ≡ *pdftex_first_extension_code* + 31
define *pdf_colorstack_node* ≡ *pdftex_first_extension_code* + 32
define *pdf_setmatrix_node* ≡ *pdftex_first_extension_code* + 33
define *pdf_save_node* ≡ *pdftex_first_extension_code* + 34
define *pdf_restore_node* ≡ *pdftex_first_extension_code* + 35
define *pdf_nobuiltin_tounicode_code* ≡ *pdftex_first_extension_code* + 36
define *pdf_interword_space_on_node* ≡ *pdftex_first_extension_code* + 37
define *pdf_interword_space_off_node* ≡ *pdftex_first_extension_code* + 38
define *pdf_fake_space_node* ≡ *pdftex_first_extension_code* + 39
define *pdf_running_link_off_node* ≡ *pdftex_first_extension_code* + 40
define *pdf_running_link_on_node* ≡ *pdftex_first_extension_code* + 41
define *pdftex_last_extension_code* ≡ *pdftex_first_extension_code* + 41

⟨ Put each of TEX's primitives into the hash table 244 ⟩ +≡
primitive(`"openout"`, *extension*, *open_node*);
primitive(`"write"`, *extension*, *write_node*); *write_loc* ← *cur_val*;
primitive(`"closeout"`, *extension*, *close_node*);

primitive("special", *extension*, *special_node*);
text(*frozen_special*) ← "special"; *eqtb*[*frozen_special*] ← *eqtb*[*cur_val*];
primitive("immediate", *extension*, *immediate_code*);
primitive("setlanguage", *extension*, *set_language_code*);
primitive("pdfliteral", *extension*, *pdf_literal_node*);
primitive("pdfcolorstack", *extension*, *pdf_colorstack_node*);
primitive("pdfsetmatrix", *extension*, *pdf_setmatrix_node*);
primitive("pdfsave", *extension*, *pdf_save_node*);
primitive("pdfrestore", *extension*, *pdf_restore_node*);
primitive("pdfobj", *extension*, *pdf_obj_code*);
primitive("pdfrefobj", *extension*, *pdf_refobj_node*);
primitive("pdfxform", *extension*, *pdf_xform_code*);
primitive("pdfrefxform", *extension*, *pdf_refxform_node*);
primitive("pdfximage", *extension*, *pdf_ximage_code*);
primitive("pdfrefximage", *extension*, *pdf_refximage_node*);
primitive("pdfannot", *extension*, *pdf_annot_node*);
primitive("pdfstartlink", *extension*, *pdf_start_link_node*);
primitive("pdfendlink", *extension*, *pdf_end_link_node*);
primitive("pdfoutline", *extension*, *pdf_outline_code*);
primitive("pdfdest", *extension*, *pdf_dest_node*);
primitive("pdfthread", *extension*, *pdf_thread_node*);
primitive("pdfstartthread", *extension*, *pdf_start_thread_node*);
primitive("pdfendthread", *extension*, *pdf_end_thread_node*);
primitive("pdfsavepos", *extension*, *pdf_save_pos_node*);
primitive("pdfsnaprefpoint", *extension*, *pdf_snap_ref_point_node*);
primitive("pdfsnapy", *extension*, *pdf_snapy_node*);
primitive("pdfsnapycomp", *extension*, *pdf_snapy_comp_node*);
primitive("pdfinfo", *extension*, *pdf_info_code*);
primitive("pdfcatalog", *extension*, *pdf_catalog_code*);
primitive("pdfnames", *extension*, *pdf_names_code*);
primitive("pdfincludechars", *extension*, *pdf_include_chars_code*);
primitive("pdffontattr", *extension*, *pdf_font_attr_code*);
primitive("pdfmapfile", *extension*, *pdf_map_file_code*);
primitive("pdfmapline", *extension*, *pdf_map_line_code*);
primitive("pdftrailer", *extension*, *pdf_trailer_code*);
primitive("pdftrailerid", *extension*, *pdf_trailer_id_code*);
primitive("pdfresettimer", *extension*, *reset_timer_code*);
primitive("pdfsetrandomseed", *extension*, *set_random_seed_code*);
primitive("pdffontexpand", *extension*, *pdf_font_expand_code*);
primitive("pdfglyphtounicode", *extension*, *pdf_glyph_to_unicode_code*);
primitive("pdfnobuiltintounicode", *extension*, *pdf_nobuiltin_tounicode_code*);
primitive("pdfinterwordspaceon", *extension*, *pdf_interword_space_on_node*);
primitive("pdfinterwordspaceoff", *extension*, *pdf_interword_space_off_node*);
primitive("pdffakespace", *extension*, *pdf_fake_space_node*);
primitive("pdfrunninglinkoff", *extension*, *pdf_running_link_off_node*);
primitive("pdfrunninglinkon", *extension*, *pdf_running_link_on_node*);

1526* ⟨Declare action procedures for use by *main_control* 1219⟩ +≡
⟨Declare procedures needed in *do_extension* 1527⟩
procedure *do_extension*;
  var *i*, *j*, *k*: *integer*; { all-purpose integers }
    p, *q*, *r*: *pointer*; { all-purpose pointers }
  begin case *cur_chr* **of**
  open_node: ⟨Implement `\openout` 1529⟩;
  write_node: ⟨Implement `\write` 1530⟩;
  close_node: ⟨Implement `\closeout` 1531⟩;
  special_node: ⟨Implement `\special` 1532*⟩;
  immediate_code: ⟨Implement `\immediate` 1620⟩;
  set_language_code: ⟨Implement `\setlanguage` 1622⟩;
  pdf_annot_node: ⟨Implement `\pdfannot` 1556⟩;
  pdf_catalog_code: ⟨Implement `\pdfcatalog` 1577⟩;
  pdf_dest_node: ⟨Implement `\pdfdest` 1563⟩;
  pdf_end_link_node: ⟨Implement `\pdfendlink` 1559⟩;
  pdf_end_thread_node: ⟨Implement `\pdfendthread` 1567⟩;
  pdf_font_attr_code: ⟨Implement `\pdffontattr` 1587⟩;
  pdf_font_expand_code: ⟨Implement `\pdffontexpand` 1533⟩;
  pdf_include_chars_code: ⟨Implement `\pdfincludechars` 1586⟩;
  pdf_info_code: ⟨Implement `\pdfinfo` 1576⟩;
  pdf_literal_node: ⟨Implement `\pdfliteral` 1536⟩;
  pdf_colorstack_node: ⟨Implement `\pdfcolorstack` 1537⟩;
  pdf_setmatrix_node: ⟨Implement `\pdfsetmatrix` 1538⟩;
  pdf_save_node: ⟨Implement `\pdfsave` 1539⟩;
  pdf_restore_node: ⟨Implement `\pdfrestore` 1540⟩;
  pdf_map_file_code: ⟨Implement `\pdfmapfile` 1588⟩;
  pdf_map_line_code: ⟨Implement `\pdfmapline` 1589⟩;
  pdf_names_code: ⟨Implement `\pdfnames` 1578⟩;
  pdf_obj_code: ⟨Implement `\pdfobj` 1542⟩;
  pdf_outline_code: ⟨Implement `\pdfoutline` 1561⟩;
  pdf_refobj_node: ⟨Implement `\pdfrefobj` 1544⟩;
  pdf_refxform_node: ⟨Implement `\pdfrefxform` 1547⟩;
  pdf_refximage_node: ⟨Implement `\pdfrefximage` 1552⟩;
  pdf_save_pos_node: ⟨Implement `\pdfsavepos` 1574⟩;
  pdf_snap_ref_point_node: ⟨Implement `\pdfsnaprefpoint` 1570⟩;
  pdf_snapy_comp_node: ⟨Implement `\pdfsnapycomp` 1573⟩;
  pdf_snapy_node: ⟨Implement `\pdfsnapy` 1572⟩;
  pdf_start_link_node: ⟨Implement `\pdfstartlink` 1558⟩;
  pdf_start_thread_node: ⟨Implement `\pdfstartthread` 1566⟩;
  pdf_thread_node: ⟨Implement `\pdfthread` 1565⟩;
  pdf_trailer_code: ⟨Implement `\pdftrailer` 1579⟩;
  pdf_trailer_id_code: ⟨Implement `\pdftrailerid` 1580⟩;
  pdf_xform_code: ⟨Implement `\pdfxform` 1546⟩;
  pdf_ximage_code: ⟨Implement `\pdfximage` 1551⟩;
  reset_timer_code: ⟨Implement `\pdfresettimer` 1584⟩;
  set_random_seed_code: ⟨Implement `\pdfsetrandomseed` 1583⟩;
  pdf_glyph_to_unicode_code: ⟨Implement `\pdfglyphtounicode` 1590⟩;
  pdf_nobuiltin_tounicode_code: ⟨Implement `\pdfnobuiltintounicode` 1591⟩;
  pdf_interword_space_on_node: ⟨Implement `\pdfinterwordspaceon` 1592⟩;
  pdf_interword_space_off_node: ⟨Implement `\pdfinterwordspaceoff` 1593⟩;
  pdf_fake_space_node: ⟨Implement `\pdffakespace` 1594⟩;

pdf_running_link_off_node: ⟨ Implement `\pdfrunninglinkoff` 1595 ⟩;
pdf_running_link_on_node: ⟨ Implement `\pdfrunninglinkon` 1596 ⟩;
othercases *confusion*("ext1")
endcases;
end;

1528*. The next subroutine uses *cur_chr* to decide what sort of whatsit is involved, and also inserts a *write_stream* number.

⟨ Declare procedures needed in *do_extension* 1527 ⟩ +≡
procedure *new_write_whatsit*(*w* : *small_number*);
begin *new_whatsit*(*cur_chr*, *w*);
if $w \neq write\_node\_size$ **then** *scan_four_bit_int*
else begin *scan_int*;
if $cur\_val < 0$ **then** $cur\_val \leftarrow 17$
else if $(cur\_val > 15) \land (cur\_val \neq 18)$ **then** $cur\_val \leftarrow 16$;
end;
$write\_stream(tail) \leftarrow cur\_val$;
if $mubyte\_out + mubyte\_zero < 0$ **then** $write\_mubyte(tail) \leftarrow 0$
else if $mubyte\_out + mubyte\_zero \geq 2 * mubyte\_zero$ **then** $write\_mubyte(tail) \leftarrow 2 * mubyte\_zero - 1$
else $write\_mubyte(tail) \leftarrow mubyte\_out + mubyte\_zero$;
end;

1532*. When '`\special{...}`' appears, we expand the macros in the token list as in `\xdef` and `\mark`.

⟨ Implement `\special` 1532* ⟩ ≡
begin *new_whatsit*(*special_node*, *write_node_size*);
if $spec\_out + mubyte\_zero < 0$ **then** $write\_stream(tail) \leftarrow 0$
else if $spec\_out + mubyte\_zero \geq 2 * mubyte\_zero$ **then** $write\_stream(tail) \leftarrow 2 * mubyte\_zero - 1$
else $write\_stream(tail) \leftarrow spec\_out + mubyte\_zero$;
if $mubyte\_out + mubyte\_zero < 0$ **then** $write\_mubyte(tail) \leftarrow 0$
else if $mubyte\_out + mubyte\_zero \geq 2 * mubyte\_zero$ **then** $write\_mubyte(tail) \leftarrow 2 * mubyte\_zero - 1$
else $write\_mubyte(tail) \leftarrow mubyte\_out + mubyte\_zero$;
if $(spec\_out = 2) \lor (spec\_out = 3)$ **then**
if $(mubyte\_out > 2) \lor (mubyte\_out = -1) \lor (mubyte\_out = -2)$ **then** $write\_noexpanding \leftarrow true$;
$p \leftarrow scan\_toks(false, true)$; $write\_tokens(tail) \leftarrow def\_ref$; $write\_noexpanding \leftarrow false$;
end

This code is used in section 1526*.

1599*. Each new type of node that appears in our data structure must be capable of being displayed, copied, destroyed, and so on. The routines that we need for write-oriented whatsits are somewhat like those for mark nodes; other extensions might, of course, involve more subtlety here.

⟨ Basic printing procedures 57 ⟩ +≡
procedure *print_write_whatsit*(*s* : *str_number*; *p* : *pointer*);
begin *print_esc*(*s*);
if $write\_stream(p) < 16$ **then** *print_int*(*write_stream*(*p*))
else if $write\_stream(p) = 16$ **then** *print_char*("*")
else *print_char*("-");
if $(s = \texttt{"write"}) \land (write\_mubyte(p) \neq mubyte\_zero)$ **then**
begin *print_char*("<"); *print_int*(*write_mubyte*(*p*) − *mubyte_zero*); *print_char*(">");
end;
end;

1600*. ⟨Display the whatsit node p 1600*⟩ ≡

```
case subtype(p) of
open_node: begin print_write_whatsit("openout", p); print_char("=");
  print_file_name(open_name(p), open_area(p), open_ext(p));
  end;
write_node: begin print_write_whatsit("write", p); print_mark(write_tokens(p));
  end;
close_node: print_write_whatsit("closeout", p);
special_node: begin print_esc("special");
  if write_stream(p) ≠ mubyte_zero then
    begin print_char("<"); print_int(write_stream(p) − mubyte_zero);
    if (write_stream(p) − mubyte_zero = 2) ∨ (write_stream(p) − mubyte_zero = 3) then
      begin print_char(":"); print_int(write_mubyte(p) − mubyte_zero);
      end;
    print_char(">");
    end;
  print_mark(write_tokens(p));
  end;
language_node: begin print_esc("setlanguage"); print_int(what_lang(p)); print("␣(hyphenmin␣");
  print_int(what_lhm(p)); print_char(","); print_int(what_rhm(p)); print_char(")");
  end;
pdf_literal_node: begin print_esc("pdfliteral");
  case pdf_literal_mode(p) of
  set_origin: do_nothing;
  direct_page: print("␣page");
  direct_always: print("␣direct");
  othercases confusion("literal2")
  endcases; print_mark(pdf_literal_data(p));
  end;
pdf_colorstack_node: begin print_esc("pdfcolorstack␣"); print_int(pdf_colorstack_stack(p));
  case pdf_colorstack_cmd(p) of
  colorstack_set: print("␣set␣");
  colorstack_push: print("␣push␣");
  colorstack_pop: print("␣pop");
  colorstack_current: print("␣current");
  othercases confusion("pdfcolorstack")
  endcases;
  if pdf_colorstack_cmd(p) ≤ colorstack_data then print_mark(pdf_colorstack_data(p));
  end;
pdf_setmatrix_node: begin print_esc("pdfsetmatrix"); print_mark(pdf_setmatrix_data(p));
  end;
pdf_save_node: begin print_esc("pdfsave");
  end;
pdf_restore_node: begin print_esc("pdfrestore");
  end;
pdf_refobj_node: begin print_esc("pdfrefobj");
  if obj_obj_is_stream(pdf_obj_objnum(p)) > 0 then
    begin if obj_obj_stream_attr(pdf_obj_objnum(p)) ≠ null then
      begin print("␣attr"); print_mark(obj_obj_stream_attr(pdf_obj_objnum(p)));
      end;
    print("␣stream");
    end;
```

```
    if obj_obj_is_file(pdf_obj_objnum(p)) > 0 then print("␣file");
    print_mark(obj_obj_data(pdf_obj_objnum(p)));
    end;
  pdf_refxform_node: begin print_esc("pdfrefxform"); print("(");
    print_scaled(obj_xform_height(pdf_xform_objnum(p))); print_char("+");
    print_scaled(obj_xform_depth(pdf_xform_objnum(p))); print(")x");
    print_scaled(obj_xform_width(pdf_xform_objnum(p)));
    end;
  pdf_refximage_node: begin print_esc("pdfrefximage"); print("(");
    print_scaled(obj_ximage_height(pdf_ximage_objnum(p))); print_char("+");
    print_scaled(obj_ximage_depth(pdf_ximage_objnum(p))); print(")x");
    print_scaled(obj_ximage_width(pdf_ximage_objnum(p)));
    end;
  pdf_annot_node: begin print_esc("pdfannot");
    ⟨Display ¡rule spec¿ for whatsit node created by pdfTEX 1598⟩;
    print_mark(pdf_annot_data(p));
    end;
  pdf_start_link_node: begin print_esc("pdfstartlink");
    ⟨Display ¡rule spec¿ for whatsit node created by pdfTEX 1598⟩;
    if pdf_link_attr(p) ≠ null then
      begin print("␣attr"); print_mark(pdf_link_attr(p));
      end;
    print("␣action");
    if pdf_action_type(pdf_link_action(p)) = pdf_action_user then
      begin print("␣user"); print_mark(pdf_action_user_tokens(pdf_link_action(p)));
      end
    else begin if pdf_action_file(pdf_link_action(p)) ≠ null then
        begin print("␣file"); print_mark(pdf_action_file(pdf_link_action(p)));
        end;
      case pdf_action_type(pdf_link_action(p)) of
      pdf_action_goto: begin if pdf_action_named_id(pdf_link_action(p)) > 0 then
          begin print("␣goto␣name"); print_mark(pdf_action_id(pdf_link_action(p)));
          end
        else begin print("␣goto␣num"); print_int(pdf_action_id(pdf_link_action(p)))
          end;
        end;
      pdf_action_page: begin print("␣page"); print_int(pdf_action_id(pdf_link_action(p)));
        print_mark(pdf_action_page_tokens(pdf_link_action(p)));
        end;
      pdf_action_thread: begin if pdf_action_named_id(pdf_link_action(p)) > 0 then
          begin print("␣thread␣name"); print_mark(pdf_action_id(pdf_link_action(p)));
          end
        else begin print("␣thread␣num"); print_int(pdf_action_id(pdf_link_action(p)));
          end;
        end;
      othercases pdf_error("displaying", "unknown␣action␣type");
      endcases;
      end
    end;
  pdf_end_link_node: print_esc("pdfendlink");
  pdf_dest_node: begin print_esc("pdfdest");
    if pdf_dest_named_id(p) > 0 then
```

```
      begin print(" name"); print_mark(pdf_dest_id(p));
      end
    else begin print(" num"); print_int(pdf_dest_id(p));
      end;
    print(" ");
    case pdf_dest_type(p) of
    pdf_dest_xyz: begin print("xyz");
      if pdf_dest_xyz_zoom(p) ≠ null then
        begin print(" zoom"); print_int(pdf_dest_xyz_zoom(p));
        end;
      end;
    pdf_dest_fitbh: print("fitbh");
    pdf_dest_fitbv: print("fitbv");
    pdf_dest_fitb: print("fitb");
    pdf_dest_fith: print("fith");
    pdf_dest_fitv: print("fitv");
    pdf_dest_fitr: begin print("fitr"); ⟨Display ¡rule spec¿ for whatsit node created by pdfTEX 1598⟩;
      end;
    pdf_dest_fit: print("fit");
    othercases print("unknown!");
    endcases;
    end;
  pdf_thread_node, pdf_start_thread_node: begin if subtype(p) = pdf_thread_node then
      print_esc("pdfthread")
    else print_esc("pdfstartthread");
    print("("); print_rule_dimen(pdf_height(p)); print_char("+"); print_rule_dimen(pdf_depth(p));
    print(")x"); print_rule_dimen(pdf_width(p));
    if pdf_thread_attr(p) ≠ null then
      begin print(" attr"); print_mark(pdf_thread_attr(p));
      end;
    if pdf_thread_named_id(p) > 0 then
      begin print(" name"); print_mark(pdf_thread_id(p));
      end
    else begin print(" num"); print_int(pdf_thread_id(p));
      end;
    end;
  pdf_end_thread_node: print_esc("pdfendthread");
  pdf_save_pos_node: print_esc("pdfsavepos");
  pdf_snap_ref_point_node: print_esc("pdfsnaprefpoint");
  pdf_snapy_node: begin print_esc("pdfsnapy"); print_char(" "); print_spec(snap_glue_ptr(p), 0);
    print_char(" "); print_spec(final_skip(p), 0);
    end;
  pdf_snapy_comp_node: begin print_esc("pdfsnapycomp"); print_char(" ");
    print_int(snapy_comp_ratio(p));
    end;
  pdf_interword_space_on_node: print_esc("pdfinterwordspaceon");
  pdf_interword_space_off_node: print_esc("pdfinterwordspaceoff");
  pdf_fake_space_node: print_esc("pdffakespace");
  pdf_running_link_off_node: print_esc("pdfrunninglinkoff");
  pdf_running_link_on_node: print_esc("pdfrunninglinkon");
  othercases print("whatsit?")
  endcases
```

This code is used in section 201.

1612* After all this preliminary shuffling, we come finally to the routines that actually send out the requested data. Let's do `\special` first (it's easier).

⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1612* ⟩ ≡
procedure *special_out*(*p* : *pointer*);
  var *old_setting*: 0 .. *max_selector*; { holds print *selector* }
    k: *pool_pointer*; { index into *str_pool* }
  begin *synch_h*; *synch_v*;
  old_setting ← *selector*; *selector* ← *new_string*; *spec_sout* ← *spec_out*;
  spec_out ← *write_stream*(*p*) − *mubyte_zero*; *mubyte_sout* ← *mubyte_out*;
  mubyte_out ← *write_mubyte*(*p*) − *mubyte_zero*; *active_noconvert* ← *true*; *mubyte_slog* ← *mubyte_log*;
  mubyte_log ← 0;
  if (*mubyte_out* > 0) ∨ (*mubyte_out* = −1) **then** *mubyte_log* ← 1;
  if (*spec_out* = 2) ∨ (*spec_out* = 3) **then**
    begin if (*mubyte_out* > 0) ∨ (*mubyte_out* = −1) **then**
      begin *special_printing* ← *true*; *mubyte_log* ← 1;
      end;
    if *mubyte_out* > 1 **then** *cs_converting* ← *true*;
    end;
  show_token_list(*link*(*write_tokens*(*p*)), *null*, *pool_size* − *pool_ptr*); *selector* ← *old_setting*; *str_room*(1);
  if *cur_length* < 256 **then**
    begin *dvi_out*(*xxx1*); *dvi_out*(*cur_length*);
    end
  else begin *dvi_out*(*xxx4*); *dvi_four*(*cur_length*);
    end;
  if (*spec_out* = 1) ∨ (*spec_out* = 3) **then**
    for *k* ← *str_start*[*str_ptr*] **to** *pool_ptr* − 1 **do** *str_pool*[*k*] ← *si*(*xchr*[*so*(*str_pool*[*k*])]);
  for *k* ← *str_start*[*str_ptr*] **to** *pool_ptr* − 1 **do** *dvi_out*(*so*(*str_pool*[*k*]));
  spec_out ← *spec_sout*; *mubyte_out* ← *mubyte_sout*; *mubyte_log* ← *mubyte_slog*; *special_printing* ← *false*;
  cs_converting ← *false*; *active_noconvert* ← *false*; *pool_ptr* ← *str_start*[*str_ptr*]; { erase the string }
  end;

See also sections 1614*, 1617*, 1716, and 1720.

This code is used in section 647*.

1614* ⟨Declare procedures needed in *hlist_out*, *vlist_out* 1612*⟩ +≡
procedure *write_out*(*p* : *pointer*);
  var *old_setting*: 0 . . *max_selector*; { holds print *selector* }
    old_mode: *integer*; { saved *mode* }
    j: *small_number*; { write stream number }
    q, *r*: *pointer*; { temporary variables for list manipulation }
    d: *integer*; { number of characters in incomplete current string }
    clobbered: *boolean*; { system string is ok? }
    runsystem_ret: *integer*; { return value from *runsystem* }
  begin *mubyte_sout* ← *mubyte_out*; *mubyte_out* ← *write_mubyte*(*p*) − *mubyte_zero*;
  if $(mubyte\_out > 2) \lor (mubyte\_out = -1) \lor (mubyte\_out = -2)$ **then** *write_noexpanding* ← *true*;
  ⟨Expand macros in the token list and make *link*(*def_ref*) point to the result 1615⟩;
  old_setting ← *selector*; *j* ← *write_stream*(*p*);
  if $j = 18$ **then** *selector* ← *new_string*
  else if *write_open*[*j*] **then** *selector* ← *j*
    else begin { write to the terminal if file isn't open }
      if $(j = 17) \land (selector = term\_and\_log)$ **then** *selector* ← *log_only*;
      print_nl("");
      end;
  active_noconvert ← *true*;
  if $mubyte\_out > 1$ **then** *cs_converting* ← *true*;
  mubyte_slog ← *mubyte_log*;
  if $(mubyte\_out > 0) \lor (mubyte\_out = -1)$ **then** *mubyte_log* ← 1
  else *mubyte_log* ← 0;
  token_show(*def_ref*); *print_ln*; *cs_converting* ← *false*; *write_noexpanding* ← *false*;
  active_noconvert ← *false*; *mubyte_out* ← *mubyte_sout*; *mubyte_log* ← *mubyte_slog*; *flush_list*(*def_ref*);
  if $j = 18$ **then**
    begin if $(tracing\_online \le 0)$ **then** *selector* ← *log_only* { Show what we're doing in the log file. }
    else *selector* ← *term_and_log*; { Show what we're doing. }
      { If the log file isn't open yet, we can only send output to the terminal. Calling *open_log_file* from here seems to result in bad data in the log. }
    if ¬*log_opened* **then** *selector* ← *term_only*;
    print_nl("`runsystem(`");
    for *d* ← 0 **to** *cur_length* − 1 **do**
      begin { *print* gives up if passed *str_ptr*, so do it by hand. }
      print(*so*(*str_pool*[*str_start*[*str_ptr*] + *d*])); { N.B.: not *print_char* }
      end;
    print("`)...`");
    if *shellenabledp* **then**
      begin *str_room*(1); *append_char*(0); { Append a null byte to the expansion. }
      clobbered ← *false*;
      for *d* ← 0 **to** *cur_length* − 1 **do** { Convert to external character set. }
        begin *str_pool*[*str_start*[*str_ptr*] + *d*] ← *xchr*[*str_pool*[*str_start*[*str_ptr*] + *d*]];
        if $(str\_pool[str\_start[str\_ptr] + d] = null\_code) \land (d < cur\_length - 1)$ **then** *clobbered* ← *true*;
            { minimal checking: NUL not allowed in argument string of *system*() }
        end;
      if *clobbered* **then** *print*("`clobbered`")
      else begin { We have the command. See if we're allowed to execute it, and report in the log. We don't check the actual exit status of the command, or do anything with the output. }
        runsystem_ret ← *runsystem*(*conststringcast*(*addressof*(*str_pool*[*str_start*[*str_ptr*]])));
        if $runsystem\_ret = -1$ **then** *print*("`quotation␣error␣in␣system␣command`")
        else if $runsystem\_ret = 0$ **then** *print*("`disabled␣(restricted)`")

```
          else if runsystem_ret = 1 then print("executed")
            else if runsystem_ret = 2 then print("executed␣safely␣(allowed)")
        end;
      end
    else begin print("disabled"); { shellenabledp false }
      end;
    print_char("."); print_nl(""); print_ln; pool_ptr ← str_start[str_ptr]; { erase the string }
    end;
  selector ← old_setting;
  end;
```

1617* The *out_what* procedure takes care of outputting whatsit nodes for *vlist_out* and *hlist_out*.

```
⟨Declare procedures needed in hlist_out, vlist_out 1612*⟩ +≡
procedure out_what(p : pointer);
  var j: small_number; { write stream number }
    old_setting: 0 .. max_selector;
  begin case subtype(p) of
  open_node, write_node, close_node: ⟨Do some work that has been queued up for \write 1619*⟩;
  special_node: special_out(p);
  language_node: do_nothing;
  pdf_save_pos_node: ⟨Save current position in DVI mode 1618⟩;
  others: begin if (pdftex_first_extension_code ≤ subtype(p)) ∧ (subtype(p) ≤ pdftex_last_extension_code)
          then pdf_error("ext4", "pdf␣node␣ended␣up␣in␣DVI␣mode")
    else confusion("ext4")
    end;
  endcases;
  end;
```

1619* We don't implement `\write` inside of leaders. (The reason is that the number of times a leader box appears might be different in different implementations, due to machine-dependent rounding in the glue calculations.)

⟨Do some work that has been queued up for `\write` 1619*⟩ ≡
```
if ¬doing_leaders then
  begin j ← write_stream(p);
  if subtype(p) = write_node then write_out(p)
  else begin if write_open[j] then
      begin a_close(write_file[j]); write_open[j] ← false;
      end;
    if subtype(p) = close_node then do_nothing   { already closed }
    else if j < 16 then
        begin cur_name ← open_name(p); cur_area ← open_area(p); cur_ext ← open_ext(p);
        if cur_ext = "" then cur_ext ← ".tex";
        pack_cur_name;
        while ¬kpse_out_name_ok(stringcast(name_of_file + 1)) ∨ ¬a_open_out(write_file[j]) do
          prompt_file_name("output␣file␣name", ".tex");
        write_open[j] ← true;   { If on first line of input, log file is not ready yet, so don't log. }
        if log_opened ∧ texmf_yesno(´log_openout´) then
          begin old_setting ← selector;
          if (tracing_online ≤ 0) then selector ← log_only   { Show what we're doing in the log file. }
          else selector ← term_and_log;   { Show what we're doing. }
          print_nl("\openout"); print_int(j); print("␣=␣`");
          print_file_name(cur_name, cur_area, cur_ext); print("´."); print_nl(""); print_ln;
          selector ← old_setting;
          end;
        end;
    end;
  end
```
This code is used in section 1617*.

1634* Threads are handled in similar way as link annotations.

```
⟨Declare procedures needed in pdf_hlist_out, pdf_vlist_out 727⟩ +≡
procedure append_bead(p : pointer);
  var a, b, c, t: integer;
  begin if ¬is_shipping_page then pdf_error("ext4", "threads␣cannot␣be␣inside␣an␣XForm");
  t ← get_obj(obj_type_thread, pdf_thread_id(p), pdf_thread_named_id(p)); b ← pdf_new_objnum;
  obj_bead_ptr(b) ← pdf_get_mem(pdfmem_bead_size); obj_bead_page(b) ← pdf_last_page;
  obj_bead_data(b) ← p;
  if pdf_thread_attr(p) ≠ null then obj_bead_attr(b) ← tokens_to_string(pdf_thread_attr(p))
  else obj_bead_attr(b) ← 0;
  if obj_thread_first(t) = 0 then
    begin obj_thread_first(t) ← b; obj_bead_next(b) ← b; obj_bead_prev(b) ← b;
    end
  else begin a ← obj_thread_first(t); c ← obj_bead_prev(a); obj_bead_prev(b) ← c; obj_bead_next(b) ← a;
    obj_bead_prev(a) ← b; obj_bead_next(c) ← b;
    end;
  pdf_append_list(b)(pdf_bead_list);
  end;
procedure do_thread(p, parent_box : pointer; x, y : scaled);
  begin if doing_leaders then return;
  if subtype(p) = pdf_start_thread_node then
    begin pdf_thread_wd ← pdf_width(p); pdf_thread_ht ← pdf_height(p); pdf_thread_dp ← pdf_depth(p);
    pdf_last_thread_id ← pdf_thread_id(p); pdf_last_thread_named_id ← (pdf_thread_named_id(p) > 0);
    if pdf_last_thread_named_id then add_token_ref(pdf_thread_id(p));
    pdf_thread_level ← cur_s;
    end;
  set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_thread_margin);
  append_bead(p); last_thread ← p;
  end;
procedure append_thread(parent_box : pointer; x, y : scaled);
  var p: pointer;
  begin p ← get_node(pdf_thread_node_size); info(p) ← max_halfword;  { this is not a whatsit node }
  link(p) ← null;  { this node will be destroyed separately }
  pdf_width(p) ← pdf_thread_wd; pdf_height(p) ← pdf_thread_ht; pdf_depth(p) ← pdf_thread_dp;
  pdf_thread_attr(p) ← null; pdf_thread_id(p) ← pdf_last_thread_id;
  if pdf_last_thread_named_id then
    begin add_token_ref(pdf_thread_id(p)); pdf_thread_named_id(p) ← 1;
    end
  else pdf_thread_named_id(p) ← 0;
  set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_thread_margin);
  append_bead(p); last_thread ← p;
  end;
procedure end_thread;
  begin if pdf_thread_level ≠ cur_s then pdf_error("ext4",
        "\pdfendthread␣ended␣up␣in␣different␣nesting␣level␣than␣\pdfstartthread");
  if is_running(pdf_thread_dp) ∧ (last_thread ≠ null) then
    pdf_bottom(last_thread) ← cur_v + pdf_thread_margin;
  if pdf_last_thread_named_id then delete_token_ref(pdf_last_thread_id);
  last_thread ← null;
  end;
function open_subentries(p : pointer): integer;
  var k, c: integer; l, r: integer;
```

```
  begin k ← 0;
  if obj_outline_first(p) ≠ 0 then
    begin l ← obj_outline_first(p);
    repeat incr(k); c ← open_subentries(l);
      if obj_outline_count(l) > 0 then k ← k + c;
      obj_outline_parent(l) ← p; r ← obj_outline_next(l);
      if r = 0 then obj_outline_last(p) ← l;
      l ← r;
    until l = 0;
    end;
  if obj_outline_count(p) > 0 then obj_outline_count(p) ← k
  else obj_outline_count(p) ← −k;
  open_subentries ← k;
  end;
procedure do_dest(p, parent_box : pointer; x, y : scaled);
  var k: integer;
  begin if ¬is_shipping_page then pdf_error("ext4", "destinations␣cannot␣be␣inside␣an␣XForm");
  if doing_leaders then return;
  k ← get_obj(obj_type_dest, pdf_dest_id(p), pdf_dest_named_id(p));
  if obj_dest_ptr(k) ≠ null then
    begin warn_dest_dup(pdf_dest_id(p), pdf_dest_named_id(p), "ext4",
        "has␣been␣already␣used,␣duplicate␣ignored"); return;
    end;
  obj_dest_ptr(k) ← p; pdf_append_list(k)(pdf_dest_list);
  case pdf_dest_type(p) of
  pdf_dest_xyz: if matrixused then
      set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_dest_margin)
    else begin pdf_left(p) ← cur_h; pdf_top(p) ← cur_v;
      end;
  pdf_dest_fith, pdf_dest_fitbh: if matrixused then
      set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_dest_margin)
    else pdf_top(p) ← cur_v;
  pdf_dest_fitv, pdf_dest_fitbv: if matrixused then
      set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_dest_margin)
    else pdf_left(p) ← cur_h;
  pdf_dest_fit, pdf_dest_fitb: do_nothing;
  pdf_dest_fitr: set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_dest_margin);
  endcases;
  end;
procedure out_form(p : pointer);
  begin synctexpdfrefxform(pdf_xform_objnum(p)); pdf_end_text; pdf_print_ln("q");
  if pdf_lookup_list(pdf_xform_list, pdf_xform_objnum(p)) = null then
    pdf_append_list(pdf_xform_objnum(p))(pdf_xform_list);
  cur_v ← cur_v + obj_xform_depth(pdf_xform_objnum(p)); pdf_print("1␣0␣0␣1␣");
  pdf_print_bp(pdf_x(cur_h)); pdf_out("␣"); pdf_print_bp(pdf_y(cur_v)); pdf_print_ln("␣cm");
  pdf_print("/Fm"); pdf_print_int(obj_info(pdf_xform_objnum(p))); pdf_print_resname_prefix;
  pdf_print_ln("␣Do"); pdf_print_ln("Q");
  end;
procedure out_image(p : pointer);
  var image, groupref: integer; img_w, img_h: integer;
  begin image ← obj_ximage_data(pdf_ximage_objnum(p));
  if (image_rotate(image) = 90) ∨ (image_rotate(image) = 270) then
```

```
      begin img_h ← image_width(image); img_w ← image_height(image);
      end
    else begin img_w ← image_width(image); img_h ← image_height(image);
      end;
  pdf_end_text; pdf_print_ln("q");
  if pdf_lookup_list(pdf_ximage_list, pdf_ximage_objnum(p)) = null then
    pdf_append_list(pdf_ximage_objnum(p))(pdf_ximage_list);
  if ¬is_pdf_image(image) then
    begin if is_png_image(image) then
      begin groupref ← get_image_group_ref(image);
      if (groupref > 0) ∧ (pdf_page_group_val = 0) then pdf_page_group_val ← groupref;
      end;
    pdf_print_real(ext_xn_over_d(pdf_width(p), ten_pow[6], one_hundred_bp), 4); pdf_print("␣0␣0␣");
    pdf_print_real(ext_xn_over_d(pdf_height(p) + pdf_depth(p), ten_pow[6], one_hundred_bp), 4);
    pdf_out("␣"); pdf_print_bp(pdf_x(cur_h)); pdf_out("␣"); pdf_print_bp(pdf_y(cur_v));
    end
  else begin { for pdf images we generate the page group object number here }
    groupref ← get_image_group_ref(image); { 0: no group, -1: to be generated; ¿0: already written }
    if (groupref ≠ 0) ∧ (pdf_page_group_val = 0) then
      begin if groupref = −1 then
        begin pdf_page_group_val ← pdf_new_objnum; set_image_group_ref(image, pdf_page_group_val);
        end
      else { groupref ¿ 0 }
      pdf_page_group_val ← groupref;
      end;
    pdf_print_real(ext_xn_over_d(pdf_width(p), ten_pow[6], img_w), 6); pdf_print("␣0␣0␣");
    pdf_print_real(ext_xn_over_d(pdf_height(p) + pdf_depth(p), ten_pow[6], img_h), 6); pdf_out("␣");
    pdf_print_bp(pdf_x(cur_h) − ext_xn_over_d(pdf_width(p), epdf_orig_x(image), img_w)); pdf_out("␣");
    pdf_print_bp(pdf_y(cur_v) − ext_xn_over_d(pdf_height(p) + pdf_depth(p), epdf_orig_y(image), img_h));
    end;
  pdf_print_ln("␣cm"); pdf_print("/Im"); pdf_print_int(obj_info(pdf_ximage_objnum(p)));
  pdf_print_resname_prefix; pdf_print_ln("␣Do"); pdf_print_ln("Q");
  end;
function gap_amount(p : pointer; cur_pos : scaled): scaled;
      { find the gap between the position of the current snap node p and the nearest point on the grid }
  var snap_unit, stretch_amount, shrink_amount: scaled; last_pos, next_pos, g, g2: scaled;
  begin snap_unit ← width(snap_glue_ptr(p));
  if stretch_order(snap_glue_ptr(p)) > normal then stretch_amount ← max_dimen
  else stretch_amount ← stretch(snap_glue_ptr(p));
  if shrink_order(snap_glue_ptr(p)) > normal then shrink_amount ← max_dimen
  else shrink_amount ← shrink(snap_glue_ptr(p));
  if subtype(p) = pdf_snapy_node then
    last_pos ← pdf_snapy_refpos + snap_unit ∗ ((cur_pos − pdf_snapy_refpos) div snap_unit)
  else pdf_error("snapping", "invalid␣parameter␣value␣for␣gap_amount");
  next_pos ← last_pos + snap_unit; @{print_nl("snap␣ref␣pos␣=␣"); print_scaled(pdf_snapy_refpos);
  print_nl("snap␣glue␣=␣"); print_spec(snap_glue_ptr(p), 0); print_nl("gap␣amount␣=␣");
  print_scaled(snap_unit); print_nl("stretch␣amount␣=␣"); print_scaled(stretch_amount);
  print_nl("shrink␣amount␣=␣"); print_scaled(shrink_amount); print_nl("last␣point␣=␣");
  print_scaled(last_pos); print_nl("cur␣point␣=␣"); print_scaled(cur_pos); print_nl("next␣point␣=␣");
  print_scaled(next_pos); @}g ← max_dimen; g2 ← max_dimen; gap_amount ← 0;
  if cur_pos − last_pos < shrink_amount then g ← cur_pos − last_pos;
  if (next_pos − cur_pos < stretch_amount) then g2 ← next_pos − cur_pos;
```

if $(g = max\_dimen) \wedge (g2 = max\_dimen)$ **then return**; { unable to snap }
if $g2 \le g$ **then** $gap\_amount \leftarrow g2$ { skip forward }
else $gap\_amount \leftarrow -g$; { skip backward }
end;
function *get_vpos*$(p, q, b : pointer)$: *pointer*;
{ find the vertical position of node q in the output PDF page; this functions is called when the current node is p and current position is *cur_v* (global variable); b is the parent box; }
var *tmp_v*: *scaled*; *g_order*: *glue_ord*; { applicable order of infinity for glue }
g_sign: *normal* .. *shrinking*; { selects type of glue }
glue_temp: *real*; { glue value before rounding }
cur_glue: *real*; { glue seen so far }
cur_g: *scaled*; { rounded equivalent of *cur_glue* times the glue ratio }
this_box: *pointer*; { pointer to containing box }
begin $tmp\_v \leftarrow cur\_v$; $this\_box \leftarrow b$; $cur\_g \leftarrow 0$; $cur\_glue \leftarrow float\_constant(0)$;
$g\_order \leftarrow glue\_order(this\_box)$; $g\_sign \leftarrow glue\_sign(this\_box)$;
while $(p \ne q) \wedge (p \ne null)$ **do**
begin if *is_char_node*(p) **then** *confusion*("get_vpos")
else begin case *type*(p) **of**
hlist_node, *vlist_node*, *rule_node*: $tmp\_v \leftarrow tmp\_v + height(p) + depth(p)$;
whatsit_node: **if** $(subtype(p) = pdf\_refxform\_node) \vee (subtype(p) = pdf\_refximage\_node)$ **then**
$tmp\_v \leftarrow tmp\_v + pdf\_height(p) + pdf\_depth(p)$;
glue_node: **begin** ⟨ Move down without outputting leaders 1635 ⟩;
$tmp\_v \leftarrow tmp\_v + rule\_ht$;
end;
kern_node: $tmp\_v \leftarrow tmp\_v + width(p)$;
othercases *do_nothing*;
endcases;
end;
$p \leftarrow link(p)$;
end;
$get\_vpos \leftarrow tmp\_v$;
end;
procedure *do_snapy_comp*$(p, b : pointer)$; { do snapping compensation in vertical direction; search for the next snap node and do the compensation if found }
var q: *pointer*; $tmp\_v, g, g2$: *scaled*;
begin if $\neg(\neg is\_char\_node(p) \wedge (type(p) = whatsit\_node) \wedge (subtype(p) = pdf\_snapy\_comp\_node))$ **then**
pdf_error("snapping", "invalid␣parameter␣value␣for␣do_snapy_comp");
$q \leftarrow p$;
while $(q \ne null)$ **do**
begin if $\neg is\_char\_node(q) \wedge (type(q) = whatsit\_node) \wedge (subtype(q) = pdf\_snapy\_node)$ **then**
begin $tmp\_v \leftarrow get\_vpos(p, q, b)$; { get the position of q }
$g \leftarrow gap\_amount(q, tmp\_v)$; { get the gap to the grid }
$g2 \leftarrow round\_xn\_over\_d(g, snapy\_comp\_ratio(p), 1000)$; { adjustment for p }
@{*print_nl*("do_snapy_comp:␣tmp_v␣=␣"); *print_scaled*(*tmp_v*);
print_nl("do_snapy_comp:␣cur_v␣=␣"); *print_scaled*(*cur_v*); *print_nl*("do_snapy_comp:␣g␣=␣");
print_scaled(*g*); *print_nl*("do_snapy_comp:␣g2␣=␣"); *print_scaled*(*g2*); @}$cur\_v \leftarrow cur\_v + g2$;
$final\_skip(q) \leftarrow g - g2$; { adjustment for q }
if $final\_skip(q) = 0$ **then** $final\_skip(q) \leftarrow 1$;
{ use 1*sp* as the magic value to record that *final_skip* has been set here }
return;
end;
$q \leftarrow link(q)$;

end;
end;
procedure *do_snapy*(*p* : *pointer*);
begin *incr*(*count_do_snapy*); @{*print_nl*("do_snapy:␣count␣=␣"); *print_int*(*count_do_snapy*);
print_nl("do_snapy:␣cur_v␣=␣"); *print_scaled*(*cur_v*); *print_nl*("do_snapy:␣final␣skip␣=␣");
print_scaled(*final_skip*(*p*)); @}
if $final\_skip(p) \neq 0$ **then** $cur\_v \leftarrow cur\_v + final\_skip(p)$
else $cur\_v \leftarrow cur\_v + gap\_amount(p, cur\_v)$;
@{*print_nl*("do_snapy:␣cur_v␣after␣snap␣=␣"); *print_scaled*(*cur_v*); @}
end;

1645*. The extended features of ε-TEX. The program has two modes of operation: (1) In TEX compatibility mode it fully deserves the name TEX and there are neither extended features nor additional primitive commands. There are, however, a few modifications that would be legitimate in any implementation of TEX such as, e.g., preventing inadequate results of the glue to `DVI` unit conversion during *ship_out*. (2) In extended mode there are additional primitive commands and the extended features of ε-TEX are available.

The distinction between these two modes of operation initially takes place when a 'virgin' `eINITEX` starts without reading a format file. Later on the values of all ε-TEX state variables are inherited when `eVIRTEX` (or `eINITEX`) reads a format file.

The code below is designed to work for cases where '**init** ... **tini**' is a run-time switch.

⟨ Enable ε-TEX, if requested 1645* ⟩ ≡
  init if $(etex\_p \lor (buffer[loc] = \texttt{"*"})) \land (format\_ident = \texttt{"␣(INITEX)"})$ **then**
    begin *no_new_control_sequence* ← *false*; ⟨ Generate all ε-TEX primitives 1646 ⟩
    if $(buffer[loc] = \texttt{"*"})$ **then** *incr*(*loc*);
    eTeX_mode ← 1; { enter extended mode }
    ⟨ Initialize variables for ε-TEX extended mode 1810 ⟩
    end;
  tini
  if ¬*no_new_control_sequence* **then** { just entered extended mode ? }
    no_new_control_sequence ← *true* **else**

This code is used in section 1515*.

1649*. **define** $eTeX\_ex \equiv (eTeX\_mode = 1)$ { is this extended mode? }

⟨ Global variables 13 ⟩ +≡
eTeX_mode: 0 .. 1; { identifies compatibility and extended mode }
etex_p: *boolean*; { was the -etex option specified }

1657*. In order to handle `\everyeof` we need an array *eof_seen* of boolean variables.

⟨ Global variables 13 ⟩ +≡
eof_seen: ↑*boolean*; { has eof been seen? }

1718*. We detach the hlist, start a new one consisting of just one kern node, append the reversed list, and set the width of the kern node.

⟨ Reverse the complete hlist and set the subtype to *reversed* 1718* ⟩ ≡
  begin *save_h* ← *cur_h*; *temp_ptr* ← *p*; *p* ← *new_kern*(0); *sync_tag*(*p* + *medium_node_size*) ← 0;
      { *SyncTEX*: do nothing, it is too late }
  link(*prev_p*) ← *p*; *cur_h* ← 0; *link*(*p*) ← *reverse*(*this_box*, *null*, *cur_g*, *cur_glue*); *width*(*p*) ← −*cur_h*;
  cur_h ← *save_h*; *set_box_lr*(*this_box*)(*reversed*);
  end

This code is used in section 1711.

1719*. We detach the remainder of the hlist, replace the math node by an edge node, and append the reversed hlist segment to it; the tail of the reversed segment is another edge node and the remainder of the original list is attached to it.

⟨ Reverse an hlist segment and **goto** *reswitch* 1719* ⟩ ≡
  begin *save_h* ← *cur_h*; *temp_ptr* ← *link*(*p*); *rule_wd* ← *width*(*p*); *free_node*(*p*, *medium_node_size*);
      { *SyncTEX*: p is a *math_node* }
  cur_dir ← *reflected*; *p* ← *new_edge*(*cur_dir*, *rule_wd*); *link*(*prev_p*) ← *p*;
  cur_h ← *cur_h* − *left_edge* + *rule_wd*; *link*(*p*) ← *reverse*(*this_box*, *new_edge*(*reflected*, 0), *cur_g*, *cur_glue*);
  edge_dist(*p*) ← *cur_h*; *cur_dir* ← *reflected*; *cur_h* ← *save_h*; **goto** *reswitch*;
  end

This code is used in section 1714.

1722* ⟨ Move the non-*char_node* p to the new list 1722* ⟩ ≡

begin $q \leftarrow link(p)$;
case $type(p)$ **of**
hlist_node, *vlist_node*, *rule_node*, *kern_node*: $rule\_wd \leftarrow width(p)$;
⟨ Cases of *reverse* that need special treatment 1723 ⟩
edge_node: *confusion*("LR2");
othercases goto *next_p*
endcases;
$cur\_h \leftarrow cur\_h + rule\_wd$;
next_p: $link(p) \leftarrow l$;
if $type(p) = kern\_node$ **then**
if $(rule\_wd = 0) \lor (l = null)$ **then**
begin $free\_node(p, medium\_node\_size)$; $p \leftarrow l$;
end;
$l \leftarrow p$; $p \leftarrow q$;
end

This code is used in section 1721.

1726* Finally we have found the end of the hlist segment to be reversed; the final math node is released and the remaining list attached to the edge node terminating the reversed segment.

⟨ Finish the reversed hlist segment and **goto** *done* 1726* ⟩ ≡

begin $free\_node(p, medium\_node\_size)$; { *SyncTEX*: p is a *kern_node* }
$link(t) \leftarrow q$; $width(t) \leftarrow rule\_wd$; $edge\_dist(t) \leftarrow -cur\_h - rule\_wd$; **goto** *done*;
end

This code is used in section 1725.

1730* When calculating the natural width, w, of the final line preceding the display, we may have to copy all or part of its hlist. We copy, however, only those parts of the original list that are relevant for the computation of *pre_display_size*.

⟨ Declare subprocedures for *init_math* 1730* ⟩ ≡
procedure *just_copy*(p, h, t : *pointer*);
  label *found*, *not_found*;
  var *r*: *pointer*; { current node being fabricated for new list }
    words: 0 . . 5; { number of words remaining to be copied }
  begin while $p \neq null$ **do**
    begin *words* ← 1; { this setting occurs in more branches than any other }
    if *is_char_node*(*p*) **then** *r* ← *get_avail*
    else case *type*(*p*) **of**
      hlist_node, *vlist_node*: **begin** *r* ← *get_node*(*box_node_size*);
        ⟨ Copy the box *SyncTEX* information 1932* ⟩;
        mem[$r + 6$] ← *mem*[$p + 6$]; *mem*[$r + 5$] ← *mem*[$p + 5$]; { copy the last two words }
        words ← 5; *list_ptr*(*r*) ← *null*; { this affects *mem*[$r + 5$] }
        end;
      rule_node: **begin** *r* ← *get_node*(*rule_node_size*); *words* ← *rule_node_size*;
        end;
      ligature_node: **begin** *r* ← *get_avail*; { only *font* and *character* are needed }
        mem[*r*] ← *mem*[*lig_char*(*p*)]; **goto** *found*;
        end;
      kern_node, *math_node*: **begin** *words* ← *medium_node_size*;
          { *SyncTEX*: proper size for math and kern }
        r ← *get_node*(*words*);
        end;
      glue_node: **begin** *r* ← *get_node*(*medium_node_size*); *add_glue_ref*(*glue_ptr*(*p*));
          { *SyncTEX*: proper size for glue }
        ⟨ Copy the medium sized node *SyncTEX* information 1934* ⟩;
        glue_ptr(*r*) ← *glue_ptr*(*p*); *leader_ptr*(*r*) ← *null*;
        end;
      whatsit_node: ⟨ Make a partial copy of the whatsit node *p* and make *r* point to it; set *words* to the number of initial words not yet copied 1601 ⟩;
      othercases goto *not_found*
      endcases;
    while *words* > 0 **do**
      begin *decr*(*words*); *mem*[*r* + *words*] ← *mem*[*p* + *words*];
      end;
  found: *link*(*h*) ← *r*; *h* ← *r*;
  not_found: *p* ← *link*(*p*);
    end;
  link(*h*) ← *t*;
  end;

See also section 1735*.

This code is used in section 1314.

1735* ⟨ Declare subprocedures for *init_math* 1730* ⟩ +≡

```
procedure just_reverse(p : pointer);
  label done;
  var l: pointer; { the new list }
    t: pointer; { tail of reversed segment }
    q: pointer; { the next node }
    m, n: halfword; { count of unmatched math nodes }
  begin m ← min_halfword; n ← min_halfword;
  if link(temp_head) = null then
    begin just_copy(link(p), temp_head, null); q ← link(temp_head);
    end
  else begin q ← link(p); link(p) ← null; flush_node_list(link(temp_head));
    end;
  t ← new_edge(cur_dir, 0); l ← t; cur_dir ← reflected;
  while q ≠ null do
    if is_char_node(q) then
      repeat p ← q; q ← link(p); link(p) ← l; l ← p;
      until ¬is_char_node(q)
    else begin p ← q; q ← link(p);
      if type(p) = math_node then ⟨ Adjust the LR stack for the just_reverse routine 1736* ⟩;
      link(p) ← l; l ← p;
      end;
  goto done; width(t) ← width(p); link(t) ← q; free_node(p, small_node_size);
done: link(temp_head) ← l;
  end;
```

1736* ⟨ Adjust the LR stack for the *just_reverse* routine 1736* ⟩ ≡

```
  if end_LR(p) then
    if info(LR_ptr) ≠ end_LR_type(p) then
      begin type(p) ← kern_node; incr(LR_problems);
          { SyncTEX node size watch point: math_node size == kern_node size }
      end
    else begin pop_LR;
      if n > min_halfword then
        begin decr(n); decr(subtype(p)); { change after into before }
        end
      else begin if m > min_halfword then decr(m) else begin width(t) ← width(p); link(t) ← q;
          free_node(p, medium_node_size); { SyncTEX: no more "goto found", and proper node size }
          goto done;
          end;
        type(p) ← kern_node; { SyncTEX node size watch point: math_node size == kern_node size }
        end;
      end
  else begin push_LR(p);
    if (n > min_halfword) ∨ (LR_dir(p) ≠ cur_dir) then
      begin incr(n); incr(subtype(p)); { change before into after }
      end
    else begin type(p) ← kern_node; incr(m);
          { SyncTEX node size watch point: math_node size == kern_node size }
      end;
    end
```

This code is used in section 1735*.

1752* ⟨ Initiate input from new pseudo file 1752* ⟩ ≡
begin_file_reading; { set up *cur_file* and new level of input }
line ← 0; *limit* ← *start*; *loc* ← *limit* + 1; { force line read }
if *tracing_scan_tokens* > 0 **then**
begin if *term_offset* > *max_print_line* − 3 **then** *print_ln*
else if (*term_offset* > 0) ∨ (*file_offset* > 0) **then** *print_char*("␣");
name ← 19; *print*("(␣"); *incr*(*open_parens*); *update_terminal*;
end
else begin *name* ← 18; ⟨ Prepare pseudo file *SyncTEX* information 1917* ⟩;
end

This code is used in section 1750.

1770* A group entered (or a conditional started) in one file may end in a different file. Such slight anomalies, although perfectly legitimate, may cause errors that are difficult to locate. In order to be able to give a warning message when such anomalies occur, ε-TEX uses the *grp_stack* and *if_stack* arrays to record the initial *cur_boundary* and *cond_ptr* values for each input file.

⟨ Global variables 13 ⟩ +≡
grp_stack: ↑*save_pointer*; { initial *cur_boundary* }
if_stack: ↑*pointer*; { initial *cond_ptr* }

1851* When reading `\patterns` while `\savinghyphcodes` is positive the current *lc_code* values are stored together with the hyphenation patterns for the current language. They will later be used instead of the *lc_code* values for hyphenation purposes.

The *lc_code* values are stored in the linked trie analogous to patterns p_1 of length 1, with *hyph_root* = *trie_r*[0] replacing *trie_root* and *lc_code*(*p_1*) replacing the *trie_op* code. This allows to compress and pack them together with the patterns with minimal changes to the existing code.

define *hyph_root* ≡ *trie_r*[0] { root of the linked trie for *hyph_codes* }

⟨ Initialize table entries (done by `INITEX` only) 182 ⟩ +≡

1864* System-dependent changes for Web2c. Here are extra variables for Web2c. (This numbering of the system-dependent section allows easy integration of Web2c and e-TEX, etc.)

⟨Global variables 13⟩ +≡
edit_name_start: *pool_pointer*; { where the filename to switch to starts }
edit_name_length, *edit_line*: *integer*; { what line to start editing at }
ipc_on: *cinttype*; { level of IPC action, 0 for none [default] }
stop_at_space: *boolean*; { whether *more_name* returns false for space }

1865* The *edit_name_start* will be set to point into *str_pool* somewhere after its beginning if TEX is supposed to switch to an editor on exit.

⟨Set initial values of key variables 21⟩ +≡
edit_name_start ← 0; *stop_at_space* ← *true*;

1866* These are used when we regenerate the representation of the first 256 strings.

⟨Global variables 13⟩ +≡
save_str_ptr: *str_number*;
save_pool_ptr: *pool_pointer*;
shellenabledp: *cinttype*;
restrictedshell: *cinttype*;
output_comment: ↑*char*;
k, *l*: 0 . . 255; { used by 'Make the first 256 strings', etc. }

1867* When debugging a macro package, it can be useful to see the exact control sequence names in the format file. For example, if ten new csnames appear, it's nice to know what they are, to help pinpoint where they came from. (This isn't a truly "basic" printing procedure, but that's a convenient module in which to put it.)

⟨Basic printing procedures 57⟩ +≡

```
procedure print_csnames(hstart : integer; hfinish : integer);
  var c, h: integer;
  begin write_ln(stderr, ´fmtdebug:csnames␣from␣´, hstart, ´␣to␣´, hfinish, ´:´);
  for h ← hstart to hfinish do
    begin if text(h) > 0 then
      begin { if have anything at this position }
      for c ← str_start[text(h)] to str_start[text(h) + 1] − 1 do
        begin put_byte(str_pool[c], stderr); { print the characters }
        end;
      write_ln(stderr, ´|´);
      end;
    end;
  end;
```

1868* Are we printing extra info as we read the format file?

⟨Global variables 13⟩ +≡
debug_format_file: *boolean*;

1869*. A helper for printing file:line:error style messages. Look for a filename in *full_source_filename_stack*, and if we fail to find one fall back on the non-file:line:error style.

⟨ Basic printing procedures 57 ⟩ +≡
procedure *print_file_line*;
  var *level*: 0 .. *max_in_open*;
  begin *level* ← *in_open*;
  while (*level* > 0) ∧ (*full_source_filename_stack*[*level*] = 0) **do** *decr*(*level*);
  if *level* = 0 **then** *print_nl*("!␣")
  else begin *print_nl*(""); *print*(*full_source_filename_stack*[*level*]); *print*(":");
    if *level* = *in_open* **then** *print_int*(*line*)
    else *print_int*(*line_stack*[*level* + 1]);
    print(":␣");
    end;
  end;

1870*. To be able to determine whether `\write18` is enabled from within TEX we also implement `\eof18`. We sort of cheat by having an additional route *scan_four_bit_int_or_18* which is the same as *scan_four_bit_int* except it also accepts the value 18.

⟨ Declare procedures that scan restricted classes of integers 459 ⟩ +≡
procedure *scan_four_bit_int_or_18*;
  begin *scan_int*;
  if (*cur_val* < 0) ∨ ((*cur_val* > 15) ∧ (*cur_val* ≠ 18)) **then**
    begin *print_err*("Bad␣number");
    help2("Since␣I␣expected␣to␣read␣a␣number␣between␣0␣and␣15,")
    ("I␣changed␣this␣one␣to␣zero."); *int_error*(*cur_val*); *cur_val* ← 0;
    end;
  end;

1871*. Dumping the *xord*, *xchr*, and *xprn* arrays. We dump these always in the format, so a TCX file loaded during format creation can set a default for users of the format.

⟨ Dump *xord*, *xchr*, and *xprn* 1871* ⟩ ≡
  dump_things(*xord*[0], 256); *dump_things*(*xchr*[0], 256); *dump_things*(*xprn*[0], 256);

This code is used in section 1483*.

1872*. Undumping the *xord*, *xchr*, and *xprn* arrays. This code is more complicated, because we want to ensure that a TCX file specified on the command line will override whatever is in the format. Since the tcx file has already been loaded, that implies throwing away the data in the format. Also, if no *translate_filename* is given, but *eight_bit_p* is set we have to make all characters printable.

⟨ Undump *xord*, *xchr*, and *xprn* 1872* ⟩ ≡
  if *translate_filename* **then**
    begin for *k* ← 0 **to** 255 **do** *undump_things*(*dummy_xord*, 1);
    for *k* ← 0 **to** 255 **do** *undump_things*(*dummy_xchr*, 1);
    for *k* ← 0 **to** 255 **do** *undump_things*(*dummy_xprn*, 1);
    end
  else begin *undump_things*(*xord*[0], 256); *undump_things*(*xchr*[0], 256); *undump_things*(*xprn*[0], 256);
    if *eight_bit_p* **then**
      for *k* ← 0 **to** 255 **do** *xprn*[*k*] ← 1;
    end;

This code is used in section 1484*.

1873*. The string recycling routines. TEX uses 2 upto 4 *new* strings when scanning a filename in an `\input`, `\openin`, or `\openout` operation. These strings are normally lost because the reference to them are not saved after finishing the operation. *search_string* searches through the string pool for the given string and returns either 0 or the found string number.

⟨ Declare additional routines for string recycling 1873* ⟩ ≡
function *search_string*(*search* : *str_number*): *str_number*;
  label *found*;
  var *result*: *str_number*; *s*: *str_number*; { running index }
    len: *integer*; { length of searched string }
  begin *result* ← 0; *len* ← *length*(*search*);
  if *len* = 0 **then** { trivial case }
    begin *result* ← `""`; **goto** *found*;
    end
  else begin *s* ← *search* − 1; { start search with newest string below *s*; *search* > 1! }
    while *s* > 255 **do** { first 256 strings depend on implementation!! }
      begin if *length*(*s*) = *len* **then**
        if *str_eq_str*(*s*, *search*) **then**
          begin *result* ← *s*; **goto** *found*;
          end;
      decr(*s*);
      end;
    end;
found: *search_string* ← *result*;
  end;

See also section 1874*.
This code is used in section 47*.

1874*. The following routine is a variant of *make_string*. It searches the whole string pool for a string equal to the string currently built and returns a found string. Otherwise a new string is created and returned. Be cautious, you can not apply *flush_string* to a replaced string!

⟨ Declare additional routines for string recycling 1873* ⟩ +≡
function *slow_make_string*: *str_number*;
  label *exit*;
  var *s*: *str_number*; { result of *search_string* }
    t: *str_number*; { new string }
  begin *t* ← *make_string*; *s* ← *search_string*(*t*);
  if *s* > 0 **then**
    begin *flush_string*; *slow_make_string* ← *s*; **return**;
    end;
  slow_make_string ← *t*;
exit: **end**;

1875*. More changes for Web2c. Sometimes, recursive calls to the *expand* routine may cause exhaustion of the run-time calling stack, resulting in forced execution stops by the operating system. To diminish the chance of this happening, a counter is used to keep track of the recursion depth, in conjunction with a constant called *expand_depth*.

This does not catch all possible infinite recursion loops, just the ones that exhaust the application calling stack. The actual maximum value of *expand_depth* is outside of our control, but the initial setting of 10000 should be enough to prevent problems.

⟨ Global variables 13 ⟩ +≡
expand_depth_count: *integer*;

1876*. ⟨ Set initial values of key variables 21 ⟩ +≡
expand_depth_count ← 0;

1877*. When *scan_file_name* starts it looks for a *left_brace* (skipping `\relax`es, as other `\toks`-like primitives). If a *left_brace* is found, then the procedure scans a file name contained in a balanced token list, expanding tokens as it goes. When the scanner finds the balanced token list, it is converted into a string and fed character-by-character to *more_name* to do its job the same as in the "normal" file name scanning.

```
procedure scan_file_name_braced;
  var save_scanner_status: small_number;  { scanner_status upon entry }
    save_def_ref: pointer;  { def_ref upon entry, important if inside '\message }
    save_cur_cs: pointer; s: str_number;  { temp string }
    p: pointer;  { temp pointer }
    i: integer;  { loop tally }
    save_stop_at_space: boolean;  { this should be in tex.ch }
    dummy: boolean;  { Initializing }
  begin save_scanner_status ← scanner_status;  { scan_toks sets scanner_status to absorbing }
  save_def_ref ← def_ref;  { scan_toks uses def_ref to point to the token list just read }
  save_cur_cs ← cur_cs;  { we set cur_cs back a few tokens to use in runaway errors }
    { Scanning a token list }
  cur_cs ← warning_index;  { for possible runaway error }
    { mimick call_func from pdfTeX }
  if scan_toks(false, true) ≠ 0 then do_nothing;  { actually do the scanning }
      { s ← tokens_to_string(def_ref); }
  old_setting ← selector; selector ← new_string; show_token_list(link(def_ref), null, pool_size − pool_ptr);
  selector ← old_setting; s ← make_string;  { turns the token list read in a string to input }
    { Restoring some variables }
  delete_token_ref(def_ref);  { remove the token list from memory }
  def_ref ← save_def_ref;  { and restore def_ref }
  cur_cs ← save_cur_cs;  { restore cur_cs }
  scanner_status ← save_scanner_status;  { restore scanner_status }
    { Passing the read string to the input machinery }
  save_stop_at_space ← stop_at_space;  { save stop_at_space }
  stop_at_space ← false;  { set stop_at_space to false to allow spaces in file names }
  begin_name;
  for i ← str_start[s] to str_start[s + 1] − 1 do dummy ← more_name(str_pool[i]);
        { add each read character to the current file name }
  stop_at_space ← save_stop_at_space;  { restore stop_at_space }
  end;
```

1878* System-dependent changes for MLTEX. The boolean variable *mltex_p* is set by web2c according to the given command line option (or an entry in the configuration file) before any TEX function is called.

⟨ Global variables 13 ⟩ +≡
mltex_p: *boolean*;

1879* The boolean variable *mltex_enabled_p* is used to enable MLTEX's character substitution. It is initialized to *false*. When loading a `FMT` it is set to the value of the boolean *mltex_p* saved in the `FMT` file. Additionally it is set to the value of *mltex_p* in IniTEX.

⟨ Global variables 13 ⟩ +≡
mltex_enabled_p: *boolean*; { enable character substitution }

1880* ⟨ Set initial values of key variables 21 ⟩ +≡
mltex_enabled_p ← *false*;

1881* The function *effective_char* computes the effective character with respect to font information. The effective character is either the base character part of a character substitution definition, if the character does not exist in the font or the character itself.

Inside *effective_char* we can not use *char_info* because the macro *char_info* uses *effective_char* calling this function a second time with the same arguments.

If neither the character c exists in font f nor a character substitution for c was defined, you can not use the function value as a character offset in *char_info* because it will access an undefined or invalid *font_info* entry! Therefore inside *char_info* and in other places, *effective_char*'s boolean parameter *err_p* is set to *true* to issue a warning and return the incorrect replacement, but always existing character *font_bc*[*f*].

⟨Declare ε-TEX procedures for scanning 1679⟩ +≡
function *effective_char*(*err_p* : *boolean*; *f* : *internal_font_number*; *c* : *quarterword*): *integer*;
  label *found*;
  var *base_c*: *integer*; { or *eightbits*: replacement base character }
    result: *integer*; { or *quarterword* }
  begin *result* ← *c*; { return *c* unless it does not exist in the font }
  if ¬*mltex_enabled_p* **then goto** *found*;
  if *font_ec*[*f*] ≥ *qo*(*c*) **then**
    if *font_bc*[*f*] ≤ *qo*(*c*) **then**
      if *char_exists*(*orig_char_info*(*f*)(*c*)) **then** { N.B.: not *char_info*(f)(c) }
        goto *found*;
  if *qo*(*c*) ≥ *char_sub_def_min* **then**
    if *qo*(*c*) ≤ *char_sub_def_max* **then**
      if *char_list_exists*(*qo*(*c*)) **then**
        begin *base_c* ← *char_list_char*(*qo*(*c*)); *result* ← *qi*(*base_c*); { return *base_c* }
        if ¬*err_p* **then goto** *found*;
        if *font_ec*[*f*] ≥ *base_c* **then**
          if *font_bc*[*f*] ≤ *base_c* **then**
            if *char_exists*(*orig_char_info*(*f*)(*qi*(*base_c*))) **then goto** *found*;
        end;
  if *err_p* **then** { print error and return existing character? }
    begin *begin_diagnostic*; *print_nl*("Missing␣character:␣There␣is␣no␣");
    print("substitution␣for␣"); *print_ASCII*(*qo*(*c*)); *print*("␣in␣font␣"); *slow_print*(*font_name*[*f*]);
    print_char("!"); *end_diagnostic*(*false*); *result* ← *qi*(*font_bc*[*f*]);
      { N.B.: not non-existing character *c*! }
    end;
found: *effective_char* ← *result*;
  end;

1882* The function *effective_char_info* is equivalent to *char_info*, except it will return *null_character* if neither the character c exists in font f nor is there a substitution definition for c. (For these cases *char_info* using *effective_char* will access an undefined or invalid *font_info* entry. See the documentation of *effective_char* for more information.)

⟨ Declare additional functions for MLTEX 1882* ⟩ ≡
function *effective_char_info*(*f* : *internal_font_number*; *c* : *quarterword*): *four_quarters*;
  label *exit*;
  var *ci*: *four_quarters*; { character information bytes for *c* }
    base_c: *integer*; { or *eightbits*: replacement base character }
  begin if ¬*mltex_enabled_p* **then**
    begin *effective_char_info* ← *orig_char_info*(*f*)(*c*); **return**;
    end;
  if *font_ec*[*f*] ≥ *qo*(*c*) **then**
    if *font_bc*[*f*] ≤ *qo*(*c*) **then**
      begin *ci* ← *orig_char_info*(*f*)(*c*); { N.B.: not *char_info*(f)(c) }
      if *char_exists*(*ci*) **then**
        begin *effective_char_info* ← *ci*; **return**;
        end;
      end;
  if *qo*(*c*) ≥ *char_sub_def_min* **then**
    if *qo*(*c*) ≤ *char_sub_def_max* **then**
      if *char_list_exists*(*qo*(*c*)) **then**
        begin { *effective_char_info* ← *char_info*(*f*)(*qi*(*char_list_char*(*qo*(*c*)))); }
        base_c ← *char_list_char*(*qo*(*c*));
        if *font_ec*[*f*] ≥ *base_c* **then**
          if *font_bc*[*f*] ≤ *base_c* **then**
            begin *ci* ← *orig_char_info*(*f*)(*qi*(*base_c*)); { N.B.: not *char_info*(f)(c) }
            if *char_exists*(*ci*) **then**
              begin *effective_char_info* ← *ci*; **return**;
              end;
            end;
        end;
  effective_char_info ← *null_character*;
exit: **end**;

This code is used in section 586*.

1883*. This code is called for a virtual character *c* in *hlist_out* during *ship_out*. It tries to built a character substitution construct for *c* generating appropriate `DVI` code using the character substitution definition for this character. If a valid character substitution exists `DVI` code is created as if *make_accent* was used. In all other cases the status of the substitution for this character has been changed between the creation of the character node in the hlist and the output of the page—the created `DVI` code will be correct but the visual result will be undefined.

Former MLTEX versions have replaced the character node by a sequence of character, box, and accent kern nodes splicing them into the original horizontal list. This version does not do this to avoid a) a memory overflow at this processing stage, b) additional code to add a pointer to the previous node needed for the replacement, and c) to avoid wrong code resulting in anomalies because of the use within a `\leaders` box.

⟨ Output a substitution, **goto** *continue* if not possible 1883* ⟩ ≡
  begin ⟨ Get substitution information, check it, goto *found* if all is ok, otherwise goto *continue* 1886* ⟩;
found: ⟨ Print character substitution tracing log 1887* ⟩;
  ⟨ Rebuild character using substitution information 1888* ⟩;
  end

This code is used in section 648*.

1884*. pdfTEX's *pdf_hlist_out* uses a similar, but slightly modified code section of the (almost) same name.

⟨ (pdfTEX) Output a substitution, **goto** *continue* if not possible 1884* ⟩ ≡
  begin ⟨ Get substitution information, check it, goto *found* if all is ok, otherwise goto *continue* 1886* ⟩;
found: ⟨ Print character substitution tracing log 1887* ⟩;
  ⟨ (pdfTEX) Rebuild character using substitution information 1889* ⟩;
  end

This code is used in section 731*.

1885*. The global variables for the code to substitute a virtual character can be declared as local. Nonetheless we declare them as global to avoid stack overflows because *hlist_out* can be called recursively.

⟨ Global variables 13 ⟩ +≡
accent_c, *base_c*, *replace_c*: *integer*;
ia_c, *ib_c*: *four_quarters*; { accent and base character information }
base_slant, *accent_slant*: *real*; { amount of slant }
base_x_height: *scaled*; { accent is designed for characters of this height }
base_width, *base_height*: *scaled*; { height and width for base character }
accent_width, *accent_height*: *scaled*; { height and width for accent }
delta: *scaled*; { amount of right shift }

1886*. Get the character substitution information in *char_sub_code* for the character *c*. The current code checks that the substitution exists and is valid and all substitution characters exist in the font, so we can *not* substitute a character used in a substitution. This simplifies the code because we have not to check for cycles in all character substitution definitions.

⟨ Get substitution information, check it, goto *found* if all is ok, otherwise goto *continue* 1886* ⟩ ≡

```
if qo(c) ≥ char_sub_def_min then
  if qo(c) ≤ char_sub_def_max then
    if char_list_exists(qo(c)) then
      begin base_c ← char_list_char(qo(c)); accent_c ← char_list_accent(qo(c));
      if (font_ec[f] ≥ base_c) then
        if (font_bc[f] ≤ base_c) then
          if (font_ec[f] ≥ accent_c) then
            if (font_bc[f] ≤ accent_c) then
              begin ia_c ← char_info(f)(qi(accent_c)); ib_c ← char_info(f)(qi(base_c));
              if char_exists(ib_c) then
                if char_exists(ia_c) then goto found;
              end;
      begin_diagnostic; print_nl("Missing␣character:␣Incomplete␣substitution␣");
      print_ASCII(qo(c)); print("␣=␣"); print_ASCII(accent_c); print("␣"); print_ASCII(base_c);
      print("␣in␣font␣"); slow_print(font_name[f]); print_char("!"); end_diagnostic(false);
      goto continue;
      end;
  begin_diagnostic; print_nl("Missing␣character:␣There␣is␣no␣"); print("substitution␣for␣");
  print_ASCII(qo(c)); print("␣in␣font␣"); slow_print(font_name[f]); print_char("!");
  end_diagnostic(false); goto continue
```

This code is used in sections 1883* and 1884*.

1887*. For *tracinglostchars* > 99 the substitution is shown in the log file.

⟨ Print character substitution tracing log 1887* ⟩ ≡

```
if tracing_lost_chars > 99 then
  begin begin_diagnostic; print_nl("Using␣character␣substitution:␣"); print_ASCII(qo(c));
  print("␣=␣"); print_ASCII(accent_c); print("␣"); print_ASCII(base_c); print("␣in␣font␣");
  slow_print(font_name[f]); print_char("."); end_diagnostic(false);
  end
```

This code is used in sections 1883* and 1884*.

1888* This outputs the accent and the base character given in the substitution. It uses code virtually identical to the *make_accent* procedure, but without the node creation steps.

Additionally if the accent character has to be shifted vertically it does *not* create the same code. The original routine in *make_accent* and former versions of MLTEX creates a box node resulting in *push* and *pop* operations, whereas this code simply produces vertical positioning operations. This can influence the pixel rounding algorithm in some DVI drivers—and therefore will probably be changed in one of the next MLTEX versions.

⟨ Rebuild character using substitution information 1888* ⟩ ≡
base_x_height ← *x_height*(*f*); *base_slant* ← *slant*(*f*)/*float_constant*(65536); *accent_slant* ← *base_slant*;
  { slant of accent character font }
base_width ← *char_width*(*f*)(*ib_c*); *base_height* ← *char_height*(*f*)(*height_depth*(*ib_c*));
accent_width ← *char_width*(*f*)(*ia_c*); *accent_height* ← *char_height*(*f*)(*height_depth*(*ia_c*));
 { compute necessary horizontal shift (don't forget slant) }
delta ← *round*((*base_width* − *accent_width*)/*float_constant*(2) + *base_height* ∗ *base_slant* − *base_x_height* ∗ *accent_slant*); *dvi_h* ← *cur_h*; { update *dvi_h*, similar to the last statement in module 620 }
 { 1. For centering/horizontal shifting insert a kern node. }
cur_h ← *cur_h* + *delta*; *synch_h*;
 { 2. Then insert the accent character possibly shifted up or down. }
if ((*base_height* ≠ *base_x_height*) ∧ (*accent_height* > 0)) **then**
 begin { the accent must be shifted up or down }
 cur_v ← *base_line* + (*base_x_height* − *base_height*); *synch_v*;
 if *accent_c* ≥ 128 **then** *dvi_out*(*set1*);
 dvi_out(*accent_c*);
 cur_v ← *base_line*;
 end
else begin *synch_v*;
 if *accent_c* ≥ 128 **then** *dvi_out*(*set1*);
 dvi_out(*accent_c*);
 end;
cur_h ← *cur_h* + *accent_width*; *dvi_h* ← *cur_h*;
 { 3. For centering/horizontal shifting insert another kern node. }
cur_h ← *cur_h* + (−*accent_width* − *delta*);
 { 4. Output the base character. }
synch_h; *synch_v*;
if *base_c* ≥ 128 **then** *dvi_out*(*set1*);
dvi_out(*base_c*);
cur_h ← *cur_h* + *base_width*; *dvi_h* ← *cur_h* { update of *dvi_h* is unnecessary, will be set in module 620 }

This code is used in section 1883*.

1889* pdfTEX's *pdf_hlist_out* uses a similar, but slightly modified code section of the (almost) same name.

⟨(pdfTEX) Rebuild character using substitution information 1889*⟩ ≡
  base_x_height ← *x_height*(*f*); *base_slant* ← *slant*(*f*)/*float_constant*(65536); *accent_slant* ← *base_slant*;
    { slant of accent character font }
  base_width ← *char_width*(*f*)(*ib_c*); *base_height* ← *char_height*(*f*)(*height_depth*(*ib_c*));
  accent_width ← *char_width*(*f*)(*ia_c*); *accent_height* ← *char_height*(*f*)(*height_depth*(*ia_c*));
   { compute necessary horizontal shift (don't forget slant) }
  delta ← *round*((*base_width* − *accent_width*)/*float_constant*(2) + *base_height* ∗ *base_slant* − *base_x_height* ∗ *accent_slant*);
   { 1. For centering/horizontal shifting insert a kern node. }
  cur_h ← *cur_h* + *delta*;
   { 2. Then insert the accent character possibly shifted up or down. }
  if ((*base_height* ≠ *base_x_height*) ∧ (*accent_height* > 0)) **then**
   begin { the accent must be shifted up or down }
   cur_v ← *base_line* + (*base_x_height* − *base_height*); *output_one_char*(*accent_c*); *cur_v* ← *base_line*;
   end
  else begin *output_one_char*(*accent_c*);
   end;
  cur_h ← *cur_h* + *accent_width*;
   { 3. For centering/horizontal shifting insert another kern node. }
  cur_h ← *cur_h* + (−*accent_width* − *delta*);
   { 4. Output the base character. }
  output_one_char(*base_c*); *cur_h* ← *cur_h* + *base_width*;

This code is used in section 1884*.

1890* Dumping MLTEX-related material. This is just the flag in the format that tells us whether MLTEX is enabled.

⟨Dump MLTEX-specific data 1890*⟩ ≡
  dump_int("4D4C5458); { MLTEX's magic constant: "MLTX" }
  if *mltex_p* **then** *dump_int*(1)
  else *dump_int*(0);

This code is used in section 1478*.

1891* Undump MLTEX-related material, which is just a flag in the format that tells us whether MLTEX is enabled.

⟨Undump MLTEX-specific data 1891*⟩ ≡
  undump_int(*x*); { check magic constant of MLTEX }
  if $x \neq$ "4D4C5458 **then goto** *bad_fmt*;
  undump_int(*x*); { undump *mltex_p* flag into *mltex_enabled_p* }
  if $x = 1$ **then** *mltex_enabled_p* ← *true*
  else if $x \neq 0$ **then goto** *bad_fmt*;

This code is used in section 1479*.

1892* System-dependent changes for encTEX.

define *encTeX_banner* ≡ ´␣encTeX␣v.␣Jun.␣2004´

1893* The boolean variable *enctex_p* is set by web2c according to the given command line option (or an entry in the configuration file) before any TEX function is called.

⟨ Global variables 13 ⟩ +≡
enctex_p: *boolean*;

1894* The boolean variable *enctex_enabled_p* is used to enable encTEX's primitives. It is initialised to *false*. When loading a `FMT` it is set to the value of the boolean *enctex_p* saved in the `FMT` file. Additionally it is set to the value of *enctex_p* in IniTEX.

⟨ Global variables 13 ⟩ +≡
enctex_enabled_p: *boolean*; { enable encTeX }

1895* ⟨ Set initial values of key variables 21 ⟩ +≡
enctex_enabled_p ← *false*;

1896*. Auxiliary functions/procedures for encTEX (by Petr Olsak) follow. These functions implement the `\mubyte` code to convert the multibytes in *buffer* to one byte or to one control sequence. These functions manipulate a mubyte tree: each node of this tree is token list with n+1 tokens (first token consist the byte from the byte sequence itself and the other tokens point to the branches). If you travel from root of the tree to a leaf then you find exactly one byte sequence which we have to convert to one byte or control sequence. There are two variants of the leaf: the "definitive end" or the "middle leaf" if a longer byte sequence exists and the mubyte tree continues under this leaf. First variant is implemented as one memory word where the link part includes the token to which we have to convert and type part includes the number 60 (normal conversion) or 1..52 (insert the control sequence). The second variant of "middle leaf" is implemented as two memory words: first one has a type advanced by 64 and link points to the second word where info part includes the token to which we have to convert and link points to the next token list with the branches of the subtree.

The inverse: one byte to multi byte (for log printing and `\write` printing) is implemented via a pool. Each multibyte sequence is stored in a pool as a string and *mubyte_write*[*printed char*] points to this string.

define *new_mubyte_node* ≡ *link*(*p*) ← *get_avail*; *p* ← *link*(*p*); *info*(*p*) ← *get_avail*; *p* ← *info*(*p*)
define *subinfo*(#) ≡ *subtype*(#)

⟨Basic printing procedures 57⟩ +≡
{ read *buffer*[*i*] and convert multibyte. *i* should have been of type 0..*buf_size*, but web2c doesn't like that construct in argument lists. }
function *read_buffer*(**var** *i* : *integer*): *ASCII_code*;
var *p*: *pointer*; *last_found*: *integer*; *last_type*: *integer*;
begin *mubyte_skip* ← 0; *mubyte_token* ← 0; *read_buffer* ← *buffer*[*i*];
if *mubyte_in* = 0 **then**
begin if *mubyte_keep* > 0 **then** *mubyte_keep* ← 0;
return;
end;
last_found ← −2;
if (*i* = *start*) ∧ (¬*mubyte_start*) **then**
begin *mubyte_keep* ← 0;
if (*end_line_char* ≥ 0) ∧ (*end_line_char* < 256) **then**
if *mubyte_read*[*end_line_char*] ≠ *null* **then**
begin *mubyte_start* ← *true*; *mubyte_skip* ← −1; *p* ← *mubyte_read*[*end_line_char*]; **goto** *continue*;
end;
end;
restart: *mubyte_start* ← *false*;
if (*mubyte_read*[*buffer*[*i*]] = *null*) ∨ (*mubyte_keep* > 0) **then**
begin if *mubyte_keep* > 0 **then** *decr*(*mubyte_keep*);
return;
end;
p ← *mubyte_read*[*buffer*[*i*]];
continue: **if** *type*(*p*) ≥ 64 **then**
begin *last_type* ← *type*(*p*) − 64; *p* ← *link*(*p*); *mubyte_token* ← *info*(*p*); *last_found* ← *mubyte_skip*;
end
else if *type*(*p*) > 0 **then**
begin *last_type* ← *type*(*p*); *mubyte_token* ← *link*(*p*); **goto** *found*;
end;
incr(*mubyte_skip*);
if *i* + *mubyte_skip* > *limit* **then**
begin *mubyte_skip* ← 0;
if *mubyte_start* **then goto** *restart*;
return;
end;

repeat $p \leftarrow link(p)$;
  if $subinfo(info(p)) = buffer[i + mubyte\_skip]$ **then**
    begin $p \leftarrow info(p)$; **goto** *continue*;
    end;
until $link(p) = null$;
$mubyte\_skip \leftarrow 0$;
if *mubyte_start* **then goto** *restart*;
if $last\_found = -2$ **then return**; { no found }
$mubyte\_skip \leftarrow last\_found$;
found: **if** $mubyte\_token < 256$ **then** { multibyte to one byte }
  begin $read\_buffer \leftarrow mubyte\_token$; $mubyte\_token \leftarrow 0$; $i \leftarrow i + mubyte\_skip$;
  if $mubyte\_start \land (i \geq start)$ **then** $mubyte\_start \leftarrow false$;
  return;
  end
else begin { multibyte to control sequence }
  $read\_buffer \leftarrow 0$;
  if $last\_type = 60$ **then** { normal conversion }
    $i \leftarrow i + mubyte\_skip$
  else begin { insert control sequence }
    $decr(i)$; $mubyte\_keep \leftarrow last\_type$;
    if $i < start$ **then** $mubyte\_start \leftarrow true$;
    if $last\_type = 52$ **then** $mubyte\_keep \leftarrow 10000$;
    if $last\_type = 51$ **then** $mubyte\_keep \leftarrow mubyte\_skip + 1$;
    $mubyte\_skip \leftarrow -1$;
    end;
  if $mubyte\_start \land (i \geq start)$ **then** $mubyte\_start \leftarrow false$;
  return;
  end;
exit: **end**;

1897* ⟨ Declare additional routines for encTEX 1897* ⟩ ≡

```
procedure mubyte_update; { saves new string to mubyte tree }
  var j: pool_pointer; p: pointer; q: pointer; in_mutree: integer;
  begin j ← str_start[str_ptr];
  if mubyte_read[so(str_pool[j])] = null then
    begin in_mutree ← 0; p ← get_avail; mubyte_read[so(str_pool[j])] ← p; subinfo(p) ← so(str_pool[j]);
    type(p) ← 0;
    end
  else begin in_mutree ← 1; p ← mubyte_read[so(str_pool[j])];
    end;
  incr(j);
  while j < pool_ptr do
    begin if in_mutree = 0 then
      begin new_mubyte_node; subinfo(p) ← so(str_pool[j]); type(p) ← 0;
      end
    else { in_mutree = 1 }
    if (type(p) > 0) ∧ (type(p) < 64) then
      begin type(p) ← type(p) + 64; q ← link(p); link(p) ← get_avail; p ← link(p); info(p) ← q;
      new_mubyte_node; subinfo(p) ← so(str_pool[j]); type(p) ← 0; in_mutree ← 0;
      end
    else begin if type(p) ≥ 64 then p ← link(p);
      repeat p ← link(p);
        if subinfo(info(p)) = so(str_pool[j]) then
          begin p ← info(p); goto continue;
          end;
      until link(p) = null;
      new_mubyte_node; subinfo(p) ← so(str_pool[j]); type(p) ← 0; in_mutree ← 0;
      end;
  continue: incr(j);
    end;
  if in_mutree = 1 then
    begin if type(p) = 0 then
      begin type(p) ← mubyte_prefix + 64; q ← link(p); link(p) ← get_avail; p ← link(p); link(p) ← q;
      info(p) ← mubyte_stoken; return;
      end;
    if type(p) ≥ 64 then
      begin type(p) ← mubyte_prefix + 64; p ← link(p); info(p) ← mubyte_stoken; return;
      end;
    end;
  type(p) ← mubyte_prefix; link(p) ← mubyte_stoken;
exit: end;

procedure dispose_munode(p : pointer); { frees a mu subtree recursivelly }
  var q: pointer;
  begin if (type(p) > 0) ∧ (type(p) < 64) then free_avail(p)
  else begin if type(p) ≥ 64 then
      begin q ← link(p); free_avail(p); p ← q;
      end;
    q ← link(p); free_avail(p); p ← q;
    while p ≠ null do
      begin dispose_munode(info(p)); q ← link(p); free_avail(p); p ← q;
      end;
    end;
```

```
  end;
procedure dispose_mutableout(cs : pointer); { frees record from out table }
  var p, q, r: pointer;
  begin p ← mubyte_cswrite[cs mod 128]; r ← null;
  while p ≠ null do
    if info(p) = cs then
      begin if r ≠ null then link(r) ← link(link(p))
      else mubyte_cswrite[cs mod 128] ← link(link(p));
      q ← link(link(p)); free_avail(link(p)); free_avail(p); p ← q;
      end
    else begin r ← link(p); p ← link(r);
      end;
  end;
```

This code is used in section 354*.

1898* The *print_buffer* procedure prints one character from *buffer*[*i*]. It also increases *i* to the next character in the buffer.

```
⟨Basic printing procedures 57⟩ +≡
    { print one char from buffer[i]. i should have been of type 0..buf_size, but web2c doesn't like that
      construct in argument lists. }
procedure print_buffer(var i : integer);
  var c: ASCII_code;
  begin if mubyte_in = 0 then print(buffer[i]) { normal TeX }
  else if mubyte_log > 0 then print_char(buffer[i])
    else begin c ← read_buffer(i);
      if mubyte_token > 0 then print_cs(mubyte_token − cs_token_flag)
      else print(c);
      end;
  incr(i);
  end;
```

1899* Additional material to dump for encTEX. This includes whether encTEXis enabled, and if it is we also have to dump the `\mubyte` arrays.

```
⟨Dump encTEX-specific data 1899*⟩ ≡
  dump_int("45435458); {encTEX's magic constant: "ECTX" }
  if ¬enctex_p then dump_int(0)
  else begin dump_int(1); dump_things(mubyte_read[0], 256); dump_things(mubyte_write[0], 256);
    dump_things(mubyte_cswrite[0], 128);
    end;
```

This code is used in section 1478*.

1900* Undumping the additional material we dumped for encTEX. This includes conditionally undumping the `\mubyte` arrays.

⟨ Undump encTEX-specific data 1900* ⟩ ≡
  undump_int(*x*); { check magic constant of encTEX }
  if $x \neq$ "`45435458` **then goto** *bad_fmt*;
  undump_int(*x*); { undump *enctex_p* flag into *enctex_enabled_p* }
  if $x = 0$ **then** *enctex_enabled_p* ← *false*
  else if $x \neq 1$ **then goto** *bad_fmt*
    else begin *enctex_enabled_p* ← *true*; *undump_things*(*mubyte_read*[0], 256);
      undump_things(*mubyte_write*[0], 256); *undump_things*(*mubyte_cswrite*[0], 128);
      end;

This code is used in section 1479*.

1901*. The *Synchronize TEXnology*. This section is devoted to the *Synchronize TEXnology* - or simply *SyncTEX* - used to synchronize between input and output. This section explains how synchronization basics are implemented. Before we enter into more technical details, let us recall in a few words what is synchronization.

TEX typesetting system clearly separates the input and the output material, and synchronization will provide a new link between both that can help text editors and viewers to work together. More precisely, forwards synchronization is the ability, given a location in the input source file, to find what is the corresponding place in the output. Backwards synchronization just performs the opposite: given a location in the output, retrieve the corresponding material in the input source file.

For better code management and maintainance, we adopt a naming convention. Throughout this program, code related to the *Synchronize TEXnology* is tagged with the "*synctex*" key word. Any code extract where *SyncTEX* plays its part, either explicitly or implicitly, (should) contain the string "*synctex*". This naming convention also holds for external files. Moreover, all the code related to *SyncTEX* is gathered in this section, except the definitions.

1902*. Enabling synchronization should be performed from the command line, *synctexoption* is used for that purpose. This global integer variable is declared here but it is not used here. This is just a placeholder where the command line controller will put the *SyncTEX* related options, and the *SyncTEX* controller will read them.

1903*. ⟨Global variables 13⟩ +≡
synctexoption: *integer*;

1904*. A convenient primitive is provided: `\synctex=1` in the input source file enables synchronization whereas `\synctex=0` disables it. Its memory address is *synctex_code*. It is initialized by the *SyncTEX* controller to the command-line option if given. The controller may filter some reserved bits.

1905*. ⟨Put each of TEX's primitives into the hash table 244⟩ +≡
primitive(`"synctex"`, *assign_int*, *int_base* + *synctex_code*);

1906*. ⟨synctex case for *print_param* 1906*⟩ ≡
synctex_code: *print_esc*(`"synctex"`);

This code is used in section 255*.

1907*. In order to give the *SyncTEX* controller read and write access to the contents of the `\synctex` primitive, we declare *synctexoffset*, such that *mem*[*synctexoffset*] and `\synctex` correspond to the same memory storage. *synctexoffset* is initialized to the correct value when quite everything is initialized.

1908*. ⟨Global variables 13⟩ +≡
synctexoffset: *integer*; { holds the true value of *synctex_code* }

1909*. ⟨Initialize whatever TEX might access 8*⟩ +≡
synctexoffset ← *int_base* + *synctex_code*;

1910*. ⟨Initialize synctex primitive 1910*⟩ ≡
synctex_init_command;

This code is used in section 1510*.

1911* Synchronization is achieved with the help of an auxiliary file named '*jobname*`.synctex`' (*jobname* is the contents of the `\jobname` macro), where a *SyncTEX* controller implemented in the external *synctex.c* file will store geometrical information. This *SyncTEX* controller will take care of every technical details concerning the *SyncTEX* file, we will only focus on the messages the controller will receive from the TEX program.

The most accurate synchronization information should allow to map any character of the input source file to the corresponding location in the output, if relevant. Ideally, the synchronization information of the input material consists of the file name, the line and column numbers of every character. The synchronization information in the output is simply the page number and either point coordinates, or box dimensions and position. The problem is that the mapping between these informations is only known at ship out time, which means that we must keep track of the input synchronization information until the pages ship out.

As TEX only knows about file names and line numbers, but forgets the column numbers, we only consider a restricted input synchronization information called *SyncTEX information*. It consists of a unique file name identifier, the *SyncTEX file tag*, and the line number.

Keeping track of such information, should be different whether characters or nodes are involved. Actually, only certain nodes are involved in *SyncTEX*, we call them *synchronized nodes*. Synchronized nodes store the *SyncTEX* information in their last two words: the first one contains a *SyncTEX file tag* uniquely identifying the input file, and the second one contains the current line number, as returned by the `\inputlineno` primitive. The *synctex_field_size* macro contains the necessary size to store the *SyncTEX* information in a node.

When declaring the size of a new node, it is recommanded to use the following convention: if the node is synchronized, use a definition similar to $my\_synchronized\_node\_size = \mathrm{xxx} + synctex\_field\_size$. Moreover, one should expect that the *SyncTEX* information is always stored in the last two words of a synchronized node.

1912* By default, every node with a sufficiently big size is initialized at creation time in the *get_node* routine with the current *SyncTEX* information, whether or not the node is synchronized. One purpose is to set this information very early in order to minimize code dependencies, including forthcoming extensions. Another purpose is to avoid the assumption that every node type has a dedicated getter, where initialization should take place. Actually, it appears that some nodes are created using directly the *get_node* routine and not the dedicated constructor. And finally, initializing the node at only one place is less error prone.

1913* ⟨Initialize bigger nodes with *SyncTEX* information 1913*⟩ ≡

if $s \ge medium\_node\_size$ **then**
  begin $sync\_tag(r+s) \leftarrow synctex\_tag$; $sync\_line(r+s) \leftarrow line$;
  end;

This code is used in section 143*.

1914* Instead of storing the input file name, it is better to store just an identifier. Each time TEX opens a new file, it notifies the *SyncTEX* controller with a *synctex_start_input* message. This controller will create a new *SyncTEX* file tag and will update the current input state record accordingly. If the input comes from the terminal or a pseudo file, the *synctex_tag* is set to 0. It results in automatically disabling synchronization for material input from the terminal or pseudo files.

1915* ⟨Prepare new file *SyncTEX* information 1915*⟩ ≡

synctex_start_input; { Give control to the *SyncTEX* controller }

This code is used in section 563*.

1916* ⟨Prepare terminal input *SyncTEX* information 1916*⟩ ≡

$synctex\_tag \leftarrow 0$;

This code is used in section 350*.

1917* ⟨ Prepare pseudo file *SyncTEX* information 1917* ⟩ ≡
synctex_tag ← 0;

This code is used in section 1752*.

1918* ⟨ Close *SyncTEX* file and write status 1918* ⟩ ≡
synctex_terminate(*log_opened*); { Let the *SyncTEX* controller close its files. }

This code is used in section 1511*.

1919* Synchronized nodes are boxes, math, kern and glue nodes. Other nodes should be synchronized too, in particular math noads. TEX assumes that math, kern and glue nodes have the same size, this is why both are synchronized. *In fine*, only horizontal lists are really used in *SyncTEX*, but all box nodes are considered the same with respect to synchronization, because a box node type is allowed to change at execution time.

The next sections are the various messages sent to the *SyncTEX* controller. The argument is either the box or the node currently shipped out. The vertical boxes are not recorded, but the code is available for clients.

1920* ⟨ Start sheet *SyncTEX* information record 1920* ⟩ ≡
synctex_sheet(*mag*);

This code is used in section 666*.

1921* ⟨ Finish sheet *SyncTEX* information record 1921* ⟩ ≡
synctex_teehs;

This code is used in section 666*.

1922* ⟨ Start vlist *SyncTEX* information record 1922* ⟩ ≡
synctex_vlist(*this_box*);

This code is used in sections 657* and 738*.

1923* ⟨ Finish vlist *SyncTEX* information record 1923* ⟩ ≡
synctex_tsilv(*this_box*);

This code is used in sections 657* and 738*.

1924* ⟨ Start hlist *SyncTEX* information record 1924* ⟩ ≡
synctex_hlist(*this_box*);

This code is used in sections 647* and 729*.

1925* ⟨ Finish hlist *SyncTEX* information record 1925* ⟩ ≡
synctex_tsilh(*this_box*);

This code is used in sections 647* and 729*.

1926* ⟨ Record void list *SyncTEX* information 1926* ⟩ ≡
if *type*(*p*) = *vlist_node* **then**
  begin *synctex_void_vlist*(*p*, *this_box*);
  end
else begin *synctex_void_hlist*(*p*, *this_box*);
  end;

This code is used in sections 651*, 660*, 733*, and 742*.

1927* ⟨ Record current point *SyncTEX* information 1927* ⟩ ≡
synctex_current;

This code is used in sections 648* and 731*.

1928* ⟨Record horizontal *rule_node* or *glue_node* *SyncTEX* information 1928*⟩ ≡
synctex_horizontal_rule_or_glue(*p*, *this_box*);

This code is used in sections 650* and 732*.

1929* ⟨Record *kern_node* *SyncTEX* information 1929*⟩ ≡
synctex_kern(*p*, *this_box*);

This code is used in sections 650* and 732*.

1930* ⟨Record *math_node* *SyncTEX* information 1930*⟩ ≡
synctex_math(*p*, *this_box*);

This code is used in sections 650* and 732*.

1931* When making a copy of a synchronized node, we might also have to duplicate the *SyncTEX* information by copying the two last words. This is the case for a *box_node* and for a *glue_node*, but not for a *math_node* nor a *kern_node*. These last two nodes always keep the *SyncTEX* information they received at creation time.

1932* ⟨Copy the box *SyncTEX* information 1932*⟩ ≡
$sync\_tag(r + box\_node\_size) \leftarrow sync\_tag(p + box\_node\_size)$;
$sync\_line(r + box\_node\_size) \leftarrow sync\_line(p + box\_node\_size)$;

This code is used in sections 224* and 1730*.

1933* ⟨Copy the rule *SyncTEX* information 1933*⟩ ≡

{ $sync\_tag(r + rule\_node\_size) \leftarrow sync\_tag(p + rule\_node\_size)$;
$sync\_line(r + rule\_node\_size) \leftarrow sync\_line(p + rule\_node\_size)$; }

This code is used in section 224*.

1934* ⟨Copy the medium sized node *SyncTEX* information 1934*⟩ ≡
$sync\_tag(r + medium\_node\_size) \leftarrow sync\_tag(p + medium\_node\_size)$;
$sync\_line(r + medium\_node\_size) \leftarrow sync\_line(p + medium\_node\_size)$;

This code is used in sections 224* and 1730*.

1935* *Nota Bene:* The *SyncTEX* code is very close to the memory model. It is not connected to any other part of the code, except for memory management. It is possible to neutralize the *SyncTEX* code rather simply. The first step is to define a null *synctex_field_size*. The second step is to comment out the code in "Initialize bigger nodes..." and every "Copy ... *SyncTEX* information". The last step will be to comment out the *synctex_tag_field* related code in the definition of *synctex_tag* and the various "Prepare ... *SyncTEX* information". Then all the remaining code should be just harmless. The resulting program would behave exactly the same as if absolutely no *SyncTEX* related code was there, including memory management. Of course, all this assumes that *SyncTEX* is turned off from the command line.

1936*. System-dependent changes.

⟨ Declare action procedures for use by *main_control* 1219 ⟩ +≡

procedure *insert_src_special*;
  var *toklist*, *p*, *q*: *pointer*;
  begin if $(source\_filename\_stack[in\_open] > 0 \land is\_new\_source(source\_filename\_stack[in\_open], line))$ **then**
    begin $toklist \leftarrow get\_avail$; $p \leftarrow toklist$; $info(p) \leftarrow cs\_token\_flag + frozen\_special$; $link(p) \leftarrow get\_avail$;
    $p \leftarrow link(p)$; $info(p) \leftarrow left\_brace\_token +$ "{";
    $q \leftarrow str\_toks(make\_src\_special(source\_filename\_stack[in\_open], line))$; $link(p) \leftarrow link(temp\_head)$;
    $p \leftarrow q$; $link(p) \leftarrow get\_avail$; $p \leftarrow link(p)$; $info(p) \leftarrow right\_brace\_token +$ "}"; $ins\_list(toklist)$;
    $remember\_source\_info(source\_filename\_stack[in\_open], line)$;
    end;
  end;

procedure *append_src_special*;
  var *q*: *pointer*;
  begin if $(source\_filename\_stack[in\_open] > 0 \land is\_new\_source(source\_filename\_stack[in\_open], line))$ **then**
    begin $new\_whatsit(special\_node, write\_node\_size)$; $write\_stream(tail) \leftarrow 0$; $def\_ref \leftarrow get\_avail$;
    $token\_ref\_count(def\_ref) \leftarrow null$; $q \leftarrow str\_toks(make\_src\_special(source\_filename\_stack[in\_open], line))$;
    $link(def\_ref) \leftarrow link(temp\_head)$; $write\_tokens(tail) \leftarrow def\_ref$;
    $remember\_source\_info(source\_filename\_stack[in\_open], line)$;
    end;
  end;

1937*. This function used to be in pdftex, but is useful in tex too.

function *get_nullstr*: *str_number*;
  begin $get\_nullstr \leftarrow$ "";
  end;

1938* Index. Here is where you can find all uses of each identifier in the program, with underlined entries pointing to where the identifier was defined. If the identifier is only one letter long, however, you get to see only the underlined entries. *All references are to section numbers instead of page numbers.*

This index also lists error messages and other aspects of the program that you might want to look up some day. For example, the entry for "system dependencies" lists all sections that should receive special attention from people who are installing TEX in a new operating environment. A list of various things that can't happen appears under "this can't happen". Approximately 40 sections are listed under "inner loop"; these account for about 60% of TEX's running time, exclusive of input and output.

The following sections were changed by the change file: 2, 4, 6, 7, 8, 11, 12, 16, 19, 20, 23, 24, 26, 27, 28, 30, 31, 32, 33, 34, 35, 37, 38, 39, 47, 49, 51, 52, 53, 54, 59, 61, 71, 73, 74, 81, 82, 84, 93, 94, 95, 104, 109, 128, 129, 130, 131, 134, 143, 153, 156, 159, 162, 165, 170, 171, 174, 176, 183, 192, 194, 204, 220, 224, 227, 229, 231, 233, 237, 238, 240, 248, 254, 255, 256, 258, 259, 270, 271, 274, 276, 277, 279, 284, 287, 288, 293, 305, 312, 322, 323, 324, 326, 328, 330, 340, 350, 353, 354, 360, 361, 363, 365, 376, 377, 378, 379, 385, 388, 398, 426, 427, 440, 527, 539, 540, 541, 542, 543, 544, 545, 546, 547, 549, 550, 551, 552, 556, 558, 560, 562, 563, 574, 575, 576, 577, 578, 580, 586, 587, 589, 590, 596, 600, 602, 603, 604, 608, 609, 619, 622, 624, 625, 626, 629, 645, 647, 648, 649, 650, 651, 657, 660, 666, 668, 670, 673, 729, 731, 732, 733, 738, 742, 750, 882, 895, 896, 914, 923, 1085, 1095, 1096, 1098, 1099, 1100, 1101, 1103, 1105, 1106, 1109, 1114, 1115, 1116, 1118, 1119, 1120, 1121, 1122, 1125, 1126, 1133, 1135, 1138, 1139, 1140, 1141, 1163, 1209, 1211, 1212, 1225, 1267, 1311, 1315, 1343, 1387, 1391, 1395, 1396, 1397, 1398, 1399, 1400, 1406, 1407, 1408, 1428, 1433, 1436, 1441, 1451, 1455, 1456, 1459, 1477, 1478, 1479, 1481, 1482, 1483, 1484, 1485, 1486, 1487, 1488, 1490, 1491, 1492, 1493, 1494, 1495, 1496, 1497, 1498, 1499, 1500, 1501, 1505, 1510, 1511, 1512, 1513, 1515, 1516, 1517, 1519, 1522, 1526, 1528, 1532, 1599, 1600, 1612, 1614, 1617, 1619, 1634, 1645, 1649, 1657, 1718, 1719, 1722, 1726, 1730, 1735, 1736, 1752, 1770, 1851, 1864, 1865, 1866, 1867, 1868, 1869, 1870, 1871, 1872, 1873, 1874, 1875, 1876, 1877, 1878, 1879, 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887, 1888, 1889, 1890, 1891, 1892, 1893, 1894, 1895, 1896, 1897, 1898, 1899, 1900, 1901, 1902, 1903, 1904, 1905, 1906, 1907, 1908, 1909, 1910, 1911, 1912, 1913, 1914, 1915, 1916, 1917, 1918, 1919, 1920, 1921, 1922, 1923, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938.

⟨ Implement \pdfliteral 1536 ⟩ Used in section 1526*.
⟨ Implement \pdfmapfile 1588 ⟩ Used in section 1526*.
⟨ Implement \pdfmapline 1589 ⟩ Used in section 1526*.
⟨ Implement \pdfnames 1578 ⟩ Used in section 1526*.
⟨ Implement \pdfnobuiltintounicode 1591 ⟩ Used in section 1526*.
⟨ Implement \pdfobj 1542 ⟩ Used in section 1526*.
⟨ Implement \pdfoutline 1561 ⟩ Used in section 1526*.
⟨ Implement \pdfprimitive 394 ⟩ Used in section 391.
⟨ Implement \pdfrefobj 1544 ⟩ Used in section 1526*.
⟨ Implement \pdfrefxform 1547 ⟩ Used in section 1526*.
⟨ Implement \pdfrefximage 1552 ⟩ Used in section 1526*.
⟨ Implement \pdfresettimer 1584 ⟩ Used in section 1526*.
⟨ Implement \pdfrestore 1540 ⟩ Used in section 1526*.
⟨ Implement \pdfrunninglinkoff 1595 ⟩ Used in section 1526*.
⟨ Implement \pdfrunninglinkon 1596 ⟩ Used in section 1526*.
⟨ Implement \pdfsavepos 1574 ⟩ Used in section 1526*.
⟨ Implement \pdfsave 1539 ⟩ Used in section 1526*.
⟨ Implement \pdfsetmatrix 1538 ⟩ Used in section 1526*.
⟨ Implement \pdfsetrandomseed 1583 ⟩ Used in section 1526*.
⟨ Implement \pdfsnaprefpoint 1570 ⟩ Used in section 1526*.
⟨ Implement \pdfsnapycomp 1573 ⟩ Used in section 1526*.
⟨ Implement \pdfsnapy 1572 ⟩ Used in section 1526*.
⟨ Implement \pdfstartlink 1558 ⟩ Used in section 1526*.
⟨ Implement \pdfstartthread 1566 ⟩ Used in section 1526*.
⟨ Implement \pdfthread 1565 ⟩ Used in section 1526*.
⟨ Implement \pdftrailerid 1580 ⟩ Used in section 1526*.
⟨ Implement \pdftrailer 1579 ⟩ Used in section 1526*.
⟨ Implement \pdfxform 1546 ⟩ Used in section 1526*.
⟨ Implement \pdfximage 1551 ⟩ Used in section 1526*.
⟨ Implement \setlanguage 1622 ⟩ Used in section 1526*.
⟨ Implement \special 1532* ⟩ Used in section 1526*.
⟨ Implement \write 1530 ⟩ Used in section 1526*.
⟨ Incorporate a whatsit node into a vbox 1603 ⟩ Used in section 843.
⟨ Incorporate a whatsit node into an hbox 1604 ⟩ Used in section 823.
⟨ Incorporate box dimensions into the dimensions of the hbox that will contain it 825 ⟩ Used in section 823.
⟨ Incorporate box dimensions into the dimensions of the vbox that will contain it 844 ⟩ Used in section 843.
⟨ Incorporate character dimensions into the dimensions of the hbox that will contain it, then move to the next node 826 ⟩ Used in section 823.
⟨ Incorporate glue into the horizontal totals 830 ⟩ Used in section 823.
⟨ Incorporate glue into the vertical totals 845 ⟩ Used in section 843.
⟨ Increase the number of parameters in the last font 607 ⟩ Used in section 605.
⟨ Increase k until x can be multiplied by a factor of 2^{-k}, and adjust y accordingly 120 ⟩ Used in section 119.
⟨ Initialize additional fields of the first active node 1842 ⟩ Used in section 1038.
⟨ Initialize bigger nodes with *SyncTEX* information 1913* ⟩ Used in section 143*.
⟨ Initialize for hyphenating a paragraph 1066 ⟩ Used in section 1037.
⟨ Initialize synctex primitive 1910* ⟩ Used in section 1510*.
⟨ Initialize table entries (done by INITEX only) 182, 240*, 246, 250, 258*, 268, 277*, 578*, 672, 1062, 1121*, 1126*, 1392, 1477*, 1613, 1650, 1815, 1851* ⟩ Used in section 8*.
⟨ Initialize the LR stack 1707 ⟩ Used in sections 821, 1711, and 1731.
⟨ Initialize the current page, insert the \topskip glue ahead of p, and **goto** *continue* 1176 ⟩ Used in section 1175.
⟨ Initialize the input routines 353* ⟩ Used in section 1515*.

⟨Initialize the output routines 55, 61*, 554, 559⟩ Used in section 1510*.
⟨Initialize the print *selector* based on *interaction* 75⟩ Used in sections 1441* and 1515*.
⟨Initialize the special list heads and constant nodes 964, 971, 994, 1156, 1163*⟩ Used in section 182.
⟨Initialize variables as *pdf_ship_out* begins 752⟩ Used in section 751.
⟨Initialize variables as *ship_out* begins 645*⟩ Used in section 668*.
⟨Initialize variables for PDF output 792⟩ Used in section 750*.
⟨Initialize variables for ε-TEX compatibility mode 1809⟩ Used in sections 1650 and 1652.
⟨Initialize variables for ε-TEX extended mode 1810⟩ Used in sections 1645* and 1652.
⟨Initialize whatever TEX might access 8*, 1909*⟩ Used in section 4*.
⟨Initialize *hlist_out* for mixed direction typesetting 1711⟩ Used in sections 647* and 729*.
⟨Initiate input from new pseudo file 1752*⟩ Used in section 1750.
⟨Initiate or terminate input from a file 404⟩ Used in section 391.
⟨Initiate the construction of an hbox or vbox, then **return** 1259⟩ Used in section 1255.
⟨Input and store tokens from the next line of the file 509⟩ Used in section 508.
⟨Input for \read from the terminal 510⟩ Used in section 509.
⟨Input from external file, **goto** *restart* if no input found 365*⟩ Used in section 363*.
⟨Input from token list, **goto** *restart* if end of list or if a parameter needs to be expanded 379*⟩ Used in section 363*.
⟨Input the first line of *read_file*[*m*] 511⟩ Used in section 509.
⟨Input the next line of *read_file*[*m*] 512⟩ Used in section 509.
⟨Insert LR nodes at the beginning of the current line and adjust the LR stack based on LR nodes in this line 1704⟩ Used in section 1054.
⟨Insert LR nodes at the end of the current line 1706⟩ Used in section 1054.
⟨Insert a delta node to prepare for breaks at *cur_p* 1017⟩ Used in section 1010.
⟨Insert a delta node to prepare for the next active node 1018⟩ Used in section 1010.
⟨Insert a dummy noad to be sub/superscripted 1353⟩ Used in section 1352.
⟨Insert a new active node from *best_place*[*fit_class*] to *cur_p* 1019⟩ Used in section 1010.
⟨Insert a new control sequence after p, then make p point to it 279*⟩ Used in section 278.
⟨Insert a new pattern into the linked trie 1138*⟩ Used in section 1136.
⟨Insert a new primitive after p, then make p point to it 282⟩ Used in section 281.
⟨Insert a new trie node between q and p, and make p point to it 1139*⟩ Used in sections 1138*, 1852, and 1853.
⟨Insert a token containing *frozen_endv* 401⟩ Used in section 388*.
⟨Insert a token saved by \afterassignment, if any 1445⟩ Used in section 1387*.
⟨Insert glue for *split_top_skip* and set $p \leftarrow null$ 1144⟩ Used in section 1143.
⟨Insert hyphens as specified in *hyph_list*[*h*] 1107⟩ Used in section 1106*.
⟨Insert macro parameter and **goto** *restart* 381⟩ Used in section 379*.
⟨Insert the appropriate mark text into the scanner 412⟩ Used in section 391.
⟨Insert the current list into its environment 986⟩ Used in section 974.
⟨Insert the pair (s,p) into the exception table 1115*⟩ Used in section 1114*.
⟨Insert the $\langle v_j \rangle$ template and **goto** *restart* 963⟩ Used in section 364.
⟨Insert token p into TEX's input 348⟩ Used in section 304.
⟨Interpret code c and **return** if done 84*⟩ Used in section 83.
⟨Introduce new material from the terminal and **return** 87⟩ Used in section 84*.
⟨Issue an error message if $cur\_val = fmem\_ptr$ 606⟩ Used in section 605.
⟨Justify the line ending at breakpoint *cur_p*, and append it to the current vertical list, together with associated penalties and other insertions 1054⟩ Used in section 1051.
⟨Last-minute procedures 1511*, 1513*, 1514, 1516*⟩ Used in section 1508.
⟨Lengthen the preamble periodically 967⟩ Used in section 966.
⟨Let *cur_h* be the position of the first box, and set $leader\_wd + lx$ to the spacing between corresponding parts of boxes 655⟩ Used in sections 654 and 736.
⟨Let *cur_v* be the position of the first box, and set $leader\_ht + lx$ to the spacing between corresponding parts of boxes 664⟩ Used in sections 663 and 745.

⟨ Scan a numeric constant 470 ⟩ Used in section 466.
⟨ Scan a parameter until its delimiter string has been found; or, if $s = null$, simply scan the delimiter string 418 ⟩ Used in section 417.
⟨ Scan a subformula enclosed in braces and **return** 1329 ⟩ Used in section 1327.
⟨ Scan ahead in the buffer until finding a nonletter; if an expanded code is encountered, reduce it and **goto** *start_cs*; otherwise if a multiletter control sequence is found, adjust *cur_cs* and *loc*, and **goto** *found* 378* ⟩ Used in section 376*.
⟨ Scan an alphabetic character code into *cur_val* 468 ⟩ Used in section 466.
⟨ Scan an optional space 469 ⟩ Used in sections 468, 474, 481, 705, 1376, 1542, 1554, 1554, 1556, and 1563.
⟨ Scan and build the body of the token list; **goto** *found* when finished 503 ⟩ Used in section 499.
⟨ Scan and build the parameter part of the macro definition 500 ⟩ Used in section 499.
⟨ Scan and evaluate an expression e of type l 1780 ⟩ Used in section 1779.
⟨ Scan decimal fraction 478 ⟩ Used in section 474.
⟨ Scan file name in the buffer 557 ⟩ Used in section 556*.
⟨ Scan for all other units and adjust *cur_val* and f accordingly; **goto** *done* in the case of scaled points 484 ⟩ Used in section 479.
⟨ Scan for `fil` units; **goto** *attach_fraction* if found 480 ⟩ Used in section 479.
⟨ Scan for `mu` units and **goto** *attach_fraction* 482 ⟩ Used in section 479.
⟨ Scan for units that are internal dimensions; **goto** *attach_sign* with *cur_val* set if found 481 ⟩ Used in section 479.
⟨ Scan preamble text until *cur_cmd* is *tab_mark* or *car_ret*, looking for changes in the tabskip glue; append an alignrecord to the preamble list 953 ⟩ Used in section 951.
⟨ Scan the argument for command c 497 ⟩ Used in section 496.
⟨ Scan the font size specification 1434 ⟩ Used in section 1433*.
⟨ Scan the next operator and set o 1781 ⟩ Used in section 1780.
⟨ Scan the parameters and make $link(r)$ point to the macro body; but **return** if an illegal `\par` is detected 417 ⟩ Used in section 415.
⟨ Scan the preamble and record it in the *preamble* list 951 ⟩ Used in section 948.
⟨ Scan the template $\langle u_j\rangle$, putting the resulting token list in *hold_head* 957 ⟩ Used in section 953.
⟨ Scan the template $\langle v_j\rangle$, putting the resulting token list in *hold_head* 958 ⟩ Used in section 953.
⟨ Scan units and set *cur_val* to $x \cdot (cur\_val + f/2^{16})$, where there are x sp per unit; **goto** *attach_sign* if the units are internal 479 ⟩ Used in section 474.
⟨ Search *eqtb* for equivalents equal to p 273 ⟩ Used in section 190.
⟨ Search *hyph_list* for pointers to p 1108 ⟩ Used in section 190.
⟨ Search *save_stack* for equivalents that point to p 307 ⟩ Used in section 190.
⟨ Select the appropriate case and **return** or **goto** *common_ending* 535 ⟩ Used in section 527*.
⟨ Set initial values of key variables 21, 23*, 24*, 74*, 77, 80, 97, 118, 184, 233*, 272, 276*, 294, 309, 390, 409, 465, 507, 516, 577*, 582, 620, 623, 633, 677, 681, 688, 697, 709, 711, 724, 818, 828, 836, 859, 945, 1103*, 1165, 1208, 1443, 1458, 1476, 1521, 1549, 1569, 1582, 1626, 1631, 1703, 1748, 1814, 1833, 1857, 1865*, 1876*, 1880*, 1895* ⟩ Used in section 8*.
⟨ Set line length parameters in preparation for hanging indentation 1023 ⟩ Used in section 1022.
⟨ Set the glue in all the unset boxes of the current list 979 ⟩ Used in section 974.
⟨ Set the glue in node r and change it from an unset node 982 ⟩ Used in section 981.
⟨ Set the unset box q and the unset boxes in it 981 ⟩ Used in section 979.
⟨ Set the value of b to the badness for shrinking the line, and compute the corresponding *fit_class* 1027 ⟩ Used in section 1025.
⟨ Set the value of b to the badness for stretching the line, and compute the corresponding *fit_class* 1026 ⟩ Used in section 1025.
⟨ Set the value of b to the badness of the last line for shrinking, compute the corresponding *fit_class*, and **goto** *found* 1845 ⟩ Used in section 1843.
⟨ Set the value of b to the badness of the last line for stretching, compute the corresponding *fit_class*, and **goto** *found* 1844 ⟩ Used in section 1843.
⟨ Set the value of *output_penalty* 1188 ⟩ Used in section 1187.