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.

If this program is changed, the resulting system should not be called '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.

define *TeX_banner_k* ≡ `´This␣is␣TeXk,␣Version␣3.1415926´` { printed when TEX starts }
define *TeX_banner* ≡ `´This␣is␣TeX,␣Version␣3.1415926´` { printed when TEX starts }
define *banner* ≡ *TeX_banner*
define *banner_k* ≡ *TeX_banner_k*

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` and `\tracingpages`.

```
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) 164 ⟩ **Tini**

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;

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 163 and 930.

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 }
```

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 ← 0 to '37 do xchr[i] ← i;
for i ← '177 to '377 do xchr[i] ← i;
```

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 $i < j < \acute{}177$, 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* ← *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] ≠ "␣".

An overflow error is given, however, if the normal actions of *input_ln* would make *last* ≥ *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*↑. 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*↑ 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
```

```
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 ← 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 ≤ 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 ← 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 ≥ 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;  { maximun 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 section 31*.

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 $-128 \ldots 127$. To accommodate such systems we access the string pool only via macros that can easily be redefined.

```
define si(#) ≡ #  { convert from ASCII_code to packed_ASCII_code }
define so(#) ≡ #  { 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 1391* ⟩
  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 }
    m, n: text_char;  { characters input from pool_file }
    g: str_number;  { garbage }
    a: integer;  { accumulator for check sum }
    c: boolean;  { check sum has been checked }
  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*[*'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 '$k \in [0, \text{'}10 \,..\, \text{'}12, \text{'}14, \text{'}15, \text{'}33, \text{'}177 \,..\, \text{'}377]$'. 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 $[\text{'}41 \,..\, \text{'}46, \text{'}60 \,..\, \text{'}71, \text{'}136, \text{'}141 \,..\, \text{'}146, \text{'}160 \,..\, \text{'}171]$ must be printable. Thus, at least 81 printable characters are needed.

```
⟨ Character k cannot be printed 49* ⟩ ≡
  (k < "␣") ∨ (k > "~")
```

This code is used in section 48.

51* **define** *bad_pool*(#) ≡
begin *wake_up_terminal*; *write_ln*(*term_out*, #); *a_close*(*pool_file*); *get_strings_started* ← *false*;
return;
end

⟨ Read the other strings from the TEX.POOL file and return *true*, or give an error message and return *false* 51* ⟩ ≡
name_length ← *strlen*(*pool_name*); *name_of_file* ← *xmalloc_array*(*ASCII_code*, *name_length* + 1);
strcpy(*stringcast*(*name_of_file* + 1), *pool_name*); { copy the string }
if *a_open_in*(*pool_file*, *kpse_texpool_format*) **then**
begin *c* ← *false*;
repeat ⟨ Read one string, but return *false* if the string memory space is getting too tight for comfort 52* ⟩;
until *c*;
a_close(*pool_file*); *get_strings_started* ← *true*;
end
else *bad_pool*(´!␣I␣can´´t␣read␣´, *pool_name*, ´;␣bad␣path?´)

This code is used in section 47*.

52* ⟨ Read one string, but return *false* if the string memory space is getting too tight for comfort 52* ⟩ ≡
begin if *eof*(*pool_file*) **then** *bad_pool*(´!␣´, *pool_name*, ´␣has␣no␣check␣sum.´);
read(*pool_file*, *m*); *read*(*pool_file*, *n*); { read two digits of string length }
if *m* = ´*´ **then** ⟨ Check the pool check sum 53* ⟩
else begin if (*xord*[*m*] < "0") ∨ (*xord*[*m*] > "9") ∨ (*xord*[*n*] < "0") ∨ (*xord*[*n*] > "9") **then**
bad_pool(´!␣´, *pool_name*, ´␣line␣doesn´´t␣begin␣with␣two␣digits.´);
l ← *xord*[*m*] ∗ 10 + *xord*[*n*] − "0" ∗ 11; { compute the length }
if *pool_ptr* + *l* + *string_vacancies* > *pool_size* **then** *bad_pool*(´!␣You␣have␣to␣increase␣POOLSIZE.´);
for *k* ← 1 **to** *l* **do**
begin if *eoln*(*pool_file*) **then** *m* ← ´␣´ **else** *read*(*pool_file*, *m*);
append_char(*xord*[*m*]);
end;
read_ln(*pool_file*); *g* ← *make_string*;
end
end

This code is used in section 51*.

53* The WEB operation @$ denotes the value that should be at the end of this TEX.POOL file; any other value means that the wrong pool file has been loaded.

⟨ Check the pool check sum 53* ⟩ ≡
begin *a* ← 0; *k* ← 1;
loop begin if (*xord*[*n*] < "0") ∨ (*xord*[*n*] > "9") **then**
bad_pool(´!␣´, *pool_name*, ´␣check␣sum␣doesn´´t␣have␣nine␣digits.´);
a ← 10 ∗ *a* + *xord*[*n*] − "0";
if *k* = 9 **then goto** *done*;
incr(*k*); *read*(*pool_file*, *n*);
end;
done: **if** *a* ≠ @$ **then**
bad_pool(´!␣´, *pool_name*, ´␣doesn´´t␣match;␣tangle␣me␣again␣(or␣fix␣the␣path).´);
c ← *true*;
end

This code is used in section 52*.

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 }
```

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 the banner and format identifier together will occupy at most *max_print_line* character positions.

```
⟨ 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 slow_print(format_ident);
  print_ln;
  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 and goes to *end_of_TEX*. This is the only nontrivial **goto** statement in the whole program. It is used when there is no recovery from a particular error.

Some Pascal compilers do not implement non-local **goto** statements. In such cases the body of *jump_out* should simply be '*close_files_and_terminate*;' followed by a call on some system procedure that quietly terminates the program.

```
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
      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 ⟩ +≡

110.* 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 }
```

111.* 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 ≠ mem_bot) ∨ (mem_max ≠ mem_top) then bad ← 10;
  tini
  if (mem_min > mem_bot) ∨ (mem_max < mem_top) then bad ← 10;
  if (min_quarterword > 0) ∨ (max_quarterword < 127) then bad ← 11;
  if (min_halfword > 0) ∨ (max_halfword < 32767) then bad ← 12;
  if (min_quarterword < min_halfword) ∨ (max_quarterword > max_halfword) then bad ← 13;
  if (mem_min < min_halfword) ∨ (mem_max ≥ max_halfword) ∨
      (mem_bot − mem_min > max_halfword + 1) then bad ← 14;
  if (max_font_max < min_halfword) ∨ (max_font_max > max_halfword) then bad ← 15;
  if font_max > font_base + max_font_max then bad ← 16;
  if (save_size > max_halfword) ∨ (max_strings > max_halfword) then bad ← 17;
  if buf_size > max_halfword then bad ← 18;
  if max_quarterword − min_quarterword < 255 then bad ← 19;
```

112.* 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.

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

113.* The reader should study the following definitions closely:

define *sc* ≡ *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 }
`#include␣"texmfmem.h";` *word_file* = **file of** *memory_word*;

116.* 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 }
```

127.* Empirical tests show that the routine in this section performs a node-merging operation about 0.75 times per allocation, on the average, after which it finds that $r > p + 1$ about 95% of the time.

```
⟨ Try to allocate within node p and its physical successors, and goto found if allocation was possible 127* ⟩ ≡
  q ← p + node_size(p);   { find the physical successor }
  while is_empty(q) do   { merge node p with node q }
    begin t ← rlink(q);
    if q = rover then rover ← t;
    llink(t) ← llink(q); rlink(llink(q)) ← t;
    q ← q + node_size(q);
    end;
  r ← q − s;
  if r > intcast(p + 1) then ⟨ Allocate from the top of node p and goto found 128 ⟩;
  if r = p then
    if rlink(p) ≠ p then ⟨ Allocate entire node p and goto found 129 ⟩;
  node_size(p) ← q − p   { reset the size in case it grew }
```

This code is used in section 125.

144:* 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:* 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

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

```
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 the font identifier for font(p) 267 ⟩;
          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 175 ⟩;
    p ← link(p);
    end;
  end;
```

176.* 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 the font identifier for font(p) 267 ⟩;
    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;
```

186.* 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) 186* ⟩ ≡
  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 184.

209:* 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`) }
define *set_box_dimen* = 83 { change dimension of box (`\wd`, `\ht`, `\dp`) }
define *set_shape* = 84 { specify fancy paragraph shape (`\parshape`) }
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`) }
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`) }
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 *max_command* = 100 { the largest command code seen at *big_switch* }

211.* 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 in the *ship_out* routine.

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;
```

213.* **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 *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` }

215.* 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*;
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; *page_depth* ← 0; *page_max_depth* ← 0;

219.* ⟨ Show the auxiliary field, *a* 219* ⟩ ≡
case *abs*(*m*) **div** (*max_command* + 1) **of**
0: **begin** *print_nl*(`"prevdepth␣"`);
if *a.sc* ≤ *ignore_depth* **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␣be␣denominator␣of:"`); *show_box*(*a.int*); **end**;
end { there are no other cases }

This code is used in section 218.

220.* 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* .. (*hash_base* − 1)] holds the current equivalents of single-character control sequences.

2) *eqtb*[*hash_base* .. (*glue_base* − 1)] holds the current equivalents of multiletter control sequences.

3) *eqtb*[*glue_base* .. (*local_base* − 1)] holds the current equivalents of glue parameters like the current baselineskip.

4) *eqtb*[*local_base* .. (*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* .. (*dimen_base* − 1)] holds the current equivalents of fullword integer parameters like the current hyphenation penalty.

6) *eqtb*[*dimen_base* .. *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.

222.* 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 *frozen_special* = *frozen_control_sequence* + 10 { permanent '`\special`' }
define *frozen_null_font* = *frozen_control_sequence* + 11 { permanent '`\nullfont`' }
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) 164 ⟩ +≡
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*];

230.* 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 *toks_base* = *local_base* + 10 { table of 256 token list registers }
define *box_base* = *toks_base* + 256 { table of 256 box registers }
define *cur_font_loc* = *box_base* + 256 { internal font number outside math mode }
define *math_font_base* = *cur_font_loc* + 1 { table of 48 math font numbers }
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*(*every_job_loc*)
define *every_cr* ≡ *equiv*(*every_cr_loc*)
define *err_help* ≡ *equiv*(*err_help_loc*)
define *toks*(#) ≡ *equiv*(*toks_base* + #)
define *box*(#) ≡ *equiv*(*box_base* + #)
define *cur_font* ≡ *equiv*(*cur_font_loc*)
define *fam_fnt*(#) ≡ *equiv*(*math_font_base* + #)
define *cat_code*(#) ≡ *equiv*(*cat_code_base* + #)
define *lc_code*(#) ≡ *equiv*(*lc_code_base* + #)
define *uc_code*(#) ≡ *equiv*(*uc_code_base* + #)
define *sf_code*(#) ≡ *equiv*(*sf_code_base* + #)
define *math_code*(#) ≡ *equiv*(*math_code_base* + #)
{ Note: *math_code*(*c*) is the true math code plus *min_halfword* }
define *char_sub_code*(#) ≡ *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 226 ⟩ +≡
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*);

236.* 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 heldover 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 *web2c_int_pars* = *web2c_int_base* + 3 { total number of web2c's integer parameters }
define *int_pars* = *web2c_int_pars* { 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*(#) ≡ *eqtb*[*del_code_base* + #].*int*
define *count*(#) ≡ *eqtb*[*count_base* + #].*int*
define *int_par*(#) ≡ *eqtb*[*int_base* + #].*int* { an integer parameter }
define *pretolerance* ≡ *int_par*(*pretolerance_code*)
define *tolerance* ≡ *int_par*(*tolerance_code*)
define *line_penalty* ≡ *int_par*(*line_penalty_code*)
define *hyphen_penalty* ≡ *int_par*(*hyphen_penalty_code*)
define *ex_hyphen_penalty* ≡ *int_par*(*ex_hyphen_penalty_code*)
define *club_penalty* ≡ *int_par*(*club_penalty_code*)
define *widow_penalty* ≡ *int_par*(*widow_penalty_code*)
define *display_widow_penalty* ≡ *int_par*(*display_widow_penalty_code*)
define *broken_penalty* ≡ *int_par*(*broken_penalty_code*)
define *bin_op_penalty* ≡ *int_par*(*bin_op_penalty_code*)
define *rel_penalty* ≡ *int_par*(*rel_penalty_code*)
define *pre_display_penalty* ≡ *int_par*(*pre_display_penalty_code*)
define *post_display_penalty* ≡ *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 *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*)

⟨ Assign the values *depth_threshold* ← *show_box_depth* and *breadth_max* ← *show_box_breadth* 236* ⟩ ≡
depth_threshold ← *show_box_depth*; *breadth_max* ← *show_box_breadth*

This code is used in section 198.

237*. 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");
othercases *print*("[unknown␣integer␣parameter!]")
endcases;
end;

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

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
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;

240.* 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) 164 ⟩ +≡
for *k* ← *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* ← 0 **to** 255 **do** *del_code*(*k*) ← −1;
del_code(".") ← 0; { this null delimiter is used in error recovery }

241.* 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 macro-defined *date_and_time* routine. *date_and_time* in turn is a C macro, which calls *get_date_and_time*, passing it the addresses of the day, month, 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.

define *fix_date_and_time* ≡ *date_and_time*(*time*, *day*, *month*, *year*)

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

⟨ Declare the procedure called *print_cmd_chr* 298 ⟩
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* ≤ *eqtb_top*)) **then** ⟨ Show equivalent *n*, in region 1 or 2 223 ⟩
else if *n* < *local_base* **then** ⟨ Show equivalent *n*, in region 3 229 ⟩
else if *n* < *int_base* **then** ⟨ Show equivalent *n*, in region 4 233 ⟩
else if *n* < *dimen_base* **then** ⟨ Show equivalent *n*, in region 5 242 ⟩
else if *n* ≤ *eqtb_size* **then** ⟨ Show equivalent *n*, in region 6 251 ⟩
else *print_char*("?"); { this can't happen either }
end;
tats

253.* 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*;

256.* 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(#) ≡ hash[#].lh    { link for coalesced lists }
define text(#) ≡ hash[#].rh    { string number for control sequence name }
define hash_is_full ≡ (hash_used = hash_base)    { test if all positions are occupied }
define font_id_text(#) ≡ 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 }
```

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

```
no_new_control_sequence ← true;    { new identifiers are usually forbidden }
```

258.* ⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡

```
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:";
```

260:* ⟨ Insert a new control sequence after p, then make p point to it 260* ⟩ ≡

```
begin if text(p) > 0 then
  begin if hash_high < hash_extra then
    begin incr(hash_high); next(p) ← hash_high + eqtb_size; p ← 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) ← hash_used; p ← hash_used;
    end;
  end;
str_room(l); d ← cur_length;
while pool_ptr > str_start[str_ptr] do
  begin decr(pool_ptr); str_pool[pool_ptr + l] ← str_pool[pool_ptr];
  end;  { move current string up to make room for another }
for k ← j to j + l − 1 do append_char(buffer[k]);
text(p) ← make_string; pool_ptr ← pool_ptr + d;
stat incr(cs_count); tats
end
```

This code is used in section 259.

262:* 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.

⟨ Basic printing procedures 57 ⟩ +≡

```
procedure print_cs(p : integer);  { prints a purported control sequence }
  begin if p < hash_base then  { single character }
    if p ≥ single_base then
      if p = null_cs then
        begin print_esc("csname"); print_esc("endcsname");
        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 print_esc(text(p)); print_char("␣");
        end;
  end;
```

271:* ⟨ 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 }
```

283:* 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 283* ⟩ ≡
```
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 282.

290:* ⟨ Check the "constant" values for consistency 14 ⟩ +≡
if *cs_token_flag* + *eqtb_size* + *hash_extra* > *max_halfword* **then** *bad* ← 21;
if (*hash_offset* < 0) ∨ (*hash_offset* > *hash_base*) **then** *bad* ← 42;

301:* ⟨ 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) }

304:* 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*;

306.* 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 306* ⟩ ≡
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 119.

308.* 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 }
```

328.* 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; line_stack[index] ← line; start ← first; state ← mid_line;
  name ← 0;  { terminal_input is now true }
  end;
```

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

```
⟨ Initialize the input routines 331* ⟩ ≡
  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; 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 1340*.

338.* ⟨ Tell the user what has run away and try to recover 338* ⟩ ≡

```
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 339* ⟩;
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 336.

339.* 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 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 339* ⟩ ≡

```
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 338*.

366.* 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*.

367.* Sometimes, recursive calls to the following *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;
```

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

```
expand_depth_count ← 0;
```

369.* 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 392 ⟩
⟨ Declare the procedure called insert_relax 382 ⟩
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;
  var t: halfword; { token that is being "expanded after" }
    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);
  if cur_cmd < call then ⟨ Expand a nonmacro 370 ⟩
  else if cur_cmd < end_template then macro_call
    else ⟨ Insert a token containing frozen_endv 378 ⟩;
  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;
```

504.* ⟨ Either process `\ifcase` or set b to the value of a boolean condition 504* ⟩ ≡

```
case this_if of
if_char_code, if_cat_code: ⟨ Test if two characters match 509 ⟩;
if_int_code, if_dim_code: ⟨ Test relation between integers or dimensions 506 ⟩;
if_odd_code: ⟨ Test if an integer is odd 507 ⟩;
if_vmode_code: b ← (abs(mode) = vmode);
if_hmode_code: b ← (abs(mode) = hmode);
if_mmode_code: b ← (abs(mode) = mmode);
if_inner_code: b ← (mode < 0);
if_void_code, if_hbox_code, if_vbox_code: ⟨ Test box register status 508 ⟩;
ifx_code: ⟨ Test if two tokens match 510 ⟩;
if_eof_code: begin scan_four_bit_int_or_18;
  if cur_val = 18 then b ← ¬shellenabledp
  else b ← (read_open[cur_val] = closed);
  end;
if_true_code: b ← true;
if_false_code: b ← false;
if_case_code: ⟨ Select the appropriate case and return or goto common_ending 512 ⟩;
end { there are no other cases }
```

This code is used in section 501.

516.* 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 }

517.* 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.

518.* 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;
```

519.* 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;
```

520.* 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 ← ""
else begin cur_area ← str_ptr; str_start[str_ptr + 1] ← str_start[str_ptr] + area_delimiter; incr(str_ptr);
  temp_str ← search_string(cur_area);
  if temp_str > 0 then
    begin cur_area ← temp_str; decr(str_ptr); { no flush_string, pool_ptr will be wrong! }
    for j ← str_start[str_ptr + 1] to pool_ptr − 1 do
      begin str_pool[j − area_delimiter] ← str_pool[j];
      end;
    pool_ptr ← pool_ptr − area_delimiter; { update pool_ptr }
    end;
  end;
if ext_delimiter = 0 then
  begin cur_ext ← ""; cur_name ← slow_make_string;
  end
else begin cur_name ← str_ptr;
  str_start[str_ptr + 1] ← str_start[str_ptr] + ext_delimiter − area_delimiter − 1; incr(str_ptr);
  cur_ext ← make_string; decr(str_ptr); { undo extension string to look at name part }
  temp_str ← search_string(cur_name);
  if temp_str > 0 then
    begin cur_name ← temp_str; decr(str_ptr); { no flush_string, pool_ptr will be wrong! }
    for j ← str_start[str_ptr + 1] to pool_ptr − 1 do
      begin str_pool[j − ext_delimiter + area_delimiter + 1] ← str_pool[j];
      end;
    pool_ptr ← pool_ptr − ext_delimiter + area_delimiter + 1; { update pool_ptr }
    end;
  cur_ext ← slow_make_string; { remake extension string }
  end;
end;
```

521.* 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.)

```
⟨ 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;
  if a ≠ 0 then
    begin j ← str_start[a];
    while (¬must_quote) ∧ (j < str_start[a + 1]) do
      begin must_quote ← str_pool[j] = "␣"; incr(j);
      end;
    end;
  if n ≠ 0 then
    begin j ← str_start[n];
    while (¬must_quote) ∧ (j < str_start[n + 1]) do
      begin must_quote ← str_pool[j] = "␣"; incr(j);
      end;
    end;
  if e ≠ 0 then
    begin j ← str_start[e];
    while (¬must_quote) ∧ (j < str_start[e + 1]) do
      begin must_quote ← str_pool[j] = "␣"; incr(j);
      end;
    end; { 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("""");
  if a ≠ 0 then
    for j ← str_start[a] to str_start[a + 1] − 1 do
      if so(str_pool[j]) ≠ """" then print(so(str_pool[j]));
  if n ≠ 0 then
    for j ← str_start[n] to str_start[n + 1] − 1 do
      if so(str_pool[j]) ≠ """" then print(so(str_pool[j]));
  if e ≠ 0 then
    for j ← str_start[e] to str_start[e + 1] − 1 do
      if so(str_pool[j]) ≠ """" then print(so(str_pool[j]));
  if must_quote then print_char("""");
  end;
```

522.* 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;
```

523.* 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;
```

524.* 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.

526.* 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 .. 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 ← a + file_name_size − n − 1 − format_ext_length;
  k ← 0;
  if name_of_file then libc_free(name_of_file);
  name_of_file ← xmalloc_array(ASCII_code, n + (b − a + 1) + format_ext_length + 1);
  for j ← 1 to n do append_to_name(xord[ucharcast(TEX_format_default[j])]);
  for j ← a to b do append_to_name(buffer[j]);
  for j ← format_default_length − format_ext_length + 1 to format_default_length do
    append_to_name(xord[ucharcast(TEX_format_default[j])]);
  if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
  name_of_file[name_length + 1] ← 0;
  end;
```

527.* 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 527* ⟩ ≡
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 1306*.

528:* 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) ∨ (str_ptr = max_strings) ∨ (cur_length > 0) then
    make_name_string ← "?"
  else begin for k ← 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 ← 1;
    while (k ≤ name_length) ∧ (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;
```

529:* 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;
  begin name_in_progress ← true; begin_name; ⟨ Get the next non-blank non-call token 409 ⟩;
  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;
done: end_name; name_in_progress ← false;
  end;
```

533.* 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 534 ⟩;
  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;
```

535.* 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 }
```

537* 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 538 ⟩;
  log_name ← a_make_name_string(log_file); selector ← log_only; log_opened ← true;
  ⟨ Print the banner line, including the date and time 539* ⟩;
  if mltex_enabled_p then
    begin wlog_cr; wlog('MLTeX v2.2 enabled');
    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;
```

539* ⟨ Print the banner line, including the date and time 539* ⟩ ≡

```
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(day); print_char("␣");
months ← ´␣JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC´;
for k ← 3 * month − 2 to 3 * month do wlog(months[k]);
print_char("␣"); print_int(year); print_char("␣"); print_two(time div 60); print_char(":");
print_two(time mod 60);
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 537*.

540.* Let's turn now to the procedure that is used to initiate file reading when an '`\input`' command is being processed.

```
procedure start_input; { TEX will \input something }
  label done;
  var temp_str: str_number;
  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;
  state ← new_line; ⟨ Read the first line of the new file 541 ⟩;
  end;
```

551.* 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*;

552.* 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*; { right 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* }

553.* 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 }

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

555:* 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) 164 ⟩ +≡

557* 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 $char\_list\_accent(\#) \equiv (ho(char\_sub\_code(\#)) \textbf{ div } 256)$
define $char\_list\_char(\#) \equiv (ho(char\_sub\_code(\#)) \textbf{ mod } 256)$
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)
```

563:* 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 1396* ⟩
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; { right 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 565 ⟩;
bad_tfm: ⟨ Report that the font won't be loaded 564* ⟩;
done: if file_opened then b_close(tfm_file);
  read_font_info ← g;
  end;
```

564.* 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 ≥ 0 then
      begin print("␣at␣"); print_scaled(s); print("pt");
      end
    else if s ≠ −1000 then
        begin print("␣scaled␣"); print_int(−s);
        end
```

⟨ Report that the font won't be loaded 564* ⟩ ≡

```
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 563*.

566.* ⟨ Open *tfm_file* for input 566* ⟩ ≡

```
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 565.

567.* 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
```

573:* 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) ∨ (# > ec) then abort
        end
define current_character_being_worked_on ≡ k + bc − fmem_ptr
```

```
⟨ Check for charlist cycle 573* ⟩ ≡
  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 572.

576:*
```
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 576* ⟩ ≡
  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 ≥ 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 ≥ nk then abort;  { check kern }
        if a < 128 then
          if k − lig_kern_base[f] + a + 1 ≥ 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 565.

578:* 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 578* ⟩ ≡
  begin for k ← 1 to np do
    if k = 1 then { the slant parameter is a pure number }
      begin fget; sw ← fbyte;
      if sw > 127 then sw ← sw − 256;
      fget; sw ← sw * ´400 + fbyte; fget; sw ← sw * ´400 + fbyte; fget;
      font_info[param_base[f]].sc ← (sw * ´20) + (fbyte div ´20);
      end
    else store_scaled(font_info[param_base[f] + k − 1].sc);
  if feof(tfm_file) then abort;
  for k ← np + 1 to 7 do font_info[param_base[f] + k − 1].sc ← 0;
  end
```

This code is used in section 565.

579:* 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(#) ≡ #[f] ← qo(#[f])  { correct for the excess min_quarterword that was added }
⟨ Make final adjustments and goto done 579* ⟩ ≡
  if np ≥ 7 then font_params[f] ← np else font_params[f] ← 7;
  hyphen_char[f] ← default_hyphen_char; skew_char[f] ← default_skew_char;
  if bch_label < nl then bchar_label[f] ← bch_label + lig_kern_base[f]
  else bchar_label[f] ← non_address;
  font_bchar[f] ← qi(bchar); font_false_bchar[f] ← qi(bchar);
  if bchar ≤ ec then
    if bchar ≥ bc then
      begin qw ← orig_char_info(f)(bchar);  { N.B.: not qi(bchar) }
      if char_exists(qw) then font_false_bchar[f] ← non_char;
      end;
  font_name[f] ← nom; font_area[f] ← aire; font_bc[f] ← bc; font_ec[f] ← ec; font_glue[f] ← null;
  adjust(char_base); adjust(width_base); adjust(lig_kern_base); adjust(kern_base); adjust(exten_base);
  decr(param_base[f]); fmem_ptr ← fmem_ptr + lf; font_ptr ← f; g ← f; goto done
```

This code is used in section 565.

585:* 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;
```

595* 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 }

598* 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* }

600* The actual output of *dvi_buf*[*a* .. *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.

605.* 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 ≤ 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 606 ⟩;
  end;
```

620.* ⟨ Initialize variables as *ship_out* begins 620* ⟩ ≡

```
dvi_h ← 0; dvi_v ← 0; cur_h ← h_offset; dvi_f ← null_font; 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 643*.

622:* 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 1371 ⟩
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;  { left edge of sub-box, or right edge 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);
  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; left_edge ← cur_h;
  while p ≠ null do ⟨ Output node p for hlist_out and move to the next node, maintaining the condition
      cur_v = base_line 623* ⟩;
  prune_movements(save_loc);
  if cur_s > 0 then dvi_pop(save_loc);
  decr(cur_s);
  end;
```

623.* 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 substitions 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 623* ⟩ ≡
reswitch: if is_char_node(p) then
    begin synch_h; synch_v;
    repeat f ← font(p); c ← character(p);
      if f ≠ dvi_f then ⟨ Change font dvi_f to f 624* ⟩;
      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 }
            begin if c ≥ qi(128) then dvi_out(set1);
            dvi_out(qo(c));
            cur_h ← 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 1398* ⟩;
    continue: p ← link(p);
    until ¬is_char_node(p);
    dvi_h ← cur_h;
    end
  else ⟨ Output the non-char_node p for hlist_out and move to the next node 625 ⟩
```

This code is used in section 622*.

624.* ⟨ Change font *dvi_f* to *f* 624* ⟩ ≡

```
begin if ¬font_used[f] then
    begin dvi_font_def(f); font_used[f] ← true;
    end;
  if f ≤ 64 + font_base then dvi_out(f − font_base − 1 + fnt_num_0)
  else if f ≤ 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 ← f;
  end
```

This code is used in section 623*.

643*. ⟨ Ship box p out 643* ⟩ ≡

```
⟨ Update the values of max_h and max_v; but if the page is too large, goto done 644 ⟩;
⟨ Initialize variables as ship_out begins 620* ⟩;
page_loc ← dvi_offset + dvi_ptr; dvi_out(bop);
for k ← 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 > 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 641.

645*. 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 645* ⟩ ≡
  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 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 646 ⟩;
    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 602 ⟩;
    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 1336*.

711*. ⟨ 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 711* ⟩ ≡

```
begin y ← x;
if (qo(y) ≥ font_bc[g]) ∧ (qo(y) ≤ font_ec[g]) then
  begin continue: q ← orig_char_info(g)(y);
  if char_exists(q) then
    begin if char_tag(q) = ext_tag then
      begin f ← g; c ← y; goto found;
      end;
    hd ← height_depth(q); u ← char_height(g)(hd) + char_depth(g)(hd);
    if u > w then
      begin f ← g; c ← y; w ← u;
      if u ≥ v then goto found;
      end;
    if char_tag(q) = list_tag then
      begin y ← rem_byte(q); goto continue;
      end;
    end;
  end
```

This code is used in section 710.

725:* 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 ← character(a); cur_f ← fam_fnt(fam(a) + cur_size);
  if cur_f = null_font then ⟨ Complain about an undefined family and set cur_i null 726 ⟩
  else begin if (qo(cur_c) ≥ font_bc[cur_f]) ∧ (qo(cur_c) ≤ font_ec[cur_f]) then
      cur_i ← orig_char_info(cur_f)(cur_c)
    else cur_i ← null_character;
    if ¬(char_exists(cur_i)) then
      begin char_warning(cur_f, qo(cur_c)); math_type(a) ← empty;
      end;
    end;
  end;
```

743:* ⟨ Switch to a larger accent if available and appropriate 743* ⟩ ≡

```
  loop begin if char_tag(i) ≠ list_tag then goto done;
    y ← rem_byte(i); i ← orig_char_info(f)(y);
    if ¬char_exists(i) then goto done;
    if char_width(f)(i) > w then goto done;
    c ← y;
    end;
done:
```

This code is used in section 741.

752.* 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 737 ⟩ +≡
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 753 ⟩;
  make_op ← delta;
  end;
```

862* ⟨ Compute the demerits, d, from r to cur_p 862* ⟩ ≡

```
begin d ← line_penalty + b;
if abs(d) ≥ 10000 then d ← 100000000 else d ← d * d;
if pi ≠ 0 then
  if pi > 0 then d ← d + pi * pi
  else if pi > eject_penalty then d ← d − pi * pi;
if (break_type = hyphenated) ∧ (type(r) = hyphenated) then
  if cur_p ≠ null then d ← d + double_hyphen_demerits
  else d ← d + final_hyphen_demerits;
if abs(intcast(fit_class) − intcast(fitness(r))) > 1 then d ← d + adj_demerits;
end
```

This code is used in section 858.

878:* The adjustment for a desired looseness is a slightly more complicated version of the loop just considered. Note that if a paragraph is broken into segments by displayed equations, each segment will be subject to the looseness calculation, independently of the other segments.

```
⟨ Find the best active node for the desired looseness 878* ⟩ ≡
  begin r ← link(active); actual_looseness ← 0;
  repeat if type(r) ≠ delta_node then
      begin line_diff ← intcast(line_number(r)) − intcast(best_line);
      if ((line_diff < actual_looseness) ∧ (looseness ≤ line_diff)) ∨
            ((line_diff > actual_looseness) ∧ (looseness ≥ line_diff)) then
        begin best_bet ← r; actual_looseness ← line_diff; fewest_demerits ← total_demerits(r);
        end
      else if (line_diff = actual_looseness) ∧ (total_demerits(r) < fewest_demerits) then
          begin best_bet ← r; fewest_demerits ← total_demerits(r);
          end;
      end;
    r ← link(r);
  until r = last_active;
  best_line ← line_number(best_bet);
  end
```

This code is used in section 876.

923.* 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 $p_1 \dots p_k$ 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 $n_0 \dots n_k$ associated with $p_1 \dots p_k$.

The theory that comparatively few different number sequences $n_0 \dots n_k$ 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 $trie\_op(z_k) \neq min\_trie\_op$, when $p_1 \dots p_k$ has matched the letters in $hc[(l-k+1) \,..\, l\,]$ 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 }

924.* 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 implented 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 }

926.* 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 926* ⟩ ≡
  for j ← 0 to hn do hyf[j] ← 0;
  ⟨ Look for the word hc[1 .. hn] in the exception table, and goto found (with hyf containing the hyphens)
      if an entry is found 933* ⟩;
  if trie_char(cur_lang + 1) ≠ qi(cur_lang) then return;  { no patterns for cur_lang }
  hc[0] ← 0; hc[hn + 1] ← 0; hc[hn + 2] ← 256;  { insert delimiters }
  for j ← 0 to hn − r_hyf + 1 do
    begin z ← trie_link(cur_lang + 1) + hc[j]; l ← j;
    while hc[l] = qo(trie_char(z)) do
      begin if trie_op(z) ≠ min_trie_op then ⟨ Store maximum values in the hyf table 927* ⟩;
      incr(l); z ← trie_link(z) + hc[l];
      end;
    end;
found: for j ← 0 to l_hyf − 1 do hyf[j] ← 0;
  for j ← 0 to r_hyf − 1 do hyf[hn − j] ← 0
```

This code is used in section 898.

927.* ⟨ Store maximum values in the *hyf* table 927* ⟩ ≡

```
  begin v ← trie_op(z);
  repeat v ← v + op_start[cur_lang]; i ← l − hyf_distance[v];
    if hyf_num[v] > hyf[i] then hyf[i] ← hyf_num[v];
    v ← hyf_next[v];
  until v = min_trie_op;
  end
```

This code is used in section 926*.

928.* 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 .. ssup_hyph_size;
      { index into hyphen exceptions hash table; enlarging this requires changing (un)dump code }
```

929* ⟨ 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 }

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

933* 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 933* ⟩ ≡
$h \leftarrow hc[1]$; *incr*(*hn*); $hc[hn] \leftarrow cur\_lang$;
for $j \leftarrow 2$ **to** *hn* **do** $h \leftarrow (h + h + hc[j])$ **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* 934* ⟩;
$h \leftarrow hyph\_link[h]$;
if $h = 0$ **then goto** *not_found*;
decr(*h*);
end;
not_found: *decr*(*hn*)
This code is used in section 926*.

934* ⟨ 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* 934* ⟩ ≡
{ This is now a simple hash list, not an ordered one, so the module title is no longer descriptive. }
$k \leftarrow hyph\_word[h]$;
if $k = 0$ **then goto** *not_found*;
if *length*(*k*) = *hn* **then**
begin $j \leftarrow 1$; $u \leftarrow str\_start[k]$;
repeat if *so*(*str_pool*[*u*]) ≠ *hc*[*j*] **then goto** *done*;
incr(*j*); *incr*(*u*);
until $j > hn$;
⟨ Insert hyphens as specified in *hyph_list*[*h*] 935 ⟩;
decr(*hn*); **goto** *found*;
end;
done:
This code is used in section 933*.

937* 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;
  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;
  ⟨ Enter as many hyphenation exceptions as are listed, until coming to a right brace; then return 938 ⟩;
exit: end;
```

942* ⟨ Enter a hyphenation exception 942* ⟩ ≡

```
  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 943* ⟩;
  end
```

This code is used in section 938.

943* ⟨ Insert the pair (s, p) into the exception table 943* ⟩ ≡

```
  if hyph_next ≤ hyph_prime then
    while (hyph_next > 0) ∧ (hyph_word[hyph_next − 1] > 0) do decr(hyph_next);
  if (hyph_count = hyph_size) ∨ (hyph_next = 0) then overflow("exception␣dictionary", hyph_size);
  incr(hyph_count);
  while hyph_word[h] ≠ 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) 944* ⟩;
    if hyph_link[h] = 0 then
      begin hyph_link[h] ← hyph_next;
      if hyph_next ≥ 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 942*.

944.* ⟨ If the string *hyph_word*[*h*] is less than or equal to *s*, interchange (*hyph_word*[*h*], *hyph_list*[*h*]) with (*s*, *p*) 944* ⟩ ≡

```
{ 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) ≠ length(s) then goto not_found;
u ← str_start[k]; v ← str_start[s];
repeat if str_pool[u] ≠ 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 943*.

946:* 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 }
```

947:* 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 947* ⟩ ≡
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(intcast(n) + 313 * intcast(d) + 361 * intcast(v) + 1009 * intcast(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 951*, 952, 956, 960, 962, 963*, and 969*.

This code is used in section 945.

948:* 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 948* ⟩ ≡
  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 955.

949:* 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) 164 ⟩ +≡
  for k ← −trie_op_size to trie_op_size do trie_op_hash[k] ← 0;
  for k ← 0 to 255 do trie_used[k] ← min_trie_op;
  max_op_used ← min_trie_op; trie_op_ptr ← 0;
```

950:* 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 ≡ 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
```

951.* Let us suppose that a linked trie has already been constructed. Experience shows that we can often reduce its size by recognizing common subtries; therefore another hash table is introduced for this purpose, somewhat similar to *trie_op_hash*. The new hash table will be initialized to zero.

The function *trie_node*(*p*) returns *p* if *p* is distinct from other nodes that it has seen, otherwise it returns the number of the first equivalent node that it has seen.

Notice that we might make subtries equivalent even if they correspond to patterns for different languages, in which the trie ops might mean quite different things. That's perfectly all right.

```
⟨ Declare procedures for preprocessing hyphenation patterns 947* ⟩ +≡
function trie_node(p : trie_pointer): trie_pointer; { converts to a canonical form }
  label exit;
  var h: trie_pointer; { trial hash location }
    q: trie_pointer; { trial trie node }
  begin h ← abs(intcast(trie_c[p]) + 1009 * intcast(trie_o[p]) +
      2718 * intcast(trie_l[p]) + 3142 * intcast(trie_r[p])) mod trie_size;
  loop begin q ← trie_hash[h];
    if q = 0 then
      begin trie_hash[h] ← p; trie_node ← p; return;
      end;
    if (trie_c[q] = trie_c[p]) ∧ (trie_o[q] = trie_o[p]) ∧ (trie_l[q] = trie_l[p]) ∧ (trie_r[q] = trie_r[p]) then
      begin trie_node ← q; return;
      end;
    if h > 0 then decr(h) else h ← trie_size;
    end;
exit: end;
```

953.* 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 ≡ trie_hash { where linked trie families go into trie }
  define trie_back(#) ≡ 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
```

954.* 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) 164 ⟩ +≡
  trie_not_ready ← true;
```

961:* 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* 961* ⟩ ≡

```
if trie_root = 0 then { no patterns were given }
  begin for r ← 0 to 256 do clear_trie;
  trie_max ← 256;
  end
else begin 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 969*.

963:* 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 947* ⟩ +≡

```
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 964 ⟩;
    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;
```

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

```
⟨ Insert a new pattern into the linked trie 966* ⟩ ≡
  begin ⟨ Compute the trie op code, v, and set l ← 0 968* ⟩;
  q ← 0; hc[0] ← cur_lang;
  while l ≤ k do
    begin c ← hc[l]; incr(l); p ← trie_l[q]; first_child ← true;
    while (p > 0) ∧ (c > so(trie_c[p])) do
      begin q ← p; p ← trie_r[q]; first_child ← false;
      end;
    if (p = 0) ∨ (c < so(trie_c[p])) then
      ⟨ Insert a new trie node between q and p, and make p point to it 967* ⟩;
    q ← p;  { now node q represents p_1 ... p_{l-1} }
    end;
  if trie_o[q] ≠ min_trie_op then
    begin print_err("Duplicate␣pattern"); help1("(See␣Appendix␣H.)"); error;
    end;
  trie_o[q] ← v;
  end
```

This code is used in section 964.

967.* ⟨ Insert a new trie node between q and p, and make p point to it 967* ⟩ ≡

```
  begin if trie_ptr = trie_size then overflow("pattern␣memory", trie_size);
  incr(trie_ptr); trie_r[trie_ptr] ← p; p ← trie_ptr; trie_l[p] ← 0;
  if first_child then trie_l[q] ← p else trie_r[q] ← p;
  trie_c[p] ← si(c); trie_o[p] ← min_trie_op;
  end
```

This code is used in section 966*.

968.* ⟨ Compute the trie op code, v, and set $l \leftarrow 0$ 968* ⟩ ≡

```
  if hc[1] = 0 then hyf[0] ← 0;
  if hc[k] = 0 then hyf[k] ← 0;
  l ← k; v ← min_trie_op;
  loop begin if hyf[l] ≠ 0 then v ← 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 966*.

969.* 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 947* ⟩ +≡
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 955 ⟩;
  if trie_root ≠ 0 then
    begin first_fit(trie_root); trie_pack(trie_root);
    end;
  ⟨ Move the data into trie 961* ⟩;
  trie_not_ready ← false;
  end;
```

1037* 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 1037* ⟩ ≡

```
if ((head = tail) ∧ (mode > 0)) then
  begin if (insert_src_special_auto) then append_src_special;
  end;
adjust_space_factor;
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;
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 1038 ⟩;

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* 1039* ⟩;

main_loop_lookahead: ⟨ Look ahead for another character, or leave *lig_stack* empty if there's none there 1041 ⟩;

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* 1042 ⟩;

main_loop_move_lig: ⟨ Move the cursor past a pseudo-ligature, then **goto** *main_loop_lookahead* or *main_lig_loop* 1040 ⟩

This code is used in section 1033.

1039.* ⟨ 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* 1039* ⟩ ≡

```
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 1037*.

1052.* 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 1046 ⟩ +≡
procedure you_cant;
  begin print_err("You␣can't␣use␣`"); print_cmd_chr(cur_cmd, cur_chr); print_in_mode(mode);
  end;
```

1094:* ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡

```
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;
```

1138:* ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡

```
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;
```

1142:* ⟨ Go into ordinary math mode 1142* ⟩ ≡
```
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 1141 and 1145.

1170:* ⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
```
mmode + vcenter: begin scan_spec(vcenter_group, false); normal_paragraph; push_nest; mode ← −vmode;
prev_depth ← ignore_depth;
if (insert_src_special_every_vbox) then insert_src_special;
if every_vbox ≠ null then begin_token_list(every_vbox, every_vbox_text);
end;
```

1218.* 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 1218* ⟩ ≡
procedure get_r_token;
  label restart;
  begin restart: repeat get_token;
  until cur_tok ≠ space_token;
  if (cur_cs = 0) ∨ (cur_cs > eqtb_top) ∨ ((cur_cs > frozen_control_sequence) ∧ (cur_cs ≤ 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 ← cs_token_flag + frozen_protection; ins_error; goto restart;
    end;
  end;
```

See also sections 1232, 1239, 1246, 1247, 1248, 1249, 1250, 1260*, and 1268*.

This code is used in section 1214.

1225.* 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 226 ⟩ +≡
  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;
```

1226.* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
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;
```

1227.* 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 1220 ⟩ +≡

```
shorthand_def: if cur_chr = char_sub_def_code then
    begin scan_char_num; p ← char_sub_code_base + cur_val; scan_optional_equals; scan_char_num;
    n ← 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 ← 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 ← cur_chr; get_r_token; p ← 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_eight_bit_int;
      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;
```

1255.* ⟨ Assignments 1220 ⟩ +≡

```
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;
```

1260.* ⟨ Declare subprocedures for *prefixed_command* 1218* ⟩ +≡

```
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 1261 ⟩;
  ⟨ If this font has already been loaded, set f to the internal font number and goto common_ending 1263* ⟩;
  f ← read_font_info(u, cur_name, cur_area, s);
common_ending: equiv(u) ← f; eqtb[font_id_base + f] ← eqtb[u]; font_id_text(f) ← t;
  end;
```

1263.* 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* 1263* ⟩ ≡

```
  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 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
```

This code is used in section 1260*.

1268.* ⟨ Declare subprocedures for *prefixed_command* 1218* ⟩ +≡
```
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;
```

1278.* ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
```
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;
```

1304.* ⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
```
if ini_version then format_ident ← "␣(INITEX)";
```

1305.* ⟨ Declare action procedures for use by main_control 1046 ⟩ +≡
```
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 1307 ⟩;
⟨ Create the format_ident, open the format file, and inform the user that dumping has begun 1331* ⟩;
⟨ Dump constants for consistency check 1310* ⟩;
⟨ Dump MLTEX-specific data 1403* ⟩;
⟨ Dump the string pool 1312* ⟩;
⟨ Dump the dynamic memory 1314* ⟩;
⟨ Dump the table of equivalents 1316 ⟩;
⟨ Dump the font information 1323* ⟩;
⟨ Dump the hyphenation tables 1327* ⟩;
⟨ Dump a couple more things and the closing check word 1329 ⟩;
⟨ Close the format file 1332 ⟩;
end;
tini
```

1306.* 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 527* ⟩
```
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 1311* ⟩;
⟨ Undump MLTEX-specific data 1404* ⟩;
⟨ Undump the string pool 1313* ⟩;
⟨ Undump the dynamic memory 1315* ⟩;
⟨ Undump the table of equivalents 1317* ⟩;
⟨ Undump the font information 1324* ⟩;
⟨ Undump the hyphenation tables 1328* ⟩;
⟨ Undump a couple more things and the closing check word 1330* ⟩;
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;
```

1308.* 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 }
```

1309.* 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 $a \le x \le b$.

```
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
```

1310.* The next few sections of the program should make it clear how we use the dump/undump macros.

```
⟨ Dump constants for consistency check 1310* ⟩ ≡
  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 mod 4); dump_int(x); dump_things(format_engine[0], x); libc_free(format_engine);
  dump_int(@$);
  ⟨ Dump xord, xchr, and xprn 1389* ⟩;
  dump_int(max_halfword);
  dump_int(hash_high); 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 1305*.

1311.* 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 1311* ⟩ ≡ Init libc_free(font_info); libc_free(str_pool);
  libc_free(str_start); libc_free(yhash); libc_free(zeqtb); libc_free(yzmem); Tinidump_int(x);
  format_debug('format␣magic␣number')(x);
  if x ≠ "57325458 then goto bad_fmt; { not a format file }
  dump_int(x); format_debug('engine␣name␣size')(x);
  if (x < 0) ∨ (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 ≠ @$ then
    begin { check that strings are the same }
    wake_up_terminal; wterm_ln('---!␣', stringcast(name_of_file + 1), '␣doesn''t␣match␣', pool_name);
    goto bad_fmt;
    end;
  ⟨ Undump xord, xchr, and xprn 1390* ⟩;
  undump_int(x);
  if x ≠ 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_int(x); format_debug('mem_bot')(x);
  if x ≠ 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 ≠ eqtb_size then goto bad_fmt;
  undump_int(x);
  if x ≠ hash_prime then goto bad_fmt;
  undump_int(x);
  if x ≠ hyph_prime then goto bad_fmt
```

This code is used in section 1306*.

1312.* **define** *dump_four_ASCII* ≡ $w.b0 \leftarrow qi(so(str\_pool[k]))$; $w.b1 \leftarrow qi(so(str\_pool[k+1]))$; $w.b2 \leftarrow qi(so(str\_pool[k+2]))$; $w.b3 \leftarrow qi(so(str\_pool[k+3]))$; *dump_qqqq*$(w)$

⟨ Dump the string pool 1312* ⟩ ≡
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 1305*.

1313.* **define** *undump_four_ASCII* ≡ *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 1313* ⟩ ≡
undump_size(0)(*sup_pool_size* − *pool_free*)(´string␣pool␣size´)(*pool_ptr*);
if *pool_size* < *pool_ptr* + *pool_free* **then** *pool_size* ← *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* ← *str_ptr* + *strings_free*;
str_start ← *xmalloc_array*(*pool_pointer*, *max_strings*);
undump_checked_things(0, *pool_ptr*, *str_start*[0], *str_ptr* + 1);
str_pool ← *xmalloc_array*(*packed_ASCII_code*, *pool_size*); *undump_things*(*str_pool*[0], *pool_ptr*);
init_str_ptr ← *str_ptr*; *init_pool_ptr* ← *pool_ptr*

This code is used in section 1306*.

1314.* 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 1314* ⟩ ≡
sort_avail; *var_used* ← 0; *dump_int*(*lo_mem_max*); *dump_int*(*rover*); *p* ← *mem_bot*; *q* ← *rover*; *x* ← 0;
repeat *dump_things*(*mem*[*p*], *q* + 2 − *p*); *x* ← *x* + *q* + 2 − *p*; *var_used* ← *var_used* + *q* − *p*;
p ← *q* + *node_size*(*q*); *q* ← *rlink*(*q*);
until *q* = *rover*;
var_used ← *var_used* + *lo_mem_max* − *p*; *dyn_used* ← *mem_end* + 1 − *hi_mem_min*;
dump_things(*mem*[*p*], *lo_mem_max* + 1 − *p*); *x* ← *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 ← *x* + *mem_end* + 1 − *hi_mem_min*; *p* ← *avail*;
while *p* ≠ *null* **do**
begin *decr*(*dyn_used*); *p* ← *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 1305*.

1315.* ⟨ Undump the dynamic memory 1315* ⟩ ≡
```
undump(lo_mem_stat_max + 1000)(hi_mem_stat_min − 1)(lo_mem_max);
undump(lo_mem_stat_max + 1)(lo_mem_max)(rover); p ← mem_bot; q ← rover;
repeat undump_things(mem[p], q + 2 − p); p ← q + node_size(q);
  if (p > lo_mem_max) ∨ ((q ≥ rlink(q)) ∧ (rlink(q) ≠ rover)) then goto bad_fmt;
  q ← 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 ← llink(rover); q ← 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 1306*.

1317.* ⟨ Undump the table of equivalents 1317* ⟩ ≡
```
⟨ Undump regions 1 to 6 of eqtb 1320* ⟩;
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 1322* ⟩
```
This code is used in section 1306*.

1318.* 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* 1318* ⟩ ≡
```
k ← active_base;
repeat j ← 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 ← int_base; goto done1; { j = int_base − 1 }
found1: incr(j); l ← 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 ← j + 1; dump_int(k − l);
until k = int_base
```
This code is used in section 1316.

1319.* ⟨ Dump regions 5 and 6 of *eqtb* 1319* ⟩ ≡

```
repeat j ← k;
  while j < eqtb_size do
    begin if eqtb[j].int = eqtb[j + 1].int then goto found2;
    incr(j);
    end;
  l ← eqtb_size + 1; goto done2; { j = eqtb_size }
found2: incr(j); l ← j;
  while j < eqtb_size do
    begin if eqtb[j].int ≠ eqtb[j + 1].int then goto done2;
    incr(j);
    end;
done2: dump_int(l − k); dump_things(eqtb[k], l − k); k ← 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 1316.

1320.* ⟨ Undump regions 1 to 6 of *eqtb* 1320* ⟩ ≡

```
k ← active_base;
repeat undump_int(x);
  if (x < 1) ∨ (k + x > eqtb_size + 1) then goto bad_fmt;
  undump_things(eqtb[k], x); k ← k + x; undump_int(x);
  if (x < 0) ∨ (k + x > eqtb_size + 1) then goto bad_fmt;
  for j ← k to k + x − 1 do eqtb[j] ← eqtb[k − 1];
  k ← 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 1317*.

1321.* A different scheme is used to compress the hash table, since its lower region is usually sparse. When *text*(*p*) ≠ 0 for *p* ≤ *hash_used*, we output two words, *p* and *hash*[*p*]. The hash table is, of course, densely packed for *p* ≥ *hash_used*, so the remaining entries are output in a block.

⟨ Dump the hash table 1321* ⟩ ≡

```
dump_int(hash_used); cs_count ← frozen_control_sequence − 1 − hash_used + hash_high;
for p ← hash_base to hash_used do
  if text(p) ≠ 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 1316.

1322.* ⟨ Undump the hash table 1322* ⟩ ≡

```
undump(hash_base)(frozen_control_sequence)(hash_used); p ← 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 1317*.

1323.* ⟨ Dump the font information 1323* ⟩ ≡

```
dump_int(fmem_ptr); dump_things(font_info[0], fmem_ptr); dump_int(font_ptr);
⟨ Dump the array info for internal font number k 1325* ⟩;
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 1305*.

1324.* ⟨ Undump the font information 1324* ⟩ ≡

```
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 1326* ⟩;
```

This code is used in section 1306*.

1325*. ⟨ Dump the array info for internal font number k 1325* ⟩ ≡

```
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 1323*.

1326*. This module should now be named 'Undump all the font arrays'.

⟨ Undump the array info for internal font number k 1326* ⟩ ≡

```
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);
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 1324*.

1327* ⟨ Dump the hyphenation tables 1327* ⟩ ≡

```
dump_int(hyph_count);
if hyph_next ≤ hyph_prime then hyph_next ← hyph_size;
dump_int(hyph_next); { minumum 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_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 1305*.

1328* Only "nonempty" parts of *op_start* need to be restored.

```
⟨ Undump the hyphenation tables 1328* ⟩ ≡
  undump_size(0)(hyph_size)('hyph_size')(hyph_count);
  undump_size(hyph_prime)(hyph_size)('hyph_size')(hyph_next); j ← 0;
  for k ← 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 ← j − hyph_next ∗ 65536;
      end
    else hyph_next ← 0;
    if (j ≥ 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 ← 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
      { 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 ← 0 to 255 do trie_used[k] ← min_quarterword;
  tini
  k ← 256;
  while j > 0 do
    begin undump(0)(k − 1)(k); undump(1)(j)(x); init trie_used[k] ← qi(x); tini
    j ← j − x; op_start[k] ← qo(j);
    end;
  init trie_not_ready ← false tini
```

This code is used in section 1306*.

1330* ⟨ Undump a couple more things and the closing check word 1330* ⟩ ≡

```
  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 ≠ 69069) ∨ feof(fmt_file) then goto bad_fmt
```

This code is used in section 1306*.

1331.* ⟨ Create the *format_ident*, open the format file, and inform the user that dumping has begun 1331* ⟩ ≡

```
selector ← new_string; print("␣(format="); print(job_name); print_char("␣"); print_int(year);
print_char("."); print_int(month); print_char("."); print_int(day); print_char(")");
if interaction = batch_mode then selector ← log_only
else selector ← term_and_log;
str_room(1); format_ident ← make_string; pack_job_name(format_extension);
while ¬w_open_out(fmt_file) do prompt_file_name("format␣file␣name", format_extension);
print_nl("Beginning␣to␣dump␣on␣file␣"); slow_print(w_make_name_string(fmt_file)); flush_string;
print_nl(""); slow_print(format_ident)
```

This code is used in section 1305*.

1335.* 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); 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);
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);
```

```
  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);
    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); Tinihistory ← 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 1340* ⟩;
  history ← spotless; { ready to go! }
  main_control; { come to life }
  final_cleanup; { prepare for death }
  close_files_and_terminate;
final_end: do_final_end;
  end { main_body }
  ;
```

1336.* 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.

This program doesn't bother to close the input files that may still be open.

```
⟨ Last-minute procedures 1336* ⟩ ≡
procedure close_files_and_terminate;
  var k: integer;  { all-purpose index }
  begin ⟨ Finish the extensions 1381 ⟩;
  stat if tracing_stats > 0 then ⟨ Output statistics about this job 1337* ⟩; tats
  wake_up_terminal; ⟨ Finish the DVI file 645* ⟩;
  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 1338*, 1339, and 1341*.

This code is used in section 1333.

1337.* 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 1337* ⟩ ≡
  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 1336*.

1338.* We get to the *final_cleanup* routine when `\end` or `\dump` has been scanned and *its_all_over*.

```
⟨ Last-minute procedures 1336* ⟩ +≡
procedure final_cleanup;
  label exit;
  var c: small_number;  { 0 for \end, 1 for \dump }
  begin c ← cur_chr;
  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(")");
    end;
  while cond_ptr ≠ null do
    begin print_nl("("); print_esc("end␣occurred␣"); print("when␣"); print_cmd_chr(if_test, cur_if);
    if if_line ≠ 0 then
      begin print("␣on␣line␣"); print_int(if_line);
      end;
    print("␣was␣incomplete)"); if_line ← if_line_field(cond_ptr); cur_if ← subtype(cond_ptr);
    temp_ptr ← cond_ptr; cond_ptr ← link(cond_ptr); free_node(temp_ptr, if_node_size);
    end;
  if history ≠ spotless then
    if ((history = warning_issued) ∨ (interaction < error_stop_mode)) then
      if selector = term_and_log then
        begin selector ← term_only;
        print_nl("(see␣the␣transcript␣file␣for␣additional␣information)");
        selector ← term_and_log;
        end;
  if c = 1 then
    begin Init for c ← top_mark_code to split_bot_mark_code do
      if cur_mark[c] ≠ null then delete_token_ref(cur_mark[c]);
    if last_glue ≠ 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;
```

1340.* 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 1340* ⟩ ≡
  begin ⟨ Initialize the input routines 331* ⟩;
  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 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;
  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;  { 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);
    font_ptr ← null_font; fmem_ptr ← 7; font_name[null_font] ← "nullfont"; font_area[null_font] ← "";
    hyphen_char[null_font] ← "-"; skew_char[null_font] ← −1; bchar_label[null_font] ← non_address;
    font_bchar[null_font] ← non_char; font_false_bchar[null_font] ← non_char; font_bc[null_font] ← 1;
    font_ec[null_font] ← 0; font_size[null_font] ← 0; font_dsize[null_font] ← 0; char_base[null_font] ← 0;
    width_base[null_font] ← 0; height_base[null_font] ← 0; depth_base[null_font] ← 0;
    italic_base[null_font] ← 0; lig_kern_base[null_font] ← 0; kern_base[null_font] ← 0;
    exten_base[null_font] ← 0; font_glue[null_font] ← null; font_params[null_font] ← 7;
    param_base[null_font] ← −1;
    for font_k ← 0 to 6 do font_info[font_k].sc ← 0;
    end;
  tini
```

```
font_used ← xmalloc_array(boolean, font_max);
for font_k ← font_base to font_max do font_used[font_k] ← false;
⟨ Compute the magic offset 768 ⟩;
⟨ Initialize the print selector based on interaction 75 ⟩;
if (loc < limit) ∧ (cat_code(buffer[loc]) ≠ escape) then start_input;  { \input assumed }
end
```

This code is used in section 1335*.

1341.* 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 1336* ⟩ +≡
  debug procedure debug_help;  { routine to display various things }
  label breakpoint, exit;
  var k, l, m, n: integer;
  begin 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 1342* ⟩
        othercases print("?")
        endcases;
        end;
    end;
exit: end;
  gubed
```

1342:* ⟨ Numbered cases for *debug_help* 1342* ⟩ ≡

```
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 1341*.

1347.* 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 }
⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  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);
```

1351.* ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡

```
⟨ Declare procedures needed in do_extension 1352 ⟩
procedure do_extension;
  var k: integer;  { all-purpose integers }
    p: pointer;  { all-purpose pointers }
  begin case cur_chr of
  open_node: ⟨ Implement \openout 1354 ⟩;
  write_node: ⟨ Implement \write 1355 ⟩;
  close_node: ⟨ Implement \closeout 1356 ⟩;
  special_node: ⟨ Implement \special 1357 ⟩;
  immediate_code: ⟨ Implement \immediate 1378 ⟩;
  set_language_code: ⟨ Implement \setlanguage 1380 ⟩;
  othercases confusion("ext1")
  endcases;
  end;
```

1353.* 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 1352 ⟩ +≡
procedure new_write_whatsit(w : small_number);
  begin new_whatsit(cur_chr, w);
  if w ≠ write_node_size then scan_four_bit_int
  else begin scan_int;
    if cur_val < 0 then cur_val ← 17
    else if (cur_val > 15) ∧ (cur_val ≠ 18) then cur_val ← 16;
    end;
  write_stream(tail) ← cur_val;
  end;
```

1373:* ⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1371 ⟩ +≡

```
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 ⟨ Expand macros in the token list and make link(def_ref) point to the result 1374 ⟩;
  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) ∧ (selector = term_and_log) then selector ← log_only;
      print_nl("");
      end;
  token_show(def_ref); print_ln; flush_list(def_ref);
  if j = 18 then
    begin 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. }
        { 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) ∧ (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;

1376.* The *out_what* procedure takes care of outputting whatsit nodes for *vlist_out* and *hlist_out*.

```
⟨ Declare procedures needed in hlist_out, vlist_out 1371 ⟩ +≡
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 1377* ⟩;
  special_node: special_out(p);
  language_node: do_nothing;
  othercases confusion("ext4")
  endcases;
  end;
```

1377.* 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 1377* ⟩ ≡
  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 a_close(write_file[j]);
      if subtype(p) = close_node then write_open[j] ← false
      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 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 1376*.

1382* 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 }

1383* 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*;

1384* 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. }

1385* 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;
```

1386* Are we printing extra info as we read the format file?

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

1387.* 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[index + 1 − (in_open − level)]);
    print(": ");
    end;
  end;
```

1388.* 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 436 ⟩ +≡
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;
```

1389.* 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 1389* ⟩ ≡
  dump_things(xord[0], 256); dump_things(xchr[0], 256); dump_things(xprn[0], 256);
```

This code is used in section 1310*.

1390.* 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 1390* ⟩ ≡
  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 1311*.

1391* 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 1391* ⟩ ≡
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 1392*.

This code is used in section 47*.

1392* 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 1391* ⟩ +≡
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;
```

1393.* 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*;

1394.* The boolean variable *mltex_enabled_p* is used to enable MLTEX's character substitution. It is initialised 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 }

1395.* ⟨ Set initial values of key variables 21 ⟩ +≡
mltex_enabled_p ← *false*;

1396.* 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 additional functions for MLTEX 1396* ⟩ ≡
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;
```

See also section 1397*.

This code is used in section 563*.

1397:* 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 1396* ⟩ +≡
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;
```

1398:* 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 substituion 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 1398* ⟩ ≡
  begin ⟨ Get substitution information, check it, goto found if all is ok, otherwise goto continue 1400* ⟩;
found: ⟨ Print character substition tracing log 1401* ⟩;
  ⟨ Rebuild character using substitution information 1402* ⟩;
  end
```

This code is used in section 623*.

1399* 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 recursivly.

```
⟨ 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 }
```

1400* Get the character substitution information in *char_sub_code* for the character c. The current code checks that the substition 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 1400* ⟩ ≡
  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 section 1398*.

1401* For *tracinglostchars* > 99 the substitution is shown in the log file.

```
⟨ Print character substition tracing log 1401* ⟩ ≡
  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 section 1398*.

1402* 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 1402* ⟩ ≡
  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 1398*.

1403* Dumping MLTEX-related material. This is just the flag in the format that tells us whether MLTEX is enabled.

```
⟨ Dump MLTEX-specific data 1403* ⟩ ≡
  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 1305*.

1404:* Undump MLTEX-related material, which is just a flag in the format that tells us whether MLTEX is enabled.

```
⟨ Undump MLTEX-specific data 1404* ⟩ ≡
  undump_int(x);  { check magic constant of MLTEX }
  if x ≠ "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 ≠ 0 then goto bad_fmt;
```

This code is used in section 1306*.

1405.* System-dependent changes. This section should be replaced, if necessary, by any special modifications of the program that are necessary to make TEX work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the published program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

1406.* ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡

```
procedure insert_src_special;
  var toklist, p, q: pointer;
  begin if (source_filename_stack[in_open] > 0 ∧ is_new_source(source_filename_stack[in_open], line)) then
    begin toklist ← get_avail; p ← toklist; info(p) ← cs_token_flag + frozen_special; link(p) ← get_avail;
    p ← link(p); info(p) ← left_brace_token + "{";
    q ← str_toks(make_src_special(source_filename_stack[in_open], line)); link(p) ← link(temp_head);
    p ← q; link(p) ← get_avail; p ← link(p); info(p) ← 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 ∧ is_new_source(source_filename_stack[in_open], line)) then
    begin new_whatsit(special_node, write_node_size); write_stream(tail) ← 0; def_ref ← get_avail;
    token_ref_count(def_ref) ← null; q ← str_toks(make_src_special(source_filename_stack[in_open], line));
    link(def_ref) ← link(temp_head); write_tokens(tail) ← def_ref;
    remember_source_info(source_filename_stack[in_open], line);
    end;
  end;
```

1407.* This function used to be in pdftex, but is useful in tex too.

```
function get_nullstr: str_number;
  begin get_nullstr ← "";
  end;
```

1408.* 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, 61, 71, 73, 74, 81, 82, 84, 93, 94, 95, 104, 109, 110, 111, 112, 113, 116, 127, 144, 165, 174, 176, 186, 209, 211, 213, 215, 219, 220, 222, 230, 236, 237, 238, 240, 241, 252, 253, 256, 257, 258, 260, 262, 271, 283, 290, 301, 304, 306, 308, 328, 331, 338, 339, 366, 367, 368, 369, 504, 516, 517, 518, 519, 520, 521, 522, 523, 524, 526, 527, 528, 529, 533, 535, 537, 539, 540, 551, 552, 553, 554, 555, 557, 563, 564, 566, 567, 573, 576, 578, 579, 585, 595, 598, 600, 605, 620, 622, 623, 624, 643, 645, 711, 725, 743, 752, 862, 878, 923, 924, 926, 927, 928, 929, 931, 933, 934, 937, 942, 943, 944, 946, 947, 948, 949, 950, 951, 953, 954, 961, 963, 966, 967, 968, 969, 1037, 1039, 1052, 1094, 1138, 1142, 1170, 1218, 1225, 1226, 1227, 1255, 1260, 1263, 1268, 1278, 1304, 1305, 1306, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1330, 1331, 1335, 1336, 1337, 1338, 1340, 1341, 1342, 1347, 1351, 1353, 1373, 1376, 1377, 1382, 1383, 1384, 1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1405, 1406, 1407, 1408.

⟨ Declare the procedure called *fire_up* 1015 ⟩ Used in section 997.
⟨ Declare the procedure called *get_preamble_token* 785 ⟩ Used in section 777.
⟨ Declare the procedure called *handle_right_brace* 1071 ⟩ Used in section 1033.
⟨ Declare the procedure called *init_span* 790 ⟩ Used in section 789.
⟨ Declare the procedure called *insert_relax* 382 ⟩ Used in section 369*.
⟨ Declare the procedure called *macro_call* 392 ⟩ Used in section 369*.
⟨ Declare the procedure called *print_cmd_chr* 298 ⟩ Used in section 252*.
⟨ Declare the procedure called *print_skip_param* 225 ⟩ Used in section 179.
⟨ Declare the procedure called *restore_trace* 284 ⟩ Used in section 281.
⟨ Declare the procedure called *runaway* 306* ⟩ Used in section 119.
⟨ Declare the procedure called *show_token_list* 292 ⟩ Used in section 119.
⟨ Decry the invalid character and **goto** *restart* 346 ⟩ Used in section 344.
⟨ Delete $c - $ `"0"` tokens and **goto** *continue* 88 ⟩ Used in section 84*.
⟨ Delete the page-insertion nodes 1022 ⟩ Used in section 1017.
⟨ Destroy the t nodes following q, and make r point to the following node 886 ⟩ Used in section 885.
⟨ Determine horizontal glue shrink setting, then **return** or **goto** *common_ending* 667 ⟩ Used in section 660.
⟨ Determine horizontal glue stretch setting, then **return** or **goto** *common_ending* 661 ⟩ Used in section 660.
⟨ Determine the displacement, d, of the left edge of the equation, with respect to the line size z, assuming that $l = \mathit{false}$ 1205 ⟩ Used in section 1202.
⟨ Determine the shrink order 668 ⟩ Used in sections 667, 679, and 799.
⟨ Determine the stretch order 662 ⟩ Used in sections 661, 676, and 799.
⟨ Determine the value of $\mathit{height}(r)$ and the appropriate glue setting; then **return** or **goto** *common_ending* 675 ⟩ Used in section 671.
⟨ Determine the value of $\mathit{width}(r)$ and the appropriate glue setting; then **return** or **goto** *common_ending* 660 ⟩ Used in section 652.
⟨ Determine vertical glue shrink setting, then **return** or **goto** *common_ending* 679 ⟩ Used in section 675.
⟨ Determine vertical glue stretch setting, then **return** or **goto** *common_ending* 676 ⟩ Used in section 675.
⟨ Discard erroneous prefixes and **return** 1215 ⟩ Used in section 1214.
⟨ Discard the prefixes `\long` and `\outer` if they are irrelevant 1216 ⟩ Used in section 1214.
⟨ Dispense with trivial cases of void or bad boxes 981 ⟩ Used in section 980.
⟨ Display adjustment p 197 ⟩ Used in section 183.
⟨ Display box p 184 ⟩ Used in section 183.
⟨ Display choice node p 698 ⟩ Used in section 693.
⟨ Display discretionary p 195 ⟩ Used in section 183.
⟨ Display fraction noad p 700 ⟩ Used in section 693.
⟨ Display glue p 189 ⟩ Used in section 183.
⟨ Display insertion p 188 ⟩ Used in section 183.
⟨ Display kern p 191 ⟩ Used in section 183.
⟨ Display leaders p 190 ⟩ Used in section 189.
⟨ Display ligature p 193 ⟩ Used in section 183.
⟨ Display mark p 196 ⟩ Used in section 183.
⟨ Display math node p 192 ⟩ Used in section 183.
⟨ Display node p 183 ⟩ Used in section 182.
⟨ Display normal noad p 699 ⟩ Used in section 693.
⟨ Display penalty p 194 ⟩ Used in section 183.
⟨ Display rule p 187 ⟩ Used in section 183.
⟨ Display special fields of the unset node p 185 ⟩ Used in section 184.
⟨ Display the current context 312 ⟩ Used in section 311.
⟨ Display the insertion split cost 1014 ⟩ Used in section 1013.
⟨ Display the page break cost 1009 ⟩ Used in section 1008.
⟨ Display the token (m, c) 294 ⟩ Used in section 293.
⟨ Display the value of b 505 ⟩ Used in section 501.

⟨ Display the value of *glue_set*(*p*) 186* ⟩ Used in section 184.
⟨ Display the whatsit node *p* 1359 ⟩ Used in section 183.
⟨ Display token *p*, and **return** if there are problems 293 ⟩ Used in section 292.
⟨ Do first-pass processing based on *type*(*q*); **goto** *done_with_noad* if a noad has been fully processed, **goto** *check_dimensions* if it has been translated into *new_hlist*(*q*), or **goto** *done_with_node* if a node has been fully processed 731 ⟩ Used in section 730.
⟨ Do ligature or kern command, returning to *main_lig_loop* or *main_loop_wrapup* or *main_loop_move* 1043 ⟩ Used in section 1042.
⟨ Do magic computation 320 ⟩ Used in section 292.
⟨ Do some work that has been queued up for `\write` 1377* ⟩ Used in section 1376*.
⟨ Drop current token and complain that it was unmatched 1069 ⟩ Used in section 1067.
⟨ Dump MLTEX-specific data 1403* ⟩ Used in section 1305*.
⟨ Dump a couple more things and the closing check word 1329 ⟩ Used in section 1305*.
⟨ Dump constants for consistency check 1310* ⟩ Used in section 1305*.
⟨ Dump regions 1 to 4 of *eqtb* 1318* ⟩ Used in section 1316.
⟨ Dump regions 5 and 6 of *eqtb* 1319* ⟩ Used in section 1316.
⟨ Dump the array info for internal font number *k* 1325* ⟩ Used in section 1323*.
⟨ Dump the dynamic memory 1314* ⟩ Used in section 1305*.
⟨ Dump the font information 1323* ⟩ Used in section 1305*.
⟨ Dump the hash table 1321* ⟩ Used in section 1316.
⟨ Dump the hyphenation tables 1327* ⟩ Used in section 1305*.
⟨ Dump the string pool 1312* ⟩ Used in section 1305*.
⟨ Dump the table of equivalents 1316 ⟩ Used in section 1305*.
⟨ Dump *xord*, *xchr*, and *xprn* 1389* ⟩ Used in section 1310*.
⟨ Either append the insertion node *p* after node *q*, and remove it from the current page, or delete *node*(*p*) 1025 ⟩ Used in section 1023.
⟨ Either insert the material specified by node *p* into the appropriate box, or hold it for the next page; also delete node *p* from the current page 1023 ⟩ Used in section 1017.
⟨ Either process `\ifcase` or set *b* to the value of a boolean condition 504* ⟩ Used in section 501.
⟨ Empty the last bytes out of *dvi_buf* 602 ⟩ Used in section 645*.
⟨ Ensure that box 255 is empty after output 1031 ⟩ Used in section 1029.
⟨ Ensure that box 255 is empty before output 1018 ⟩ Used in section 1017.
⟨ Ensure that *trie_max* $\geq h + 256$ 957 ⟩ Used in section 956.
⟨ Enter a hyphenation exception 942* ⟩ Used in section 938.
⟨ Enter all of the patterns into a linked trie, until coming to a right brace 964 ⟩ Used in section 963*.
⟨ Enter as many hyphenation exceptions as are listed, until coming to a right brace; then **return** 938 ⟩ Used in section 937*.
⟨ Enter *skip_blanks* state, emit a space 349 ⟩ Used in section 347.
⟨ Error handling procedures 78, 81*, 82*, 93*, 94*, 95* ⟩ Used in section 4*.
⟨ Examine node *p* in the hlist, taking account of its effect on the dimensions of the new box, or moving it to the adjustment list; then advance *p* to the next node 654 ⟩ Used in section 652.
⟨ Examine node *p* in the vlist, taking account of its effect on the dimensions of the new box; then advance *p* to the next node 672 ⟩ Used in section 671.
⟨ Expand a nonmacro 370 ⟩ Used in section 369*.
⟨ Expand macros in the token list and make *link*(*def_ref*) point to the result 1374 ⟩ Used in section 1373*.
⟨ Expand the next part of the input 481 ⟩ Used in section 480.
⟨ Expand the token after the next token 371 ⟩ Used in section 370.
⟨ Explain that too many dead cycles have occurred in a row 1027 ⟩ Used in section 1015.
⟨ Express astonishment that no number was here 449 ⟩ Used in section 447.
⟨ Express consternation over the fact that no alignment is in progress 1131 ⟩ Used in section 1130.
⟨ Express shock at the missing left brace; **goto** *found* 478 ⟩ Used in section 477.
⟨ Feed the macro body and its parameters to the scanner 393 ⟩ Used in section 392.

⟨ Undump MLTEX-specific data 1404* ⟩ Used in section 1306*.
⟨ Undump a couple more things and the closing check word 1330* ⟩ Used in section 1306*.
⟨ Undump constants for consistency check 1311* ⟩ Used in section 1306*.
⟨ Undump regions 1 to 6 of *eqtb* 1320* ⟩ Used in section 1317*.
⟨ Undump the array info for internal font number k 1326* ⟩ Used in section 1324*.
⟨ Undump the dynamic memory 1315* ⟩ Used in section 1306*.
⟨ Undump the font information 1324* ⟩ Used in section 1306*.
⟨ Undump the hash table 1322* ⟩ Used in section 1317*.
⟨ Undump the hyphenation tables 1328* ⟩ Used in section 1306*.
⟨ Undump the string pool 1313* ⟩ Used in section 1306*.
⟨ Undump the table of equivalents 1317* ⟩ Used in section 1306*.
⟨ Undump *xord*, *xchr*, and *xprn* 1390* ⟩ Used in section 1311*.
⟨ Update the active widths, since the first active node has been deleted 864 ⟩ Used in section 863.
⟨ Update the current height and depth measurements with respect to a glue or kern node p 979 ⟩ Used in section 975.
⟨ Update the current page measurements with respect to the glue or kern specified by node p 1007 ⟩ Used in section 1000.
⟨ Update the value of *printed_node* for symbolic displays 861 ⟩ Used in section 832.
⟨ Update the values of *first_mark* and *bot_mark* 1019 ⟩ Used in section 1017.
⟨ Update the values of *last_glue*, *last_penalty*, and *last_kern* 999 ⟩ Used in section 997.
⟨ Update the values of *max_h* and *max_v*; but if the page is too large, **goto** *done* 644 ⟩ Used in section 643*.
⟨ Update width entry for spanned columns 801 ⟩ Used in section 799.
⟨ Use code c to distinguish between generalized fractions 1185 ⟩ Used in section 1184.
⟨ Use node p to update the current height and depth measurements; if this node is not a legal breakpoint, **goto** *not_found* or *update_heights*, otherwise set pi to the associated penalty at the break 976 ⟩ Used in section 975.
⟨ Use size fields to allocate font information 569 ⟩ Used in section 565.
⟨ Wipe out the whatsit node p and **goto** *done* 1361 ⟩ Used in section 202.
⟨ Wrap up the box specified by node r, splitting node p if called for; set $wait \leftarrow true$ if node p holds a remainder after splitting 1024 ⟩ Used in section 1023.