0.* About TEXGPC. TEXGPC is a Unix implementation of Donald E. Knuth's TEX82 in the version 3.1415926 from March 2008. It is based on GNU Pascal. The accompaning `README` file tells you how to build and run TEXGPC. To help you identify the differences of TEX82 and TEXGPC, the numbers of modified modules carry an asterisk. Letters in the left margin indicate the reason for changes. They mean:

E an error in TEX82 fixed
e a small error in TEX82 fixed
F a feature suggested by Knuth added
f a feature I feel worthwhile added
P a Pascal language violation that I managed to avoid
G explain and use a GNU Pascal extension
U adaption to Unix
u usability in a Unix environment enhanced

TEX82's error that is fixed in TEXGPC shows up in a paragraph whose last line contains only glue. TEX82 does not finish that line with `\parfillskip`, TEXGPC does. I sent an error report to Barbara Beeton, and an anonymous bug checker answered 'No Bug'. He or she reasoned with the source file of TEX82—taking the program for its specification!

The small error: TEX82 sometimes saves only the name proper of an input file, TEXGPC saves its name including path and extension, since this is what needs to be passed to the system editor. Note that this error does not show with the first input file opened in a job. Since that is the file you usually want to edit, this error went unnoticed so long.

TEXGPC's uppercase features include: It treats the command line as the first input line; it invokes `vi` if the user types '`e`' during error recovery; and you can interrupt TEXGPC by typing `^C`.

Two lowercase features I consider being bug fixes (1) With input from your terminal, TEX82 clutters it and the log file with spurious empty lines; TEXGPC doesn't—just like TEX78, as I noted in the video "TeX For Beginners", session 3. (2) Since January 1983 TEX82 removes trailing spaces from input lines for compatibility with IBM's 80 column cards; (see error number 114 in `tex82.bug`.) TEXGPC does not whereby addressing an annoyance: When during error recovery you insert a line that ends with a number you want to mark the end of the number with a space. (At least I want to do that.) But TEX82 removes it (and the end-of-line character), and reads further input until it realizes that the sequence of digits ended. To witness, type `i\showbox0\` during error recovery while the input comes from the terminal.

And here is a notable violation of Pascal: TEX82 assumes that the terminal input file is positioned *before* the first character after being opened, whereas TEXGPC assumes that it is positioned *at* the first character. Not realizing that the Pascal-H runtime system works like this, I even mailed an error report to Barbara Beeton just to revoke it two days later much to my embarassment. With 'Pascal' I mean the language as defined in the third edition of *The Pascal User Manual and Report* published 1985.

GNU Pascal extensions are needed to specify a file name at run time and to check the existence of files. Identifiers from GNU Pascal are prefixed with *gpc_* to help distinguish them from Pascal and `WEB` identifiers and to avoid name clashes. Therefore all GNU Pascal identifiers will appear together in the index.

There is only one adaption to Unix needed, namely using '`/`' instead of '`:`' as the file name separator.

The production version of TEXGPC loads the default format file *before* it reads the first input line, in this respect simulating what Knuth calls `TEX`, whereas web2c based implementations load the format file after reading the first input line, thus resembling `VIRTEX`. Consequently, TEXGPC includes the format name in the banner line on the terminal and web2c based implementations don't.

Valid input characters are the 94 visible ASCII characters together with the three control characters horizontal tabulator, form feed, and space.

On exit, TEXGPC passes its 'history' to the operating system. This integer is zero when everything is fine, one when something less serious like an overfull box was detected, two when an error happened like an undefined control sequence, and three when the program aborted because one of its tables overflowed or because it couldn't find an input file while running in batch mode.

TEX82 does arithmetic overflow checking by way of its Pascal-H runtime system. As far as I know, GNU Pascal does no overflow checking. Since I don't feel like adding this to GNU Pascal or to the Pascal source, TEXGPC must do without it.

TEXGPC is somewhat slower than web2c based programs. To compile the device independend file for this document, teTEX 3.0 spent 7.54 s and TEXGPC needed 11.40 s.

Going with Dijkstra, see `http://www.cs.utexas.edu/users/EWD/videos/noorderlicht.mpg`, I don't believe in version numbers, since I don't believe in maintaining software—I consider TEXGPC finished—and it must go without a number. This does not mean that I don't care any more about TEXGPC; comments or questions are quite welcome. In fact, I tried to explain why I changed what and how in order to encourage you to undertake further modifications or bugfixes yourself and I'll be glad to help.

I wish to thank Frank Heckenbach and Emil Jerabek from the GNU Pascal mailing list for clarifying GPC's I/O buffering strategies, and David Kastrup from the `de.comp.text.tex` news group for enlightening articles on some of TEX's more obscure features and for discussing the 'empty last line error', which David does not consider an error at all. And sometimes I agree. It might be better to tell the user that there is an empty last line with an `Underfull \hbox` warning, since those lines are usually not wanted, and if they are, you are free to ignore the warnings.

Juli 2008

Wolfgang Helbig, Programmierer `http://wwwlehre.ba-stuttgart.de/~helbig/`
Waiblingen, Baden-Württemberg `helbig@lehre.ba-stuttgart.de`

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].

define *banner* ≡ `'This␣is␣TeX-GPC'`

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.

P Complying with Pascal, TEXGPC puts the identifiers of the standard text files *input* and *output* in the parameterlist of the program header. One of the WEB macros is named *input* as well. To make TANGLE write INPUT into the Pascal source file instead of the expansion of the macro, you code the name as a concatenation of one letter identifiers since one letter identifiers cannot be macro names and are not indexed by WEAVE. The same applies to *type*.

G To access declarations from GPC's runtime system you need to **import** *gpc*. The '*only*' feature avoids name clashes.

```
define term_in ≡ i@&n@&p@&u@&t
define term_out ≡ output
define mtype ≡ t@&y@&p@&e
format import ≡ label
format only ≡ then
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 (term_in, term_out);
  import gpc only (gpc_execute, gpc_install_signal_handler, gpc_t_signal_handler, gpc_sig_int);
  label ⟨ Labels in the outer block 6 ⟩
  const ⟨ Constants in the outer block 11* ⟩
  mtype ⟨ Types in the outer block 18 ⟩
  var ⟨ Global variables 13 ⟩
  procedure set_interrupt(signal : gpc_integer); forward;
          { initialize installs set_interrupt as a Unix signal handler }
  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 ⟩
```

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* ≡ { change this to '*debug* ≡ `@{`' when not debugging }
define *gubed* ≡ { change this to '*gubed* ≡ `@}`' when not debugging }
format *debug* ≡ *begin*
format *gubed* ≡ *end*

define *stat* ≡ { change this to '*stat* ≡ `@{`' to turn off statistics }
define *tats* ≡ { change this to '*tats* ≡ `@}`' to turn off statistics }
format *stat* ≡ *begin*
format *tats* ≡ *end*

G **9*** If the first character of a Pascal comment is a dollar sign, GNU Pascal treats the comment as a list of "compiler directives" that will affect the translation of this program into machine language. The directive shown below specifies that an I/O error will not terminate the program. That way TEX may search different directories for an input file and the user gets a chance to correct the spelling of a file name without stopping the job.

GNU Pascal does no arithmetic overflow checking but TEX82 does. Here TEXGPC does not comply with TEX82.

⟨ Compiler directives 9* ⟩ ≡
`@{@&$I-@}` { no I/O checking }
This code is used in section 4*.

10* This TEX implementation conforms to the rules of the *Pascal User Manual* published by Jensen and Wirth in 1975, except where system-dependent code is necessary to make a useful system program, and except in another respect where such conformity would unnecessarily obscure the meaning and clutter up the code: We assume that **case** statements may include a default case that applies if no matching label is found. Thus, we shall use constructions like

case x **of**
1: ⟨ code for $x = 1$ ⟩;
3: ⟨ code for $x = 3$ ⟩;
othercases ⟨ code for $x \neq 1$ and $x \neq 3$ ⟩
endcases

since most Pascal compilers have plugged this hole in the language by incorporating some sort of default mechanism. For example, the Pascal-H compiler allows '*others*:' as a default label, and other Pascals allow syntaxes like '**else**' or '**otherwise**' or '*otherwise*:', etc. The definitions of **othercases** and **endcases** should be changed to agree with local conventions. Note that no semicolon appears before **endcases** in this program, so the definition of **endcases** should include a semicolon if the compiler wants one. (Of course, if no default mechanism is available, the **case** statements of TEX will have to be laboriously extended by listing all remaining cases. People who are stuck with such Pascals have, in fact, done this, successfully but not happily!)

G **define** *othercases* ≡ **else** { default for cases not listed explicitly }
define *endcases* ≡ **end** { follows the default case in an extended **case** statement }
format *othercases* ≡ *else*
format *endcases* ≡ *end*

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.

⟨ Constants in the outer block 11* ⟩ ≡
mem_max = 40000; { increased for mfbook 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* }
mem_min = 0; { 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* }
buf_size = 500; { 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* }
error_line = 72; { width of context lines on terminal error messages }
half_error_line = 42; { width of first lines of contexts in terminal error messages; should be between 30 and *error_line* − 15 }
max_print_line = 79; { width of longest text lines output; should be at least 60 }
stack_size = 200; { maximum number of simultaneous input sources }
max_in_open = 6;
{ maximum number of input files and error insertions that can be going on simultaneously }
font_max = 75; { maximum internal font number; must not exceed *max_quarterword* and must be at most *font_base* + 256 }
font_mem_size = 30000; { increased for ibycus4, number of words of *font_info* for all fonts }
param_size = 60; { maximum number of simultaneous macro parameters }
nest_size = 40; { maximum number of semantic levels simultaneously active }
max_strings = 3000; { maximum number of strings; must not exceed *max_halfword* }
string_vacancies = 8000; { 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 = 32000; { 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 }
save_size = 600; { space for saving values outside of current group; must be at most *max_halfword* }
trie_size = 8000; { space for hyphenation patterns; should be larger for `INITEX` than it is in production versions of TEX }
trie_op_size = 500; { space for "opcodes" in the hyphenation patterns }
dvi_buf_size = 800; { size of the output buffer; must be a multiple of 8 }
file_name_size = 40; { file names shouldn't be longer than this }
pool_name = `'TeXformats/tex.pool␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'`;
U { string of length *file_name_size*; tells where the string pool appears }

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 *mem_bot* = 0
{ smallest index in the *mem* array dumped by `INITEX`; must not be less than *mem_min* }
define *mem_top* ≡ 40000 { largest index in the *mem* array dumped by `INITEX`; must be substantially larger than *mem_bot* and not greater than *mem_max* }
define *font_base* = 0 { smallest internal font number; must not be less than *min_quarterword* }
define *hash_size* = 2100 { maximum number of control sequences; it should be at most about (*mem_max* − *mem_min*)/10 }
define *hash_prime* = 1777 { a prime number equal to about 85% of *hash_size* }
define *hyph_size* = 307 { another prime; the number of `\hyphenation` exceptions }

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 $'\sqcup'$ on the right of these assignment statements to $chr(i)$.

```
⟨ Set initial values of key variables 21 ⟩ +≡
  for i ← 0 to '37 do xchr[i] ← '␣';
u xchr['11] ← chr('11); { accept horizontal tab }
  xchr['14] ← chr('14); { accept form feed }
  for i ← '177 to '377 do xchr[i] ← '␣';
```

25.* Input and output. The bane of portability is the fact that different operating systems treat input and output quite differently, perhaps because computer scientists have not given sufficient attention to this problem. People have felt somehow that input and output are not part of "real" programming. Well, it is true that some kinds of programming are more fun than others. With existing input/output conventions being so diverse and so messy, the only sources of joy in such parts of the code are the rare occasions when one can find a way to make the program a little less bad than it might have been. We have two choices, either to attack I/O now and get it over with, or to postpone I/O until near the end. Neither prospect is very attractive, so let's get it over with.

The basic operations we need to do are (1) inputting and outputting of text, to or from a file or the user's terminal; (2) inputting and outputting of eight-bit bytes, to or from a file; (3) instructing the operating system to initiate ("open") or to terminate ("close") input or output from a specified file; (4) testing whether the end of an input file has been reached.

TEX needs to deal with two kinds of files. We shall use the term *alpha_file* for a file that contains textual data, and the term *byte_file* for a file that contains eight-bit binary information. These two types turn out to be the same on many computers, but sometimes there is a significant distinction, so we shall be careful to distinguish between them. Standard protocols for transferring such files from computer to computer, via high-speed networks, are now becoming available to more and more communities of users.

The program actually makes use also of a third kind of file, called a *word_file*, when dumping and reloading base information for its own initialization. We shall define a word file later; but it will be possible for us to specify simple operations on word files before they are defined.

G Violating Pascal, GNU Pascal wants you to declare the *subrange* type as **packed** . It seems to ignore **packed** with file types. GNU Pascal allows buffered output on *untyped* files only and we want to take advantage of buffering when writing the DVI file. Again, I stumble over a name clash: The Pascal predeclared type for text files and a `WEB` macro both are called *text*.

```
⟨ Types in the outer block 18 ⟩ +≡
G   eight_bits = packed 0 .. 255;   { unsigned one-byte quantity }
P   alpha_file = t@&e@&x@&t;   { Pascal requires text }
G   untyped_file = file ;   { for the DVI file }
    byte_file = packed file of eight_bits;   { files that contain binary data }
```

G 27*. GNU Pascal extends the rules of Pascal in a very convenient way. To open file f, you write

reset$(f, name)$	for input;
rewrite$(f, name)$	for output.

The '*name*' parameter holds the name of the external file that is being opened for input or output.

For the DVI file, we need to open an 'untyped' file. Here *rewrite* takes a third parameter, which is always set to one.

The GNU Pascal function *gpc_trim* removes trailing spaces. GNU Pascal's *gpc_io_result* function returns an error number. This number is set to a nonzero value if an I/O operation failed. A successfull I/O operation does not set the error number to zero, but *gpc_io_result* resets the error number on each call. Thus, TEX invokes *gpc_io_result* before opening a file to clear the error number and afterwards to check it.

```
define gpc_trim ≡ t@&r@&i@&m
define gpc_io_result ≡ i@&o@&r@&e@&s@&u@&l@&t
define reset_OK(#) ≡ gpc_io_result = 0
define rewrite_OK(#) ≡ gpc_io_result = 0
define clear_io_result ≡ if gpc_io_result = 0 then do_nothing
function a_open_in(var f : alpha_file): boolean; { open a text file for input }
  begin clear_io_result; reset(f, gpc_trim(name_of_file)); a_open_in ← reset_OK(f);
  end;
function a_open_out(var f : alpha_file): boolean; { open a text file for output }
  begin clear_io_result; rewrite(f, gpc_trim(name_of_file)); a_open_out ← rewrite_OK(f);
  end;
function b_open_in(var f : byte_file): boolean; { open a binary file for input }
  begin clear_io_result; reset(f, gpc_trim(name_of_file)); b_open_in ← reset_OK(f);
  end;
function u_open_out(var f : untyped_file): boolean; { open a binary file for buffered output }
  begin clear_io_result; rewrite(f, gpc_trim(name_of_file), 1); u_open_out ← rewrite_OK(f);
  end;
function w_open_in(var f : word_file): boolean; { open a word file for input }
  begin clear_io_result; reset(f, gpc_trim(name_of_file)); w_open_in ← reset_OK(f);
  end;
function w_open_out(var f : word_file): boolean; { open a word file for output }
  begin clear_io_result; rewrite(f, gpc_trim(name_of_file)); w_open_out ← rewrite_OK(f);
  end;
```

G **28.*** Files are closed by the GNU Pascal routine '*close*(*f*)', which should be used when all input or output with respect to *f* has been completed. This makes *f* available to be opened again, if desired; and if *f* was used for output, the *close* operation makes the corresponding external file appear on the user's area, ready to be read.

These procedures should not generate error messages if a file is being closed before it has been successfully opened.

```
procedure a_close(var f : alpha_file);  { close a text file }
  begin close(f);
  end;
procedure b_close(var f : byte_file);  { close a binary file }
  begin close(f);
  end;
procedure w_close(var f : word_file);  { close a word file }
  begin close(f);
  end;
```

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).

P TEXGPC's *input_ln* ignores *bypass_eoln*. Instead it assumes the precondition that *f*↑ holds the first character of the line which is established by Pascal's *reset* and maintained by *input_ln* in that it bypasses the end-of-line marker before returning. Pascal-H does not establish the condition if opening a terminal file, and you need to call *get*(*f*) before you access *f*↑. TEX82 controlls this with the *bypass_eoln* parameter: It is set to *true* to read the first line from the terminal and set to *false* to read the first line from a disk file. In the first case, *input_ln* does not bypass an end-of-line marker but advances the file pointer to the first character. GNU Pascal employs 'lazy input' as suggested by the Pascal User Manual: The program does not wait for input on behalf of a *reset*(*f*) or *get*(*f*) but delays reading until it uses *f*↑. Unlike TEX82 TEXGPC
f leaves trailing spaces in the input line.

G Frank Heckenbach pointed out that GNU Pascal employes buffered I/O on input files—no need to avoid high system overhead here.

```
function input_ln(var f : alpha_file; bypass_eoln : boolean): boolean;
        { inputs the next line or returns false }
  begin   { ignore bypass_eoln. Assuming f being positioned at the first character }
  last ← first;   { cf. Matthew 19:30 }
  if eof(f) then input_ln ← false
  else begin while ¬eoln(f) do
      begin if last ≥ max_buf_stack then
        begin max_buf_stack ← last + 1;
        if max_buf_stack = buf_size then
          begin read_ln(f);   { complete the current line }
          ⟨ Report overflow of the input buffer, and abort 35 ⟩;
          end;
        end;
      buffer[last] ← xord[f↑]; get(f); incr(last);
      end;
    get(f);   { Advance f to the first character of the next line }
    input_ln ← true;
    end;
  end;
```

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
P considered an output file the file variable is *term_out*. Pascal's standard text files are declared implicitly.

P **33.*** The Pascal-H implementation opened *term_in* with the file positioned 'before' the first character, violating Pascal. Pascal's standard text files are opened implicitly with *term_in* positioned at the first character.

define *t_open_in* ≡ *do_nothing* { open the terminal for text input }
define *t_open_out* ≡ *do_nothing* { open the terminal 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.

G Nothing needs to be done to update the terminal, since GNU Pascal does not employ buffered output on typed files. I do not know how to clear the type ahead buffer, so TEXGPC does nothing here. Unix holds terminal output, when it receives `^S` and continues writing to the terminal, when it recieves `^Q`. These 'flow control' characters only work when sent from the terminal but not when sent to the terminal. Here I give up, since I don't know how to restart the output from the 'wrong' side so TEXGPC does nothing.

define *update_terminal* ≡ *do_nothing* { empty the terminal output buffer }
define *clear_terminal* ≡ *do_nothing* { clear the terminal input buffer }
define *wake_up_terminal* ≡ *do_nothing* { cancel the user's cancellation of output }

36.* Different systems have different ways to get started. But regardless of what conventions are adopted, the routine that initializes the terminal should satisfy the following specifications:

1) It should open file *term_in* for input from the terminal. (The file *term_out* will already be open for output to the terminal.)
2) If the user has given a command line, this line should be considered the first line of terminal input. Otherwise the user should be prompted with '`**`', and the first line of input should be whatever is typed in response.
3) The first line of input, which might or might not be a command line, should appear in locations *first* to *last* − 1 of the *buffer* array.
4) The global variable *loc* should be set so that the character to be read next by TEX is in *buffer*[*loc*]. This character should not be blank, and we should have *loc* < *last*.

(It may be necessary to prompt the user several times before a non-blank line comes in. The prompt is '`**`' instead of the later '`*`' because the meaning is slightly different: '`\input`' need not be typed immediately after '`**`'.)

F This procedure puts the command line arguments separated by spaces into *buffer*. Like *input_ln* it updates *last* so that *buffer*[*first* .. *last*) will contain the command line.

G GNU Pascal's function *gpc_param_count* gives the number of command line arguments. The function *gpc_param_str*(*n*) returns the n-th argument for 1 ≤ *n* ≤ *gpc_param_count* in a *gpc_string*, whose length is returned by the function *gpc_length*. Again we need to resolve a naming conflict here with a `WEB` macro. A *gpc_string* is like a **packed array** [1 .. *gpc_length*] **of** *char* with varying length.

```
define loc ≡ cur_input.loc_field  { location of first unread character in buffer }
define gpc_string ≡ s@&t@&r@&i@&n@&g  { a string with varying length }
define gpc_length ≡ l@&e@&n@&g@&t@&h
define gpc_param_count ≡ p@&a@&r@&a@&m@&c@&o@&u@&n@&t
define gpc_param_str ≡ p@&a@&r@&a@&m@&s@&t@&r
        { GPC function returning the length of a gpc_string }
procedure input_command_ln;  { get the command line in buffer }
  var argc: integer;  { argument counter }
    arg: gpc_string;  { argument }
    cc: integer;  { character counter in argument }
  begin last ← first; argc ← 1;
  while argc ≤ gpc_param_count do
    begin cc ← 1; arg ← gpc_param_str(argc); incr(argc);
    while cc ≤ gpc_length(arg) do
      begin if last + 1 ≥ buf_size then ⟨ Report overflow of the input buffer, and abort 35 ⟩;
      buffer[last] ← xord[arg[cc]]; incr(last); incr(cc);
      end;
    if (argc ≤ gpc_param_count) then
      begin buffer[last] ← "␣"; incr(last);  { insert a space between arguments }
      end;
    end;
  end;
```

F 37.* The following program treats a nonempty command line as the first line.

```
function init_terminal: boolean;  { gets the terminal input started }
  label exit;
  begin t_open_in; input_command_ln;
  while first = last do
    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;
    if first = last then write_ln(term_out, 'Please type the name of your input file.');
    end;
  loc ← first;  { trim leading spaces }
  while buffer[loc] = " " do incr(loc);
  init_terminal ← true;
exit: end;
```

79:* Individual lines of help are recorded in the array *help_line*, which contains entries in positions 0 .. (*help_ptr* − 1). They should be printed in reverse order, i.e., with *help_line*[0] appearing last.

```
define hlp1(#) ≡ help_line[0] ← #; end
define hlp2(#) ≡ help_line[1] ← #; hlp1
define hlp3(#) ≡ help_line[2] ← #; hlp2
define hlp4(#) ≡ help_line[3] ← #; hlp3
define hlp5(#) ≡ help_line[4] ← #; hlp4
define hlp6(#) ≡ help_line[5] ← #; hlp5
define help0 ≡ help_ptr ← 0  { sometimes there might be no help }
define help1 ≡ begin help_ptr ← 1; hlp1  { use this with one help line }
define help2 ≡ begin help_ptr ← 2; hlp2  { use this with two help lines }
define help3 ≡ begin help_ptr ← 3; hlp3  { use this with three help lines }
define help4 ≡ begin help_ptr ← 4; hlp4  { use this with four help lines }
define help5 ≡ begin help_ptr ← 5; hlp5  { use this with five help lines }
define help6 ≡ begin help_ptr ← 6; hlp6  { use this with six help lines }
⟨ Global variables 13 ⟩ +≡
help_line: array [0 .. 5] of str_number;  { helps for the next error }
help_ptr: 0 .. 6;  { the number of help lines present }
use_err_help: boolean;  { should the err_help list be shown? }
F edit_line: integer;  { line number to be passed to the system editor }
edit_file_name: str_number;  { file name to be passed to the system editor }
```

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

```
help_ptr ← 0; use_err_help ← false; edit_line ← 0; edit_file_name ← 0;
F     { initialize system edit arguments }
```

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. But such an extension requires some system wizardry, so the present implementation simply types out the name of the file that should be edited and the relevant line number.

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

```
⟨ 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
F     begin  { save values to be passed to the system editor }
      edit_file_name ← input_stack[base_ptr].name_field; edit_line ← line; interaction ← scroll_mode;
      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.

96:* Users occasionally want to interrupt TEX while it's running. If the Pascal runtime system allows this, one can implement a routine that sets the global variable *interrupt* to some nonzero value when such an interrupt is signalled. Otherwise there is probably at least a way to make *interrupt* nonzero using the Pascal debugger.

G GNU Pascal reserves the identifier *interrupt*, which seems a bug. `WEB` provides an simple workaround.

```
define interrupt ≡ buginterrupt
format interrupt ≡ true
define check_interrupt ≡
        begin if interrupt ≠ 0 then pause_for_instructions;
        end
```

⟨ Global variables 13 ⟩ +≡
```
interrupt: integer;  { should TEX pause for instructions? }
OK_to_interrupt: boolean;  { should interrupts be observed? }
```

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. In GNU Pascal a *short_real* has the desired size. 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 ⟩ +≡

```
G   glue_ratio = short_real;   { one-word representation of a glue expansion factor in GNU Pascal }
```

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

```
define sc ≡ int  { scaled data is equivalent to integer }
⟨ Types in the outer block 18 ⟩ +≡
G   quarterword = packed min_quarterword .. max_quarterword;  { 1/4 of a word }
G   halfword = packed min_halfword .. max_halfword;  { 1/2 of a word }
    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 }
    two_halves = packed record rh: halfword;
      case two_choices of
      1: (lh : halfword);
      2: (b0 : quarterword; b1 : quarterword);
      end;
    four_quarters = packed record b0: quarterword;
      b1: quarterword;
      b2: quarterword;
      b3: quarterword;
      end;
    memory_word = record
      case four_choices of
      1: (int : integer);
      2: (gr : glue_ratio);
      3: (hh : two_halves);
      4: (qqqq : four_quarters);
      end;
    word_file = file of memory_word;
```

241.* The following procedure, which is called just before TEX initializes its input and output, establishes the initial values of the date and time. Since standard Pascal cannot provide such information, something special is needed. The program here simply specifies July 4, 1776, at noon; but users probably want a better approximation to the truth.

G GNU Pascal provides the *get_time_stamp* function, which stores the system time in its argument. Since *day, month,* and *year* are `WEB` macros I need to resolve the naming conflict.

```
procedure fix_date_and_time;
  var t: time_stamp;
  begin get_time_stamp(t); time ← t.minute + t.hour * 60; { minutes since midnight }
  day ← t.d@&a@&y; month ← t.m@&o@&n@&t@&h; year ← t.y@&e@&a@&r; { Anno Domini }
  end;
```

360.* All of the easy branches of *get_next* have now been taken care of. There is one more branch.

```
define end_line_char_inactive ≡ (end_line_char < 0) ∨ (end_line_char > 255)
⟨ Move to next line of file, or goto restart if there is no next line, or return if a \read line has
      finished 360* ⟩ ≡
  if name > 17 then ⟨ Read next line of file into buffer, or goto restart if the file has ended 362 ⟩
  else begin if ¬terminal_input then { \read line has ended }
      begin cur_cmd ← 0; cur_chr ← 0; return;
      end;
    if input_ptr > 0 then { text was inserted during error recovery }
      begin end_file_reading; goto restart; { resume previous level }
      end;
    if selector < log_only then open_log_file;
    if interaction > nonstop_mode then
      begin if end_line_char_inactive then incr(limit);
      if limit = start then { previous line was empty }
        print_nl("(Please␣type␣a␣command␣or␣say␣`\end´)");
f     print_nl(""); first ← start; { avoid empty lines on terminal and log file }
      prompt_input("*"); { input on-line into buffer }
      limit ← last;
      if end_line_char_inactive then decr(limit)
      else buffer[limit] ← end_line_char;
      first ← limit + 1; loc ← start;
      end
    else fatal_error("***␣(job␣aborted,␣no␣legal␣\end␣found)");
          { nonstop mode, which is intended for overnight batch processing, never waits for on-line input }
    end
```

This code is used in section 343.

U **514.*** Input files that can't be found in the working directory may appear in a directory called *TEX_area*. Font metric files whose directory path are not given explicitly are assumed to appear in the working directory
f or in a standard system directory called *TEX_font_area*. In this implementation the system directories are sub directories of the working directory.

```
define TEX_area ≡ "TeXinputs/"  { i.e., a subdirectory of the working directory }
define TEX_font_area ≡ "TeXfonts/"  { dito }
```

516.* 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 = "␣" then more_name ← false
  else begin str_room(1); append_char(c);  { contribute c to the current string }
U   if c = "/" then  { use "/" as a file name separator }
      begin area_delimiter ← cur_length; ext_delimiter ← 0;
      end
    else if (c = ".") ∧ (ext_delimiter = 0) then ext_delimiter ← cur_length;
    more_name ← true;
    end;
  end;
```

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

```
U   TEX_format_default ← ´TeXformats/plain.fmt´;  { "/" is the Unix file name separator }
```

532.* 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 ensure_dvi_open ≡
        if output_file_name = 0 then
          begin if job_name = 0 then open_log_file;
          pack_job_name(".dvi");
          while ¬u_open_out(dvi_file) do prompt_file_name("file␣name␣for␣output", ".dvi");
          output_file_name ← make_name_string;
          end
```

⟨ Global variables 13 ⟩ +≡

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

e Keep the complete file name since it might be needed to be passed to the system editor. (TEX82 strips off area and extension to conserve string pool space.)

```
procedure start_input; { TEX will \input something }
  label done;
  begin scan_file_name; { set cur_name to desired file name }
  if cur_ext = "" then cur_ext ← ".tex";
  pack_cur_name;
  loop begin begin_file_reading; { set up cur_file and new level of input }
    if a_open_in(cur_file) then goto done;
    if cur_area = "" then
      begin pack_file_name(cur_name, TEX_area, cur_ext);
      if a_open_in(cur_file) then goto done;
      end;
    end_file_reading; { remove the level that didn't work }
    prompt_file_name("input␣file␣name", ".tex");
    end;
done: name ← a_make_name_string(cur_file);
  if job_name = 0 then
    begin 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(name) > 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(name); update_terminal; state ← new_line;
  ⟨ Read the first line of the new file 538 ⟩;
  end;
```

597:* 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.

G In fact, buffering dramatically cuts down system overhead. To compile this document, a program without buffering spent 48.45 s in the kernel but with buffering only 0.57 s. The total times were 64.45 s vs. 11.40 s.

GNU Pascal's *block_write* procedure takes a untyped **file** , an **array** and the number of blocks to be written. When opening the file, we specified the block length to be one, so the number of blocks equals the number of bytes. The **array** here is given as a 'slice', another extension of GNU Pascal. It should be clear, what *buffer[a..b]* means. This simple change was suggested by Emil Jerabek.

```
procedure write_dvi(a, b : dvi_index);
  var k: dvi_index;
G begin block_write(dvi_file, dvi_buf[a .. b], b − a + 1);  { buffered output with GNU Pascal }
  end;
```

642:* 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 642* ⟩ ≡
  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 643 ⟩;
    dvi_out(post_post); dvi_four(last_bop); dvi_out(id_byte);
    k ← 4 + ((dvi_buf_size − dvi_ptr) mod 4);  { the number of 223's }
    while k > 0 do
      begin dvi_out(223); decr(k);
      end;
    ⟨ Empty the last bytes out of dvi_buf 599 ⟩;
    print_nl("Output␣written␣on␣"); slow_print(output_file_name); print("␣("); print_int(total_pages);
    print("␣page");
    if total_pages ≠ 1 then print_char("s");
    print(",␣"); print_int(dvi_offset + dvi_ptr); print("␣bytes)."); close(dvi_file);
    end
```

This code is used in section 1333*.

816:* The first task is to move the list from *head* to *temp_head* and go into the enclosing semantic level. We also append the `\parfillskip` glue to the end of the paragraph, removing a space (or other glue node) if it was there, since spaces usually precede blank lines and instances of '`$$`'. The *par_fill_skip* is preceded by an infinite penalty, so it will never be considered as a potential breakpoint.

This code assumes that a *glue_node* and a *penalty_node* occupy the same number of *mem* words.

```
⟨ Get ready to start line breaking 816* ⟩ ≡
  link(temp_head) ← link(head);
  if is_char_node(tail) then tail_append(new_penalty(inf_penalty))
  else if type(tail) ≠ glue_node then tail_append(new_penalty(inf_penalty))
    else begin type(tail) ← penalty_node; delete_glue_ref(glue_ptr(tail)); flush_node_list(leader_ptr(tail));
      penalty(tail) ← inf_penalty;
      end;
E  non_prunable_p ← tail; { save a pointer to the penalty before \parfillskip }
  link(tail) ← new_param_glue(par_fill_skip_code); init_cur_lang ← prev_graf mod ´200000;
  init_l_hyf ← prev_graf div ´20000000; init_r_hyf ← (prev_graf div ´200000) mod ´100; pop_nest;
```

See also sections 827, 834, and 848.

This code is used in section 815.

862* Breaking paragraphs into lines, continued. So far we have gotten a little way into the *line_break* routine, having covered its important *try_break* subroutine. Now let's consider the rest of the process.

The main loop of *line_break* traverses the given hlist, starting at *link*(*temp_head*), and calls *try_break* at each legal breakpoint. A variable called *auto_breaking* is set to true except within math formulas, since glue nodes are not legal breakpoints when they appear in formulas.

The current node of interest in the hlist is pointed to by *cur_p*. Another variable, *prev_p*, is usually one step behind *cur_p*, but the real meaning of *prev_p* is this: If *type*(*cur_p*) = *glue_node* then *cur_p* is a legal breakpoint if and only if *auto_breaking* is true and *prev_p* does not point to a glue node, penalty node, explicit kern node, or math node.

The following declarations provide for a few other local variables that are used in special calculations.

⟨ Local variables for line breaking 862* ⟩ ≡
auto_breaking: *boolean*; { is node *cur_p* outside a formula? }
E *non_prunable_p*: *pointer*; { pointer to the infinite penalty node at the end of the paragraph }
prev_p: *pointer*; { helps to determine when glue nodes are breakpoints }
q, *r*, *s*, *prev_s*: *pointer*; { miscellaneous nodes of temporary interest }
f: *internal_font_number*; { used when calculating character widths }

See also section 893.

This code is used in section 815.

876* Once the best sequence of breakpoints has been found (hurray), we call on the procedure *post_line_break* to finish the remainder of the work. (By introducing this subprocedure, we are able to keep *line_break* from getting extremely long.)

⟨ Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and append them to the current vertical list 876* ⟩ ≡
E *post_line_break*(*final_widow_penalty*, *non_prunable_p*) { pass *non_prunable_p* }

This code is used in section 815.

877* The total number of lines that will be set by *post_line_break* is *best_line* − *prev_graf* − 1. The last breakpoint is specified by *break_node*(*best_bet*), and this passive node points to the other breakpoints via the *prev_break* links. The finishing-up phase starts by linking the relevant passive nodes in forward order, changing *prev_break* to *next_break*. (The *next_break* fields actually reside in the same memory space as the *prev_break* fields did, but we give them a new name because of their new significance.) Then the lines are justified, one by one.

```
define next_break ≡ prev_break  { new name for prev_break after links are reversed }
⟨ Declare subprocedures for line_break 826 ⟩ +≡
procedure post_line_break(final_widow_penalty : integer; non_prunable_p : pointer);
E       { add another parameter needed when pruning a line }
  label done, done1;
  var q, r, s: pointer;  { temporary registers for list manipulation }
    disc_break: boolean;  { was the current break at a discretionary node? }
    post_disc_break: boolean;  { and did it have a nonempty post-break part? }
    cur_width: scaled;  { width of line number cur_line }
    cur_indent: scaled;  { left margin of line number cur_line }
    t: quarterword;  { used for replacement counts in discretionary nodes }
    pen: integer;  { use when calculating penalties between lines }
    cur_line: halfword;  { the current line number being justified }
  begin ⟨ Reverse the links of the relevant passive nodes, setting cur_p to the first breakpoint 878 ⟩;
  cur_line ← prev_graf + 1;
  repeat ⟨ Justify the line ending at breakpoint cur_p, and append it to the current vertical list, together
      with associated penalties and other insertions 880 ⟩;
    incr(cur_line); cur_p ← next_break(cur_p);
    if cur_p ≠ null then
      if ¬post_disc_break then ⟨ Prune unwanted nodes at the beginning of the next line 879* ⟩;
  until cur_p = null;
  if (cur_line ≠ best_line) ∨ (link(temp_head) ≠ null) then confusion("line␣breaking");
  prev_graf ← best_line − 1;
  end;
```

879*. Glue and penalty and kern and math nodes are deleted at the beginning of a line, except in the anomalous case that the node to be deleted is actually one of the chosen breakpoints. Otherwise the pruning done here is designed to match the lookahead computation in *try_break*, where the *break_width* values are computed for non-discretionary breakpoints.

E The pointer *non_prunable_p* references the infinite penalty node preceding the `\parfillskip` node at the end of the paragraph. These nodes must not be pruned. TEX82 deletes them whenever the last line of a paragraph containes only glue and thus violates the specification in The TEXbook.

```
⟨ Prune unwanted nodes at the beginning of the next line 879* ⟩ ≡
  begin r ← temp_head;
  loop begin q ← link(r);
    if q = cur_break(cur_p) then goto done1;  { cur_break(cur_p) is the next breakpoint }
        { now q cannot be null }
    if is_char_node(q) then goto done1;
    if non_discardable(q) then goto done1;
E   if q = non_prunable_p then goto done1;  { keep \parfillskip }
    if type(q) = kern_node then
      if subtype(q) ≠ explicit then goto done1;
    r ← q;  { now type(q) = glue_node, kern_node, math_node or penalty_node }
    end;
done1: if r ≠ temp_head then
    begin link(r) ← null; flush_node_list(link(temp_head)); link(temp_head) ← q;
    end;
  end
```

This code is used in section 877*.

1332* Now this is really it: TEX starts and ends here. The function *gpc_execute* will start the system editor (vi) and *gpc_halt* passes the *history* as an exit code to the system.

```
define gpc_halt ≡ h@&a@&l@&t
begin { start_here }
history ← fatal_error_stop; { in case we quit during initialization }
t_open_out; { open the terminal for output }
if ready_already = 314159 then goto start_of_TEX;
⟨ Check the "constant" values for consistency 14 ⟩
if bad > 0 then
  begin wterm_ln('Ouch---my␣internal␣constants␣have␣been␣clobbered!', '---case␣', bad : 1);
  goto final_end;
  end;
initialize; { set global variables to their starting values }
init if ¬get_strings_started then goto final_end;
init_prim; { call primitive for each primitive }
init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr; fix_date_and_time;
tini
ready_already ← 314159;
start_of_TEX: ⟨ Preload the default format file 1380* ⟩;
⟨ Initialize the output routines 55 ⟩;
⟨ Get the first line of input and prepare to start 1337 ⟩;
history ← spotless; { ready to go! }
main_control; { come to life }
final_cleanup; { prepare for death }
end_of_TEX: close_files_and_terminate;
F final_end: if edit_file_name > 0 then start_editor; { user typed 'E' }
u gpc_halt(history); { pass history as the exit value to the system }
end.
```

u **1333*** Here we do whatever is needed to complete TEX's job gracefully. Special care is taken to put an end-of-line marker at the end of the last line since that looks better. 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 1333* ⟩ ≡
procedure close_files_and_terminate;
  var k: integer; { all-purpose index }
  begin ⟨ Finish the extensions 1378 ⟩;
  stat if tracing_stats > 0 then ⟨ Output statistics about this job 1334 ⟩; tats
  wake_up_terminal; ⟨ Finish the DVI file 642* ⟩;
  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␣"); slow_print(log_name); print_char(".");
      end;
    end;
u   if term_offset > 0 then wterm_cr; { write eoln if necessary }
  end;
```

See also sections 1335, 1336, and 1338*.

This code is used in section 1330.

1338.* 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 1333* ⟩ +≡
  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;
P   if eof(term_in) then return;  { don't try to read past end of file }
    read(term_in, m);
    if m < 0 then return
    else if m = 0 then
        begin goto breakpoint; @\  { go to every label at least once }
      breakpoint: m ← 0; @{˙BREAKPOINT˙@}@\
        end
P   else begin if eof(term_in) then return;  { don't try to read past end of file }
      read(term_in, n);
      case m of
      ⟨ Numbered cases for debug_help 1339* ⟩
      othercases print("?")
      endcases;
      end;
    end;
exit: end;
  gubed
```

1339*. ⟨ Numbered cases for *debug_help* 1339* ⟩ ≡

```
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: print_word(font_info[n]);
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 }
P 13: begin if eof(term_in) then return;  { don't try to read past end of file }
   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 1338*.

1379* 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.

f **1380*** Try to preload the default format file. This is called even before the first line is read from the terminal, and thus turns `VIRTEX` into `TEX`, at least as seen by the user. `INITEX` sets *format_ident* to '`INITEX`' and won't load a format file here.

```
⟨ Preload the default format file 1380* ⟩ ≡
  if format_ident = 0 then
    begin pack_buffered_name(format_default_length − format_ext_length, 1, 0);
    if ¬w_open_in(fmt_file) then
      begin wterm_ln('I␣can''t␣find␣the␣format␣file␣', name_of_file); goto final_end
      end;
    if ¬load_fmt_file then
      begin w_close(fmt_file); goto final_end
      end;
    w_close(fmt_file);
    end
```

This code is used in section 1332*.

F **1381*** If the user typed '`E`' to edit a file after confronted with an error message, TEX will clean up and then call *start_editor* as its last feat. The file name and line number to be passed to the system editor are saved in *edit_file_name* and *edit_line*.

This procedure must not print error messages, since all files are already closed.

Beware of using any `WEB` strings like `"vi +"` since that would change the string pool file and you'll need to rebuild all format files with the new string pool in case you disagree which editor is the system editor.

G An overflow of *name_of_file* cannot happen, since *name_of_file* kept the file name while the file was being opened. The *gpc_write_str* function writes its arguments into a *string* to build the command line. The *gpc_execute* function is part of the GNU Pascal Runtime System. Its parameter is a *gpc_string* which holds the command line to be executed.

```
define gpc_execute ≡ e@&x@&e@&c@&u@&t@&e
define gpc_write_str ≡ w@&r@&i@&t@&e@&s@&t@&r
⟨ Error handling procedures 78 ⟩ +≡
procedure start_editor;
  var i: integer; { index into name_of_file }
    j: pool_pointer; { index into str_pool }
    cmd_line: gpc_string(200); { area to build the command line }
  begin i ← 1; j ← str_start[edit_file_name];
  while j < str_start[edit_file_name + 1] do
    begin name_of_file[i] ← xchr[str_pool[j]]; incr(i); incr(j)
    end;
  while i ≤ file_name_size do
    begin name_of_file[i] ← '␣'; incr(i)
    end;
  gpc_write_str(cmd_line, 'vi␣+', line, '␣', gpc_trim(name_of_file));
  if 0 ≠ gpc_execute(cmd_line) then
    write_ln(gpc_param_str(0), ':␣could␣not␣start␣editor␣with:␣"', cmd_line, '"');
  end;
```

F **1382:*** The next modules declare and install the interrupt procedure *set_interrupt*.

The identifiers are truncated by `TANGLE` to twelve characters. We use this trick to persuade `TANGLE` to transfer the complete name to the Pascal source.

define *gpc_t_signal_handler* ≡ *t@&s@&i@&g@&n@&a@&l@&h@&a@&n@&d@&l@&e@&r*
define *gpc_install_signal_handler* ≡ *i@&n@&s@&t@&a@&l@&l@&s@&i@&g@&n@&a@&l@&h@&a@&n@&d@&l@&e@&r*
define *gpc_sig_int* ≡ *s@&i@&g@&i@&n@&t*
define *gpc_null* ≡ *n@&u@&l@&l*
define *gpc_integer* ≡ *integer* { for later versions of GPC (3.4+) replace *integer* by *cinteger* }

⟨ Error handling procedures 78 ⟩ +≡
procedure *set_interrupt*(*signal* : *gpc_integer*);
begin *interrupt* ← 1 **end**;

1383:* To install *set_interrupt* as our 'signal handler', I use procedure *gpc_install_signal_handler*. It works with these arguments, but don't ask why. GNU Pascal's *gpc_sig_int* constant denotes the Unix interrupt signal, which is sent when the user types `^C`. Then *set_interrupt* is called, which sets the global variable *interrupt* to one, thus causing TEX to invoke *error* to ask the user what he wants.

⟨ Initialize whatever TEX might access 8 ⟩ +≡
if *gpc_install_signal_handler*(*gpc_sig_int*, *set_interrupt*, *true*, *true*, *gpc_null*, *gpc_null*) **then** *do_nothing*;

1384.* 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, 7, 9, 10, 11, 12, 23, 25, 27, 28, 31, 32, 33, 34, 36, 37, 79, 80, 84, 96, 109, 113, 241, 360, 514, 516, 521, 532, 537, 597, 642, 816, 862, 876, 877, 879, 1332, 1333, 1338, 1339, 1379, 1380, 1381, 1382, 1383, 1384.

⟨ Check single-word *avail* list 168 ⟩ Used in section 167.
⟨ Check that another $ follows 1197 ⟩ Used in sections 1194, 1194, and 1206.
⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← *true* 1195 ⟩ Used in sections 1194 and 1194.
⟨ Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been found, otherwise **goto** *done1* 899 ⟩ Used in section 894.
⟨ Check the "constant" values for consistency 14, 111, 290, 522, 1249 ⟩ Used in section 1332*.
⟨ Check the pool check sum 53 ⟩ Used in section 52.
⟨ Check variable-size *avail* list 169 ⟩ Used in section 167.
⟨ Clean up the memory by removing the break nodes 865 ⟩ Used in sections 815 and 863.
⟨ Clear dimensions to zero 650 ⟩ Used in sections 649 and 668.
⟨ Clear off top level from *save_stack* 282 ⟩ Used in section 281.
⟨ Close the format file 1329 ⟩ Used in section 1302.
⟨ Coerce glue to a dimension 451 ⟩ Used in sections 449 and 455.
⟨ Compiler directives 9* ⟩ Used in section 4*.
⟨ Complain about an undefined family and set *cur_i* null 723 ⟩ Used in section 722.
⟨ Complain about an undefined macro 370 ⟩ Used in section 367.
⟨ Complain about missing \endcsname 373 ⟩ Used in section 372.
⟨ Complain about unknown unit and **goto** *done2* 459 ⟩ Used in section 458.
⟨ Complain that \the can't do this; give zero result 428 ⟩ Used in section 413.
⟨ Complain that the user should have said \mathaccent 1166 ⟩ Used in section 1165.
⟨ Compleat the incompleat noad 1185 ⟩ Used in section 1184.
⟨ Complete a potentially long \show command 1298 ⟩ Used in section 1293.
⟨ Compute result of *multiply* or *divide*, put it in *cur_val* 1240 ⟩ Used in section 1236.
⟨ Compute result of *register* or *advance*, put it in *cur_val* 1238 ⟩ Used in section 1236.
⟨ Compute the amount of skew 741 ⟩ Used in section 738.
⟨ Compute the badness, *b*, of the current page, using *awful_bad* if the box is too full 1007 ⟩ Used in section 1005.
⟨ Compute the badness, *b*, using *awful_bad* if the box is too full 975 ⟩ Used in section 974.
⟨ Compute the demerits, *d*, from *r* to *cur_p* 859 ⟩ Used in section 855.
⟨ Compute the discretionary *break_width* values 840 ⟩ Used in section 837.
⟨ Compute the hash code *h* 261 ⟩ Used in section 259.
⟨ Compute the magic offset 765 ⟩ Used in section 1337.
⟨ Compute the minimum suitable height, *w*, and the corresponding number of extension steps, *n*; also set *width*(*b*) 714 ⟩ Used in section 713.
⟨ Compute the new line width 850 ⟩ Used in section 835.
⟨ Compute the register location *l* and its type *p*; but **return** if invalid 1237 ⟩ Used in section 1236.
⟨ Compute the sum of two glue specs 1239 ⟩ Used in section 1238.
⟨ Compute the trie op code, *v*, and set *l* ← 0 965 ⟩ Used in section 963.
⟨ Compute the values of *break_width* 837 ⟩ Used in section 836.
⟨ Consider a node with matching width; **goto** *found* if it's a hit 612 ⟩ Used in section 611.
⟨ Consider the demerits for a line from *r* to *cur_p*; deactivate node *r* if it should no longer be active; then **goto** *continue* if a line from *r* to *cur_p* is infeasible, otherwise record a new feasible break 851 ⟩ Used in section 829.
⟨ Constants in the outer block 11* ⟩ Used in section 4*.
⟨ Construct a box with limits above and below it, skewed by *delta* 750 ⟩ Used in section 749.
⟨ Construct a sub/superscript combination box *x*, with the superscript offset by *delta* 759 ⟩ Used in section 756.
⟨ Construct a subscript box *x* when there is no superscript 757 ⟩ Used in section 756.
⟨ Construct a superscript box *x* 758 ⟩ Used in section 756.
⟨ Construct a vlist box for the fraction, according to *shift_up* and *shift_down* 747 ⟩ Used in section 743.

⟨ Read box dimensions 571 ⟩ Used in section 562.
⟨ Read character data 569 ⟩ Used in section 562.
⟨ Read extensible character recipes 574 ⟩ Used in section 562.
⟨ Read font parameters 575 ⟩ Used in section 562.
⟨ Read ligature/kern program 573 ⟩ Used in section 562.
⟨ Read next line of file into *buffer*, or **goto** *restart* if the file has ended 362 ⟩ Used in section 360*.
⟨ Read one string, but return *false* if the string memory space is getting too tight for comfort 52 ⟩ Used in section 51.
⟨ Read the first line of the new file 538 ⟩ Used in section 537*.
⟨ Read the other strings from the `TEX.POOL` file and return *true*, or give an error message and return *false* 51 ⟩ Used in section 47.
⟨ Read the `TFM` header 568 ⟩ Used in section 562.
⟨ Read the `TFM` size fields 565 ⟩ Used in section 562.
⟨ Readjust the height and depth of *cur_box*, for `\vtop` 1087 ⟩ Used in section 1086.
⟨ Reconstitute nodes for the hyphenated word, inserting discretionary hyphens 913 ⟩ Used in section 903.
⟨ Record a new feasible break 855 ⟩ Used in section 851.
⟨ Recover from an unbalanced output routine 1027 ⟩ Used in section 1026.
⟨ Recover from an unbalanced write command 1372 ⟩ Used in section 1371.
⟨ Recycle node p 999 ⟩ Used in section 997.
⟨ Remove the last box, unless it's part of a discretionary 1081 ⟩ Used in section 1080.
⟨ Replace nodes *ha* .. *hb* by a sequence of nodes that includes the discretionary hyphens 903 ⟩ Used in section 895.
⟨ Replace the tail of the list by p 1187 ⟩ Used in section 1186.
⟨ Replace z by z' and compute α, β 572 ⟩ Used in section 571.
⟨ Report a runaway argument and abort 396 ⟩ Used in sections 392 and 399.
⟨ Report a tight hbox and **goto** *common_ending*, if this box is sufficiently bad 667 ⟩ Used in section 664.
⟨ Report a tight vbox and **goto** *common_ending*, if this box is sufficiently bad 678 ⟩ Used in section 676.
⟨ Report an extra right brace and **goto** *continue* 395 ⟩ Used in section 392.
⟨ Report an improper use of the macro and abort 398 ⟩ Used in section 397.
⟨ Report an overfull hbox and **goto** *common_ending*, if this box is sufficiently bad 666 ⟩ Used in section 664.
⟨ Report an overfull vbox and **goto** *common_ending*, if this box is sufficiently bad 677 ⟩ Used in section 676.
⟨ Report an underfull hbox and **goto** *common_ending*, if this box is sufficiently bad 660 ⟩ Used in section 658.
⟨ Report an underfull vbox and **goto** *common_ending*, if this box is sufficiently bad 674 ⟩ Used in section 673.
⟨ Report overflow of the input buffer, and abort 35 ⟩ Used in sections 31* and 36*.
⟨ Report that an invalid delimiter code is being changed to null; set *cur_val* ← 0 1161 ⟩ Used in section 1160.
⟨ Report that the font won't be loaded 561 ⟩ Used in section 560.
⟨ Report that this dimension is out of range 460 ⟩ Used in section 448.
⟨ Resume the page builder after an output routine has come to an end 1026 ⟩ Used in section 1100.
⟨ Reverse the links of the relevant passive nodes, setting *cur_p* to the first breakpoint 878 ⟩ Used in section 877*.
⟨ Scan a control sequence and set *state* ← *skip_blanks* or *mid_line* 354 ⟩ Used in section 344.
⟨ Scan a numeric constant 444 ⟩ Used in section 440.
⟨ Scan a parameter until its delimiter string has been found; or, if $s = null$, simply scan the delimiter string 392 ⟩ Used in section 391.
⟨ Scan a subformula enclosed in braces and **return** 1153 ⟩ Used in section 1151.
⟨ Scan ahead in the buffer until finding a nonletter; if an expanded code is encountered, reduce it and **goto** *start_cs*; otherwise if a multiletter control sequence is found, adjust *cur_cs* and *loc*, and **goto** *found* 356 ⟩ Used in section 354.
⟨ Scan an alphabetic character code into *cur_val* 442 ⟩ Used in section 440.
⟨ Scan an optional space 443 ⟩ Used in sections 442, 448, 455, and 1200.
⟨ Scan and build the body of the token list; **goto** *found* when finished 477 ⟩ Used in section 473.
⟨ Scan and build the parameter part of the macro definition 474 ⟩ Used in section 473.

⟨ Undump the hyphenation tables 1325 ⟩ Used in section 1303.
⟨ Undump the string pool 1310 ⟩ Used in section 1303.
⟨ Undump the table of equivalents 1314 ⟩ Used in section 1303.
⟨ Update the active widths, since the first active node has been deleted 861 ⟩ Used in section 860.
⟨ Update the current height and depth measurements with respect to a glue or kern node p 976 ⟩ Used in section 972.
⟨ Update the current page measurements with respect to the glue or kern specified by node p 1004 ⟩ Used in section 997.
⟨ Update the value of *printed_node* for symbolic displays 858 ⟩ Used in section 829.
⟨ Update the values of *first_mark* and *bot_mark* 1016 ⟩ Used in section 1014.
⟨ Update the values of *last_glue*, *last_penalty*, and *last_kern* 996 ⟩ Used in section 994.
⟨ Update the values of *max_h* and *max_v*; but if the page is too large, **goto** *done* 641 ⟩ Used in section 640.
⟨ Update width entry for spanned columns 798 ⟩ Used in section 796.
⟨ Use code c to distinguish between generalized fractions 1182 ⟩ Used in section 1181.
⟨ 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 973 ⟩ Used in section 972.
⟨ Use size fields to allocate font information 566 ⟩ Used in section 562.
⟨ Wipe out the whatsit node p and **goto** *done* 1358 ⟩ 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 1021 ⟩ Used in section 1020.