0* About \TeX\FPC\, \TeX\FPC\ is a port of Donald E. Knuth’s typesetting program \TeX, version 3.141592653 from February 2021 to Free Pascal (FPC) and Unix. To help you identify the differences of \TeX and \TeX\FPC, the numbers of modified modules carry an asterisk. Letters in the left margin indicate the reason for a change. They mean:

E fixes an error in \TeX82
F adds a feature as suggested by Knuth
X describes an FPC extension
U necessary change in Unix
u enhances usability in a Unix environment

E (1) \TeX82 deletes area and extension of an input file name and then only shows the base name of the file during error recovery.
(2) \TeX82 prunes discardable nodes from the beginning of a new line until it reaches a nondiscardable node. This might leave you with an empty box resulting in an Underfull box warning. Btw, I discovered this bug while trying to prove the line breaking algorithm, not while plain testing it. If you have time, prove, if not test.
(3) \TeX fails to respect end of file (Control-D) from terminal input during debug dialog.
(4) Igor Liferenko reported an extra space in the transcript file after the user switched to `/batchmode` during error recovery.

F (1) \TeX\FPC treats the command line as the first input line;
(2) \TeX\FPC starts `ed, the unix system editor, if the user types `E` during error recovery.
(3) You can interrupt \TeX\FPC by typing `Control-C`.

P (1) \TeX82 assumes that the terminal input file is positioned before the first character after being opened, whereas \TeX\FPC assumes that it is positioned at the first character, thus complying with the Pascal standard.
(2) The names of the standard text files must occur in the program header whenever they are used.
(3) The standard text files must not be declared. Declared files with the name of the standard text files are new internal files.
(4) The program must not open the standard text files.

X (1) FPC’s extensions are needed to specify a file name at run time, to check the existence of files and to access the system date and time. Identifiers from FPC Pascal are prefixed with `fp` to help distinguish them from Pascal and `WEB` identifiers and to avoid name clashes. Furthermore all FPC Pascal identifiers will appear together in the index.

U (1) The Unix file separator is `/` instead of `:`.

u (1) On exit, \TeX\FPC 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.
(2) Valid input characters are the 94 visible ASCII characters together with the three control characters horizontal tabulator, form feed, and space.
(3) Terminate last line on terminal. This is Unix, not DOS!
(4) Teach \TeX and user how to end the terminal input by `Control-D`. 

February 2021

Wolfgang Helbig, Programmierer
Waiblingen, Baden-Württemberg
helbig@mailbox.org
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 prototype 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 1983 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 \texttt{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 “\texttt{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].

Even though \TeX\textsc{fpc} does not differ from \TeX I proudly change the banner! And take responsibility for any error.

\texttt{define \texttt{banner} \equiv \texttt{This is \TeX-FPC, 3rd ed.}}
4\* The program begins with a normal Pascal program heading whose components will be filled in later, using the conventions of \Verb+: For example, the portion of the program called \texttt{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.

Actually the heading shown here is not quite normal: The \texttt{program} line does not mention any \texttt{output} file, because Pascal-H would ask the \TeX user to specify a file name if \texttt{output} were specified here.

Pascal wants the identifiers of the standard text files \texttt{input} and \texttt{output} in the parameter list of the program header:

\begin{verbatim}
define \texttt{term\_in} \equiv \texttt{i\@\&\&\&u\@\&t}
define \texttt{term\_out} \equiv \texttt{o\@\&\&\&p\@\&u\@\&t}
define \texttt{mttype} \equiv \texttt{t\@\&y\@\&p\&e}
format \texttt{mttype} \equiv \texttt{type} \{ \texttt{`mttype} will be equivalent to \texttt{`type}\} 
format \texttt{type} \equiv \texttt{true} \{ \texttt{but \texttt{`type} will not be treated as a reserved word}\}
\end{verbatim}

\begin{verbatim}
\langle Compiler directives 9\*\rangle
\end{verbatim}

\begin{verbatim}
program \texttt{TEX}(\texttt{term\_in}, \texttt{term\_out});
label \langle Labels in the outer block 6\rangle
const \langle Constants in the outer block 11\*\rangle
mtype \langle Types in the outer block 18\rangle
var \langle Global variables 13\rangle
procedure \texttt{catch\_signal}(i \colon \texttt{integer}); \texttt{interrupt forward};
procedure \texttt{initialize}; \{ this procedure gets things started properly \}
  var \langle Local variables for initialization 19\rangle
  begin \{ Initialize whatever \TeX might access 8\}
  end;
\langle Basic printing procedures 57\rangle
\langle Error handling procedures 78\rangle
\end{verbatim}

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 \texttt{debug \dots gubed}, with apologies to people who wish to preserve the purity of English.

Similarly, there is some conditional code delimited by \texttt{stat \dots tats} that is intended for use when statistics are to be kept about \TeX's memory usage. The \texttt{stat \dots tats} code also implements diagnostic information for \texttt{\\textbackslash tracing\textbackslash paragraphs}, \texttt{\textbackslash tracing\textbackslash pages}, and \texttt{\textbackslash tracing\textbackslash restores}.

\begin{verbatim}
define \texttt{debug} \equiv \texttt{0}\{ change this to \texttt{`debug} \equiv \texttt{`} when debugging \}
define \texttt{gubed} \equiv \texttt{0}\{ change this to \texttt{`gubed} \equiv \texttt{`} when debugging \}
format \texttt{debug} \equiv \texttt{begin}
format \texttt{gubed} \equiv \texttt{end}
define \texttt{stat} \equiv \{ change this to \texttt{`stat} \equiv \texttt{0}\` to turn off statistics \}
define \texttt{tats} \equiv \{ change this to \texttt{`tats} \equiv \texttt{0}\` to turn off statistics \}
format \texttt{stat} \equiv \texttt{begin}
format \texttt{tats} \equiv \texttt{end}
\end{verbatim}
9* If the first character of a Pascal comment is a dollar sign, Pascal-H treats the comment as a list of “compiler directives” that will affect the translation of this program into machine language. The directives shown below specify full checking and inclusion of the Pascal debugger when \TeX is being debugged, but they cause range checking and other redundant code to be eliminated when the production system is being generated. Arithmetic overflow will be detected in all cases.

\TeX If the first character of a Pascal comment is a dollar sign, Free Pascal treats the comment as a “compiler directive”. Turn off checking since the debugger might trigger a range check when it accesses subfields of a memory word without knowing what it is reading. Overflow is checked if the result of an integer operation overflows the range of 64-bit \textit{integer}. FPC in default mode neither provides \texttt{goto} nor the I/O procedures \texttt{get} and \texttt{put}, and 16-bit \textit{integer}. The compiler directive \texttt{MODE ISO} fixes all of it.

\begin{verbatim}
\langle Compiler directives 9* \rangle \equiv
  @\{\$MODE ISO\} \{ turn on mode ISO \}
  @\{\$Q+\} \{ turn on overflow checking \}
  @\{\$R+\} \{ turn on range checking \}
  debug @\{\$Q-\} @\{\$R-\} gubed \{ turn off all checks when debugging \}
\end{verbatim}

This code is used in section 4*.

10* This \TeX implementation conforms to the rules of the \textit{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 \texttt{case} statements may include a default case that applies if no matching label is found. Thus, we shall use constructions like

\begin{verbatim}
case x of
  1: (code for \texttt{x = 1});
  3: (code for \texttt{x = 3});
othercases (code for \texttt{x \neq 1} and \texttt{x \neq 3})
endcases
\end{verbatim}

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 ‘other:’ as a default label, and other Pascals allow syntaxes like ‘\texttt{else}’ or ‘\texttt{otherwise}’ or ‘\texttt{otherwise:}’, etc. The definitions of \texttt{othercases} and \texttt{endcases} should be changed to agree with local conventions. Note that no semicolon appears before \texttt{endcases} in this program, so the definition of \texttt{endcases} should include a semicolon if the compiler wants one. (Of course, if no default mechanism is available, the \texttt{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!)

\TeX This is the only place I voluntarily use an FPC extension to Pascal.

\begin{verbatim}
define othercases \equiv else \{ default for cases not listed explicitly \}
define endcases \equiv end \{ follows the default case in an extended case statement \}
format othercases \equiv else
format endcases \equiv end
\end{verbatim}
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{}.

\begin{verbatim}
\langle Constants in the outer block \texttt{11} \rangle \equiv

mem_max = 30000; \{ greatest index in \TeX{}'s internal \texttt{mem} array; must be strictly less than \texttt{max_halfword}; must be equal to \texttt{mem_top} in INITEX, otherwise $\geq$ \texttt{mem_top} \}

mem_min = 0; \{ smallest index in \TeX{}'s internal \texttt{mem} array; must be \texttt{min_halfword} or more; must be equal to \texttt{mem_bot} in INITEX, otherwise $\leq$ \texttt{mem_bot} \}

buf_size = 500; \{ maximum number of characters simultaneously present in current lines of open files and in control sequences between \texttt{\csname} and \texttt{\endcsname}; must not exceed \texttt{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_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 \texttt{max_quarterword} and must be at most \texttt{font_base} + 256 \}

font_mem_size = 20000; \{ number of words of \texttt{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_string = 3000; \{ maximum number of strings; must not exceed \texttt{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 \texttt{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 \texttt{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 "\texttt{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'; \{ Unix filename. \}

\end{verbatim}

\texttt{This code is used in section 4*}.  

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 \TeX\book 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 \texttt{xchr[0...37]}, but the safest policy is to blank everything out by using the code shown below.

However, other settings of \texttt{xchr} will make \TeX\ more friendly on computers that have an extended character set, so that users can type things like ‘#’ instead of ‘\times’. People with extended character sets can assign codes arbitrarily, giving an \texttt{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 ’/0. To get the most “permissive” character set, change ‘u’ on the right of these assignment statements to \texttt{chr(i)}.

\begin{verbatim}
\url{\texttt{\\set initial values of key variables 21} \equiv}
    \texttt{for i = 0 to 37 do xchr[i] \leftarrow \textquoteleft u\textquoteright ;};
\url{\texttt{xchr[11] \leftarrow chr(11); \{ accept horizontal tab \}}}
\url{\texttt{xchr[14] \leftarrow chr(14); \{ accept form feed \}}}
\url{\texttt{for i = 177 to 377 do xchr[i] \leftarrow \textquoteleft u\textquoteright ;}}
\end{verbatim}

In Unix tab and form feed are valid characters. The plain format categorizes the tab as a spacer and form feed as an active character defined as \textquoteleft outer\textquoteright .

\url{\texttt{\\}}
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 8-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 8-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.

\begin{verbatim}
\texttt{eight_bits} = 0..255;  \{ unsigned one-byte quantity \}
\texttt{alpha_file} = t0&x0kx0kt;  \{ the type of text files is \textit{text} \}
\texttt{byte_file} = \texttt{packed file of eight_bits};  \{ files that contain binary data \}
\texttt{untyped_file} = \texttt{file};  \{ untyped files for buffered output \}
\end{verbatim}
The Pascal-H compiler with which the present version of \TeX{} was prepared has extended the rules of Pascal in a very convenient way. To open file \texttt{f}, we can write

\begin{verbatim}
reset (f, name, '/0')  for input;
rewrite (f, name, '/0') for output.
\end{verbatim}

The ‘name’ parameter, which is of type ‘\texttt{packed array [\langle any\rangle] of char}’, stands for the name of the external file that is being opened for input or output. Blank spaces that might appear in \texttt{name} are ignored.

The ‘/0’ parameter tells the operating system not to issue its own error messages if something goes wrong. If a file of the specified name cannot be found, or if such a file cannot be opened for some other reason (e.g., someone may already be trying to write the same file), we will have \texttt{erstat(f) \neq 0} after an unsuccessful \texttt{reset} or \texttt{rewrite}. This allows \TeX{} to undertake appropriate corrective action.

The procedure \texttt{fpc_assign} assigns an external file name to a file. The function \texttt{fpc_iow_result} returns a nonzero value if any error occurred since the last invocation of \texttt{fpc_iow_result}. The runtime system halts the program when it experiences an I/O error. Since \TeX{}FPC wants to survive while trying to open a nonexistence file, it turns off I/O checking for the open procedures.

\begin{verbatim}
define fpc_iow_result \equiv \&i0sor0x0s8s000f0t
#define fpc_assign \equiv \&a8s8s0s8s0s000n
#define reset_OK (#) \equiv fpc_iow_result \neq 0
#define rewrite_OK (#) \equiv fpc_iow_result = 0
#define clear_iow_result \equiv if fpc_iow_result = 0 then do nothing
0\&0\&0\&0\&0 \{ turn of I/O checking \}

function a_open_in(var f : alpha_file): boolean;
    begin clear_iow_result; fpc_assign (f, name_of_file); reset (f); a_open_in \leftarrow reset_OK (f);
    end;

function a_open_out(var f : alpha_file): boolean;  \{ open a text file for output \}
    begin clear_iow_result; fpc_assign (f, name_of_file); rewrite (f); a_open_out \leftarrow rewrite_OK (f);
    end;

function b_open_in (var f : byte_file): boolean;  \{ open a binary file for input \}
    begin clear_iow_result; fpc_assign (f, name_of_file); reset (f); b_open_in \leftarrow reset_OK (f);
    end;

function b_open_out(var f : byte_file): boolean;  \{ open a binary file for output \}
    begin clear_iow_result; fpc_assign (f, name_of_file); rewrite (f); b_open_out \leftarrow rewrite_OK (f);
    end;

function w_open_in(var f : word_file): boolean;  \{ open a word file for input \}
    begin clear_iow_result; fpc_assign (f, name_of_file); reset (f); w_open_in \leftarrow reset_OK (f);
    end;

function w_open_out(var f : word_file): boolean;  \{ open a word file for output \}
    begin clear_iow_result; fpc_assign (f, name_of_file); rewrite (f); w_open_out \leftarrow rewrite_OK (f);
    end;
0\&0\&0\&0 \{ turn on I/O checking \}
\end{verbatim}
The `input_in` 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] ≠ "\n"`.

An overflow error is given, however, if the normal actions of `input_in` 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_in`, 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_in`.

If the `bypass_coln` parameter is `true`, `input_in` will do a `get` before looking at the first character of the line; this skips over an `coln` 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 `coln` immediately before `eof`, but TeX needs only a weaker restriction: If `eof` occurs in the middle of a line, the system function `coln` should return a `true` result (even though `f†` will be undefined).

Since the inner loop of `input_in` 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.

---

**function** `input_in` (var `f` : `alpha_file`; `bypass_coln` : boolean) : boolean;

{ inputs the next line or returns false }

```pascal
var last_nonblank : 0 .. buf[size]; { last with trailing blanks removed }
begin { input the first character of the line into `f†` }
last ← first; { cf. Matthew 19:30 }
if `eof(f)` then `input_in` ← `false`
else begin `last_nonblank` ← `first`;
    while `¬coln(f)` do
        begin if `last ≥ max_buf_stack` then
            begin `max_buf_stack` ← `last + 1`;
                if `max_buf_stack = buf_size` then { Report overflow of the input buffer, and abort 35 };
            end;
            `buffer[last]` ← `ord(f†)`; `get(f†)`; `incr(last)`;
            if `buffer[last - 1] ≠ "\n"` then `last_nonblank` ← `last`;
        end;
        `last` ← `last_nonblank`;
    end;
end;
```

**P** Standard Pascal never suppresses the first `get`, so `input_in` must not bypass the first character of the first line. To maintain this rule for subsequent lines, `input_in` is changed to bypass the end of line character at the end of line.

**31** 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`.

**P** No need to declare standard input/output in standard Pascal.
33* Here is how to open the terminal files in Pascal-H. The '/T' switch suppresses the first `get'.

In Pascal, the standard text files are opened implicitly.

```pascal
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.

In Unix, nothing needs to be done here.

```pascal
define fpc_flush ≡ f_getl f_getu f_gets f_geth
define update_terminal ≡ fpc_flush(term_out) { 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 }
```
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`.

(If 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 ‘**’.)

**X** An `fp_c_string` is a packed array `[1 .. fp_c_length]` of `char` with varying length. The function `fp_c_length(s)` returns the length of the `fp_c_string`s. The function `fp_c_param_count` returns the number of command line arguments less one. The function `fp_c_param_str(n)` returns the n-th argument for `0 <= n <= fp_c_param_count`.

**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.

```plaintext
define loc ← cur_input.loc_field  { location of first unread character in buffer }
define fp_c_string ≡ s[0:<0:maxsize][0:0:maxsize]
define fp_c_length ≡ loc+1[loc+1=0]t0`
define fp_c_param_count ≡ p[0:0:maxsize][0:0:maxsize]
define fp_c_param_str(0) ≡ p[0:0:maxsize][0:0:maxsize]
procedure input_command Ln;  { get the command line in buffer }
var argc: integer;  { argument counter }
arg: fp_c_string;  { argument }
cc: integer;  { character counter in argument }
begin last ← first; argc ← 1;
while argc ≤ fp_c_param_count do
  begin cc ← 1; arg ← fp_c_param_str(argc); incr(argc);
  while cc ≤ fp_c_length(arg) do
    begin if last + 1 ≥ bufsize then (Report overflow of the input buffer, and abort 35);
      if xord[arg[cc]] ≠ invalid_code then buffer[last] ← xord[arg[cc]];
      incr(last); incr(cc)
    end;
    if (argc ≤ fp_c_param_count) then
      begin buffer[last] ← "_"; incr(last)  { insert a space between arguments }
    end
  end
end
```
The following program does the required initialization without retrieving a possible command line. The command line is treated as the first terminal line.

Tell user to end the terminal file by Control-D.

```pascal
function init_terminal: boolean; { gets the terminal input started }
  begin
    label exit;
    begin
      open_in; input_command_in; loc ← first;
      if loc < last then
        begin
          init_terminal ← true; return; { first line is the command line }
        end;
    end;
    loop begin
      write(term_out, "**");
      if ~input_in(term_in, true) then { this shouldn’t happen }
        begin
          write_in(term_out);
          write(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_in(term_out, "Please type the name of your input file or Control-D.");
    end;
  end;
```
51* define bad_pool(#) ≡
   begin wake_up_terminal; write_in(term_out,#); 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.〉;
name_of_file ← pool_name;  { we needn’t set name_length }  
if a_open_in(pool_file) then
   begin c ← false;
   repeat 〈Read one string, but return false if the string memory space is getting too tight for comfort.〉;
      until c;
      a_close(pool_file); get_strings_started ← true;
   end  
   else bad_pool(’!uIuCan’uTeXformats/tex.pool.’)  { Unix file name } 
      This code is used in section 52.

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.〉 ≡
   begin a ← 0; k ← 1;
   loop begin if (xord[n] < "0") \∨ (xord[n] > "9") then
      bad_pool(’!uTEX.POOL_u.check_u.sum_u.doesn’t_u.have_u.nine_u.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(’!uTeXformats/tex.pool_u.doesn’t_u.match_u.Not_u.installed?’);
   end  
      { Unix file name }
      c ← true;
end

This code is used in section 52.
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.

\[
\text{define help1(#)} \equiv \text{help_line[0]} \leftarrow \#; \text{end}
\]
\[
\text{define help2(#)} \equiv \text{help_line[1]} \leftarrow \# ; \text{help1}
\]
\[
\text{define help3(#)} \equiv \text{help_line[2]} \leftarrow \# ; \text{help2}
\]
\[
\text{define help4(#)} \equiv \text{help_line[3]} \leftarrow \# ; \text{help3}
\]
\[
\text{define help5(#)} \equiv \text{help_line[4]} \leftarrow \# ; \text{help4}
\]
\[
\text{define help6(#)} \equiv \text{help_line[5]} \leftarrow \# ; \text{help5}
\]
\[
\text{define help0} \equiv \text{help_ptr} \leftarrow 0 \quad \{ \text{sometimes there might be no help} \}
\]
\[
\text{define help1} \equiv \text{begin help_ptr} \leftarrow 1 ; \text{help1} \quad \{ \text{use this with one help line} \}
\]
\[
\text{define help2} \equiv \text{begin help_ptr} \leftarrow 2 ; \text{help2} \quad \{ \text{use this with two help lines} \}
\]
\[
\text{define help3} \equiv \text{begin help_ptr} \leftarrow 3 ; \text{help3} \quad \{ \text{use this with three help lines} \}
\]
\[
\text{define help4} \equiv \text{begin help_ptr} \leftarrow 4 ; \text{help4} \quad \{ \text{use this with four help lines} \}
\]
\[
\text{define help5} \equiv \text{begin help_ptr} \leftarrow 5 ; \text{help5} \quad \{ \text{use this with five help lines} \}
\]
\[
\text{define help6} \equiv \text{begin help_ptr} \leftarrow 6 ; \text{help6} \quad \{ \text{use this with six help lines} \}
\]

\{(Global variables 13) +≡

\[\text{help_line: array [0 . . . 5] of str_number; \{ helps for the next error \}}\]
\[\text{help_ptr: 0 . . 6; \{ the number of help lines present \}}\]
\[\text{use_err_help: boolean; \{ should the err_help list be shown? \}}\]
\[\text{want_edit: boolean; \{ start vi? \}}\]

80* (Set initial values of key variables 21) +≡

\[\text{help_ptr} \leftarrow 0 ; \text{use_err_help} \leftarrow \text{false} ;\]
\[\text{F want_edit} \leftarrow \text{false}; \quad \{ \text{don't start ed} \}\]

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* ) +≡

\text{case c of}

"0", "1", "2", "3", "4", "5", "6", "7", "8", "9": \text{if deletions_allowed then}
\{(Delete c = "0" tokens and goto continue 88)\};
\text{debug "D": begin debug_help; goto continue; end; gubed}

"E": \text{if base_ptr > 0 then}
\{(Print the help information and goto continue 89)\};

"H": \{Print the help information and goto continue 87\};

"T": \{Introduce new material from the terminal and return 87\};

"Q", "R", "S": \{Change the interaction level and return 86*\};

"X": \text{begin interaction \leftarrow scroll_mode; jump_out;}
\text{end;}
\text{othercases do_nothing}
\text{endcases;}\}

\{(Print the menu of available options 85\}

This code is used in section 83.
Here the author of \TeX{} apologizes for making use of the numerical relation between "Q", "R", "S", and the desired interaction settings \texttt{batch\_mode}, \texttt{nonstop\_mode}, \texttt{scroll\_mode}.

Change the interaction level and \texttt{return 86*}:

\begin{verbatim}
begin error\_count ← 0; interaction ← batch\_mode + c − "Q"; print("OK, entering");
case c of
   "Q": print\_esc ("batchmode"); {don't turn off terminal now}
   "R": print\_esc ("nonstopmode");
   "S": print\_esc ("scrollmode");
end; {there are no other cases}
   print("..."); print\_ln; update\_terminal;
if c = "Q" then decr (selector);
return; {but now}
end
\end{verbatim}

This code is used in section 84*. 
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 \texttt{glue\_ratio} for such proportionality ratios. A glue ratio should take the same amount of memory as an \texttt{integer} (usually 32 bits) if it is to blend smoothly with \TeX{}'s other data structures. Thus \texttt{glue\_ratio} should be equivalent to \texttt{short\_real} in some implementations of Pascal. Alternatively, it is possible to deal with glue ratios using nothing but fixed-point arithmetic; see \textit{TUGboat} 3,1 (March 1982), 10-27. (But the routines cited there must be modified to allow negative glue ratios.)

In FPC Pascal the type \texttt{fp\_single} seems appropriate.

\begin{verbatim}
define \texttt{fp\_single} \equiv \texttt{short\_real}  
define \texttt{set\_glue\_ratio\_zero} \equiv \texttt{#} \leftarrow 0.0 \quad \text{(store the representation of zero ratio)}  
define \texttt{set\_glue\_ratio\_one} \equiv \texttt{#} \leftarrow 1.0 \quad \text{(store the representation of unit ratio)}  
define \texttt{float} \equiv \texttt{#} \quad \text{(convert from \texttt{glue\_ratio} to type \texttt{real})}  
define \texttt{unfloat} \equiv \texttt{#} \quad \text{(convert from \texttt{real} to type \texttt{glue\_ratio})}  
define \texttt{float\_constant} \equiv \texttt{#} \leftarrow 0 \quad \text{(convert \texttt{integer} constant to \texttt{real})}  
\end{verbatim}

\begin{verbatim}
(\text{Types in the outer block})  +\equiv
\begin{align*}
glue\_ratio &= \texttt{fp\_single} ; \quad \text{\{one-word representation of a glue expansion factor in FPC Pascal\}}
\end{align*}
\end{verbatim}
The operation of adding or subtracting \textit{min-quarterword} occurs quite frequently in \TeX, so it is convenient to abbreviate this operation by using the macros \texttt{qi} and \texttt{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 ‘\texttt{x+0}’ and ‘\texttt{x-0}’, if these macros are simplified in the obvious way when \textit{min-quarterword} = 0.

\begin{verbatim}
X  Which is the case with FPC.

define qi (#) ≡ #  { to put an \textit{eight-bits} item into a quarterword }
define qo (#) ≡ #  { to take an \textit{eight-bits} item out of a quarterword }
define hi (#) ≡ #  { to put a sixteen-bit item into a halfword }
define ho (#) ≡ #  { to take a sixteen-bit item from a halfword }
\end{verbatim}
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 assumes that suitable values appear in the global variables \texttt{sys\_time}, \texttt{sys\_day}, \texttt{sys\_month}, and \texttt{sys\_year} (which are initialized to noon on 4 July 1776, in case the implementor is careless).

\TeX{} The functions \texttt{now}, \texttt{decode\_date}, and \texttt{decode\_time} are provided by the unit \texttt{sysutils}. The command line option \texttt{fpcu-Fsysutils.tex.p} links that unit. When FPC is in ISO mode, it does not accept declaring a unit in the source file.

\begin{verbatim}
define fpc\_now \equiv \texttt{now}  
define fpc\_decode\_date \equiv \texttt{decode\_date}  
define fpc\_decode\_time \equiv \texttt{decode\_time}  

procedure fix\_date\_and\_time;  
var yy, mm, dd: \texttt{word}; hh, ss, ms: \texttt{word};  
begin fpc\_decode\_date(fpc\_now, yy, mm, dd);  \{ current date \}  
  sys\_day \leftarrow dd;  \texttt{day} \leftarrow sys\_day;  \texttt{sys\_month} \leftarrow mm;  \texttt{month} \leftarrow sys\_month;  \texttt{sys\_year} \leftarrow yy;  
  \texttt{year} \leftarrow sys\_year;  
  fpc\_decode\_time(fpc\_now, hh, mm, ss, ms);  \{ current time \}  
  \texttt{sys\_time} \leftarrow hh * 60 + mm;  \texttt{time} \leftarrow sys\_time;  \{ minutes since midnight \}  
end;  
\end{verbatim}