1. Introduction. The DVItype utility program reads binary device-independent ("DVI") files that are produced by document compilers such as TEX, and converts them into symbolic form. This program has two chief purposes: (1) It can be used to determine whether a DVI file is valid or invalid, when diagnosing compiler errors; and (2) it serves as an example of a program that reads DVI files correctly, for system programmers who are developing DVI-related software.

Goal number (2) needs perhaps a bit more explanation. Programs for typesetting need to be especially careful about how they do arithmetic; if rounding errors accumulate, margins won't be straight, vertical rules won't line up, and so on. But if rounding is done everywhere, even in the midst of words, there will be uneven spacing between the letters, and that looks bad. Human eyes notice differences of a thousandth of an inch in the positioning of lines that are close together; on low resolution devices, where rounding produces effects four times as great as this, the problem is especially critical. Experience has shown that unusual care is needed even on high-resolution equipment; for example, a mistake in the sixth significant hexadecimal place of a constant once led to a difficult-to-find bug in some software for the Alphatype CRS, which has a resolution of 5333 pixels per inch (make that 5333.33333333 pixels per inch). The document compilers that generate DVI files make certain assumptions about the arithmetic that will be used by DVI-reading software, and if these assumptions are violated the results will be of inferior quality. Therefore the present program is intended as a guide to proper procedure in the critical places where a bit of subtlety is involved.

The first DVItype program was designed by David Fuchs in 1979, and it went through several versions on different computers as the format of DVI files was evolving to its present form. Peter Breitenlohner helped with the latest revisions.

The *banner* string defined here should be changed whenever DVItype gets modified.

define *banner* ≡ ´This␣is␣DVItype,␣Version␣3.6´ { printed when the program starts }

2. This program is written in standard Pascal, except where it is necessary to use extensions; for example, DVItype must read files whose names are dynamically specified, and that would be impossible in pure Pascal. All places where nonstandard constructions are used have been listed in the index under "system dependencies."

One of the extensions to standard Pascal that we shall deal with is the ability to move to a random place in a binary file; another is to determine the length of a binary file. Such extensions are not necessary for reading DVI files, and they are not important for efficiency reasons either—an infrequently used program like DVItype does not have to be efficient. But they are included there because of DVItype's rôle as a model of a DVI reading routine, since other DVI processors ought to be highly efficient. If DVItype is being used with Pascals for which random file positioning is not efficiently available, the following definition should be changed from *true* to *false*; in such cases, DVItype will not include the optional feature that reads the postamble first.

Another extension is to use a default **case** as in TANGLE, WEAVE, etc.

define *random_reading* ≡ *true* { should we skip around in the file? }
define *othercases* ≡ *others*: { default for cases not listed explicitly }
define *endcases* ≡ **end** { follows the default case in an extended **case** statement }
format *othercases* ≡ *else*
format *endcases* ≡ *end*

3. The binary input comes from *dvi_file*, and the symbolic output is written on Pascal's standard *output* file. The term *print* is used instead of *write* when this program writes on *output*, so that all such output could easily be redirected if desired.

```
define print(#) ≡ write(#)
define print_ln(#) ≡ write_ln(#)
program DVI_type(dvi_file, output);
  label ⟨ Labels in the outer block 4 ⟩
  const ⟨ Constants in the outer block 5 ⟩
  type ⟨ Types in the outer block 8 ⟩
  var ⟨ Globals in the outer block 10 ⟩
  procedure initialize; { this procedure gets things started properly }
    var i: integer; { loop index for initializations }
    begin print_ln(banner);
    ⟨ Set initial values 11 ⟩
    end;
```

4. If the program has to stop prematurely, it goes to the '*final_end*'. Another label, *done*, is used when stopping normally.

```
define final_end = 9999 { label for the end of it all }
define done = 30 { go here when finished with a subtask }
⟨ Labels in the outer block 4 ⟩ ≡
  final_end, done;
```

This code is used in section 3.

5. The following parameters can be changed at compile time to extend or reduce `DVItype`'s capacity.

```
⟨ Constants in the outer block 5 ⟩ ≡
  max_fonts = 100; { maximum number of distinct fonts per DVI file }
  max_widths = 10000; { maximum number of different characters among all fonts }
  line_length = 79; { bracketed lines of output will be at most this long }
  terminal_line_length = 150;
      { maximum number of characters input in a single line of input from the terminal }
  stack_size = 100; { DVI files shouldn't push beyond this depth }
  name_size = 1000; { total length of all font file names }
  name_length = 50; { a file name shouldn't be longer than this }
```

This code is used in section 3.

6. Here are some macros for common programming idioms.

```
define incr(#) ≡ # ← # + 1 { increase a variable by unity }
define decr(#) ≡ # ← # − 1 { decrease a variable by unity }
define do_nothing ≡ { empty statement }
```

7. If the DVI file is badly malformed, the whole process must be aborted; DVItype will give up, after issuing an error message about the symptoms that were noticed.

Such errors might be discovered inside of subroutines inside of subroutines, so a procedure called *jump_out* has been introduced. This procedure, which simply transfers control to the label *final_end* at the end of the program, contains the only non-local **goto** statement in DVItype.

```
define abort(#) ≡
        begin print(´␣´,#); jump_out;
        end
define bad_dvi(#) ≡ abort(´Bad␣DVI␣file:␣´,#,´!´)
procedure jump_out;
  begin goto final_end;
  end;
```

8. The character set. Like all programs written with the `WEB` system, `DVItype` can be used with any character set. But it uses ASCII code internally, because the programming for portable input-output is easier when a fixed internal code is used, and because `DVI` files use ASCII code for file names and certain other strings.

The next few sections of `DVItype` have therefore been copied from the analogous ones in the `WEB` system routines. They have been considerably simplified, since `DVItype` need not deal with the controversial ASCII codes less than *´40* or greater than *´176*. If such codes appear in the `DVI` file, they will be printed as question marks.

⟨ Types in the outer block 8 ⟩ ≡
  ASCII_code = "␣" .. "~"; { a subrange of the integers }

See also sections 9 and 21.

This code is used in section 3.

9. The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets in a convenient way, especially in a program like `DVItype`. So we shall assume that the Pascal system being used for `DVItype` has a character set containing at least the standard visible characters of ASCII code ("!" through "~").

Some Pascal compilers use the original name *char* for the data type associated with the characters in text files, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name. In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters in the output file. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define *text_char* ≡ *char* { the data type of characters in text files }
define *first_text_char* = 0 { ordinal number of the smallest element of *text_char* }
define *last_text_char* = 127 { ordinal number of the largest element of *text_char* }

⟨ Types in the outer block 8 ⟩ +≡
  text_file = **packed file of** *text_char*;

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

⟨ Globals in the outer block 10 ⟩ ≡
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [0 .. 255] **of** *text_char*; { specifies conversion of output characters }

See also sections 22, 24, 25, 30, 33, 39, 41, 42, 45, 48, 57, 64, 67, 72, 73, 78, 97, 101, and 108.

This code is used in section 3.

11. Under our assumption that the visible characters of standard ASCII are all present, the following assignment statements initialize the *xchr* array properly, without needing any system-dependent changes.

⟨ Set initial values 11 ⟩ ≡

```
for i ← 0 to '37 do xchr[i] ← '?';
xchr['40] ← ' '; xchr['41] ← '!'; xchr['42] ← '"'; xchr['43] ← '#'; xchr['44] ← '$';
xchr['45] ← '%'; xchr['46] ← '&'; xchr['47] ← ''';
xchr['50] ← '('; xchr['51] ← ')'; xchr['52] ← '*'; xchr['53] ← '+'; xchr['54] ← ',';
xchr['55] ← '-'; xchr['56] ← '.'; xchr['57] ← '/';
xchr['60] ← '0'; xchr['61] ← '1'; xchr['62] ← '2'; xchr['63] ← '3'; xchr['64] ← '4';
xchr['65] ← '5'; xchr['66] ← '6'; xchr['67] ← '7';
xchr['70] ← '8'; xchr['71] ← '9'; xchr['72] ← ':'; xchr['73] ← ';'; xchr['74] ← '<';
xchr['75] ← '='; xchr['76] ← '>'; xchr['77] ← '?';
xchr['100] ← '@'; xchr['101] ← 'A'; xchr['102] ← 'B'; xchr['103] ← 'C'; xchr['104] ← 'D';
xchr['105] ← 'E'; xchr['106] ← 'F'; xchr['107] ← 'G';
xchr['110] ← 'H'; xchr['111] ← 'I'; xchr['112] ← 'J'; xchr['113] ← 'K'; xchr['114] ← 'L';
xchr['115] ← 'M'; xchr['116] ← 'N'; xchr['117] ← 'O';
xchr['120] ← 'P'; xchr['121] ← 'Q'; xchr['122] ← 'R'; xchr['123] ← 'S'; xchr['124] ← 'T';
xchr['125] ← 'U'; xchr['126] ← 'V'; xchr['127] ← 'W';
xchr['130] ← 'X'; xchr['131] ← 'Y'; xchr['132] ← 'Z'; xchr['133] ← '['; xchr['134] ← '\';
xchr['135] ← ']'; xchr['136] ← '^'; xchr['137] ← '_';
xchr['140] ← '`'; xchr['141] ← 'a'; xchr['142] ← 'b'; xchr['143] ← 'c'; xchr['144] ← 'd';
xchr['145] ← 'e'; xchr['146] ← 'f'; xchr['147] ← 'g';
xchr['150] ← 'h'; xchr['151] ← 'i'; xchr['152] ← 'j'; xchr['153] ← 'k'; xchr['154] ← 'l';
xchr['155] ← 'm'; xchr['156] ← 'n'; xchr['157] ← 'o';
xchr['160] ← 'p'; xchr['161] ← 'q'; xchr['162] ← 'r'; xchr['163] ← 's'; xchr['164] ← 't';
xchr['165] ← 'u'; xchr['166] ← 'v'; xchr['167] ← 'w';
xchr['170] ← 'x'; xchr['171] ← 'y'; xchr['172] ← 'z'; xchr['173] ← '{'; xchr['174] ← '|';
xchr['175] ← '}'; xchr['176] ← '~';
for i ← '177 to 255 do xchr[i] ← '?';
```

See also sections 12, 31, 43, 58, 65, 68, 74, and 98.

This code is used in section 3.

12. The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*.

⟨ Set initial values 11 ⟩ +≡

```
for i ← first_text_char to last_text_char do xord[chr(i)] ← '40;
for i ← " " to "~" do xord[xchr[i]] ← i;
```

13. Device-independent file format. Before we get into the details of `DVItype`, we need to know exactly what `DVI` files are. The form of such files was designed by David R. Fuchs in 1979. Almost any reasonable typesetting device can be driven by a program that takes `DVI` files as input, and dozens of such `DVI`-to-whatever programs have been written. Thus, it is possible to print the output of document compilers like TEX on many different kinds of equipment.

A `DVI` file is a stream of 8-bit bytes, which may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the '*set_rule*' command has two parameters, each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters, and shorter parameters that denote distances, can be either positive or negative. Such parameters are given in two's complement notation. For example, a two-byte-long distance parameter has a value between -2^{15} and $2^{15}-1$.

A `DVI` file consists of a "preamble," followed by a sequence of one or more "pages," followed by a "postamble." The preamble is simply a *pre* command, with its parameters that define the dimensions used in the file; this must come first. Each "page" consists of a *bop* command, followed by any number of other commands that tell where characters are to be placed on a physical page, followed by an *eop* command. The pages appear in the order that they were generated, not in any particular numerical order. If we ignore *nop* commands and *fnt_def* commands (which are allowed between any two commands in the file), each *eop* command is immediately followed by a *bop* command, or by a *post* command; in the latter case, there are no more pages in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in `DVI` commands are "pointers." These are four-byte quantities that give the location number of some other byte in the file; the first byte is number 0, then comes number 1, and so on. For example, one of the parameters of a *bop* command points to the previous *bop*; this makes it feasible to read the pages in backwards order, in case the results are being directed to a device that stacks its output face up. Suppose the preamble of a `DVI` file occupies bytes 0 to 99. Now if the first page occupies bytes 100 to 999, say, and if the second page occupies bytes 1000 to 1999, then the *bop* that starts in byte 1000 points to 100 and the *bop* that starts in byte 2000 points to 1000. (The very first *bop*, i.e., the one that starts in byte 100, has a pointer of -1.)

14. The `DVI` format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information implicit instead of explicit. When a `DVI`-reading program reads the commands for a page, it keeps track of several quantities: (a) The current font f is an integer; this value is changed only by *fnt* and *fnt_num* commands. (b) The current position on the page is given by two numbers called the horizontal and vertical coordinates, h and v. Both coordinates are zero at the upper left corner of the page; moving to the right corresponds to increasing the horizontal coordinate, and moving down corresponds to increasing the vertical coordinate. Thus, the coordinates are essentially Cartesian, except that vertical directions are flipped; the Cartesian version of (h, v) would be $(h, -v)$. (c) The current spacing amounts are given by four numbers w, x, y, and z, where w and x are used for horizontal spacing and where y and z are used for vertical spacing. (d) There is a stack containing (h, v, w, x, y, z) values; the `DVI` commands *push* and *pop* are used to change the current level of operation. Note that the current font f is not pushed and popped; the stack contains only information about positioning.

The values of h, v, w, x, y, and z are signed integers having up to 32 bits, including the sign. Since they represent physical distances, there is a small unit of measurement such that increasing h by 1 means moving a certain tiny distance to the right. The actual unit of measurement is variable, as explained below.

15. Here is a list of all the commands that may appear in a DVI file. Each command is specified by its symbolic name (e.g., *bop*), its opcode byte (e.g., 139), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, 'p[4]' means that parameter p is four bytes long.

set_char_0 0. Typeset character number 0 from font f such that the reference point of the character is at (h, v). Then increase h by the width of that character. Note that a character may have zero or negative width, so one cannot be sure that h will advance after this command; but h usually does increase.

set_char_1 through *set_char_127* (opcodes 1 to 127). Do the operations of *set_char_0*; but use the character whose number matches the opcode, instead of character 0.

set1 128 c[1]. Same as *set_char_0*, except that character number c is typeset. TEX82 uses this command for characters in the range $128 \le c < 256$.

set2 129 c[2]. Same as *set1*, except that c is two bytes long, so it is in the range $0 \le c < 65536$. TEX82 never uses this command, which is intended for processors that deal with oriental languages; but DVItype will allow character codes greater than 255, assuming that they all have the same width as the character whose code is c mod 256.

set3 130 c[3]. Same as *set1*, except that c is three bytes long, so it can be as large as $2^{24} - 1$.

set4 131 c[4]. Same as *set1*, except that c is four bytes long, possibly even negative. Imagine that.

set_rule 132 a[4] b[4]. Typeset a solid black rectangle of height a and width b, with its bottom left corner at (h, v). Then set $h \leftarrow h + b$. If either $a \le 0$ or $b \le 0$, nothing should be typeset. Note that if $b < 0$, the value of h will decrease even though nothing else happens. Programs that typeset from DVI files should be careful to make the rules line up carefully with digitized characters, as explained in connection with the *rule_pixels* subroutine below.

put1 133 c[1]. Typeset character number c from font f such that the reference point of the character is at (h, v). (The 'put' commands are exactly like the 'set' commands, except that they simply put out a character or a rule without moving the reference point afterwards.)

put2 134 c[2]. Same as *set2*, except that h is not changed.

put3 135 c[3]. Same as *set3*, except that h is not changed.

put4 136 c[4]. Same as *set4*, except that h is not changed.

put_rule 137 a[4] b[4]. Same as *set_rule*, except that h is not changed.

nop 138. No operation, do nothing. Any number of *nop*'s may occur between DVI commands, but a *nop* cannot be inserted between a command and its parameters or between two parameters.

bop 139 c_0[4] c_1[4] ... c_9[4] p[4]. Beginning of a page: Set $(h, v, w, x, y, z) \leftarrow (0, 0, 0, 0, 0, 0)$ and set the stack empty. Set the current font f to an undefined value. The ten c_i parameters can be used to identify pages, if a user wants to print only part of a DVI file; TEX82 gives them the values of \count0 ... \count9 at the time \shipout was invoked for this page. The parameter p points to the previous *bop* command in the file, where the first *bop* has $p = -1$.

eop 140. End of page: Print what you have read since the previous *bop*. At this point the stack should be empty. (The DVI-reading programs that drive most output devices will have kept a buffer of the material that appears on the page that has just ended. This material is largely, but not entirely, in order by v coordinate and (for fixed v) by h coordinate; so it usually needs to be sorted into some order that is appropriate for the device in question. DVItype does not do such sorting.)

push 141. Push the current values of (h, v, w, x, y, z) onto the top of the stack; do not change any of these values. Note that f is not pushed.

pop 142. Pop the top six values off of the stack and assign them to (h, v, w, x, y, z). The number of pops should never exceed the number of pushes, since it would be highly embarrassing if the stack were empty at the time of a *pop* command.

right1 143 b[1]. Set $h \leftarrow h + b$, i.e., move right b units. The parameter is a signed number in two's complement notation, $-128 \le b < 128$; if $b < 0$, the reference point actually moves left.

right2 144 b[2]. Same as *right1*, except that b is a two-byte quantity in the range $-32768 \le b < 32768$.

right3 145 b[3]. Same as *right1*, except that b is a three-byte quantity in the range $-2^{23} \le b < 2^{23}$.

right4 146 b[4]. Same as *right1*, except that b is a four-byte quantity in the range $-2^{31} \le b < 2^{31}$.

w0 147. Set $h \leftarrow h + w$; i.e., move right w units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how w gets particular values.

w1 148 b[1]. Set $w \leftarrow b$ and $h \leftarrow h + b$. The value of b is a signed quantity in two's complement notation, $-128 \le b < 128$. This command changes the current w spacing and moves right by b.

w2 149 b[2]. Same as *w1*, but b is a two-byte-long parameter, $-32768 \le b < 32768$.

w3 150 b[3]. Same as *w1*, but b is a three-byte-long parameter, $-2^{23} \le b < 2^{23}$.

w4 151 b[4]. Same as *w1*, but b is a four-byte-long parameter, $-2^{31} \le b < 2^{31}$.

x0 152. Set $h \leftarrow h + x$; i.e., move right x units. The 'x' commands are like the 'w' commands except that they involve x instead of w.

x1 153 b[1]. Set $x \leftarrow b$ and $h \leftarrow h + b$. The value of b is a signed quantity in two's complement notation, $-128 \le b < 128$. This command changes the current x spacing and moves right by b.

x2 154 b[2]. Same as *x1*, but b is a two-byte-long parameter, $-32768 \le b < 32768$.

x3 155 b[3]. Same as *x1*, but b is a three-byte-long parameter, $-2^{23} \le b < 2^{23}$.

x4 156 b[4]. Same as *x1*, but b is a four-byte-long parameter, $-2^{31} \le b < 2^{31}$.

down1 157 a[1]. Set $v \leftarrow v + a$, i.e., move down a units. The parameter is a signed number in two's complement notation, $-128 \le a < 128$; if $a < 0$, the reference point actually moves up.

down2 158 a[2]. Same as *down1*, except that a is a two-byte quantity in the range $-32768 \le a < 32768$.

down3 159 a[3]. Same as *down1*, except that a is a three-byte quantity in the range $-2^{23} \le a < 2^{23}$.

down4 160 a[4]. Same as *down1*, except that a is a four-byte quantity in the range $-2^{31} \le a < 2^{31}$.

y0 161. Set $v \leftarrow v + y$; i.e., move down y units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how y gets particular values.

y1 162 a[1]. Set $y \leftarrow a$ and $v \leftarrow v + a$. The value of a is a signed quantity in two's complement notation, $-128 \le a < 128$. This command changes the current y spacing and moves down by a.

y2 163 a[2]. Same as *y1*, but a is a two-byte-long parameter, $-32768 \le a < 32768$.

y3 164 a[3]. Same as *y1*, but a is a three-byte-long parameter, $-2^{23} \le a < 2^{23}$.

y4 165 a[4]. Same as *y1*, but a is a four-byte-long parameter, $-2^{31} \le a < 2^{31}$.

z0 166. Set $v \leftarrow v + z$; i.e., move down z units. The 'z' commands are like the 'y' commands except that they involve z instead of y.

z1 167 a[1]. Set $z \leftarrow a$ and $v \leftarrow v + a$. The value of a is a signed quantity in two's complement notation, $-128 \le a < 128$. This command changes the current z spacing and moves down by a.

z2 168 a[2]. Same as *z1*, but a is a two-byte-long parameter, $-32768 \le a < 32768$.

z3 169 a[3]. Same as *z1*, but a is a three-byte-long parameter, $-2^{23} \le a < 2^{23}$.

z4 170 a[4]. Same as *z1*, but a is a four-byte-long parameter, $-2^{31} \le a < 2^{31}$.

fnt_num_0 171. Set $f \leftarrow 0$. Font 0 must previously have been defined by a *fnt_def* instruction, as explained below.

fnt_num_1 through *fnt_num_63* (opcodes 172 to 234). Set $f \leftarrow 1, \ldots, f \leftarrow 63$, respectively.

fnt1 235 k[1]. Set $f \leftarrow k$. TEX82 uses this command for font numbers in the range $64 \le k < 256$.

fnt2 236 k[2]. Same as *fnt1*, except that k is two bytes long, so it is in the range $0 \le k < 65536$. TEX82 never generates this command, but large font numbers may prove useful for specifications of color or texture, or they may be used for special fonts that have fixed numbers in some external coding scheme.

fnt3 237 $k[3]$. Same as *fnt1*, except that k is three bytes long, so it can be as large as $2^{24}-1$.

fnt4 238 $k[4]$. Same as *fnt1*, except that k is four bytes long; this is for the really big font numbers (and for the negative ones).

xxx1 239 $k[1]$ $x[k]$. This command is undefined in general; it functions as a $(k+2)$-byte *nop* unless special `DVI`-reading programs are being used. TEX82 generates *xxx1* when a short enough `\special` appears, setting k to the number of bytes being sent. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 240 $k[2]$ $x[k]$. Like *xxx1*, but $0 \le k < 65536$.

xxx3 241 $k[3]$ $x[k]$. Like *xxx1*, but $0 \le k < 2^{24}$.

xxx4 242 $k[4]$ $x[k]$. Like *xxx1*, but k can be ridiculously large. TEX82 uses *xxx4* when *xxx1* would be incorrect.

fnt_def1 243 $k[1]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 256$; font definitions will be explained shortly.

fnt_def2 244 $k[2]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 65536$.

fnt_def3 245 $k[3]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 2^{24}$.

fnt_def4 246 $k[4]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $-2^{31} \le k < 2^{31}$.

pre 247 $i[1]$ $num[4]$ $den[4]$ $mag[4]$ $k[1]$ $x[k]$. Beginning of the preamble; this must come at the very beginning of the file. Parameters i, num, den, mag, k, and x are explained below.

post 248. Beginning of the postamble, see below.

post_post 249. Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

16. **define** *set_char_0* = 0 { typeset character 0 and move right }
define *set1* = 128 { typeset a character and move right }
define *set_rule* = 132 { typeset a rule and move right }
define *put1* = 133 { typeset a character }
define *put_rule* = 137 { typeset a rule }
define *nop* = 138 { no operation }
define *bop* = 139 { beginning of page }
define *eop* = 140 { ending of page }
define *push* = 141 { save the current positions }
define *pop* = 142 { restore previous positions }
define *right1* = 143 { move right }
define *w0* = 147 { move right by w }
define *w1* = 148 { move right and set w }
define *x0* = 152 { move right by x }
define *x1* = 153 { move right and set x }
define *down1* = 157 { move down }
define *y0* = 161 { move down by y }
define *y1* = 162 { move down and set y }
define *z0* = 166 { move down by z }
define *z1* = 167 { move down and set z }
define *fnt_num_0* = 171 { set current font to 0 }
define *fnt1* = 235 { set current font }
define *xxx1* = 239 { extension to DVI primitives }
define *xxx4* = 242 { potentially long extension to DVI primitives }
define *fnt_def1* = 243 { define the meaning of a font number }
define *pre* = 247 { preamble }
define *post* = 248 { postamble beginning }
define *post_post* = 249 { postamble ending }
define *undefined_commands* ≡ 250, 251, 252, 253, 254, 255

17. The preamble contains basic information about the file as a whole. As stated above, there are six parameters:

$$i[1]\ num[4]\ den[4]\ mag[4]\ k[1]\ x[k].$$

The i byte identifies DVI format; currently this byte is always set to 2. (The value $i = 3$ is currently used for an extended format that allows a mixture of right-to-left and left-to-right typesetting. Some day we will set $i = 4$, when DVI format makes another incompatible change—perhaps in the year 2048.)

The next two parameters, *num* and *den*, are positive integers that define the units of measurement; they are the numerator and denominator of a fraction by which all dimensions in the DVI file could be multiplied in order to get lengths in units of 10^{-7} meters. (For example, there are exactly 7227 TEX points in 254 centimeters, and TEX82 works with scaled points where there are 2^{16} sp in a point, so TEX82 sets $num = 25400000$ and $den = 7227 \cdot 2^{16} = 473628672$.)

The *mag* parameter is what TEX82 calls \mag, i.e., 1000 times the desired magnification. The actual fraction by which dimensions are multiplied is therefore $mn/1000d$. Note that if a TEX source document does not call for any 'true' dimensions, and if you change it only by specifying a different \mag setting, the DVI file that TEX creates will be completely unchanged except for the value of *mag* in the preamble and postamble. (Fancy DVI-reading programs allow users to override the *mag* setting when a DVI file is being printed.)

Finally, k and x allow the DVI writer to include a comment, which is not interpreted further. The length of comment x is k, where $0 \le k < 256$.

define *id_byte* = 2 { identifies the kind of DVI files described here }

18. Font definitions for a given font number k contain further parameters

$$c[4]\ s[4]\ d[4]\ a[1]\ l[1]\ n[a+l].$$

The four-byte value c is the check sum that TEX (or whatever program generated the DVI file) found in the TFM file for this font; c should match the check sum of the font found by programs that read this DVI file.

Parameter s contains a fixed-point scale factor that is applied to the character widths in font k; font dimensions in TFM files and other font files are relative to this quantity, which is always positive and less than 2^{27}. It is given in the same units as the other dimensions of the DVI file. Parameter d is similar to s; it is the "design size," and (like s) it is given in DVI units. Thus, font k is to be used at $mag \cdot s/1000d$ times its normal size.

The remaining part of a font definition gives the external name of the font, which is an ASCII string of length $a + l$. The number a is the length of the "area" or directory, and l is the length of the font name itself; the standard local system font area is supposed to be used when $a = 0$. The n field contains the area in its first a bytes.

Font definitions must appear before the first use of a particular font number. Once font k is defined, it must not be defined again; however, we shall see below that font definitions appear in the postamble as well as in the pages, so in this sense each font number is defined exactly twice, if at all. Like *nop* commands, font definitions can appear before the first *bop*, or between an *eop* and a *bop*.

19. The last page in a DVI file is followed by '*post*'; this command introduces the postamble, which summarizes important facts that TEX has accumulated about the file, making it possible to print subsets of the data with reasonable efficiency. The postamble has the form

post $p[4]$ *num*$[4]$ *den*$[4]$ *mag*$[4]$ $l[4]$ $u[4]$ $s[2]$ $t[2]$
⟨ font definitions ⟩
post_post $q[4]$ $i[1]$ 223's$[\geq 4]$

Here p is a pointer to the final *bop* in the file. The next three parameters, *num*, *den*, and *mag*, are duplicates of the quantities that appeared in the preamble.

Parameters l and u give respectively the height-plus-depth of the tallest page and the width of the widest page, in the same units as other dimensions of the file. These numbers might be used by a DVI-reading program to position individual "pages" on large sheets of film or paper; however, the standard convention for output on normal size paper is to position each page so that the upper left-hand corner is exactly one inch from the left and the top. Experience has shown that it is unwise to design DVI-to-printer software that attempts cleverly to center the output; a fixed position of the upper left corner is easiest for users to understand and to work with. Therefore l and u are often ignored.

Parameter s is the maximum stack depth (i.e., the largest excess of *push* commands over *pop* commands) needed to process this file. Then comes t, the total number of pages (*bop* commands) present.

The postamble continues with font definitions, which are any number of *fnt_def* commands as described above, possibly interspersed with *nop* commands. Each font number that is used in the DVI file must be defined exactly twice: Once before it is first selected by a *fnt* command, and once in the postamble.

20. The last part of the postamble, following the *post_post* byte that signifies the end of the font definitions, contains q, a pointer to the *post* command that started the postamble. An identification byte, i, comes next; this currently equals 2, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., *´337* in octal). TEX puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a `DVI` file makes it feasible for `DVI`-reading programs to find the postamble first, on most computers, even though TEX wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the `DVI` reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 (*post*); now the postamble can be read, so the `DVI` reader discovers all the information needed for typesetting the pages. Note that it is also possible to skip through the `DVI` file at reasonably high speed to locate a particular page, if that proves desirable. This saves a lot of time, since `DVI` files used in production jobs tend to be large.

Unfortunately, however, standard Pascal does not include the ability to access a random position in a file, or even to determine the length of a file. Almost all systems nowadays provide the necessary capabilities, so `DVI` format has been designed to work most efficiently with modern operating systems. As noted above, `DVItype` will limit itself to the restrictions of standard Pascal if *random_reading* is defined to be *false*.

21. Input from binary files. We have seen that a `DVI` file is a sequence of 8-bit bytes. The bytes appear physically in what is called a '**packed file of** 0 .. 255' in Pascal lingo.

Packing is system dependent, and many Pascal systems fail to implement such files in a sensible way (at least, from the viewpoint of producing good production software). For example, some systems treat all byte-oriented files as text, looking for end-of-line marks and such things. Therefore some system-dependent code is often needed to deal with binary files, even though most of the program in this section of `DVItype` is written in standard Pascal.

One common way to solve the problem is to consider files of *integer* numbers, and to convert an integer in the range $-2^{31} \le x < 2^{31}$ to a sequence of four bytes (a, b, c, d) using the following code, which avoids the controversial integer division of negative numbers:

```
if x ≥ 0 then a ← x div ´100000000
else begin x ← (x + ´10000000000) + ´10000000000; a ← x div ´100000000 + 128;
  end
x ← x mod ´100000000;
b ← x div ´200000; x ← x mod ´200000;
c ← x div ´400; d ← x mod ´400;
```

The four bytes are then kept in a buffer and output one by one. (On 36-bit computers, an additional division by 16 is necessary at the beginning. Another way to separate an integer into four bytes is to use/abuse Pascal's variant records, storing an integer and retrieving bytes that are packed in the same place; *caveat implementor!*) It is also desirable in some cases to read a hundred or so integers at a time, maintaining a larger buffer.

We shall stick to simple Pascal in this program, for reasons of clarity, even if such simplicity is sometimes unrealistic.

```
⟨ Types in the outer block 8 ⟩ +≡
  eight_bits = 0 .. 255;  { unsigned one-byte quantity }
  byte_file = packed file of eight_bits;  { files that contain binary data }
```

22. The program deals with two binary file variables: *dvi_file* is the main input file that we are translating into symbolic form, and *tfm_file* is the current font metric file from which character-width information is being read.

```
⟨ Globals in the outer block 10 ⟩ +≡
dvi_file: byte_file;  { the stuff we are DVItyping }
tfm_file: byte_file;  { a font metric file }
```

23. To prepare these files for input, we *reset* them. An extension of Pascal is needed in the case of *tfm_file*, since we want to associate it with external files whose names are specified dynamically (i.e., not known at compile time). The following code assumes that '*reset*(f, s)' does this, when f is a file variable and s is a string variable that specifies the file name. If *eof*(f) is true immediately after *reset*(f, s) has acted, we assume that no file named s is accessible.

```
procedure open_dvi_file;  { prepares to read packed bytes in dvi_file }
  begin reset(dvi_file); cur_loc ← 0;
  end;
procedure open_tfm_file;  { prepares to read packed bytes in tfm_file }
  begin reset(tfm_file, cur_name);
  end;
```

24. If you looked carefully at the preceding code, you probably asked, "What are *cur_loc* and *cur_name*?" Good question. They're global variables: *cur_loc* is the number of the byte about to be read next from *dvi_file*, and *cur_name* is a string variable that will be set to the current font metric file name before *open_tfm_file* is called.

⟨ Globals in the outer block 10 ⟩ +≡
cur_loc: *integer*; { where we are about to look, in *dvi_file* }
cur_name: **packed array** [1 .. *name_length*] **of** *char*; { external name, with no lower case letters }

25. It turns out to be convenient to read four bytes at a time, when we are inputting from TFM files. The input goes into global variables *b0*, *b1*, *b2*, and *b3*, with *b0* getting the first byte and *b3* the fourth.

⟨ Globals in the outer block 10 ⟩ +≡
b0, *b1*, *b2*, *b3*: *eight_bits*; { four bytes input at once }

26. The *read_tfm_word* procedure sets *b0* through *b3* to the next four bytes in the current TFM file.

procedure *read_tfm_word*;
begin *read*(*tfm_file*, *b0*); *read*(*tfm_file*, *b1*); *read*(*tfm_file*, *b2*); *read*(*tfm_file*, *b3*);
end;

27. We shall use another set of simple functions to read the next byte or bytes from *dvi_file*. There are seven possibilities, each of which is treated as a separate function in order to minimize the overhead for subroutine calls.

```
function get_byte: integer; { returns the next byte, unsigned }
  var b: eight_bits;
  begin if eof(dvi_file) then get_byte ← 0
  else begin read(dvi_file, b); incr(cur_loc); get_byte ← b;
    end;
  end;
function signed_byte: integer; { returns the next byte, signed }
  var b: eight_bits;
  begin read(dvi_file, b); incr(cur_loc);
  if b < 128 then signed_byte ← b else signed_byte ← b − 256;
  end;
function get_two_bytes: integer; { returns the next two bytes, unsigned }
  var a, b: eight_bits;
  begin read(dvi_file, a); read(dvi_file, b); cur_loc ← cur_loc + 2; get_two_bytes ← a ∗ 256 + b;
  end;
function signed_pair: integer; { returns the next two bytes, signed }
  var a, b: eight_bits;
  begin read(dvi_file, a); read(dvi_file, b); cur_loc ← cur_loc + 2;
  if a < 128 then signed_pair ← a ∗ 256 + b
  else signed_pair ← (a − 256) ∗ 256 + b;
  end;
function get_three_bytes: integer; { returns the next three bytes, unsigned }
  var a, b, c: eight_bits;
  begin read(dvi_file, a); read(dvi_file, b); read(dvi_file, c); cur_loc ← cur_loc + 3;
  get_three_bytes ← (a ∗ 256 + b) ∗ 256 + c;
  end;
function signed_trio: integer; { returns the next three bytes, signed }
  var a, b, c: eight_bits;
  begin read(dvi_file, a); read(dvi_file, b); read(dvi_file, c); cur_loc ← cur_loc + 3;
  if a < 128 then signed_trio ← (a ∗ 256 + b) ∗ 256 + c
  else signed_trio ← ((a − 256) ∗ 256 + b) ∗ 256 + c;
  end;
function signed_quad: integer; { returns the next four bytes, signed }
  var a, b, c, d: eight_bits;
  begin read(dvi_file, a); read(dvi_file, b); read(dvi_file, c); read(dvi_file, d); cur_loc ← cur_loc + 4;
  if a < 128 then signed_quad ← ((a ∗ 256 + b) ∗ 256 + c) ∗ 256 + d
  else signed_quad ← (((a − 256) ∗ 256 + b) ∗ 256 + c) ∗ 256 + d;
  end;
```

28. Finally we come to the routines that are used only if *random_reading* is *true*. The driver program below needs two such routines: *dvi_length* should compute the total number of bytes in *dvi_file*, possibly also causing *eof*(*dvi_file*) to be true; and *move_to_byte*(n) should position *dvi_file* so that the next *get_byte* will read byte n, starting with $n = 0$ for the first byte in the file.

Such routines are, of course, highly system dependent. They are implemented here in terms of two assumed system routines called *set_pos* and *cur_pos*. The call *set_pos*(f, n) moves to item n in file f, unless n is negative or larger than the total number of items in f; in the latter case, *set_pos*(f, n) moves to the end of file f. The call *cur_pos*(f) gives the total number of items in f, if *eof*(f) is true; we use *cur_pos* only in such a situation.

```
function dvi_length: integer;
  begin set_pos(dvi_file, -1); dvi_length ← cur_pos(dvi_file);
  end;
procedure move_to_byte(n : integer);
  begin set_pos(dvi_file, n); cur_loc ← n;
  end;
```

29. Reading the font information. DVI file format does not include information about character widths, since that would tend to make the files a lot longer. But a program that reads a DVI file is supposed to know the widths of the characters that appear in *set_char* commands. Therefore DVItype looks at the font metric (TFM) files for the fonts that are involved.

The character-width data appears also in other files (e.g., in GF files that specify bit patterns for digitized characters); thus, it is usually possible for DVI reading programs to get by with accessing only one file per font. DVItype has a comparatively easy task in this regard, since it needs only a few words of information from each font; other DVI-to-printer programs may have to go to some pains to deal with complications that arise when a large number of large font files all need to be accessed simultaneously.

30. For purposes of this program, we need to know only two things about a given character c in a given font f: (1) Is c a legal character in f? (2) If so, what is the width of c? We also need to know the symbolic name of each font, so it can be printed out, and we need to know the approximate size of inter-word spaces in each font.

The answers to these questions appear implicitly in the following data structures. The current number of known fonts is nf. Each known font has an internal number f, where $0 \le f < nf$; the external number of this font, i.e., its font identification number in the DVI file, is $font\_num[f]$, and the external name of this font is the string that occupies positions $font\_name[f]$ through $font\_name[f+1]-1$ of the array *names*. The latter array consists of *ASCII_code* characters, and $font\_name[nf]$ is its first unoccupied position. A horizontal motion in the range $-4 * font\_space[f] < h < font\_space[f]$ will be treated as a 'kern' that is not indicated in the printouts that DVItype produces between brackets. The legal characters run from $font\_bc[f]$ to $font\_ec[f]$, inclusive; more precisely, a given character c is valid in font f if and only if $font\_bc[f] \le c \le font\_ec[f]$ and $char\_width(f)(c) \neq invalid\_width$. Finally, $char\_width(f)(c) = width[width\_base[f]+c]$, and *width_ptr* is the first unused position of the *width* array.

```
define char_width_end(#) ≡ # ]
define char_width(#) ≡ width [ width_base[#] + char_width_end
define invalid_width ≡ '17777777777
define invalid_font ≡ max_fonts
```

⟨ Globals in the outer block 10 ⟩ +≡
```
font_num: array [0 .. max_fonts] of integer;   { external font numbers }
font_name: array [0 .. max_fonts] of 1 .. name_size;   { starting positions of external font names }
names: array [1 .. name_size] of ASCII_code;   { characters of names }
font_check_sum: array [0 .. max_fonts] of integer;   { check sums }
font_scaled_size: array [0 .. max_fonts] of integer;   { scale factors }
font_design_size: array [0 .. max_fonts] of integer;   { design sizes }
font_space: array [0 .. max_fonts] of integer;   { boundary between "small" and "large" spaces }
font_bc: array [0 .. max_fonts] of integer;   { beginning characters in fonts }
font_ec: array [0 .. max_fonts] of integer;   { ending characters in fonts }
width_base: array [0 .. max_fonts] of integer;   { index into width table }
width: array [0 .. max_widths] of integer;   { character widths, in DVI units }
nf: 0 .. max_fonts;   { the number of known fonts }
width_ptr: 0 .. max_widths;   { the number of known character widths }
```

31. ⟨ Set initial values 11 ⟩ +≡
```
nf ← 0; width_ptr ← 0; font_name[0] ← 1;
font_space[invalid_font] ← 0;   { for out_space and out_vmove }
font_bc[invalid_font] ← 1; font_ec[invalid_font] ← 0;
```

32. It is, of course, a simple matter to print the name of a given font.

```
procedure print_font(f : integer); { f is an internal font number }
  var k: 0 .. name_size; { index into names }
  begin if f = invalid_font then print('UNDEFINED!')
  else begin for k ← font_name[f] to font_name[f + 1] − 1 do print(xchr[names[k]]);
    end;
  end;
```

33. An auxiliary array *in_width* is used to hold the widths as they are input. The global variables *tfm_check_sum* and *tfm_design_size* are set to the check sum and design size that appear in the current TFM file.

```
⟨ Globals in the outer block 10 ⟩ +≡
in_width: array [0 .. 255] of integer; { TFM width data in DVI units }
tfm_check_sum: integer; { check sum found in tfm_file }
tfm_design_size: integer; { design size found in tfm_file, in DVI units }
tfm_conv: real; { DVI units per absolute TFM unit }
```

34. Here is a procedure that absorbs the necessary information from a TFM file, assuming that the file has just been successfully reset so that we are ready to read its first byte. (A complete description of TFM file format appears in the documentation of TFtoPL and will not be repeated here.) The procedure does not check the TFM file for validity, nor does it give explicit information about what is wrong with a TFM file that proves to be invalid; DVI-reading programs need not do this, since TFM files are almost always valid, and since the TFtoPL utility program has been specifically designed to diagnose TFM errors. The procedure simply returns *false* if it detects anything amiss in the TFM data.

There is a parameter, z, which represents the scaling factor being used to compute the font dimensions; it must be in the range $0 < z < 2^{27}$.

```
function in_TFM(z : integer): boolean; { input TFM data or return false }
  label 9997, { go here when the format is bad }
    9998, { go here when the information cannot be loaded }
    9999; { go here to exit }
  var k: integer; { index for loops }
    lh: integer; { length of the header data, in four-byte words }
    nw: integer; { number of words in the width table }
    wp: 0 .. max_widths; { new value of width_ptr after successful input }
    alpha, beta: integer; { quantities used in the scaling computation }
  begin ⟨ Read past the header data; goto 9997 if there is a problem 35 ⟩;
  ⟨ Store character-width indices at the end of the width table 36 ⟩;
  ⟨ Read and convert the width values, setting up the in_width table 37 ⟩;
  ⟨ Move the widths from in_width to width, and append pixel_width values 40 ⟩;
  width_ptr ← wp; in_TFM ← true; goto 9999;
9997: print_ln('---not␣loaded,␣TFM␣file␣is␣bad');
9998: in_TFM ← false;
9999: end;
```

35. ⟨ Read past the header data; **goto** 9997 if there is a problem 35 ⟩ ≡

```
read_tfm_word; lh ← b2 * 256 + b3; read_tfm_word; font_bc[nf] ← b0 * 256 + b1;
font_ec[nf] ← b2 * 256 + b3;
if font_ec[nf] < font_bc[nf] then font_bc[nf] ← font_ec[nf] + 1;
if width_ptr + font_ec[nf] − font_bc[nf] + 1 > max_widths then
  begin print_ln('---not␣loaded,␣DVItype␣needs␣larger␣width␣table'); goto 9998;
  end;
wp ← width_ptr + font_ec[nf] − font_bc[nf] + 1; read_tfm_word; nw ← b0 * 256 + b1;
if (nw = 0) ∨ (nw > 256) then goto 9997;
for k ← 1 to 3 + lh do
  begin if eof(tfm_file) then goto 9997;
  read_tfm_word;
  if k = 4 then
    if b0 < 128 then tfm_check_sum ← ((b0 * 256 + b1) * 256 + b2) * 256 + b3
    else tfm_check_sum ← (((b0 − 256) * 256 + b1) * 256 + b2) * 256 + b3
  else if k = 5 then
      if b0 < 128 then tfm_design_size ← round(tfm_conv * (((b0 * 256 + b1) * 256 + b2) * 256 + b3))
      else goto 9997;
  end;
```

This code is used in section 34.

36. ⟨ Store character-width indices at the end of the *width* table 36 ⟩ ≡

```
if wp > 0 then
  for k ← width_ptr to wp − 1 do
    begin read_tfm_word;
    if b0 > nw then goto 9997;
    width[k] ← b0;
    end;
```

This code is used in section 34.

37. The most important part of *in_TFM* is the width computation, which involves multiplying the relative widths in the `TFM` file by the scaling factor in the `DVI` file. This fixed-point multiplication must be done with precisely the same accuracy by all `DVI`-reading programs, in order to validate the assumptions made by `DVI`-writing programs like TEX82.

Let us therefore summarize what needs to be done. Each width in a `TFM` file appears as a four-byte quantity called a *fix_word*. A *fix_word* whose respective bytes are (a, b, c, d) represents the number

$$x = \begin{cases} b \cdot 2^{-4} + c \cdot 2^{-12} + d \cdot 2^{-20}, & \text{if } a = 0; \\ -16 + b \cdot 2^{-4} + c \cdot 2^{-12} + d \cdot 2^{-20}, & \text{if } a = 255. \end{cases}$$

(No other choices of a are allowed, since the magnitude of a `TFM` dimension must be less than 16.) We want to multiply this quantity by the integer z, which is known to be less than 2^{27}. If $z < 2^{23}$, the individual multiplications $b \cdot z$, $c \cdot z$, $d \cdot z$ cannot overflow; otherwise we will divide z by 2, 4, 8, or 16, to obtain a multiplier less than 2^{23}, and we can compensate for this later. If z has thereby been replaced by $z' = z/2^e$, let $\beta = 2^{4-e}$; we shall compute

$$\lfloor (b + c \cdot 2^{-8} + d \cdot 2^{-16})\, z'/\beta \rfloor$$

if $a = 0$, or the same quantity minus $\alpha = 2^{4+e} z'$ if $a = 255$. This calculation must be done exactly, for the reasons stated above; the following program does the job in a system-independent way, assuming that arithmetic is exact on numbers less than 2^{31} in magnitude.

```
⟨ Read and convert the width values, setting up the in_width table 37 ⟩ ≡
  ⟨ Replace z by z′ and compute α, β 38 ⟩;
  for k ← 0 to nw − 1 do
    begin read_tfm_word; in_width[k] ← (((((b3 ∗ z) div ´400) + (b2 ∗ z)) div ´400) + (b1 ∗ z)) div beta;
    if b0 > 0 then
      if b0 < 255 then goto 9997
      else in_width[k] ← in_width[k] − alpha;
    end
```

This code is used in section 34.

```
38. ⟨ Replace z by z′ and compute α, β 38 ⟩ ≡
  begin alpha ← 16;
  while z ≥ ´40000000 do
    begin z ← z div 2; alpha ← alpha + alpha;
    end;
  beta ← 256 div alpha; alpha ← alpha ∗ z;
  end
```

This code is used in section 37.

39. A DVI-reading program usually works with font files instead of TFM files, so DVItype is atypical in that respect. Font files should, however, contain exactly the same character width data that is found in the corresponding TFMs; check sums are used to help ensure this. In addition, font files usually also contain the widths of characters in pixels, since the device-independent character widths of TFM files are generally not perfect multiples of pixels.

The *pixel_width* array contains this information; when *width*[*k*] is the device-independent width of some character in DVI units, *pixel_width*[*k*] is the corresponding width of that character in an actual font. The macro *char_pixel_width* is set up to be analogous to *char_width*.

```
define char_pixel_width(#) ≡ pixel_width [ width_base[#] + char_width_end
```

```
⟨ Globals in the outer block 10 ⟩ +≡
pixel_width: array [0 .. max_widths] of integer;  { actual character widths, in pixels }
conv: real;  { converts DVI units to pixels }
true_conv: real;  { converts unmagnified DVI units to pixels }
numerator, denominator: integer;  { stated conversion ratio }
mag: integer;  { magnification factor times 1000 }
```

40. The following code computes pixel widths by simply rounding the TFM widths to the nearest integer number of pixels, based on the conversion factor *conv* that converts DVI units to pixels. However, such a simple formula will not be valid for all fonts, and it will often give results that are off by ±1 when a low-resolution font has been carefully hand-fitted. For example, a font designer often wants to make the letter 'm' a pixel wider or narrower in order to make the font appear more consistent. DVI-to-printer programs should therefore input the correct pixel width information from font files whenever there is a chance that it may differ. A warning message may also be desirable in the case that at least one character is found whose pixel width differs from *conv* ∗ *width* by more than a full pixel.

```
define pixel_round(#) ≡ round(conv ∗ (#))
```

```
⟨ Move the widths from in_width to width, and append pixel_width values 40 ⟩ ≡
  if in_width[0] ≠ 0 then goto 9997;  { the first width should be zero }
  width_base[nf] ← width_ptr − font_bc[nf];
  if wp > 0 then
    for k ← width_ptr to wp − 1 do
      if width[k] = 0 then
        begin width[k] ← invalid_width; pixel_width[k] ← 0;
        end
      else begin width[k] ← in_width[width[k]]; pixel_width[k] ← pixel_round(width[k]);
        end
```

This code is used in section 34.

41. Optional modes of output. `DVItype` will print different quantities of information based on some options that the user must specify: The *out_mode* level is set to one of five values (*errors_only*, *terse*, *mnemonics_only*, *verbose*, *the_works*), giving different degrees of output; and the typeout can be confined to a restricted subset of the pages by specifying the desired starting page and the maximum number of pages. Furthermore there is an option to specify the resolution of an assumed discrete output device, so that pixel-oriented calculations will be shown; and there is an option to override the magnification factor that is stated in the `DVI` file.

The starting page is specified by giving a sequence of 1 to 10 numbers or asterisks separated by dots. For example, the specification '`1.*.-5`' can be used to refer to a page output by TEX when `\count0` = 1 and `\count2` = −5. (Recall that *bop* commands in a `DVI` file are followed by ten 'count' values.) An asterisk matches any number, so the '`*`' in '`1.*.-5`' means that `\count1` is ignored when specifying the first page. If several pages match the given specification, `DVItype` will begin with the earliest such page in the file. The default specification '`*`' (which matches all pages) therefore denotes the page at the beginning of the file.

When `DVItype` begins, it engages the user in a brief dialog so that the options will be specified. This part of `DVItype` requires nonstandard Pascal constructions to handle the online interaction; so it may be preferable in some cases to omit the dialog and simply to stick to the default options (*out_mode* = *the_works*, starting page '`*`', *max_pages* = 1000000, *resolution* = 300.0, *new_mag* = 0). On other hand, the system-dependent routines that are needed are not complicated, so it will not be terribly difficult to introduce them.

```
define errors_only = 0  { value of out_mode when minimal printing occurs }
define terse = 1  { value of out_mode for abbreviated output }
define mnemonics_only = 2  { value of out_mode for medium-quantity output }
define verbose = 3  { value of out_mode for detailed tracing }
define the_works = 4  { verbose, plus check of postamble if random_reading }
```

```
⟨ Globals in the outer block 10 ⟩ +≡
out_mode: errors_only .. the_works;  { controls the amount of output }
max_pages: integer;  { at most this many bop .. eop pages will be printed }
resolution: real;  { pixels per inch }
new_mag: integer;  { if positive, overrides the postamble's magnification }
```

42. The starting page specification is recorded in two global arrays called *start_count* and *start_there*. For example, '`1.*.-5`' is represented by *start_there*[0] = *true*, *start_count*[0] = 1, *start_there*[1] = *false*, *start_there*[2] = *true*, *start_count*[2] = −5. We also set *start_vals* = 2, to indicate that count 2 was the last one mentioned. The other values of *start_count* and *start_there* are not important, in this example.

```
⟨ Globals in the outer block 10 ⟩ +≡
start_count: array [0 .. 9] of integer;  { count values to select starting page }
start_there: array [0 .. 9] of boolean;  { is the start_count value relevant? }
start_vals: 0 .. 9;  { the last count considered significant }
count: array [0 .. 9] of integer;  { the count values on the current page }
```

43.
```
⟨ Set initial values 11 ⟩ +≡
  out_mode ← the_works; max_pages ← 1000000; start_vals ← 0; start_there[0] ← false;
```

44. Here is a simple subroutine that tests if the current page might be the starting page.

```
function start_match: boolean;  { does count match the starting spec? }
  var k: 0 .. 9;  { loop index }
    match: boolean;  { does everything match so far? }
  begin match ← true;
  for k ← 0 to start_vals do
    if start_there[k] ∧ (start_count[k] ≠ count[k]) then match ← false;
  start_match ← match;
  end;
```

45. The *input_ln* routine waits for the user to type a line at his or her terminal; then it puts ASCII-code equivalents for the characters on that line into the *buffer* array. The *term_in* file is used for terminal input, and *term_out* for terminal output.

```
⟨ Globals in the outer block 10 ⟩ +≡
buffer: array [0 .. terminal_line_length] of ASCII_code;
term_in: text_file;   { the terminal, considered as an input file }
term_out: text_file;   { the terminal, considered as an output file }
```

46. Since the terminal is being used for both input and output, some systems need a special routine to make sure that the user can see a prompt message before waiting for input based on that message. (Otherwise the message may just be sitting in a hidden buffer somewhere, and the user will have no idea what the program is waiting for.) We shall invoke a system-dependent subroutine *update_terminal* in order to avoid this problem.

```
define update_terminal ≡ break(term_out)   { empty the terminal output buffer }
```

47. During the dialog, `DVItype` will treat the first blank space in a line as the end of that line. Therefore *input_ln* makes sure that there is always at least one blank space in *buffer*.

```
procedure input_ln;   { inputs a line from the terminal }
  var k: 0 .. terminal_line_length;
  begin update_terminal; reset(term_in);
  if eoln(term_in) then read_ln(term_in);
  k ← 0;
  while (k < terminal_line_length) ∧ ¬eoln(term_in) do
    begin buffer[k] ← xord[term_in↑]; incr(k); get(term_in);
    end;
  buffer[k] ← "␣";
  end;
```

48. The global variable *buf_ptr* is used while scanning each line of input; it points to the first unread character in *buffer*.

```
⟨ Globals in the outer block 10 ⟩ +≡
buf_ptr: 0 .. terminal_line_length;   { the number of characters read }
```

49. Here is a routine that scans a (possibly signed) integer and computes the decimal value. If no decimal integer starts at *buf_ptr*, the value 0 is returned. The integer should be less than 2^{31} in absolute value.

```
function get_integer: integer;
  var x: integer;   { accumulates the value }
    negative: boolean;   { should the value be negated? }
  begin if buffer[buf_ptr] = "-" then
    begin negative ← true; incr(buf_ptr);
    end
  else negative ← false;
  x ← 0;
  while (buffer[buf_ptr] ≥ "0") ∧ (buffer[buf_ptr] ≤ "9") do
    begin x ← 10 * x + buffer[buf_ptr] − "0"; incr(buf_ptr);
    end;
  if negative then get_integer ← −x else get_integer ← x;
  end;
```

50. The selected options are put into global variables by the *dialog* procedure, which is called just as `DVItype` begins.

```
procedure dialog;
  label 1,2,3,4,5;
  var k: integer;  { loop variable }
  begin rewrite(term_out);  { prepare the terminal for output }
  write_ln(term_out,banner); ⟨ Determine the desired out_mode 51 ⟩;
  ⟨ Determine the desired start_count values 52 ⟩;
  ⟨ Determine the desired max_pages 53 ⟩;
  ⟨ Determine the desired resolution 54 ⟩;
  ⟨ Determine the desired new_mag 55 ⟩;
  ⟨ Print all the selected options 56 ⟩;
  end;
```

51. ⟨ Determine the desired *out_mode* 51 ⟩ ≡

```
1: write(term_out,'Output level (default=4, ? for help): ');
   out_mode ← the_works; input_ln;
  if buffer[0] ≠ " " then
    if (buffer[0] ≥ "0") ∧ (buffer[0] ≤ "4") then out_mode ← buffer[0] − "0"
    else begin write(term_out,'Type 4 for complete listing,');
      write(term_out,' 0 for errors and fonts only,');
      write_ln(term_out,' 1 or 2 or 3 for something in between.'); goto 1;
      end
```

This code is used in section 50.

52. ⟨ Determine the desired *start_count* values 52 ⟩ ≡

```
2: write(term_out,'Starting page (default=*): ');
   start_vals ← 0; start_there[0] ← false; input_ln;
  buf_ptr ← 0; k ← 0;
  if buffer[0] ≠ " " then
    repeat if buffer[buf_ptr] = "*" then
        begin start_there[k] ← false; incr(buf_ptr);
        end
      else begin start_there[k] ← true; start_count[k] ← get_integer;
        end;
      if (k < 9) ∧ (buffer[buf_ptr] = ".") then
        begin incr(k); incr(buf_ptr);
        end
      else if buffer[buf_ptr] = " " then start_vals ← k
        else begin write(term_out,'Type, e.g., 1.*.-5 to specify the ');
          write_ln(term_out,'first page with \count0=1, \count2=-5.');
          goto 2;
          end;
    until start_vals = k
```

This code is used in section 50.

53. ⟨ Determine the desired *max_pages* 53 ⟩ ≡

```
3: write(term_out, ´Maximum␣number␣of␣pages␣(default=1000000):␣´); max_pages ← 1000000;
   input_ln; buf_ptr ← 0;
   if buffer[0] ≠ "␣" then
     begin max_pages ← get_integer;
     if max_pages ≤ 0 then
       begin write_ln(term_out, ´Please␣type␣a␣positive␣number.´); goto 3;
       end;
     end
```

This code is used in section 50.

54. ⟨ Determine the desired *resolution* 54 ⟩ ≡

```
4: write(term_out, ´Assumed␣device␣resolution´);
   write(term_out, ´␣in␣pixels␣per␣inch␣(default=300/1):␣´); resolution ← 300.0; input_ln;
   buf_ptr ← 0;
   if buffer[0] ≠ "␣" then
     begin k ← get_integer;
     if (k > 0) ∧ (buffer[buf_ptr] = "/") ∧ (buffer[buf_ptr + 1] > "0") ∧ (buffer[buf_ptr + 1] ≤ "9") then
       begin incr(buf_ptr); resolution ← k/get_integer;
       end
     else begin write(term_out, ´Type␣a␣ratio␣of␣positive␣integers;´);
       write_ln(term_out, ´␣(1␣pixel␣per␣mm␣would␣be␣254/10).´); goto 4;
       end;
     end
```

This code is used in section 50.

55. ⟨ Determine the desired *new_mag* 55 ⟩ ≡

```
5: write(term_out, ´New␣magnification␣(default=0␣to␣keep␣the␣old␣one):␣´); new_mag ← 0;
   input_ln; buf_ptr ← 0;
   if buffer[0] ≠ "␣" then
     if (buffer[0] ≥ "0") ∧ (buffer[0] ≤ "9") then new_mag ← get_integer
     else begin write(term_out, ´Type␣a␣positive␣integer␣to␣override␣´);
       write_ln(term_out, ´the␣magnification␣in␣the␣DVI␣file.´); goto 5;
       end
```

This code is used in section 50.

56. After the dialog is over, we print the options so that the user can see what `DVItype` thought was specified.

```
⟨ Print all the selected options 56 ⟩ ≡
  print_ln('Options␣selected:'); print('␣␣Starting␣page␣=␣');
  for k ← 0 to start_vals do
    begin if start_there[k] then print(start_count[k] : 1)
    else print('*');
    if k < start_vals then print('.')
    else print_ln('␣');
    end;
  print_ln('␣␣Maximum␣number␣of␣pages␣=␣', max_pages : 1);
  print('␣␣Output␣level␣=␣', out_mode : 1);
  case out_mode of
  errors_only: print_ln('␣(showing␣bops,␣fonts,␣and␣error␣messages␣only)');
  terse: print_ln('␣(terse)');
  mnemonics_only: print_ln('␣(mnemonics)');
  verbose: print_ln('␣(verbose)');
  the_works: if random_reading then print_ln('␣(the␣works)')
    else begin out_mode ← verbose; print_ln('␣(the␣works:␣same␣as␣level␣3␣in␣this␣DVItype)');
      end;
  end;
  print_ln('␣␣Resolution␣=␣', resolution : 12 : 8, '␣pixels␣per␣inch');
  if new_mag > 0 then print_ln('␣␣New␣magnification␣factor␣=␣', new_mag/1000 : 8 : 3)
```

This code is used in section 50.

57. Defining fonts. When *out_mode* = *the_works*, `DVItype` reads the postamble first and loads all of the fonts defined there; then it processes the pages. In this case, a *fnt_def* command should match a previous definition if and only if the *fnt_def* being processed is not in the postamble. But if *out_mode* < *the_works*, `DVItype` reads the pages first and the postamble last, so the conventions are reversed: a *fnt_def* should match a previous *fnt_def* if and only if the current one is a part of the postamble.

A global variable *in_postamble* is provided to tell whether we are processing the postamble or not.

```
⟨ Globals in the outer block 10 ⟩ +≡
in_postamble: boolean;  { are we reading the postamble? }
```

58.
```
⟨ Set initial values 11 ⟩ +≡
  in_postamble ← false;
```

59. The following subroutine does the necessary things when a *fnt_def* command is being processed.

```
procedure define_font(e : integer);  { e is an external font number }
  var f: 0 .. max_fonts; p: integer;  { length of the area/directory spec }
    n: integer;  { length of the font name proper }
    c, q, d, m: integer;  { check sum, scaled size, design size, magnification }
    r: 0 .. name_length;  { index into cur_name }
    j, k: 0 .. name_size;  { indices into names }
    mismatch: boolean;  { do names disagree? }
  begin if nf = max_fonts then
    abort('DVItype␣capacity␣exceeded␣(max␣fonts=', max_fonts : 1, ')!');
  font_num[nf] ← e; f ← 0;
  while font_num[f] ≠ e do incr(f);
  ⟨ Read the font parameters into position for font nf, and print the font name 61 ⟩;
  if ((out_mode = the_works) ∧ in_postamble) ∨ ((out_mode < the_works) ∧ ¬in_postamble) then
    begin if f < nf then print_ln('---this␣font␣was␣already␣defined!');
    end
  else begin if f = nf then print_ln('---this␣font␣wasn''t␣loaded␣before!');
    end;
  if f = nf then ⟨ Load the new font, unless there are problems 62 ⟩
  else ⟨ Check that the current font definition matches the old one 60 ⟩;
  end;
```

60.
```
⟨ Check that the current font definition matches the old one 60 ⟩ ≡
  begin if font_check_sum[f] ≠ c then
    print_ln('---check␣sum␣doesn''t␣match␣previous␣definition!');
  if font_scaled_size[f] ≠ q then print_ln('---scaled␣size␣doesn''t␣match␣previous␣definition!');
  if font_design_size[f] ≠ d then print_ln('---design␣size␣doesn''t␣match␣previous␣definition!');
  j ← font_name[f]; k ← font_name[nf];
  if font_name[f + 1] − j ≠ font_name[nf + 1] − k then mismatch ← true
  else begin mismatch ← false;
    while j < font_name[f + 1] do
      begin if names[j] ≠ names[k] then mismatch ← true;
      incr(j); incr(k);
      end;
    end;
  if mismatch then print_ln('---font␣name␣doesn''t␣match␣previous␣definition!');
  end
```

This code is used in section 59.

61. ⟨ Read the font parameters into position for font *nf*, and print the font name 61 ⟩ ≡

```
c ← signed_quad; font_check_sum[nf] ← c;
q ← signed_quad; font_scaled_size[nf] ← q;
d ← signed_quad; font_design_size[nf] ← d;
if (q ≤ 0) ∨ (d ≤ 0) then m ← 1000
else m ← round((1000.0 * conv * q)/(true_conv * d));
p ← get_byte; n ← get_byte;
if font_name[nf] + n + p > name_size then
  abort('DVItype␣capacity␣exceeded␣(name␣size=', name_size : 1, ')!');
font_name[nf + 1] ← font_name[nf] + n + p;
if showing then print(':␣') { when showing is true, the font number has already been printed }
else print('Font␣', e : 1, ':␣');
if n + p = 0 then print('null␣font␣name!')
else for k ← font_name[nf] to font_name[nf + 1] − 1 do names[k] ← get_byte;
print_font(nf);
if ¬showing then
  if m ≠ 1000 then print('␣scaled␣', m : 1)
```

This code is used in section 59.

62. ⟨ Load the new font, unless there are problems 62 ⟩ ≡

```
begin ⟨ Move font name into the cur_name string 66 ⟩;
open_tfm_file;
if eof(tfm_file) then print('---not␣loaded,␣TFM␣file␣can''t␣be␣opened!')
else begin if (q ≤ 0) ∨ (q ≥ '1000000000') then print('---not␣loaded,␣bad␣scale␣(', q : 1, ')!')
  else if (d ≤ 0) ∨ (d ≥ '1000000000') then print('---not␣loaded,␣bad␣design␣size␣(', d : 1, ')!')
    else if in_TFM(q) then ⟨ Finish loading the new font info 63 ⟩;
  end;
if out_mode = errors_only then print_ln('␣');
end
```

This code is used in section 59.

63. ⟨ Finish loading the new font info 63 ⟩ ≡

```
begin font_space[nf] ← q div 6; { this is a 3-unit "thin space" }
if (c ≠ 0) ∧ (tfm_check_sum ≠ 0) ∧ (c ≠ tfm_check_sum) then
  begin print_ln('---beware:␣check␣sums␣do␣not␣agree!');
  print_ln('␣␣␣(', c : 1, '␣vs.␣', tfm_check_sum : 1, ')'); print('␣␣␣');
  end;
if abs(tfm_design_size − d) > 2 then
  begin print_ln('---beware:␣design␣sizes␣do␣not␣agree!');
  print_ln('␣␣␣(', d : 1, '␣vs.␣', tfm_design_size : 1, ')'); print('␣␣␣');
  end;
print('---loaded␣at␣size␣', q : 1, '␣DVI␣units'); d ← round((100.0 * conv * q)/(true_conv * d));
if d ≠ 100 then
  begin print_ln('␣'); print('␣(this␣font␣is␣magnified␣', d : 1, '%)');
  end;
incr(nf); { now the new font is officially present }
end
```

This code is used in section 62.

64. If $p = 0$, i.e., if no font directory has been specified, `DVItype` is supposed to use the default font directory, which is a system-dependent place where the standard fonts are kept. The string variable *default_directory* contains the name of this area.

```
define default_directory_name ≡ 'TeXfonts:'   { change this to the correct name }
define default_directory_name_length = 9   { change this to the correct length }
⟨ Globals in the outer block 10 ⟩ +≡
default_directory: packed array [1 .. default_directory_name_length] of char;
```

65.
```
⟨ Set initial values 11 ⟩ +≡
  default_directory ← default_directory_name;
```

66. The string *cur_name* is supposed to be set to the external name of the `TFM` file for the current font. This usually means that we need to prepend the name of the default directory, and to append the suffix '`.TFM`'. Furthermore, we change lower case letters to upper case, since *cur_name* is a Pascal string.

```
⟨ Move font name into the cur_name string 66 ⟩ ≡
  for k ← 1 to name_length do cur_name[k] ← ' ';
  if p = 0 then
    begin for k ← 1 to default_directory_name_length do cur_name[k] ← default_directory[k];
    r ← default_directory_name_length;
    end
  else r ← 0;
  for k ← font_name[nf] to font_name[nf + 1] − 1 do
    begin incr(r);
    if r + 4 > name_length then
      abort('DVItype capacity exceeded (max font name length=', name_length : 1, ')!');
    if (names[k] ≥ "a") ∧ (names[k] ≤ "z") then cur_name[r] ← xchr[names[k] − '40]
    else cur_name[r] ← xchr[names[k]];
    end;
  cur_name[r + 1] ← '.'; cur_name[r + 2] ← 'T'; cur_name[r + 3] ← 'F'; cur_name[r + 4] ← 'M'
```

This code is used in section 62.

67. Low level output routines. Simple text in the DVI file is saved in a buffer until *line_length* − 2 characters have accumulated, or until some non-simple DVI operation occurs. Then the accumulated text is printed on a line, surrounded by brackets. The global variable *text_ptr* keeps track of the number of characters currently in the buffer.

```
⟨ Globals in the outer block 10 ⟩ +≡
text_ptr: 0 .. line_length;  { the number of characters in text_buf }
text_buf: array [1 .. line_length] of ASCII_code;  { saved characters }
```

68.
```
⟨ Set initial values 11 ⟩ +≡
  text_ptr ← 0;
```

69. The *flush_text* procedure will empty the buffer if there is something in it.

```
procedure flush_text;
  var k: 0 .. line_length;  { index into text_buf }
  begin if text_ptr > 0 then
    begin if out_mode > errors_only then
      begin print('[');
      for k ← 1 to text_ptr do print(xchr[text_buf[k]]);
      print_ln(']');
      end;
    text_ptr ← 0;
    end;
  end;
```

70. And the *out_text* procedure puts something in it.

```
procedure out_text(c : ASCII_code);
  begin if text_ptr = line_length − 2 then flush_text;
  incr(text_ptr); text_buf[text_ptr] ← c;
  end;
```

71. Translation to symbolic form. The main work of `DVItype` is accomplished by the *do_page* procedure, which produces the output for an entire page, assuming that the *bop* command for that page has already been processed. This procedure is essentially an interpretive routine that reads and acts on the `DVI` commands.

72. The definition of `DVI` files refers to six registers, (h, v, w, x, y, z), which hold integer values in `DVI` units. In practice, we also need registers *hh* and *vv*, the pixel analogs of h and v, since it is not always true that $hh = pixel\_round(h)$ or $vv = pixel\_round(v)$.

The stack of (h, v, w, x, y, z) values is represented by eight arrays called *hstack*, ..., *zstack*, *hhstack*, and *vvstack*.

```
⟨ Globals in the outer block 10 ⟩ +≡
h, v, w, x, y, z, hh, vv: integer;  { current state values }
hstack, vstack, wstack, xstack, ystack, zstack: array [0 .. stack_size] of integer;
        { pushed down values in DVI units }
hhstack, vvstack: array [0 .. stack_size] of integer;  { pushed down values in pixels }
```

73. Three characteristics of the pages (their *max_v*, *max_h*, and *max_s*) are specified in the postamble, and a warning message is printed if these limits are exceeded. Actually *max_v* is set to the maximum height plus depth of a page, and *max_h* to the maximum width, for purposes of page layout. Since characters can legally be set outside of the page boundaries, it is not an error when *max_v* or *max_h* is exceeded. But *max_s* should not be exceeded.

The postamble also specifies the total number of pages; `DVItype` checks to see if this total is accurate.

```
⟨ Globals in the outer block 10 ⟩ +≡
max_v: integer;  { the value of abs(v) should probably not exceed this }
max_h: integer;  { the value of abs(h) should probably not exceed this }
max_s: integer;  { the stack depth should not exceed this }
max_v_so_far, max_h_so_far, max_s_so_far: integer;  { the record high levels }
total_pages: integer;  { the stated total number of pages }
page_count: integer;  { the total number of pages seen so far }
```

74.
```
⟨ Set initial values 11 ⟩ +≡
  max_v ← ´17777777777 − 99; max_h ← ´17777777777 − 99; max_s ← stack_size + 1;
  max_v_so_far ← 0; max_h_so_far ← 0; max_s_so_far ← 0; page_count ← 0;
```

75. Before we get into the details of *do_page*, it is convenient to consider a simpler routine that computes the first parameter of each opcode.

```
define four_cases(#) ≡ #, # + 1, # + 2, # + 3
define eight_cases(#) ≡ four_cases(#), four_cases(# + 4)
define sixteen_cases(#) ≡ eight_cases(#), eight_cases(# + 8)
define thirty_two_cases(#) ≡ sixteen_cases(#), sixteen_cases(# + 16)
define sixty_four_cases(#) ≡ thirty_two_cases(#), thirty_two_cases(# + 32)
function first_par(o : eight_bits): integer;
  begin case o of
  sixty_four_cases(set_char_0), sixty_four_cases(set_char_0 + 64): first_par ← o − set_char_0;
  set1, put1, fnt1, xxx1, fnt_def1: first_par ← get_byte;
  set1 + 1, put1 + 1, fnt1 + 1, xxx1 + 1, fnt_def1 + 1: first_par ← get_two_bytes;
  set1 + 2, put1 + 2, fnt1 + 2, xxx1 + 2, fnt_def1 + 2: first_par ← get_three_bytes;
  right1, w1, x1, down1, y1, z1: first_par ← signed_byte;
  right1 + 1, w1 + 1, x1 + 1, down1 + 1, y1 + 1, z1 + 1: first_par ← signed_pair;
  right1 + 2, w1 + 2, x1 + 2, down1 + 2, y1 + 2, z1 + 2: first_par ← signed_trio;
  set1 + 3, set_rule, put1 + 3, put_rule, right1 + 3, w1 + 3, x1 + 3, down1 + 3, y1 + 3, z1 + 3, fnt1 + 3,
      xxx1 + 3, fnt_def1 + 3: first_par ← signed_quad;
  nop, bop, eop, push, pop, pre, post, post_post, undefined_commands: first_par ← 0;
  w0: first_par ← w;
  x0: first_par ← x;
  y0: first_par ← y;
  z0: first_par ← z;
  sixty_four_cases(fnt_num_0): first_par ← o − fnt_num_0;
  end;
  end;
```

76. Here is another subroutine that we need: It computes the number of pixels in the height or width of a rule. Characters and rules will line up properly if the sizes are computed precisely as specified here. (Since *conv* is computed with some floating-point roundoff error, in a machine-dependent way, format designers who are tailoring something for a particular resolution should not plan their measurements to come out to an exact integer number of pixels; they should compute things so that the rule dimensions are a little less than an integer number of pixels, e.g., 4.99 instead of 5.00.)

```
function rule_pixels(x : integer): integer;  { computes ⌈conv · x⌉ }
  var n: integer;
  begin n ← trunc(conv ∗ x);
  if n < conv ∗ x then rule_pixels ← n + 1 else rule_pixels ← n;
  end;
```

77. Strictly speaking, the *do_page* procedure is really a function with side effects, not a '**procedure**'; it returns the value *false* if `DVItype` should be aborted because of some unusual happening. The subroutine is organized as a typical interpreter, with a multiway branch on the command code followed by **goto** statements leading to routines that finish up the activities common to different commands. We will use the following labels:

```
define fin_set = 41  { label for commands that set or put a character }
define fin_rule = 42  { label for commands that set or put a rule }
define move_right = 43  { label for commands that change h }
define move_down = 44  { label for commands that change v }
define show_state = 45  { label for commands that change s }
define change_font = 46  { label for commands that change cur_font }
```

78. Some Pascal compilers severely restrict the length of procedure bodies, so we shall split *do_page* into two parts, one of which is called *special_cases*. The different parts communicate with each other via the global variables mentioned above, together with the following ones:

```
⟨ Globals in the outer block 10 ⟩ +≡
s: integer;  { current stack size }
ss: integer;  { stack size to print }
cur_font: integer;  { current internal font number }
showing: boolean;  { is the current command being translated in full? }
```

79. Here is the overall setup.

```
⟨ Declare the function called special_cases 82 ⟩
function do_page: boolean;
  label fin_set, fin_rule, move_right, show_state, done, 9998, 9999;
  var o: eight_bits;  { operation code of the current command }
    p, q: integer;  { parameters of the current command }
    a: integer;  { byte number of the current command }
    hhh: integer;  { h, rounded to the nearest pixel }
  begin cur_font ← invalid_font;  { set current font undefined }
  s ← 0; h ← 0; v ← 0; w ← 0; x ← 0; y ← 0; z ← 0; hh ← 0; vv ← 0;  { initialize the state variables }
  while true do ⟨ Translate the next command in the DVI file; goto 9999 with do_page = true if it was
      eop; goto 9998 if premature termination is needed 80 ⟩;
9998: print_ln('!'); do_page ← false;
9999: end;
```

80. Commands are broken down into "major" and "minor" categories: A major command is always shown in full, while a minor one is put into the buffer in abbreviated form. Minor commands, which account for the bulk of most `DVI` files, involve horizontal spacing and the typesetting of characters in a line; these are shown in full only if *out_mode* $\geq$ *verbose*.

```
define show(#) ≡
        begin flush_text; showing ← true; print(a : 1, ´:␣´, #);
        end
define major(#) ≡
        if out_mode > errors_only then show(#)
define minor(#) ≡
        if out_mode > terse then
          begin showing ← true; print(a : 1, ´:␣´, #);
          end
define error(#) ≡
        if ¬showing then show(#)
        else print(´␣´, #)
⟨ Translate the next command in the DVI file; goto 9999 with do_page = true if it was eop; goto 9998 if
      premature termination is needed 80 ⟩ ≡
  begin a ← cur_loc; showing ← false; o ← get_byte; p ← first_par(o);
  if eof(dvi_file) then bad_dvi(´the␣file␣ended␣prematurely´);
  ⟨ Start translation of command o and goto the appropriate label to finish the job 81 ⟩;
fin_set: ⟨ Finish a command that either sets or puts a character, then goto move_right or done 89 ⟩;
fin_rule: ⟨ Finish a command that either sets or puts a rule, then goto move_right or done 90 ⟩;
move_right: ⟨ Finish a command that sets h ← h + q, then goto done 91 ⟩;
show_state: ⟨ Show the values of ss, h, v, w, x, y, z, hh, and vv; then goto done 93 ⟩;
done: if showing then print_ln(´␣´);
  end
```

This code is used in section 79.

81. The multiway switch in *first_par*, above, was organized by the length of each command; the one in *do_page* is organized by the semantics.

```
⟨ Start translation of command o and goto the appropriate label to finish the job 81 ⟩ ≡
  if o < set_char_0 + 128 then ⟨ Translate a set_char command 88 ⟩
  else case o of
    four_cases(set1): begin major(´set´, o − set1 + 1 : 1, ´␣´, p : 1); goto fin_set;
      end;
    four_cases(put1): begin major(´put´, o − put1 + 1 : 1, ´␣´, p : 1); goto fin_set;
      end;
    set_rule: begin major(´setrule´); goto fin_rule;
      end;
    put_rule: begin major(´putrule´); goto fin_rule;
      end;
    ⟨ Cases for commands nop, bop, . . . , pop 83 ⟩
    ⟨ Cases for horizontal motion 84 ⟩
    othercases if special_cases(o, p, a) then goto done else goto 9998
    endcases
```

This code is used in section 80.

82. ⟨ Declare the function called *special_cases* 82 ⟩ ≡

```
function special_cases(o : eight_bits; p, a : integer): boolean;
  label change_font, move_down, done, 9998;
  var q: integer;  { parameter of the current command }
    k: integer;  { loop index }
    bad_char: boolean;  { has a non-ASCII character code appeared in this xxx? }
    pure: boolean;  { is the command error-free? }
    vvv: integer;  { v, rounded to the nearest pixel }
  begin pure ← true;
  case o of
  ⟨ Cases for vertical motion 85 ⟩
  ⟨ Cases for fonts 86 ⟩
  four_cases(xxx1): ⟨ Translate an xxx command and goto done 87 ⟩;
  pre: begin error('preamble␣command␣within␣a␣page!'); goto 9998;
    end;
  post, post_post: begin error('postamble␣command␣within␣a␣page!'); goto 9998;
    end;
  othercases begin error('undefined␣command␣', o : 1, '!'); goto done;
    end
  endcases;
move_down: ⟨ Finish a command that sets v ← v + p, then goto done 92 ⟩;
change_font: ⟨ Finish a command that changes the current font, then goto done 94 ⟩;
9998: pure ← false;
done: special_cases ← pure;
  end;
```

This code is used in section 79.

83. ⟨ Cases for commands *nop*, *bop*, ..., *pop* 83 ⟩ ≡

```
nop: begin minor('nop'); goto done;
  end;
bop: begin error('bop occurred before eop!'); goto 9998;
  end;
eop: begin major('eop');
  if s ≠ 0 then error('stack not empty at end of page (level ', s : 1, ')!');
  do_page ← true; print_ln(' '); goto 9999;
  end;
push: begin major('push');
  if s = max_s_so_far then
    begin max_s_so_far ← s + 1;
    if s = max_s then error('deeper than claimed in postamble!');
    if s = stack_size then
      begin error('DVItype capacity exceeded (stack size=', stack_size : 1, ')'); goto 9998;
      end;
    end;
  hstack[s] ← h; vstack[s] ← v; wstack[s] ← w; xstack[s] ← x; ystack[s] ← y; zstack[s] ← z;
  hhstack[s] ← hh; vvstack[s] ← vv; incr(s); ss ← s − 1; goto show_state;
  end;
pop: begin major('pop');
  if s = 0 then error('(illegal at level zero)!')
  else begin decr(s); hh ← hhstack[s]; vv ← vvstack[s]; h ← hstack[s]; v ← vstack[s]; w ← wstack[s];
    x ← xstack[s]; y ← ystack[s]; z ← zstack[s];
    end;
  ss ← s; goto show_state;
  end;
```

This code is used in section 81.

84. Rounding to the nearest pixel is best done in the manner shown here, so as to be inoffensive to the eye: When the horizontal motion is small, like a kern, *hh* changes by rounding the kern; but when the motion is large, *hh* changes by rounding the true position *h* so that accumulated rounding errors disappear. We allow a larger space in the negative direction than in the positive one, because TEX makes comparatively large backspaces when it positions accents.

```
define out_space(#) ≡
        if (p ≥ font_space[cur_font]) ∨ (p ≤ −4 ∗ font_space[cur_font]) then
          begin out_text("␣"); hh ← pixel_round(h + p);
          end
        else hh ← hh + pixel_round(p);
      minor(#, ' ', p : 1); q ← p; goto move_right
```

⟨ Cases for horizontal motion 84 ⟩ ≡

```
four_cases(right1): begin out_space('right', o − right1 + 1 : 1);
  end;
w0, four_cases(w1): begin w ← p; out_space('w', o − w0 : 1);
  end;
x0, four_cases(x1): begin x ← p; out_space('x', o − x0 : 1);
  end;
```

This code is used in section 81.

85. Vertical motion is done similarly, but with the threshold between "small" and "large" increased by a factor of five. The idea is to make fractions like "$\frac{1}{2}$" round consistently, but to absorb accumulated rounding errors in the baseline-skip moves.

```
define out_vmove(#) ≡
        if abs(p) ≥ 5 * font_space[cur_font] then vv ← pixel_round(v + p)
        else vv ← vv + pixel_round(p);
      major(#, ´␣´, p : 1); goto move_down
```

⟨ Cases for vertical motion 85 ⟩ ≡

```
four_cases(down1): begin out_vmove(´down´, o − down1 + 1 : 1);
  end;
y0, four_cases(y1): begin y ← p; out_vmove(´y´, o − y0 : 1);
  end;
z0, four_cases(z1): begin z ← p; out_vmove(´z´, o − z0 : 1);
  end;
```

This code is used in section 82.

86. ⟨ Cases for fonts 86 ⟩ ≡

```
sixty_four_cases(fnt_num_0): begin major(´fntnum´, p : 1); goto change_font;
  end;
four_cases(fnt1): begin major(´fnt´, o − fnt1 + 1 : 1, ´␣´, p : 1); goto change_font;
  end;
four_cases(fnt_def1): begin major(´fntdef´, o − fnt_def1 + 1 : 1, ´␣´, p : 1); define_font(p); goto done;
  end;
```

This code is used in section 82.

87. ⟨ Translate an *xxx* command and **goto** *done* 87 ⟩ ≡

```
begin major(´xxx␣´´´); bad_char ← false;
if p < 0 then error(´string␣of␣negative␣length!´);
for k ← 1 to p do
  begin q ← get_byte;
  if (q < "␣") ∨ (q > "~") then bad_char ← true;
  if showing then print(xchr[q]);
  end;
if showing then print(´´´´);
if bad_char then error(´non-ASCII␣character␣in␣xxx␣command!´);
goto done;
end
```

This code is used in section 82.

88. ⟨ Translate a *set_char* command 88 ⟩ ≡

```
begin if (o > "␣") ∧ (o ≤ "~") then
  begin out_text(p); minor(´setchar´, p : 1);
  end
else major(´setchar´, p : 1);
goto fin_set;
end
```

This code is used in section 81.

89. ⟨ Finish a command that either sets or puts a character, then **goto** *move_right* or *done* 89 ⟩ ≡

```
if p < 0 then p ← 255 − ((−1 − p) mod 256)
else if p ≥ 256 then p ← p mod 256;  { width computation for oriental fonts }
if (p < font_bc[cur_font]) ∨ (p > font_ec[cur_font]) then q ← invalid_width
else q ← char_width(cur_font)(p);
if q = invalid_width then
  begin error(´character␣´, p : 1, ´␣invalid␣in␣font␣´); print_font(cur_font);
  if cur_font ≠ invalid_font then print(´!´);  { the invalid font has '!' in its name }
  end;
if o ≥ put1 then goto done;
if q = invalid_width then q ← 0
else hh ← hh + char_pixel_width(cur_font)(p);
goto move_right
```

This code is used in section 80.

90. ⟨ Finish a command that either sets or puts a rule, then **goto** *move_right* or *done* 90 ⟩ ≡

```
q ← signed_quad;
if showing then
  begin print(´␣height␣´, p : 1, ´,␣width␣´, q : 1);
  if out_mode > mnemonics_only then
    if (p ≤ 0) ∨ (q ≤ 0) then print(´␣(invisible)´)
    else print(´␣(´, rule_pixels(p) : 1, ´x´, rule_pixels(q) : 1, ´␣pixels)´);
  end;
if o = put_rule then goto done;
if showing then
  if out_mode > mnemonics_only then print_ln(´␣´);
hh ← hh + rule_pixels(q); goto move_right
```

This code is used in section 80.

91. A sequence of consecutive rules, or consecutive characters in a fixed-width font whose width is not an integer number of pixels, can cause *hh* to drift far away from a correctly rounded value. `DVItype` ensures that the amount of drift will never exceed *max_drift* pixels.

Since `DVItype` is intended to diagnose strange errors, it checks carefully to make sure that h and v do not get out of range. Normal `DVI`-reading programs need not do this.

```
define infinity ≡ '17777777777  { ∞ (approximately) }
define max_drift = 2  { we insist that abs(hh − pixel_round(h)) ≤ max_drift }
⟨ Finish a command that sets h ← h + q, then goto done 91 ⟩ ≡
  if (h > 0) ∧ (q > 0) then
    if h > infinity − q then
      begin error('arithmetic␣overflow!␣parameter␣changed␣from␣', q : 1, '␣to␣', infinity − h : 1);
      q ← infinity − h;
      end;
  if (h < 0) ∧ (q < 0) then
    if −h > q + infinity then
      begin error('arithmetic␣overflow!␣parameter␣changed␣from␣', q : 1, '␣to␣', (−h) − infinity : 1);
      q ← (−h) − infinity;
      end;
  hhh ← pixel_round(h + q);
  if abs(hhh − hh) > max_drift then
    if hhh > hh then hh ← hhh − max_drift
    else hh ← hhh + max_drift;
  if showing then
    if out_mode > mnemonics_only then
      begin print('␣h:=', h : 1);
      if q ≥ 0 then print('+');
      print(q : 1, '=', h + q : 1, ',␣hh:=', hh : 1);
      end;
  h ← h + q;
  if abs(h) > max_h_so_far then
    begin if abs(h) > max_h + 99 then
      begin error('warning:␣|h|>', max_h : 1, '!'); max_h ← abs(h);
      end;
    max_h_so_far ← abs(h);
    end;
  goto done
```

This code is used in section 80.

92. ⟨ Finish a command that sets $v \leftarrow v + p$, then **goto** *done* 92 ⟩ ≡

```
if (v > 0) ∧ (p > 0) then
  if v > infinity − p then
    begin error('arithmetic␣overflow!␣parameter␣changed␣from␣', p : 1, '␣to␣', infinity − v : 1);
    p ← infinity − v;
    end;
if (v < 0) ∧ (p < 0) then
  if −v > p + infinity then
    begin error('arithmetic␣overflow!␣parameter␣changed␣from␣', p : 1, '␣to␣', (−v) − infinity : 1);
    p ← (−v) − infinity;
    end;
vvv ← pixel_round(v + p);
if abs(vvv − vv) > max_drift then
  if vvv > vv then vv ← vvv − max_drift
  else vv ← vvv + max_drift;
if showing then
  if out_mode > mnemonics_only then
    begin print('␣v:=', v : 1);
    if p ≥ 0 then print('+');
    print(p : 1, '=', v + p : 1, ',␣vv:=', vv : 1);
    end;
v ← v + p;
if abs(v) > max_v_so_far then
  begin if abs(v) > max_v + 99 then
    begin error('warning:␣|v|>', max_v : 1, '!'); max_v ← abs(v);
    end;
  max_v_so_far ← abs(v);
  end;
goto done
```

This code is used in section 82.

93. ⟨ Show the values of ss, h, v, w, x, y, z, hh, and vv; then **goto** *done* 93 ⟩ ≡

```
if showing then
  if out_mode > mnemonics_only then
    begin print_ln('␣'); print('level␣', ss : 1, ':(h=', h : 1, ',v=', v : 1, ',w=', w : 1, ',x=', x : 1,
        ',y=', y : 1, ',z=', z : 1, ',hh=', hh : 1, ',vv=', vv : 1, ')');
    end;
goto done
```

This code is used in section 80.

94. ⟨ Finish a command that changes the current font, then **goto** *done* 94 ⟩ ≡

```
font_num[nf] ← p; cur_font ← 0;
while font_num[cur_font] ≠ p do incr(cur_font);
if cur_font = nf then
  begin cur_font ← invalid_font;
  error('invalid␣font␣selection:␣font␣', p : 1, '␣was␣never␣defined!');
  end;
if showing then
  if out_mode > mnemonics_only then
    begin print('␣current␣font␣is␣'); print_font(cur_font);
    end;
goto done
```

This code is used in section 82.

95. Skipping pages. A routine that's much simpler than *do_page* is used to pass over pages that are not being translated. The *skip_pages* subroutine is assumed to begin just after the preamble has been read, or just after a *bop* has been processed. It continues until either finding a *bop* that matches the desired starting page specifications, or until running into the postamble.

```
⟨ Declare the procedure called scan_bop 99 ⟩
procedure skip_pages(bop_seen : boolean);
  label 9999;  { end of this subroutine }
  var p: integer;  { a parameter }
    k: 0 .. 255;  { command code }
    down_the_drain: integer;  { garbage }
  begin showing ← false;
  while true do
    begin if ¬bop_seen then
      begin scan_bop;
      if in_postamble then goto 9999;
      if ¬started then
        if start_match then
          begin started ← true; goto 9999;
          end;
      end;
    ⟨ Skip until finding eop 96 ⟩;
    bop_seen ← false;
    end;
9999: end;
```

96. ⟨ Skip until finding *eop* 96 ⟩ ≡

```
  repeat if eof(dvi_file) then bad_dvi('the␣file␣ended␣prematurely');
    k ← get_byte; p ← first_par(k);
    case k of
    set_rule, put_rule: down_the_drain ← signed_quad;
    four_cases(fnt_def1): begin define_font(p); print_ln('␣');
      end;
    four_cases(xxx1): while p > 0 do
        begin down_the_drain ← get_byte; decr(p);
        end;
    bop, pre, post, post_post, undefined_commands: bad_dvi('illegal␣command␣at␣byte␣', cur_loc − 1 : 1);
    othercases do_nothing
    endcases;
  until k = eop;
```

This code is used in section 95.

97. Global variables called *old_backpointer* and *new_backpointer* are used to check whether the back pointers are properly set up. Another one tells whether we have already found the starting page.

```
⟨ Globals in the outer block 10 ⟩ +≡
old_backpointer: integer;  { the previous bop command location }
new_backpointer: integer;  { the current bop command location }
started: boolean;  { has the starting page been found? }
```

98. ⟨ Set initial values 11 ⟩ +≡

```
  old_backpointer ← −1; started ← false;
```

99. The *scan_bop* procedure reads `DVI` commands following the preamble or following *eop*, until finding either *bop* or the postamble.

```
⟨ Declare the procedure called scan_bop 99 ⟩ ≡
procedure scan_bop;
  var k: 0 .. 255;  { command code }
  begin repeat if eof(dvi_file) then bad_dvi('the␣file␣ended␣prematurely');
    k ← get_byte;
    if (k ≥ fnt_def1) ∧ (k < fnt_def1 + 4) then
      begin define_font(first_par(k)); k ← nop;
      end;
  until k ≠ nop;
  if k = post then in_postamble ← true
  else begin if k ≠ bop then bad_dvi('byte␣', cur_loc − 1 : 1, '␣is␣not␣bop');
    new_backpointer ← cur_loc − 1; incr(page_count);
    for k ← 0 to 9 do count[k] ← signed_quad;
    if signed_quad ≠ old_backpointer then
      print_ln('backpointer␣in␣byte␣', cur_loc − 4 : 1, '␣should␣be␣', old_backpointer : 1, '!');
    old_backpointer ← new_backpointer;
    end;
  end;
```

This code is used in section 95.

100. Using the backpointers. The routines in this section of the program are brought into play only if *random_reading* is *true* (and only if *out_mode* = *the_works*). First comes a routine that illustrates how to find the postamble quickly.

```
⟨ Find the postamble, working back from the end 100 ⟩ ≡
  n ← dvi_length;
  if n < 53 then bad_dvi(´only␣´, n : 1, ´␣bytes␣long´);
  m ← n − 4;
  repeat if m = 0 then bad_dvi(´all␣223s´);
    move_to_byte(m); k ← get_byte; decr(m);
  until k ≠ 223;
  if k ≠ id_byte then bad_dvi(´ID␣byte␣is␣´, k : 1);
  move_to_byte(m − 3); q ← signed_quad;
  if (q < 0) ∨ (q > m − 33) then bad_dvi(´post␣pointer␣´, q : 1, ´␣at␣byte␣´, m − 3 : 1);
  move_to_byte(q); k ← get_byte;
  if k ≠ post then bad_dvi(´byte␣´, q : 1, ´␣is␣not␣post´);
  post_loc ← q; first_backpointer ← signed_quad
```

This code is used in section 107.

101. Note that the last steps of the above code save the locations of the the *post* byte and the final *bop*. We had better declare these global variables, together with two more that we will need shortly.

```
⟨ Globals in the outer block 10 ⟩ +≡
post_loc: integer;  { byte location where the postamble begins }
first_backpointer: integer;  { the pointer following post }
start_loc: integer;  { byte location of the first page to process }
after_pre: integer;  { byte location immediately following the preamble }
```

102. The next little routine shows how the backpointers can be followed to move through a DVI file in reverse order. Ordinarily a DVI-reading program would do this only if it wants to print the pages backwards or if it wants to find a specified starting page that is not necessarily the first page in the file; otherwise it would of course be simpler and faster just to read the whole file from the beginning.

```
⟨ Count the pages and move to the starting page 102 ⟩ ≡
  q ← post_loc; p ← first_backpointer; start_loc ← −1;
  if p < 0 then in_postamble ← true
  else begin repeat  { now q points to a post or bop command; p ≥ 0 is prev pointer }
      if p > q − 46 then bad_dvi(´page␣link␣´, p : 1, ´␣after␣byte␣´, q : 1);
      q ← p; move_to_byte(q); k ← get_byte;
      if k = bop then incr(page_count)
      else bad_dvi(´byte␣´, q : 1, ´␣is␣not␣bop´);
      for k ← 0 to 9 do count[k] ← signed_quad;
      p ← signed_quad;
      if start_match then
        begin start_loc ← q; old_backpointer ← p;
        end;
    until p < 0;
    if start_loc < 0 then abort(´starting␣page␣number␣could␣not␣be␣found!´);
    if old_backpointer < 0 then start_loc ← after_pre;  { we want to check everything }
    move_to_byte(start_loc);
    end;
  if page_count ≠ total_pages then
    print_ln(´there␣are␣really␣´, page_count : 1, ´␣pages,␣not␣´, total_pages : 1, ´!´)
```

This code is used in section 107.

103. Reading the postamble. Now imagine that we are reading the `DVI` file and positioned just four bytes after the *post* command. That, in fact, is the situation, when the following part of `DVItype` is called upon to read, translate, and check the rest of the postamble.

```
procedure read_postamble;
  var k: integer;  { loop index }
    p, q, m: integer;  { general purpose registers }
  begin showing ← false; post_loc ← cur_loc − 5;
  print_ln('Postamble starts at byte ', post_loc : 1, '.');
  if signed_quad ≠ numerator then print_ln('numerator doesn''t match the preamble!');
  if signed_quad ≠ denominator then print_ln('denominator doesn''t match the preamble!');
  if signed_quad ≠ mag then
    if new_mag = 0 then print_ln('magnification doesn''t match the preamble!');
  max_v ← signed_quad; max_h ← signed_quad;
  print('maxv=', max_v : 1, ',maxh=', max_h : 1);
  max_s ← get_two_bytes; total_pages ← get_two_bytes;
  print_ln(',maxstackdepth=', max_s : 1, ',totalpages=', total_pages : 1);
  if out_mode < the_works then ⟨ Compare the lust parameters with the accumulated facts 104 ⟩;
  ⟨ Process the font definitions of the postamble 106 ⟩;
  ⟨ Make sure that the end of the file is well-formed 105 ⟩;
  end;
```

104. No warning is given when *max_h_so_far* exceeds *max_h* by less than 100, since 100 units is invisibly small; it's approximately the wavelength of visible light, in the case of TEX output. Rounding errors can be expected to make h and v slightly more than *max_h* and *max_v*, every once in a while; hence small discrepancies are not cause for alarm.

```
⟨ Compare the lust parameters with the accumulated facts 104 ⟩ ≡
  begin if max_v + 99 < max_v_so_far then print_ln('warning: observed maxv was ', max_v_so_far : 1);
  if max_h + 99 < max_h_so_far then print_ln('warning: observed maxh was ', max_h_so_far : 1);
  if max_s < max_s_so_far then print_ln('warning: observed maxstackdepth was ', max_s_so_far : 1);
  if page_count ≠ total_pages then
    print_ln('there are really ', page_count : 1, ' pages, not ', total_pages : 1, '!');
  end
```

This code is used in section 103.

105. When we get to the present code, the *post_post* command has just been read.

```
⟨ Make sure that the end of the file is well-formed 105 ⟩ ≡
  q ← signed_quad;
  if q ≠ post_loc then print_ln('bad postamble pointer in byte ', cur_loc − 4 : 1, '!');
  m ← get_byte;
  if m ≠ id_byte then
    print_ln('identification in byte ', cur_loc − 1 : 1, ' should be ', id_byte : 1, '!');
  k ← cur_loc; m ← 223;
  while (m = 223) ∧ ¬eof(dvi_file) do m ← get_byte;
  if ¬eof(dvi_file) then bad_dvi('signature in byte ', cur_loc − 1 : 1, ' should be 223')
  else if cur_loc < k + 4 then
      print_ln('not enough signature bytes at end of file (', cur_loc − k : 1, ')');
```

This code is used in section 103.

106. ⟨ Process the font definitions of the postamble 106 ⟩ ≡

```
repeat k ← get_byte;
  if (k ≥ fnt_def1) ∧ (k < fnt_def1 + 4) then
    begin p ← first_par(k); define_font(p); print_ln(´␣´); k ← nop;
    end;
until k ≠ nop;
if k ≠ post_post then print_ln(´byte␣´, cur_loc − 1 : 1, ´␣is␣not␣postpost!´)
```

This code is used in section 103.

107. The main program. Now we are ready to put it all together. This is where `DVItype` starts, and where it ends.

```
begin initialize; { get all variables initialized }
dialog; { set up all the options }
⟨ Process the preamble 109 ⟩;
if out_mode = the_works then { random_reading = true }
  begin ⟨ Find the postamble, working back from the end 100 ⟩;
  in_postamble ← true; read_postamble; in_postamble ← false;
  ⟨ Count the pages and move to the starting page 102 ⟩;
  end;
skip_pages(false);
if ¬in_postamble then ⟨ Translate up to max_pages pages 111 ⟩;
if out_mode < the_works then
  begin if ¬in_postamble then skip_pages(true);
  if signed_quad ≠ old_backpointer then
    print_ln('backpointer␣in␣byte␣', cur_loc − 4 : 1, '␣should␣be␣', old_backpointer : 1, '!');
  read_postamble;
  end;
final_end: end.
```

108. The main program needs a few global variables in order to do its work.

```
⟨ Globals in the outer block 10 ⟩ +≡
k, m, n, p, q: integer; { general purpose registers }
```

109. A `DVI`-reading program that reads the postamble first need not look at the preamble; but `DVItype` looks at the preamble in order to do error checking, and to display the introductory comment.

```
⟨ Process the preamble 109 ⟩ ≡
  open_dvi_file; p ← get_byte; { fetch the first byte }
  if p ≠ pre then bad_dvi('First␣byte␣isn''t␣start␣of␣preamble!');
  p ← get_byte; { fetch the identification byte }
  if p ≠ id_byte then print_ln('identification␣in␣byte␣1␣should␣be␣', id_byte : 1, '!');
  ⟨ Compute the conversion factors 110 ⟩;
  p ← get_byte; { fetch the length of the introductory comment }
  print('''');
  while p > 0 do
    begin decr(p); print(xchr[get_byte]);
    end;
  print_ln(''''); after_pre ← cur_loc
```

This code is used in section 107.

110. The conversion factor *conv* is figured as follows: There are exactly n/d decimicrons per `DVI` unit, and 254000 decimicrons per inch, and *resolution* pixels per inch. Then we have to adjust this by the stated amount of magnification.

```
⟨ Compute the conversion factors 110 ⟩ ≡
  numerator ← signed_quad; denominator ← signed_quad;
  if numerator ≤ 0 then bad_dvi(´numerator␣is␣´, numerator : 1);
  if denominator ≤ 0 then bad_dvi(´denominator␣is␣´, denominator : 1);
  print_ln(´numerator/denominator=´, numerator : 1, ´/´, denominator : 1);
  tfm_conv ← (25400000.0/numerator) ∗ (denominator/473628672)/16.0;
  conv ← (numerator/254000.0) ∗ (resolution/denominator); mag ← signed_quad;
  if new_mag > 0 then mag ← new_mag
  else if mag ≤ 0 then bad_dvi(´magnification␣is␣´, mag : 1);
  true_conv ← conv; conv ← true_conv ∗ (mag/1000.0);
  print_ln(´magnification=´, mag : 1, ´;␣´, conv : 16 : 8, ´␣pixels␣per␣DVI␣unit´)
```

This code is used in section 109.

111. The code shown here uses a convention that has proved to be useful: If the starting page was specified as, e.g., '`1.*.-5`', then all page numbers in the file are displayed by showing the values of counts 0, 1, and 2, separated by dots. Such numbers can, for example, be displayed on the console of a printer when it is working on that page.

```
⟨ Translate up to max_pages pages 111 ⟩ ≡
  begin while max_pages > 0 do
    begin decr(max_pages); print_ln(´␣´); print(cur_loc − 45 : 1, ´:␣beginning␣of␣page␣´);
    for k ← 0 to start_vals do
      begin print(count[k] : 1);
      if k < start_vals then print(´.´)
      else print_ln(´␣´);
      end;
    if ¬do_page then bad_dvi(´page␣ended␣unexpectedly´);
    scan_bop;
    if in_postamble then goto done;
    end;
done: end
```

This code is used in section 107.

112. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make `DVItype` 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 printed program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

113. Index. Pointers to error messages appear here together with the section numbers where each identifier is used.

The DVItype processor

(Version 3.6, December 1995)

The preparation of this report was supported in part by the National Science Foundation under grants IST-8201926 and MCS-8300984, and by the System Development Foundation.