1* Introduction. The DVIcopy utility program copies (selected pages of) binary device-independent ("DVI") files that are produced by document compilers such as \TeX, and replaces all references to characters from virtual fonts by the typesetting instructions specified for them in binary virtual-font ("VF") files. This program has two chief purposes: (1) It can be used as preprocessor for existing DVI-related software in cases where this software is unable to handle virtual fonts or (given suitable VF files) where this software cannot handle fonts with more than 128 characters; and (2) it serves as an example of a program that reads DVI andVF files correctly, for system programmers who are developing DVI-related software.

Goal number (1) is important since quite a few existing programs have to be adapted to the extended capabilities of Version 3 of \TeX which will require some time. Moreover some existing programs are 'as is' and the source code is, unfortunately, not available. 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 (see the documentation of \DVItype for more details). This program is written as if it were a DVI-driver for a hypothetical typesetting device \texttt{out\_file}, the output file receiving the copy of the input \texttt{dvi\_file}. In addition all code related to \texttt{out\_file} is concentrated in two chapters at the end of this program and quite independent of the rest of the code concerned with the decoding of DVI and VF files and with font substitutions. Thus it should be relatively easy to replace the device dependent code of this program by the corresponding code required for a real typesetting device. Having this in mind \DVItype’s pixel rounding algorithms are included as conditional code not used by DVIcopy.

The \texttt{banner} and \texttt{preamble\_comment} strings defined here should be changed whenever DVIcopy gets modified.

\begin{verbatim}
define my\_name ≡ 'dvicopy'
define banner ≡ 'This is DVIcopy, Version 1.6' \{printed when the program starts\}
define title ≡ 'DVIcopy' \{the name of this program, used in some messages\}
define copyright ≡ 'Copyright (C) 1990,2009, Peter Breitenlohner'
define preamble\_comment ≡ 'DVIcopy, 1.6, output from\_'
define comm\_length = 24 \{length of preamble\_comment\}
define from\_length = 6 \{length of its `\_from\_` part\}
\end{verbatim}

2* This program is written in standard Pascal, except where it is necessary to use extensions; for example, DVIcopy 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; since DVIcopy is (a model for) a production program it should, however, be made as efficient as possible for a particular system. If DVIcopy is being used with Pascals for which random file positioning is not efficiently available, the following definition should be changed from \texttt{true} to \texttt{false}; in such cases, DVIcopy will not include the optional feature that reads the postamble first.

\begin{verbatim}
(random\_reading: boolean; \{should we skip around in the file?\}) ≡
\end{verbatim}

See also sections 17, 21, 32, 37, 46, 49, 62*, 65, 71, 77, 80, 81, 84, 90, 96, 100, 108*, 117, 120, 122, 124, 125, 128, 134, 142, 146, 157, 158, 173, 177, 183, 185, 193, 199, 220, 231, 244, 255, 259, and 301*.

This code is used in section 3*. 

\[
\text{DVIcopy} \quad \S 1
\]
3* The program begins with a fairly normal header, made up of pieces that will mostly be filled in later. The DVI input comes from file \textit{dvi file}, the DVI output goes to file \textit{out file}, and messages go to Pascal's standard output file. The TFM and VF files are defined later since their external names are determined dynamically.

If it is necessary to abort the job because of a fatal error, the program calls the \texttt{'jump out'} procedure.

(Compiler directives 9)

\texttt{procedure initialize; \{ this procedure gets things started properly \}}

(Errors handling procedures 23*)

\texttt{begin kpse_set_program_name(argv[0], my name); parse_arguments; print(banner);
print ln(version_string); print ln(copyright);
print ln(\texttt{'Distributed under terms of GNU General Public License'})
\{ Set initial values 18 \}}

end;

5* The following parameters can be changed at compile time to extend or reduce \texttt{DVIcopy}'s capacity.

\texttt{define max select = 10 \{ maximum number of page selection ranges \}}

( constants in the outer block 5*)

\texttt{max fonts = 400; \{ maximum number of distinct fonts \}}

\texttt{max chars = 750000; \{ maximum number of different characters among all fonts \}}

\texttt{max widths = 16000; \{ maximum number of different characters widths \}}

\texttt{max packets = 65530; \{ maximum number of different characters packets; must be less than 65536 \}}

\texttt{max bytes = 250000; \{ maximum number of bytes for characters packets \}}

\texttt{max recursion = 10; \{ VF files shouldn't recurse beyond this level \}}

\texttt{stack size = 100; \{ DVI files shouldn't push beyond this depth \}}

\texttt{terminal line length = 256; \{ maximum number of characters input in a single line of input from the terminal \}}

This code is used in section 3*.
7* Introduction (continued). On some systems it is necessary to use various integer subrange types in order to make DVIcopy efficient; this is true in particular for frequently used variables such as loop indices. Consider an integer variable \( x \) with values in the range 0..255: on most small systems \( x \) should be a one or two byte integer whereas on most large systems \( x \) should be a four byte integer. Clearly the author of a program knows best which range of values is required for each variable; thus DVIcopy never uses Pascal’s integer type. All integer variables are declared as one of the integer subrange types defined below as WEB macros or Pascal types; these definitions can be used without system-dependent changes, provided the signed 32 bit integers are a subset of the standard type integer, and the compiler automatically uses the optimal representation for integer subranges (both conditions need not be satisfied for a particular system).

The complementary problem of storing large arrays of integer type variables as compactly as possible is efficiently; this is true in particular for frequently used variables such as loop indices. Consider an integer variable \( x \) with values in the range 0..255: on most small systems \( x \) should be a one or two byte integer whereas on most large systems \( x \) should be a four byte integer. Clearly the author of a program knows best which range of values is required for each variable; thus DVIcopy never uses Pascal’s integer type. All integer variables are declared as one of the integer subrange types defined below as WEB macros or Pascal types; these definitions can be used without system-dependent changes, provided the signed 32 bit integers are a subset of the standard type integer, and the compiler automatically uses the optimal representation for integer subranges (both conditions need not be satisfied for a particular system).

The complementary problem of storing large arrays of integer type variables as compactly as possible is addressed differently; here DVIcopy uses a Pascal type declaration for each kind of array element.

Note that the primary purpose of these definitions is optimizations, not range checking. All places where optimization for a particular system is highly desirable have been listed in the index under “optimization.”

\[
\begin{align*}
\text{define } & \text{int}_32 \equiv \text{integer} \quad \{ \text{signed 32 bit integers} \} \\
\text{define } & \text{int}_31 \equiv \text{int}_31_t \\
\text{define } & \text{int}_24 u \equiv \text{int}_24 u_t \\
\text{define } & \text{int}_24 \equiv \text{int}_24_t \\
\text{define } & \text{int}_23 \equiv \text{int}_23_t \\
\text{define } & \text{int}_16 u \equiv \text{int}_16 u_t \\
\text{define } & \text{int}_16 \equiv \text{int}_16_t \\
\text{define } & \text{int}_8 u \equiv \text{int}_8 u_t \\
\text{define } & \text{int}_8 \equiv \text{int}_8_t \\
\text{define } & \text{int}_7 \equiv \text{int}_7_t
\end{align*}
\]

(Types in the outer block 7* ) \equiv
\[
\begin{align*}
\text{int}_31 & = 0 \ldots \text{"}7\text{FFFFFFF"} ; \quad \{ \text{unsigned 31 bit integer} \} \\
\text{int}_24 u & = 0 \ldots \text{"}7\text{FFFFF"} ; \quad \{ \text{unsigned 24 bit integer} \} \\
\text{int}_24 & = -\text{"}8\text{00000 .. "}7\text{FFFFF"} ; \quad \{ \text{signed 24 bit integer} \} \\
\text{int}_23 & = 0 \ldots \text{"}7\text{FFFFF"} ; \quad \{ \text{unsigned 23 bit integer} \} \\
\text{int}_16 u & = 0 \ldots \text{"}7\text{FFF"} ; \quad \{ \text{unsigned 16 bit integer} \} \\
\text{int}_16 & = -\text{"}8\text{000 .. "}7\text{FFF"} ; \quad \{ \text{signed 16 bit integer} \} \\
\text{int}_8 u & = 0 \ldots \text{"}7\text{FF"} ; \quad \{ \text{unsigned 8 bit integer} \} \\
\text{int}_8 & = -\text{"}8\text{0 .. "}7\text{F"} ; \quad \{ \text{signed 8 bit integer} \} \\
\text{int}_7 & = 0 \ldots \text{"}7\text{F"} ; \quad \{ \text{unsigned 7 bit integer} \}
\end{align*}
\]


This code is used in section 3*.

11* 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; the term d_print is used for conditional output if we are debugging.

\[
\begin{align*}
\text{define } & \text{print} ( \# ) \equiv \text{write} ( \text{term}_\text{out}, \# ) \\
\text{define } & \text{printLn} ( \# ) \equiv \text{writeLn} ( \text{term}_\text{out}, \# ) \\
\text{define } & \text{newLine} \equiv \text{writeLn} ( \text{term}_\text{out} ) \quad \{ \text{start new line} \} \\
\text{define } & \text{printNL} ( \# ) \equiv \{ \text{print information starting on a new line} \} \\
& \quad \text{begin newLine} ; \text{print} ( \# ) ; \\
& \quad \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{define } & \text{d_print} ( \# ) \equiv \\
& \quad \text{debug} \text{ print} ( \# ) \text{ gubed}
\end{align*}
\]

\[
\begin{align*}
\text{define } & \text{d_printLn} ( \# ) \equiv \\
& \quad \text{debug} \text{ printLn} ( \# ) \text{ gubed}
\end{align*}
\]
14* The character set. Like all programs written with the WEB system, DVIcopy 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 and VF files use ASCII code for file names and certain other strings.

The next few sections of DVIcopy have therefore been copied from the analogous ones in the WEB system routines. They have been considerably simplified, since DVIcopy 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.

15* 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 DVIcopy. So we shall assume that the Pascal system being used for DVIcopy 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 \( \text{chr}(\text{first_text_char}) \) through \( \text{chr}(\text{last_text_char}) \), inclusive. The following definitions should be adjusted if necessary.

\[
\text{define } \text{text_char} \equiv \text{ASCII code} \quad \{ \text{the data type of characters in text files} \} \\
\text{define } \text{first_text_char} = 0 \quad \{ \text{ordinal number of the smallest element of text_char} \} \\
\text{define } \text{last_text_char} = 255 \quad \{ \text{ordinal number of the largest element of text_char} \}
\]

\[
\text{define } \text{text_char} \equiv \text{ASCII code} \quad \{ \text{the data type of characters in text files} \} \\
\text{define } \text{first_text_char} = 0 \quad \{ \text{ordinal number of the smallest element of text_char} \} \\
\text{define } \text{last_text_char} = 255 \quad \{ \text{ordinal number of the largest element of text_char} \}
\]

\[
\langle \text{Types in the outer block 7*} \rangle +\equiv \\
\text{ASCII_code} = 0 \ldots 255; \quad \{ \text{a subrange of the integers} \} \\
\]
If an input (DVI, TFM, VF, or other) file is badly malformed, the whole process must be aborted; DVIcopy will give up, after issuing an error message about what caused the error. These messages will, however, in most cases just indicate which input file caused the error. One of the programs DVItype, TFtoPL, or VFtoVP should then be used to diagnose the error in full detail.

Such errors might be discovered inside of subroutines inside of subroutines, so a procedure called *jump_out* has been introduced.

```pascal
format noreturn ≡ procedure
define abort(#) ≡
  begin write ln(stderr, ´␣´, #, ´.´); jump_out;
  end
⟨Error handling procedures 23*⟩ ≡
⟨Basic printing procedures 48⟩

procedure close_files_and_terminate; forward;
  noreturn procedure jump_out;
    begin mark_fatal; close_files_and_terminate; uexit(1);
    end;
See also sections 24*, 25*, 94*, and 109*.
This code is used in section 3*.

Sometimes the program’s behavior is far different from what it should be, and DVIcopy prints an error message that is really for the DVIcopy maintenance person, not the user. In such cases the program says *confusion* (indication of where we are).

```pascal
⟨Error handling procedures 23*⟩ +≡
noreturn procedure confusion(p: pckt_pointer);
  begin print(´␣´, title, ´␣´, capacity exceeded[´´]); print_packet(p); print ln(´.´); jump_out;
  end;

An overflow stop occurs if DVIcopy’s tables aren’t large enough.

```pascal
⟨Error handling procedures 23*⟩ +≡
noreturn procedure overflow(p: pckt_pointer; n: int_16u);
  begin print(´␣´, title, ´␣´, capacity exceeded[´´]); print_packet(p);
   print ln(´=´, n: 1, ´.´); jump_out;
  end;

```
§26  DVIcopy  BINARY DATA AND BINARY FILES  7

62* Before a font file can be opened for input we must build a string with its external name.

\(\text{ Globals in the outer block } z^* \) \(+\equiv\)

cur_name: \(\uparrow\text{char}\);

\(l_{\text{cur name}}: \text{int}_{15}\)  \{ this many characters are actually relevant in \(\text{cur name}\) \}

63* Since files are actually searched through path definitions, the area definitions are ignored here. To reduce the required changes we simply ignore the parameters given to \(\text{make font name}\).

\(\text{define} \quad \text{append to name}(\#) \equiv\)

\(\quad \text{begin} \quad \text{cur name}[l_{\text{cur name}}] \leftarrow \#; \quad \text{incr}(l_{\text{cur name}})\)

\(\quad \text{end}\)

\(\text{define} \quad \text{make font name end}(\#) \equiv \text{make name}\)

\(\text{define} \quad \text{make font name}(\#) \equiv l_{\text{cur name}} \leftarrow 0; \quad \text{make font name end}\)

67* The \(\text{make name}\) procedure used to build the external file name. The global variable \(l_{\text{cur name}}\) contains the length of a default area which has been copied to \(\text{cur name}\) before \(\text{make name}\) is called.

\(\text{procedure} \quad \text{make name}(e : \text{pkt_pointer});\)

\(\quad \text{var} \ b: \text{eight_bits}; \quad \{ \text{a byte extracted from byte_mem} \}\)

\(\quad \ n: \text{pkt_pointer}; \quad \{ \text{file name packet} \}\)

\(\quad \ l_{\text{cur loc}}, l_{\text{cur limit}}: \text{byte_pointer}; \quad \{ \text{indices into byte_mem} \}\)

\(\quad \text{device} \ l_{\text{ll}}: \text{int}_{15}; \quad \{ \text{loop index} \}\)

\(\text{ecived}\)

\(\quad \text{begin} \quad n \leftarrow \text{font name}(\text{cur fnt}); \quad \text{cur name} \leftarrow \text{xmalloc_array(char, pkt_length(n) + pkt_length(e))};\)

\(\quad \ l_{\text{cur loc}} \leftarrow \text{pkt_start}[n]; \quad l_{\text{cur limit}} \leftarrow \text{pkt_start}[n + 1]; \quad \text{pkt_extract}(b); \quad \{ \text{length of area part} \}\)

\(\quad \text{if} \ b > 0 \quad \text{then} \quad l_{\text{cur name}} \leftarrow 0;\)

\(\quad \text{while} \ l_{\text{cur loc}} < l_{\text{cur limit}} \text{ do}\)

\(\quad \text{begin} \quad \text{pkt_extract}(b); \quad \text{append to name}(\text{xchr}[b]);\)

\(\quad \text{end};\)

\(\quad \text{cur name}[l_{\text{cur name}}] \leftarrow 0;\)

\(\text{end};\)
91* (Initialize predefined strings 45) ⊆
\text{\texttt{id4(".")("t")("f")("m")(\texttt{tfm}\	exttt{ext});} \text{ \{} \text{file name extension for TFM files \}}

92* If no font directory has been specified, we search paths.

93* (No initialization to be done. Keep this module to preserve numbering.)

94* If a TFM file is badly malformed, we say \texttt{bad\_font}; for a TFM file the \texttt{bad\_tfm} procedure is used to give an error message which refers the user to \texttt{TFtoPL} and \texttt{PLtoTF}, and terminates \texttt{DVIcopy}.

\langle \text{Error handling procedures 23*} \rangle ≡
noreturn procedure \texttt{bad\_tfm};
\begin{align*}
& \text{begin print\((\text{`Bad\_TFM\_file'})\); print\_font(cur\_fnt); print\_ln\((`!')\);}
& \text{abort\((\text{`Use\_TFtoPL/PLtoTF\_to\_diagnose\_and\_correct\_the\_problem'})\);}
& \text{end;}
\end{align*}
noreturn procedure \texttt{bad\_font};
\begin{align*}
& \text{begin new\_line;}
& \text{case font\_type(cur\_fnt) of}
& \text{defined\_font: confusion(str\_fonts);}
& \text{loaded\_font: bad\_tfm;}
& \langle \text{Cases for bad\_font 136} \rangle
& \text{othercases abort\((`\text{internal\_error'})\);}
& \text{endcases;}
& \text{end;}
\end{align*}

95* To prepare \texttt{tfm\_file} for input we \texttt{reset} it.
\texttt{TFM\_default\_area\_name\_length} and \texttt{TFM\_default\_area} will not be used by \texttt{make\_font\_name}.

\langle \text{TFM: Open \texttt{tfm\_file} 95*} \rangle ≡
\begin{align*}
& \text{make\_font\_name(TFM\_default\_area\_name\_length)(TFM\_default\_area)(\texttt{tfm\_ext});}
& \text{full\_name ← kpse\_find\_tfm(cur\_name);}
& \text{if full\_name then}
& \text{begin resetbin(tfm\_file,full\_name); free(cur\_name); free(full\_name);}
& \text{end}
& \text{else abort\((`---not\_loaded\_\text{TFM\_file\_can\_t\_be\_opened!'}\))}
\end{align*}

This code is used in section 99.

104* (Replace \texttt{z} by \texttt{z'} and compute \alpha, \beta 104*) ≡
\begin{align*}
& \text{alpha ← 16;}
& \text{if } z ≥ `1000000000 \text{ then abort\((`Character\_size\_is\_too\_large!')\);}
& \text{while } z ≥ `40000000 \text{ do}
& \text{begin } z ← z \text{ div } 2; \text{ alpha ← alpha + alpha;}
& \text{end;}
& \text{beta ← 256 \text{ div } alpha; alpha ← alpha * z}
\end{align*}

This code is used in sections 105 and 152.
108* Low-level DVI input routines. The program uses the binary file variable \texttt{dvi\_file} for its main input file; \texttt{dvi\_loc} is the number of the byte about to be read next from \texttt{dvi\_file}.

\begin{verbatim}
Globals in the outer block 2* \equiv 
\begin{verbatim}
dvi\_file: byte\_file;  \{ the stuff we are DVIcopying \}
dvi\_loc: int\_32;  \{ where we are about to look, in \texttt{dvi\_file} \}
full\_name: ↑char;
\end{verbatim}
\end{verbatim}

109* If the DVI file is badly malformed, we say \texttt{bad\_dvi}; this procedure gives an error message which refers the user to \texttt{DVItype}, and terminates \texttt{DVIcopy}.

\begin{verbatim}
Error handling procedures 23* \equiv 
noreturn procedure bad\_dvi;
\begin{verbatim}
begin new\_line; print\_ln(´Bad DVI file: loc=", dvi\_loc : 1,"!´);
print(´Use DVItype with output level´);
if random\_reading then print(´=4´) else print(´<4´);
abort(´to diagnose the problem´);
end;
\end{verbatim}
\end{verbatim}

110* To prepare \texttt{dvi\_file} for input, we \texttt{reset} it.

\begin{verbatim}
Open input file(s) 110* \equiv 
dvi\_loc ← 0;
\end{verbatim}
This code is used in section 241*.

112* Next we come to the routines that are used only if \texttt{random\_reading} is \texttt{true}. The driver program below needs two such routines: \texttt{dvi\_length} should compute the total number of bytes in \texttt{dvi\_file}, possibly also causing \texttt{eof(dvi\_file)} to be \texttt{true}; and \texttt{dvi\_move(n)} should position \texttt{dvi\_file} so that the next \texttt{dvi\_byte} will read byte \texttt{n}, starting with \texttt{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 \texttt{set\_pos} and \texttt{cur\_pos}. The call \texttt{set\_pos(f,n)} moves to item \texttt{n} in file \texttt{f}, unless \texttt{n} is negative or larger than the total number of items in \texttt{f}; in the latter case, \texttt{set\_pos(f,n)} moves to the end of file \texttt{f}. The call \texttt{cur\_pos(f)} gives the total number of items in \texttt{f}, if \texttt{eof(f)} is \texttt{true}; we use \texttt{cur\_pos} only in such a situation.

\begin{verbatim}
function dvi\_length: int\_32;
\begin{verbatim}
begin xfseek(dvi\_file,0,2,dvi\_name); dvi\_loc ← xtell(dvi\_file, dvi\_name); dvi\_length ← dvi\_loc;
end;
\end{verbatim}
\end{verbatim}

\begin{verbatim}
procedure dvi\_move(n : int\_32);
\begin{verbatim}
begin xfseek(dvi\_file,n,0,dvi\_name); dvi\_loc ← n;
end;
\end{verbatim}
\end{verbatim}
(Initialize predefined strings 45) $\equiv$

\[ id3(".")("v")("f")\text{\(vf\_ext\);} \quad \{ \text{file name extension for VF files} \} \]

If no font directory has been specified, \texttt{DVIcopy} is supposed to use the default \texttt{VF} directory, which is a system-dependent place where the \texttt{VF} files for standard fonts are kept.

Actually, under UNIX the standard area is defined in an external file \texttt{site.h}. And the users have a path searched for fonts, by setting the \texttt{VFFONTS} environment variable.

(No initialization to be done. Keep this module to preserve numbering.)

To prepare \texttt{vf\_file} for input we \texttt{reset} it.

Do path searching. But the \texttt{VF} file may not exist.

\[ \langle \text{VF: Open \texttt{vf\_file} or \texttt{goto not\_found} 139*} \rangle \equiv \]

\[
\text{make\_font\_name}(\text{\(VF\_default\_area\_name\_length\)})(\text{\(VF\_default\_area\)})(\text{\(vf\_ext\)});
\]

\[
\text{full\_name} \leftarrow \text{kpse\_find\_uf}(\text{\(cur\_name\)});
\]

\[
\text{if } \text{full\_name} \text{ then}
\]

\[
\begin{align*}
\text{begin} & \text{ resetbin(\(vf\_file, full\_name\))} \quad \text{free(\(cur\_name\))} \quad \text{free(\(full\_name\))}; \\
\text{end}
\end{align*}
\]

\[
\text{else } \text{goto not\_found};
\]

\[
\text{vf\_loc} \leftarrow 0
\]

This code is used in section 151.
§151 DVIcopy

READING VF FILES

163* web2c does not like array assignments. So we need to do them through a macro replacement.

\texttt{define do\_vf\_move(\#) \equiv vf\_move[vf\_ptr]\# \leftarrow vf\_move[vf\_ptr - 1]\#}
\texttt{define vf\_move\_assign \equiv}
\texttt{\hspace{1em}begin do\_vf\_move([0][0]); do\_vf\_move([0][1]); do\_vf\_move([1][0]); do\_vf\_move([1][1])}
\texttt{end}

\texttt{(VF: Start a new level 163*) \equiv}
\texttt{append\_one(push); vf\_move\_assign; vf\_push\_loc[vf\_ptr] \leftarrow byte\_ptr; vf\_last\_end[vf\_ptr] \leftarrow byte\_ptr;}
\texttt{vf\_last[vf\_ptr] \leftarrow vf\_other}

This code is used in sections 162 and 172.

170* (VF: Apply rule 3 or 4 170*) \equiv
\texttt{begin if vf\_push\_num[vf\_ptr] > 0 then}
\texttt{\hspace{1em}begin decr(vf\_push\_num[vf\_ptr]); vf\_move\_assign;}
\texttt{end}
\texttt{else begin decr(byte\_ptr); decr(vf\_ptr);}
\texttt{\hspace{1em}end;}
\texttt{if cur.class \neq pop.cl then goto reswitch; \{ this is rule 4 \}
\hspace{1em}end}

This code is used in section 168.
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 `byte_mem` array as a temporary string. Pascal's standard `input` file is used for terminal input, as `output` is used for terminal output.

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.

```pascal
define update_terminal ≡ fflush(stdout)  { empty the terminal output buffer }
define scan_blank(#) ≡  { tests for ‘blank’ when scanning (command line) options }
  ((byte_mem[#] = bi("\n")) ∨ (byte_mem[#] = bi(opt_separator)))
define scan_skip ≡  { skip ‘blanks’ }
  while scan_blank(scan_ptr) ∧ (scan_ptr < byte_ptr) do  incr(scan_ptr)
define scan_init ≡  { initialize scan_ptr }
  byte_mem[byte_ptr] ← bi("\n"); scan_ptr ← pckt_start[pckt_ptr − 1]; scan_skip
```

See also sections 178, 179, and 189.

This code is used in section 180.
Now we are ready to put it all together. Here is where `DVIcopy` starts, and where it ends.

```plaintext
begin initialize;  { get all variables initialized }
⟨Initialize predefined strings 45⟩
⟨Open input file(s) 110*⟩
⟨Open output file(s) 246*⟩
do_dvi;  { process the entire DVI file }
close_files_and_terminate;
end.
```
To prepare *out_file* for output, we *rewrite* it.

* ⟨ Open output file(s) 246* ) ≡ 

This code is used in section 241*.

Writing the *out_file* should be done as efficient as possible for a particular system; on many systems this means that a large number of bytes will be accumulated in a buffer and is then written from that buffer to *out_file*. In order to simplify such system dependent changes we use the WEB macro *out_byte* to write the next DVI byte. Here we give a simple minded definition for this macro in terms of standard Pascal.

`define out_byte(#) ≡ put_byte(#, out_file)  { write next DVI byte}`
These are the local variables (if any) needed for do_pre.

\[ \text{var } k: \text{int}_15 ; \{ \text{general purpose variable} \} \]
\[ p, q, r: \text{byte}_\text{pointer} ; \{ \text{indices into byte}_{\text{mem}} \} \]
\[ \text{comment: const}_{e}\text{string}; \{ \text{preamble comment prefix} \} \]

This code is used in section 204.

And here is the device dependent code for do_pre; the DVI preamble comment written to out_file is similar to the one produced by GFtoPK, but we want to apply our preamble comment prefix only once.

\[
\text{(OUT: Process the pre 261*)} \equiv \\
\text{out\.one(pre);} \text{ out\.one(dvi\.id);} \text{ out\.four(dvi\.num);} \text{ out\.four(dvi\.den);} \text{ out\.four(out\.mag);} \\
p \leftarrow \text{pkt\.start[pkt.ptr - 1]} ; q \leftarrow \text{byte.ptr} ; \{ \text{location of old DVI comment} \} \\
\text{comment} \leftarrow \text{preamble\.comment;} \text{ pkt\.room(comm\.length);} \\
\text{for } k \leftarrow 0 \text{ to comm\.length - 1 } \text{do} \text{ append\.byte(xord[ucharcast(comment]}[k]))} \\
\text{while byte.mem[p] = bi("\_\") do incr(p);} \{ \text{remove leading blanks} \} \\
\text{if } p = q \text{ then Decr(byte.ptr)(from\.length)} \\
\text{end;} \\
k \leftarrow \text{byte.ptr} - p ; \{ \text{total length} \} \\
\text{if } k > 255 \text{ then} \\
\text{begin } k \leftarrow 255 ; q \leftarrow p + 255 - \text{comm\.length} ; \{ \text{at most 255 bytes} \} \\
\text{end;} \\
\text{out\.one(k);} \text{ out\.packet(new\.packet);} \text{ flush\.packet;} \\
\text{for } r \leftarrow p \text{ to } q - 1 \text{ do out\.one(bo(byte.mem}[r]))} \\
\text{This code is used in section 204.}
\]
**293** System-dependent changes. Parse a Unix-style command line.

This macro tests if its argument is the current option, as represented by the index variable `option_index`.

```c
#define argument_is(#) \( \equiv (\text{strcmp}(\text{long_options}[\text{option_index}].\text{name}, #) = 0) \)
```

(Define `parse_arguments` 293*)

```c
procedure parse_arguments;

const n_options = 5; \{ Pascal won’t count array lengths for us. \}

var long_options: array [0 .. n_options] of getopt_struct;

getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options; k, m: c_int_type;

begin \{ Define the option table 294* \};

\begin{enumerate}
\item \textbf{Initialize options 187}: \texttt{repeat} getopt_return_val \leftarrow getopt_long_only(argc, argv, “”, long_options, address_of (option_index)); \texttt{if} getopt_return_val = \texttt{-1} \texttt{then} \{ End of arguments; we exit the loop below. \}
\item \texttt{else if} getopt_return_val = “?” \texttt{then} \{ End of arguments; we exit the loop below. \}
\item \texttt{else if} argument_is (“help”) \texttt{then} \{ End of arguments; we exit the loop below. \}
\item \texttt{else if} argument_is (“version”) \texttt{then} \{ End of arguments; we exit the loop below. \}
\item \texttt{else if} argument_is (“magnification”) \texttt{then} \{ End of arguments; we exit the loop below. \}
\item \texttt{else if} argument_is (“max-pages”) \texttt{then} \{ End of arguments; we exit the loop below. \}
\item \texttt{else if} argument_is (“page-start”) \texttt{then} \{ End of arguments; we exit the loop below. \}
\end{enumerate}

\begin{enumerate}
\item until getopt_return_val = \texttt{-1}; \{ Now optind is the index of first non-option on the command line. We can have zero, one, or two remaining arguments. \}
\item if \((\text{optind} > \text{argc}) \lor (\text{optind} + 2 < \text{argc})\) \texttt{then} \{ End of arguments; we exit the loop below. \}
\item \texttt{endif}\; \texttt{begin} writeLn(stdin.err, my_name, “:\:\textbf{Need at least two file arguments.”); usage(my_name); \texttt{end}\;
\item if \texttt{optind} = \texttt{argc} \texttt{then} \{ End of arguments; we exit the loop below. \}
\item \texttt{endif}\; \texttt{begin} dvi_name \leftarrow “<stdin>”\; \texttt{dvi_file} \leftarrow \texttt{make_binary_file(stdin)}\; \texttt{random_reading} \leftarrow \texttt{false}\; \texttt{end}\;
\item else begin dvi_name \leftarrow \texttt{extend_filename(cmdline(optind), “dvi”)}\; \texttt{resetbin(dvi_file, dvi_name)}\; \texttt{random_reading} \leftarrow \texttt{true}\; \texttt{end}\;
\item if \texttt{optind} + 2 = \texttt{argc} \texttt{then} \{ End of arguments; we exit the loop below. \}
\item \texttt{endif}\; \texttt{begin} rewritebin(out_file, extend_filename(cmdline(optind + 1), “dvi”))\; \texttt{term_out} \leftarrow \texttt{stdout}\; \texttt{end}\;
\item else begin out_file \leftarrow \texttt{make_binary_file(stdout)}\; \texttt{term_out} \leftarrow \texttt{stdout}\; \texttt{end}\;
\item \texttt{end}\;
\end{enumerate}
```

This code is used in section 3*.
§294  Here is the first of the options we allow.

\begin{verbatim}
Define the option table 294*
\end{verbatim}
\begin{verbatim}
current_option ← 0; long_options[0].name ← `help`; long_options[0].has_arg ← 0;
long_options[0].flag ← 0; long_options[0].val ← 0; incr(current_option);
\end{verbatim}

See also sections 295*, 296*, 297*, 298*, and 300*.
This code is used in section 293*.

§295  Another of the standard options.

\begin{verbatim}
Define the option table 294*
\end{verbatim}
\begin{verbatim}
+≡
long_options[current_option].name ← `version`; long_options[current_option].has_arg ← 0;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
\end{verbatim}

§296  Magnification to apply.

\begin{verbatim}
Define the option table 294*
\end{verbatim}
\begin{verbatim}
+≡
long_options[current_option].name ← `magnification`; long_options[current_option].has_arg ← 1;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
\end{verbatim}

§297  How many pages to do.

\begin{verbatim}
Define the option table 294*
\end{verbatim}
\begin{verbatim}
+≡
long_options[current_option].name ← `max-pages`; long_options[current_option].has_arg ← 1;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
\end{verbatim}

§298  What page to start at.

\begin{verbatim}
Define the option table 294*
\end{verbatim}
\begin{verbatim}
+≡
long_options[current_option].name ← `page-start`; long_options[current_option].has_arg ← 1;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
\end{verbatim}
 Parsing the starting page specification is a bit complicated. (This is the same as in DVItype.)

\[ \text{(Determine the desired start_count values from optarg 299*)} \equiv \]

\[ \begin{array}{l}
    k \leftarrow 0; \quad \{ \text{which \texttt{count} register we're on} \} \\
    m \leftarrow 0; \quad \{ \text{position in optarg} \}
\end{array} \]

while optarg[m] do
    begin
      if optarg[m] = "*" then
        begin
          start_where[k] \leftarrow false; incr(m);
        end
      else if optarg[m] = "." then
        begin
          incr(k);
          if k \geq 10 then
            begin
              writeLn(stderr, my_name, "More than ten count registers specified.");
              uexit(1);
            end;
          incr(m);
        end
      else begin
        start_count[k] \leftarrow strtol(optarg + m, address_of(end_num), 10);
        if end_num = optarg + m then
          begin
            writeLn(stderr, my_name, "-page-start values must be numeric or *.");
            uexit(1);
          end;
        start_where[k] \leftarrow true; m \leftarrow m + end_num - (optarg + m);
        end;
      end;
    end;

start_vals \leftarrow k; selected \leftarrow false;

This code is used in section 293*.

 An element with all zeros always ends the list.

\[ \text{(Define the option table 294*)} \equiv \]

\[ \begin{array}{l}
    \text{long_options[current_option].name} \leftarrow 0; \text{long_options[current_option].has_arg} \leftarrow 0; \\
    \text{long_options[current_option].flag} \leftarrow 0; \text{long_options[current_option].val} \leftarrow 0;
\end{array} \]

\[ \text{(Globals in the outer block 2*)} \equiv \]

term_out: text;

\texttt{dvi\_name: const\_c\_string;}
302* Index. Pointers to error messages appear here together with the section numbers where each identifier is used.

The following sections were changed by the change file: 1, 2, 3, 5, 7, 11, 14, 15, 23, 24, 25, 62, 63, 67, 91, 92, 93, 94, 95, 104, 108, 109, 110, 112, 135, 137, 138, 139, 163, 170, 176, 241, 246, 248, 260, 261, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302.

- help: 294*
- magnification: 296*
- max-pages: 297*
- page-start: 298*
- version: 295*

a: 50, 51, 52, 53, 258.
address_of: 293*, 299*
all_done: 185, 204, 207, 235.
alpha: 103, 104*, 142.
any: 155.
append_one:
append_to_name: 63*, 64*, 67*.
argy: 293*
argument_is: 293*
argy: 3*, 293*
atou: 293*
b: 51, 52, 53, 67*.
Bad char c: 98.
Bad DVI file: 109*.
Bad TFM file: 94*.
Bad VF file: 136.
bad_tfm: 94*.
banner: 1*, 3*, 293*.
be: 69, 99, 101, 102, 106.
begin: 6, 8, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59.
begin_byte: 50, 113, 141.
begin_char: 57, 252.
begin_four: 56, 251.
begin_one: 54.
begin_pair: 51, 113, 141.
begin_quad: 53, 113, 141.
begin_signed: 59, 254.
begin_trio: 52, 113, 141.
begin_two: 55.
begin_unsigned: 58, 253.
beta: 103, 104*, 142.
beware: char widths do not agree: 98.
beware: check sums do not agree: 98.
beware: design sizes do not agree: 98.
bo: 31, 41, 48, 49, 60, 61, 250, 258, 261*.
boolean: 2* 57, 84, 86, 98, 107, 122, 125, 132, 151, 154, 158, 178, 179, 183, 185, 186, 213, 222, 252.
bop: 26, 119, 121, 185, 205, 206, 229, 234, 235, 263.
buidl_packet: 89, 160.
byte: 175.
byte_file: 29, 90, 108*, 134, 244.
c: 52, 53, 98.
c_int_type: 293*.
chain_flag: 83, 87, 88.
char widths do not agree: 98.
char_cl: 119, 121, 156, 161, 164, 225, 236.
char_code: 115.
char_ext: 115.
char_offset: 76, 79, 81.
char_packets: 76, 77, 81, 106.
char_pixels: 198, 199, 214, 216, 222.
char_pointer: 76, 77, 99, 125, 199, 216.
char_res: 115.
check sums do not agree: 98.
check_check_sum: 98, 101, 107, 152.
check_design_size: 98, 101, 107, 152.
check_width: 98, 160.
chr: 15* 17, 19.
close_files_and_terminate: 23* 240, 241*.
close_in: 30, 99, 151, 240.
cmd_cl: 119, 120, 124.
cmd_par: 116, 117.
cmdline: 293*.
comm_length: 1*, 261*.
large_h_space: 202, 203.
large_v_space: 202, 203.
last_pop: 151, 161.
last_text_char: 15, 19.
le1nf: 146, 147, 150, 242.
ll: 64, 67*.
load: 107.
load_font: 99, 103, 107, 130.
loaded_font: 4, 94* 99, 216, 217.
long_char: 133, 151, 160.
long_options: 293* 294* 295* 296* 297* 298* 300*.
loop: 12.
m: 61, 293*.
make_binary_file: 293*.
make_font_name: 63*, 95* 139*.
make_font_name_end: 63*.
make_font_res: 64.
make_font_res_end: 64.
make_name: 63*, 64, 67*.
make_packet: 36, 40, 44, 47, 89, 132, 150.
make_res: 64, 66*.
make_width: 73, 105.
mark_error: 21, 86, 89, 98.
mark_fatal: 21, 23*.
mark_harmless: 21, 98, 290.
match: 186.
max_chars: 5*, 76, 102, 105, 242.
max_c1: 119.
max_font_type: 4, 79.
max_h_drift: 203, 259.
max_packets: 5* 31, 38, 40, 47, 242.
max_pages: 185, 187, 190, 204, 207, 293*.
max_par: 116.
max_pix_value: 198.
max_recursion: 5* 219, 227, 228, 242.
max_select: 5* 185, 188, 190.
max_v_drift: 203, 259.
max_widths: 5* 70, 73, 242.
mem: 175.
missing_character_packet...: 86.
move: 155.
move_zero: 151, 164, 165.
my_name: 1*, 3* 293* 299*.
name: 293* 294* 295* 296* 297* 298* 300*.
negative: 179.
n_new_line: 11* 60, 94* 98, 109* 228.
n_new_packet: 47, 152, 180, 204, 209, 230, 261*.
nf: 79, 80, 82, 107, 132, 150, 242.
nil: 12.
not_found: 10, 88, 139* 151.
num_select: 185, 204, 207.
nuu: 218.
nnum4: 218.
nnum4_par: 116, 118, 127, 145.
o: 57, 58, 59, 253, 254.
opt separator: 175, 176* 177, 180.
optary: 293*, 299*.
optimization: 7* 50, 54, 97, 103, 111, 140, 248*.
optind: 293*.
option_index: 293*.
ord: 17.
othercases: 13.
others: 13.
out_back: 244, 245, 263, 291.
out_byte: 248* 249, 250, 251, 258.
out_char: 252, 258.
out_file: 1* 3* 244, 246* 247, 248* 250, 251, 255,
261, 290, 293*.
out_fnt: 255, 263, 288.
out_four: 251, 258, 261* 263, 279, 281, 291.
out_loc: 244, 245, 249, 250, 251, 258, 263, 290, 291.
out_mag: 61, 185, 187, 190, 204, 230, 261* 291, 293*.
out_max_h: 244, 245, 274, 291.
out_max_v: 244, 245, 277, 291.
out_one: 249, 252, 253, 254, 261* 263, 265, 267,
269, 273, 276, 279, 281, 289, 290, 291.
out_packet: 250, 261, 271.
out_pages: 244, 245, 263, 290, 291.
out_signed: 254, 273, 276.
out_stack: 244, 245, 267, 291.
out_unsigned: 253, 258, 271, 288.
output: 3* 11* 176*.
stack: 193, 208.

stack_index: 192, 193.
stack_pointer: 157, 192, 193.
stack_ptr: 157, 193, 205, 207, 208, 267, 289.
stack_record: 192, 193.
stack_size: 5*162, 192, 232, 242.
stack_used: 157, 159, 162, 242.
start_count: 185, 186, 189, 299*.
start_match: 186, 206, 234.
start_there: 185, 186, 189, 299*.
start_vals: 185, 186, 189, 190, 299*.
stat: 8.
stderr: 23*293*299*
stdin: 293*.
stdout: 176*293*.
str_bytes: 34, 45, 46.
str_chars: 45, 46, 102, 105.
str_fonts: 45, 46, 94*107, 132, 150, 216, 285, 288.
str_magn: 185, 190, 191.
str_name_length: 45, 46.
str_packets: 40, 45, 46, 47, 49, 88, 225.
str_recursion: 45, 46, 228.
str_select: 185, 190, 191.
str_stack: 45, 46, 162, 232.
str_widths: 45, 46, 73.
strmp: 293*.
strtol: 299*.

substituted character packet...: 86.
system dependencies: 2*3*7*9, 13, 15*23*27, 28, 31, 50, 54, 61, 63*64, 67*95*97, 101, 103, 111, 112*140, 175, 176*180, 202, 240, 241*243, 248*.

t: 88.
tats: 8.
temp_byte: 151, 152, 229, 230, 233, 237.
temp_pix: 199, 203.
term_out: 11*293*301*.
terminal_line_length: 5*176*.
text: 301*.
text_char: 15*17, 177.
text_file: 15*.
TFM files: 68.

TFM file can’t be opened: 95*.
tfm_byte: 97.
tfm_b0: 96, 97, 101, 102, 103, 105, 143, 144.
tfm_b01: 101.
tfm_b1: 96, 97, 101, 103, 105, 143, 144.
tfm_b2: 96, 97, 101, 103, 105, 143, 144.
tfm_b3: 96, 97, 101, 103, 105, 143, 144.
tfm_conv: 100, 101, 150, 152, 230.

TFM_default_area: 95*.
TFM_default_area_name_length: 95*
tfm_ext: 90, 91*, 95*
tfm_file: 90, 95*, 97, 99.
tfm_fix1: 103, 143.
tfm_fix2: 103, 143.
tfm_fix3: 103, 143.
tfm_fix3u: 103, 143, 144.
tfm_fix4: 103, 105, 143.
tfm_squad: 101.
tfm_uquad: 101.

This can’t happen: 24*
three_cases: 115, 161, 218.
title: 1*, 25*.
twentyfour_bits: 27.
two_cases: 115, 127, 161.
type_flag: 83, 84, 88.
type_setting: 183, 184, 205, 206, 207, 215, 235.
u: 98.
ucharcast: 261*. uexit: 23* 299*
upd: 57, 252.
update_terminal: 176*.
usage: 293*. usage_help: 293*.

Use DVItype: 109*
Use TFtoPL/PLtoTF: 94*
Use VFtoVP/VPtoVF: 136.
v: 193.
v_conv: 202, 204, 259.
v_file: 192, 193, 194.
v_pixel_round: 202, 203, 211, 259.
v_resolution: 204, 259.
v_rule_pixels: 202, 213.
v_upd_end: 203.
v_upd_move: 203, 211.

version_string: 3*.
vf_byte: 140, 141, 143, 144.
vf_char_type: 158, 159, 165.
vf_complex: 160, 173.
VF_default_area: 139*.
VF_default_area_name_length: 139*.
vf_do_font: 150, 153.
vf_e_fnts: 146, 149.

vf_eof: 140.

vf_ext: 134, 135*, 139*.
vf_file: 134, 139*, 140, 141, 151.
vf_first_par: 145, 161.
vf_fixp: 144, 150.
vf_fix1: 143, 145.
vf_fix2: 143, 145.
vf_fix3: 143, 145.
vf_fix3u: 143, 160.
vf_font: 151, 161, 164, 165.
vf_font_type: 4, 136, 146, 151, 226, 238.
v_group: 156, 168.
v_i_fnts: 146, 148, 150, 153.
v_id: 133, 152.
v_move: 157, 159, 161, 163*.
v_move_assign: 163*, 170*.
v_nf: 146, 148, 149, 150, 153.
v_pquad: 144, 150, 152.
v_push_num: 156, 157, 162, 168, 170*, 172.
v_put: 156, 159, 160.
v_rule: 156, 159, 168.
v_rule_type: 158, 159, 166.
v_set: 156, 159.
v_simple: 160, 173, 226.
v_squad: 141, 144, 145, 150, 152, 153.
v_strio: 141, 160.
v_type: 156, 157, 158.
v_upair: 141, 145, 153.
v_uquad: 144, 145, 160.
v_utrio: 141, 145, 153.
v_wp: 151, 160.
visible: 213, 280.
vv: 200, 203, 259.
vv_field: 200, 201.
w: 73, 99, 193.

w_hash: 70, 71, 72, 75.
w_link: 70, 71, 72, 75.
w_x_field: 192, 193, 194.
WEB: 33.

width_dimen: 173, 174, 236, 279.

width_pointer: 70, 71, 73, 76, 77, 98, 99, 125, 151, 222.

widths: 70, 71, 72, 73, 75, 81, 98, 99, 214, 216, 222.

wp: 98.

write: 11*.

write_in: 11*, 23*, 293*, 299*

w0: 26, 121, 154, 273.

w0_cl: 119, 121, 161, 210, 225, 236.

w1: 26, 118, 121, 122, 123.

x: 54, 55, 56, 58, 59, 73, 114, 143, 144, 179, 193, 251, 253, 254.

x_cl: 119, 121, 122, 123.

xchr: 17, 18, 19, 48, 60, 61, 66, 67*, 177, 180.

xclause: 12.

xfseek: 112*.

xftell: 112*.

xmalloc_array: 67*.

xord: 17, 19, 64, 176*, 261*.

xx: 59.

xxx: 58, 155, 164, 253.

xxx_cl: 119, 121, 164, 225, 236, 237.

xxx1: 26, 118, 121, 167, 271.

xxx4: 26.

x0: 26, 121, 154, 273.

x0_cl: 119, 121.

x1: 26, 118, 121, 122, 123.

x4: 173.

y: 193.

y_cl: 119, 121, 123, 161, 211.

y_zfield: 192, 193, 194.

y0: 26, 121, 154, 276.

y0_cl: 119, 121, 161, 211, 225, 236.

y1: 26, 118, 121, 122, 123.

z: 103, 193.

z_cl: 119, 121, 122, 123.

zero: 155.

zero_stack: 193, 194, 201, 205.

zero_width: 72.

z0: 26, 121, 154, 276.

z0_cl: 119, 121.

zl: 26, 118, 121, 122, 123.

z4: 155.
(Action procedures for *dialog* 176*, 178, 179, 189*) Used in section 180.
(Basic printing procedures 48, 60, 61, 181) Used in section 23*.
(Cases for options 190) Used in section 180.
(Cases for *bad_font* 136) Used in section 94*.
(Close output file(s) 247) Used in section 240.
(Compare packet *p* with current packet, *goto* found if equal 43) Used in section 42.
(Compiler directives 9) Used in section 3*.
(Compute the packet hash code *h* 41) Used in section 40.
(Compute the packet location *p* 42) Used in section 40.
(Compute the width hash code *h* 74) Used in section 73.
(Compute the width location *p*, *goto* found unless the value is new 75) Used in section 73.
(Constants in the outer block 5*) Used in section 3*.
(DVI: Find the postamble 233) Used in section 232.
(DVI: Find the starting page 234) Used in section 232.
(DVI: Locate font *cur_parm* 131) Used in sections 130 and 132.
(DVI: Process a page; then *goto* done 236) Used in section 235.
(DVI: Process one page 235) Used in section 229.
(DVI: Process the postamble 232) Used in section 229.
(DVI: Process the preamble 230) Used in section 229.
(DVI: Skip a page; then *goto* done 237) Used in section 235.
(DVI: Typeset a *char* 238) Used in section 236.
(Declare device dependent font data arrays 195) Used in section 81.
(Declare device dependent types 198) Used in section 192.
(Declare typesetting procedures 250, 251, 252, 253, 254, 258) Used in section 182.
(Define the option table 294*, 295*, 296*, 297*, 298*, 300*) Used in section 293*.
(Define *parse_arguments* 293*) Used in section 3*.
(Determine the desired *start_count* values from *optarg* 299*) Used in section 293*.
(Determine whether this page should be processed or skipped 206) Used in section 205.
(Device dependent stack record fields 200) Used in section 192.
(Error handling procedures 23*, 24*, 25*, 94*, 109*) Used in section 3*.
(Finish output file(s) 215) Used in section 240.
(Globals in the outer block 2*, 17, 21, 32, 37, 46, 49, 62*, 65, 71, 77, 80, 81, 84, 90, 96, 100, 108*, 117, 120, 122, 124, 125, 128, 134, 142, 146, 157, 158, 173, 177, 183, 185, 193, 199, 220, 231, 244, 255, 259, 301*) Used in section 3*.
(Initialize device dependent data for a font 197) Used in section 99.
(Initialize device dependent font data 196) Used in section 82.
(Initialize device dependent stack record fields 201) Used in section 194.
(Initialize options 187) Used in sections 180 and 293*.
(Initialize predefined strings 45, 91*, 135*, 191) Used in section 241*.
(Local variables for initialization 16, 39) Used in section 3*.
(Locate a character packet and *goto* found if found 87) Used in sections 86 and 88.
(OUT: Declare additional local variables for *do_font* 283) Used in section 216.
(OUT: Declare additional local variables for *do_xx* 270) Used in section 209.
(OUT: Declare additional local variables *do_bop* 262) Used in section 205.
(OUT: Declare additional local variables *do_rule* 280) Used in section 213.
(OUT: Declare local variables (if any) for *do_char* 287) Used in section 214.
(OUT: Declare local variables (if any) for *do_down* 275) Used in section 211.
(OUT: Declare local variables (if any) for *do_eop* 264) Used in section 207.
(OUT: Declare local variables (if any) for *do_pop* 268) Used in section 208.
(OUT: Declare local variables (if any) for *do_pre* 260*) Used in section 204.
(OUT: Declare local variables (if any) for *do_push* 266) Used in section 208.
(OUT: Declare local variables (if any) for *do_right* 272) Used in section 210.
(OUT: Declare local variables (if any) for *do_width* 278) Used in section 212.
(OUT: Finish incomplete page 289) Used in section 215.
(OUT: Finish output file(s) 290) Used in section 215.
(OUT: Look for a font file after trying to read the VF file 285) Used in section 216.
(OUT: Look for a font file before trying to read the VF file; if found goto done 284) Used in section 216.
(OUT: Move down 277) Used in section 211.
(OUT: Move right 274) Used in sections 210, 212, 213, and 214.
(OUT: Prepare to use font cur_fnt 286) Used in section 217.
(OUT: Process a pop 263) Used in section 205.
(OUT: Process a down or y or z 276) Used in section 211.
(OUT: Process a pop 269) Used in section 208.
(OUT: Process a push 267) Used in section 208.
(OUT: Process a right or w or x 273) Used in section 210.
(OUT: Process an eop 265) Used in section 207.
(OUT: Process an xxx 271) Used in section 209.
(OUT: Process the pre 261*) Used in section 204.
(OUT: Typeset a visible rule 281) Used in sections 213 and 282.
(OUT: Typeset a char 288) Used in section 214.
(OUT: Typeset a width 279) Used in section 212.
(OUT: Typeset an invisible rule 282) Used in section 213.
(OUT: Write the postamble 291) Used in section 290.
(Open input file(s) 110*) Used in section 241*.
(Open output file(s) 246*) Used in section 241*.
(Prepare to use font cur_fnt 217) Used in sections 226 and 238.
(Print memory usage statistics 242) Used in section 240.
(Print more font usage statistics 257) Used in section 242.
(Print more memory usage statistics 292) Used in section 242.
(Print the job history 243) Used in section 240.
(Print valid options 188) Used in section 181.
(Replace z by z’ and compute α, β 104*) Used in sections 105 and 152.
(Set initial values 18, 19, 22, 38, 72, 78, 82, 85, 118, 121, 123, 126, 129, 147, 159, 174, 175, 184, 194, 221, 245, 256) Used in section 3*.
(TFM: Convert character-width indices to character-width pointers 106) Used in section 99.
(TFM: Open tfm_file 95*) Used in section 99.
(TFM: Read and convert the width values 105) Used in section 99.
(TFM: Read past the header data 101) Used in section 99.
(TFM: Store character-width indices 102) Used in section 99.
(VF: Append DVI commands to the character packet 161) Used in section 160.
(VF: Apply rule 3 or 4 170*) Used in section 168.
(VF: Apply rule 5 172) Used in section 168.
(VF: Apply rule 6 171) Used in section 168.
(VF: Build a character packet 160) Used in section 151.
(VF: Display the recursion traceback and terminate 228) Used in section 227.
(VF: Do a char, rule, or xxx 164) Used in section 161.
(VF: Do a fnt, a char, or both 165) Used in section 164.
(VF: Do a pop 168) Used in section 161.
(VF: Do a push 162) Used in section 161.
(VF: Do a rule 166) Used in section 164.
(VF: Do an xxx 167) Used in section 164.
(VF: Enter a new recursion level 227) Used in section 226.
(VF: Interpret the DVI commands in the packet 225) Used in section 222.
(VF: Locate font cur_parm 149) Used in sections 148 and 150.
\{ VF: Open \texttt{vf\_file} or \texttt{goto not\_found} 139* \} Used in section 151.
\{ VF: Prepare for rule 4 169 \} Used in section 168.
\{ VF: Process the font definitions 153 \} Used in section 151.
\{ VF: Process the preamble 152 \} Used in section 151.
\{ VF: Restore values on exit from \texttt{do\_vf\_packet} 224 \} Used in section 222.
\{ VF: Save values on entry to \texttt{do\_vf\_packet} 223 \} Used in section 222.
\{ VF: Start a new level 163* \} Used in sections 162 and 172.
\{ VF: Typeset a \texttt{char} 226 \} Used in section 225.
\{ Variables for scaling computation 103 \} Used in sections 99 and 142.