The DVIcopy processor

(Version 1.6, September 2009)

This program was developed at the Max-Planck-Institut für Physik (Werner-Heisenberg-Institut), Munich, Germany. 'TEX' is a trademark of the American Mathematical Society. 'METAFONT' is a trademark of Addison-Wesley Publishing Company.

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 and VF 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 *out_file*, the output file receiving the copy of the input *dvi_file*. In addition all code related to *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 *banner* and *preamble_comment* strings defined here should be changed whenever DVIcopy gets modified.

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 }

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 *true* to *false*; in such cases, DVIcopy will not include the optional feature that reads the postamble first.

⟨ Globals in the outer block 2* ⟩ ≡
random_reading: *boolean*; { should we skip around in the file? }

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*.

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 *dvi_file*, the DVI output goes to file *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 '*jump_out*' procedure.

```
⟨ Compiler directives 9 ⟩
program DVI_copy(dvi_file, out_file, output);
  const ⟨ Constants in the outer block 5* ⟩
  type ⟨ Types in the outer block 7* ⟩
  var ⟨ Globals in the outer block 2* ⟩
    ⟨ Error handling procedures 23* ⟩
    ⟨ Define parse_arguments 293* ⟩
  procedure initialize; { this procedure gets things started properly }
    var ⟨ Local variables for initialization 16 ⟩
    begin kpse_set_program_name(argv[0], my_name); parse_arguments; print(banner);
    print_ln(version_string); print_ln(copyright);
    print_ln('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 DVIcopy's capacity.

```
  define max_select = 10 { maximum number of page selection ranges }
⟨ Constants in the outer block 5* ⟩ ≡
  max_fonts = 400; { maximum number of distinct fonts }
  max_chars = 750000; { maximum number of different characters among all fonts }
  max_widths = 16000; { maximum number of different characters widths }
  max_packets = 65530; { maximum number of different characters packets; must be less than 65536 }
  max_bytes = 250000; { maximum number of bytes for characters packets }
  max_recursion = 10; { VF files shouldn't recurse beyond this level }
  stack_size = 100; { DVI files shouldn't push beyond this depth }
  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 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."

```
define int_32 ≡ integer  { signed 32 bit integers }
define int_31 ≡ int_31_t
define int_24u ≡ int_24u_t
define int_24 ≡ int_24_t
define int_23 ≡ int_23_t
define int_16u ≡ int_16u_t
define int_16 ≡ int_16_t
define int_15 ≡ int_15_t
define int_8u ≡ int_8u_t
define int_8 ≡ int_8_t
define int_7 ≡ int_7_t
```

⟨ Types in the outer block 7* ⟩ ≡

```
int_31 = 0 .. "7FFFFFFF;  { unsigned 31 bit integer }
int_24u = 0 .. "FFFFFF;  { unsigned 24 bit integer }
int_24 = -"800000 .. "7FFFFF;  { signed 24 bit integer }
int_23 = 0 .. "7FFFFF;  { unsigned 23 bit integer }
int_16u = 0 .. "FFFF;  { unsigned 16 bit integer }
int_16 = -"8000 .. "7FFF;  { signed 16 bit integer }
int_15 = 0 .. "7FFF;  { unsigned 15 bit integer }
int_8u = 0 .. "FF;  { unsigned 8 bit integer }
int_8 = -"80 .. "7F;  { signed 8 bit integer }
int_7 = 0 .. "7F;  { unsigned 7 bit integer }
```

See also sections 14*, 15*, 27, 29, 31, 36, 70, 76, 79, 83, 116, 119, 154, 156, 192, and 219.

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.

```
define print(#) ≡ write(term_out,#)
define print_ln(#) ≡ write_ln(term_out,#)
define new_line ≡ write_ln(term_out)  { start new line }
define print_nl(#) ≡  { print information starting on a new line }
    begin new_line; print(#);
    end
define d_print(#) ≡
    debug print(#) gubed
define d_print_ln(#) ≡
    debug print_ln(#) gubed
```

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.

⟨ Types in the outer block 7* ⟩ +≡
  ASCII_code = 0 .. 255; { a subrange of the integers }

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 *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define *text_char* ≡ *ASCII_code* { the data type of characters in text files }
define *first_text_char* = 0 { ordinal number of the smallest element of *text_char* }
define *last_text_char* = 255 { ordinal number of the largest element of *text_char* }

⟨ Types in the outer block 7* ⟩ +≡
  text_file = **packed file of** *text_char*;

23:* 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.

```
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*.

24:* 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).

```
⟨ Error handling procedures 23* ⟩ +≡
noreturn procedure confusion(p : pckt_pointer);
    begin print(´␣!This␣can´´t␣happen␣(´); print_packet(p); print_ln(´).´); jump_out;
    end;
```

25:* An overflow stop occurs if `DVIcopy`'s tables aren't large enough.

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

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

```
⟨ Globals in the outer block 2* ⟩ +≡
cur_name: ↑char;
l_cur_name: int_15;  { this many characters are actually relevant in 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 *make_font_name*.

```
define append_to_name(#) ≡
        begin cur_name[l_cur_name] ← #; incr(l_cur_name);
        end
define make_font_name_end(#) ≡ make_name
define make_font_name(#) ≡ l_cur_name ← 0; make_font_name_end
```

67:* The *make_name* procedure used to build the external file name. The global variable *l_cur_name* contains the length of a default area which has been copied to *cur_name* before *make_name* is called.

```
procedure make_name(e : pckt_pointer);
  var b: eight_bits;  { a byte extracted from byte_mem }
    n: pckt_pointer;  { file name packet }
    cur_loc, cur_limit: byte_pointer;  { indices into byte_mem }
  device ll: int_15;  { loop index }
  ecived
  begin n ← font_name(cur_fnt); cur_name ← xmalloc_array(char, pckt_length(n) + pckt_length(e));
  cur_loc ← pckt_start[n]; cur_limit ← pckt_start[n + 1]; pckt_extract(b);  { length of area part }
  if b > 0 then l_cur_name ← 0;
  while cur_loc < cur_limit do
    begin pckt_extract(b); append_to_name(xchr[b]);
    end;
  cur_name[l_cur_name] ← 0;
  end;
```

91.* ⟨ Initialize predefined strings 45 ⟩ +≡
id4(".")("t")("f")("m")(*tfm_ext*); { 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 *bad_font*; for a TFM file the *bad_tfm* procedure is used to give an error message which refers the user to TFtoPL and PLtoTF, and terminates DVIcopy.

```
⟨ Error handling procedures 23* ⟩ +≡
noreturn procedure bad_tfm;
  begin print(´Bad␣TFM␣file´); print_font(cur_fnt); print_ln(´!´);
  abort(´Use␣TFtoPL/PLtoTF␣to␣diagnose␣and␣correct␣the␣problem´);
  end;
noreturn procedure bad_font;
  begin new_line;
  case font_type(cur_fnt) of
  defined_font: confusion(str_fonts);
  loaded_font: bad_tfm;
    ⟨ Cases for bad_font 136 ⟩
  othercases abort(´internal␣error´);
  endcases;
  end;
```

95.* To prepare *tfm_file* for input we *reset* it.
TFM_default_area_name_length and *TFM_default_area* will not be used by *make_font_name*.

```
⟨ TFM: Open tfm_file 95* ⟩ ≡
  make_font_name(TFM_default_area_name_length)(TFM_default_area)(tfm_ext);
  full_name ← kpse_find_tfm(cur_name);
  if full_name then
    begin resetbin(tfm_file, full_name); free(cur_name); free(full_name);
    end
  else abort(´---not␣loaded,␣TFM␣file␣can´´t␣be␣opened!´)
```

This code is used in section 99.

104.* ⟨ Replace z by z' and compute α, β 104* ⟩ ≡

```
  alpha ← 16;
  if z ≥ ´1000000000 then abort(´Character␣size␣is␣too␣large!´);
  while z ≥ ´40000000 do
    begin z ← z div 2; alpha ← alpha + alpha;
    end;
  beta ← 256 div alpha; alpha ← alpha * z
```

This code is used in sections 105 and 152.

108.* Low-level DVI input routines. The program uses the binary file variable *dvi_file* for its main input file; *dvi_loc* is the number of the byte about to be read next from *dvi_file*.

```
⟨ Globals in the outer block 2* ⟩ +≡
dvi_file: byte_file;  { the stuff we are DVIcopying }
dvi_loc: int_32;  { where we are about to look, in dvi_file }
full_name: ↑char;
```

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

```
⟨ Error handling procedures 23* ⟩ +≡
noreturn procedure bad_dvi;
  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;
```

110.* To prepare *dvi_file* for input, we *reset* it.

```
⟨ Open input file(s) 110* ⟩ ≡
  dvi_loc ← 0;
```

This code is used in section 241*.

112.* Next 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 *dvi_move*(*n*) should position *dvi_file* so that the next *dvi_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: int_32;
  begin xfseek(dvi_file, 0, 2, dvi_name); dvi_loc ← xftell(dvi_file, dvi_name); dvi_length ← dvi_loc;
  end;
procedure dvi_move(n : int_32);
  begin xfseek(dvi_file, n, 0, dvi_name); dvi_loc ← n;
  end;
```

135.* ⟨ Initialize predefined strings 45 ⟩ +≡
id3(".")("v")("f")(*vf_ext*); { file name extension for VF files }

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

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

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

139.* To prepare *vf_file* for input we *reset* it.

Do path searching. But the VF file may not exist.

⟨ VF: Open *vf_file* or **goto** *not_found* 139* ⟩ ≡
make_font_name(*VF_default_area_name_length*)(*VF_default_area*)(*vf_ext*);
full_name ← *kpse_find_vf*(*cur_name*);
if *full_name* **then**
begin *resetbin*(*vf_file*, *full_name*); *free*(*cur_name*); *free*(*full_name*);
end
else goto *not_found*;
vf_loc ← 0

This code is used in section 151.

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

```
define do_vf_move(#) ≡ vf_move[vf_ptr]# ← vf_move[vf_ptr − 1]#
define vf_move_assign ≡
        begin do_vf_move([0][0]); do_vf_move([0][1]); do_vf_move([1][0]); do_vf_move([1][1])
        end
```

⟨ VF: Start a new level 163* ⟩ ≡

```
append_one(push); vf_move_assign; vf_push_loc[vf_ptr] ← byte_ptr; vf_last_end[vf_ptr] ← byte_ptr;
vf_last[vf_ptr] ← vf_other
```

This code is used in sections 162 and 172.

170:* ⟨ VF: Apply rule 3 or 4 170* ⟩ ≡

```
begin if vf_push_num[vf_ptr] > 0 then
  begin decr(vf_push_num[vf_ptr]); vf_move_assign;
  end
else begin decr(byte_ptr); decr(vf_ptr);
  end;
if cur_class ≠ pop_cl then goto reswitch;   { this is rule 4 }
end
```

This code is used in section 168.

176:* 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.

```
define update_terminal ≡ fflush(stdout)  { empty the terminal output buffer }
define scan_blank(#) ≡  { tests for 'blank' when scanning (command line) options }
        ((byte_mem[#] = bi("␣")) ∨ (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("␣"); scan_ptr ← pckt_start[pckt_ptr − 1]; scan_skip
```

⟨ Action procedures for *dialog* 176* ⟩ ≡

```
procedure input_ln;  { inputs a line from the terminal }
  var k: 0 .. terminal_line_length;
  begin print('Enter␣option:␣'); update_terminal;  { if eoln(input) then read_ln(input); }
  k ← 0; pckt_room(terminal_line_length);
  while (k < terminal_line_length) ∧ ¬eoln(input) do
    begin append_byte(xord[getc(input)]); incr(k);
    end;
  end;
```

See also sections 178, 179, and 189.

This code is used in section 180.

241:* Now we are ready to put it all together. Here is where `DVIcopy` starts, and where it ends.

```
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.
```

246:* To prepare *out_file* for output, we *rewrite* it.

⟨ Open output file(s) 246* ⟩ ≡

This code is used in section 241*.

248:* 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 }

260:* These are the local variables (if any) needed for *do_pre*.

```
⟨ OUT: Declare local variables (if any) for do_pre 260* ⟩ ≡
var k: int_15; { general purpose variable }
  p, q, r: byte_pointer; { indices into byte_mem }
  comment: const_c_string; { preamble comment prefix }
```

This code is used in section 204.

261:* 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.

```
⟨ OUT: Process the pre 261* ⟩ ≡
  out_one(pre); out_one(dvi_id); out_four(dvi_num); out_four(dvi_den); out_four(out_mag);
  p ← pckt_start[pckt_ptr − 1]; q ← byte_ptr; { location of old DVI comment }
  comment ← preamble_comment; pckt_room(comm_length);
  for k ← 0 to comm_length − 1 do append_byte(xord[ucharcast(comment[k])]);
  while byte_mem[p] = bi("␣") do incr(p); { remove leading blanks }
  if p = q then Decr(byte_ptr)(from_length)
  else begin k ← 0;
    while (k < comm_length) ∧ (byte_mem[p + k] = byte_mem[q + k]) do incr(k);
    if k = comm_length then Incr(p)(comm_length);
    end;
  k ← byte_ptr − p; { total length }
  if k > 255 then
    begin k ← 255; q ← p + 255 − comm_length; { at most 255 bytes }
    end;
  out_one(k); out_packet(new_packet); flush_packet;
  for r ← p to q − 1 do out_one(bo(byte_mem[r]));
```

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*.

```
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
```

```
⟨ Define parse_arguments 293* ⟩ ≡
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;
    end_num: ↑char;
  begin ⟨ Define the option table 294* ⟩;
  ⟨ Initialize options 187 ⟩;
  repeat getopt_return_val ← getopt_long_only(argc, argv, ´´, long_options, address_of(option_index));
    if getopt_return_val = −1 then
      begin do_nothing;  { End of arguments; we exit the loop below. }
      end
    else if getopt_return_val = "?" then
        begin usage(my_name);
        end
      else if argument_is(´help´) then
          begin usage_help(DVICOPY_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(banner, ´Peter␣Breitenlohner´, nil, nil);
            end
          else if argument_is(´magnification´) then
              begin out_mag ← atou(optarg);
              end
            else if argument_is(´max-pages´) then
                begin max_pages ← atou(optarg); incr(cur_select);
                end
              else if argument_is(´page-start´) then
                  begin ⟨ Determine the desired start_count values from optarg 299* ⟩;
                  end;  { Else it was a flag; getopt has already done the assignment. }
  until getopt_return_val = −1;  { Now optind is the index of first non-option on the command line. We
        can have zero, one, or two remaining arguments. }
  if (optind > argc) ∨ (optind + 2 < argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣at␣most␣two␣file␣arguments.´); usage(my_name);
    end;
  if optind = argc then
    begin dvi_name ← ´<stdin>´; dvi_file ← make_binary_file(stdin); random_reading ← false;
    end
  else begin dvi_name ← extend_filename(cmdline(optind), ´dvi´); resetbin(dvi_file, dvi_name);
    random_reading ← true;
    end;
  if optind + 2 = argc then
    begin rewritebin(out_file, extend_filename(cmdline(optind + 1), ´dvi´)); term_out ← stdout;
    end
  else begin out_file ← make_binary_file(stdout); term_out ← stderr;
    end;
  end;
```

This code is used in section 3*.

294.* Here is the first of the options we allow.

```
⟨ Define the option table 294* ⟩ ≡
  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);
```

See also sections 295*, 296*, 297*, 298*, and 300*.

This code is used in section 293*.

295.* Another of the standard options.

```
⟨ Define the option table 294* ⟩ +≡
  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);
```

296.* Magnification to apply.

```
⟨ Define the option table 294* ⟩ +≡
  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);
```

297.* How many pages to do.

```
⟨ Define the option table 294* ⟩ +≡
  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);
```

298.* What page to start at.

```
⟨ Define the option table 294* ⟩ +≡
  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);
```

299.* Parsing the starting page specification is a bit complicated. (This is the same as in `DVItype`.)

```
⟨ Determine the desired start_count values from optarg 299* ⟩ ≡
  k ← 0;  { which \count register we're on }
  m ← 0;  { position in optarg }
  while optarg[m] do
    begin if optarg[m] = "*" then
      begin start_there[k] ← false; incr(m);
      end
    else if optarg[m] = "." then
        begin incr(k);
        if k ≥ 10 then
          begin write_ln(stderr, my_name, ': More than ten count registers specified.');
          uexit(1);
          end;
        incr(m);
        end
      else begin start_count[k] ← strtol(optarg + m, address_of(end_num), 10);
        if end_num = optarg + m then
          begin write_ln(stderr, my_name, ': -page-start values must be numeric or *.');
          uexit(1);
          end;
        start_there[k] ← true; m ← m + end_num − (optarg + m);
        end;
    end;
  start_vals ← k; selected ← false;
```

This code is used in section 293*.

300.* An element with all zeros always ends the list.

```
⟨ Define the option table 294* ⟩ +≡
  long_options[current_option].name ← 0; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0;
```

301.* ⟨ Globals in the outer block 2* ⟩ +≡

```
term_out: text;
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.

⟨ VF: Open *vf_file* or **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 *do_vf_packet* 224 ⟩ Used in section 222.
⟨ VF: Save values on entry to *do_vf_packet* 223 ⟩ Used in section 222.
⟨ VF: Start a new level 163* ⟩ Used in sections 162 and 172.
⟨ VF: Typeset a *char* 226 ⟩ Used in section 225.
⟨ Variables for scaling computation 103 ⟩ Used in sections 99 and 142.