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.
'TEX' is a trademark of the American Mathematical Society.

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 *my_name* ≡ `'dvitype'`
define *banner* ≡ `'This␣is␣DVItype,␣Version␣3.6'` { printed when the program starts }

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*(*stdout*, #)
define *print_ln*(#) ≡ *write_ln*(*stdout*, #)

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 ⟩
⟨ Define *parse_arguments* 112* ⟩
procedure *initialize*; { this procedure gets things started properly }
var *i*: *integer*; { loop index for initializations }
begin *kpse_set_program_name*(*argv*[0], *my_name*); *parse_arguments*; *print*(*banner*);
print_ln(*version_string*); ⟨ Set initial values 11 ⟩
end;

4* Label *done* is used when stopping normally.

define *done* = 30 { go here when finished with a subtask }

⟨ Labels in the outer block 4* ⟩ ≡
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 = 500;  { maximum number of distinct fonts per DVI file }
  max_widths = 25000;  { maximum number of different characters among all fonts }
  line_length = 79;  { bracketed lines of output will be at most this long }
  stack_size = 100;  { DVI files shouldn't push beyond this depth }
  name_size = 10000;  { total length of all font file names }
```

This code is used in section 3*.

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.

```
define jump_out ≡ uexit(1)
define abort(#) ≡
        begin write_ln(stderr, #); jump_out;
        end
define bad_dvi(#) ≡ abort('Bad␣DVI␣file:␣', #, '!')
```

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 = 0 . . 255; { 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* ≡ *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 8* ⟩ +≡
  text_file = **packed file of** *text_char*;

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 resetbin(dvi_file, dvi_name); cur_loc ← 0;
  end;
procedure open_tfm_file; { prepares to read packed bytes in tfm_file }
  var full_name: ↑char;
  begin full_name ← kpse_find_tfm(cur_name);
  if full_name then
    begin tfm_file ← fopen(full_name, FOPEN_RBIN_MODE);
    end
  else begin tfm_file ← nil;
    end;
  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: ↑char; { external name }
```

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 xfseek(dvi_file, 0, 2, dvi_name); cur_loc ← xftell(dvi_file, dvi_name); dvi_length ← cur_loc;
  end;
procedure move_to_byte(n : integer);
  begin xfseek(dvi_file, n, 0, dvi_name); cur_loc ← n;
  end;
```

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.* Initializations are done sooner now.

45.* No dialog.

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*.

48.* No dialog.

49.* No dialog.

50.* No dialog (50).

51.* No dialog (51).

52.* No dialog (52).

53.* No dialog (53).

54.* No dialog (54).

55.* No dialog (55).

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('␣␣Resolution␣=␣'); print_real(resolution, 12, 8); print_ln('␣pixels␣per␣inch');
  if new_mag > 0 then
    begin print('␣␣New␣magnification␣factor␣=␣'); print_real(new_mag/1000.0, 8, 3); print_ln('')
    end
```

This code is used in section 107*.

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_size;  { current filename length }
    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;
```

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(' ');
if tfm_file then xfclose(tfm_file, cur_name);  { should be the kpse_find_tfm result }
free(cur_name);  { We xmalloc'd this before we got called. }
end
```

This code is used in section 59*.

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.

Under Unix, users have a path searched for fonts, there's no single default directory.

65:* (No initialization needs to be done. Keep this module to preserve numbering.)

66.* The string *cur_name* is supposed to be set to the external name of the `TFM` file for the current font. We do not impose a maximum limit here. It's too bad there is a limit on the total length of all filenames, but it doesn't seem worth reprogramming all that.

```
define name_start ≡ font_name[nf]
define name_end ≡ font_name[nf + 1]
```

⟨ Move font name into the *cur_name* string 66* ⟩ ≡

```
r ← name_end − name_start; cur_name ← xmalloc_array(char, r);
    { strncpy might be faster, but it's probably a good idea to keep the xchr translation. }
for k ← name_start to name_end do
  begin cur_name[k − name_start] ← xchr[names[k]];
  end;
cur_name[r] ← 0; { Append null byte for C. }
```

This code is used in section 62*.

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;
  othercases abort('internal␣error');
  endcases;
  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, ':␣', #);
        if show_opcodes ∧ (o ≥ 128) then print('␣{', o : 1, '}');
        end
define major(#) ≡
        if out_mode > errors_only then show(#)
define minor(#) ≡
        if out_mode > terse then
          begin showing ← true; print(a : 1, ':␣', #);
          if show_opcodes ∧ (o ≥ 128) then print('␣{', o : 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 $h \leftarrow h + q$, 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.

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));
      if out_mode ≠ errors_only then print_ln('␣');
      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.

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 }
⟨ Print all the selected options 56* ⟩;
⟨ 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;
end.
```

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('magnification=', mag : 1, ';␣');
  print_real(conv, 16, 8); print_ln('␣pixels␣per␣DVI␣unit')
```

This code is used in section 109.

112.* System-dependent changes. Parse a Unix-style command line.

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

```
⟨ Define parse_arguments 112* ⟩ ≡
procedure parse_arguments;
  const n_options = 8;  { 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; end_num: ↑char;
        { for page-start }
  begin ⟨ Define the option table 113* ⟩;
  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(DVITYPE_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(banner, nil, ´D.E.␣Knuth´, nil);
            end
          else if argument_is(´output-level´) then
              begin if (optarg[0] < ´0´) ∨ (optarg[0] > ´4´) ∨ (optarg[1] ≠ 0) then
                begin write_ln(stderr, ´Value␣for␣--output-level␣must␣be␣>=␣0␣and␣<=␣4.´);
                uexit(1);
                end;
              out_mode ← optarg[0] − ´0´;
              end
            else if argument_is(´page-start´) then
                begin ⟨ Determine the desired start_count values from optarg 117* ⟩;
                end
              else if argument_is(´max-pages´) then
                  begin max_pages ← atou(optarg);
                  end
                else if argument_is(´dpi´) then
                    begin resolution ← atof(optarg);
                    end
                  else if argument_is(´magnification´) then
                      begin new_mag ← atou(optarg);
                      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. }
  if (optind + 1 ≠ argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣exactly␣one␣file␣argument.´); usage(my_name);
    end;
  dvi_name ← extend_filename(cmdline(optind), ´dvi´);
  end;
```

This code is used in section 3*.

113.* Here are the options we allow. The first is one of the standard GNU options.

```
⟨ Define the option table 113* ⟩ ≡
  current_option ← 0; long_options[current_option].name ← ´help´;
  long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
  long_options[current_option].val ← 0; incr(current_option);
```

See also sections 114*, 115*, 116*, 118*, 119*, 120*, 121*, and 123*.

This code is used in section 112*.

114.* Another of the standard options.

```
⟨ Define the option table 113* ⟩ +≡
  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);
```

115.* How verbose to be.

```
⟨ Define the option table 113* ⟩ +≡
  long_options[current_option].name ← ´output-level´; long_options[current_option].has_arg ← 1;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
  out_mode ← the_works; { default }
```

116.* What page to start at.

```
⟨ Define the option table 113* ⟩ +≡
  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);
```

117.* Parsing the starting page specification is a bit complicated.

```
⟨ Determine the desired start_count values from optarg 117* ⟩ ≡
  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;
```

This code is used in section 112*.

118.* How many pages to do.

⟨ Define the option table 113* ⟩ +≡
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*);
max_pages ← 1000000; { default }

119.* Resolution, in pixels per inch.

⟨ Define the option table 113* ⟩ +≡
long_options[*current_option*].*name* ← ´`dpi`´; *long_options*[*current_option*].*has_arg* ← 1;
long_options[*current_option*].*flag* ← 0; *long_options*[*current_option*].*val* ← 0; *incr*(*current_option*);
resolution ← 300.0; { default }

120.* Magnification to apply.

⟨ Define the option table 113* ⟩ +≡
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*);
new_mag ← 0; { default is to keep the old one }

121.* Whether to show numeric opcodes.

⟨ Define the option table 113* ⟩ +≡
long_options[*current_option*].*name* ← ´`show-opcodes`´; *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← *address_of*(*show_opcodes*); *long_options*[*current_option*].*val* ← 1;
incr(*current_option*);

122.* ⟨ Globals in the outer block 10 ⟩ +≡
show_opcodes: *c_int_type*;

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

⟨ Define the option table 113* ⟩ +≡
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;

124.* Global filenames.

⟨ Globals in the outer block 10 ⟩ +≡
dvi_name: *const_c_string*;

125.* 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, 3, 4, 5, 7, 8, 9, 23, 24, 28, 42, 43, 45, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 59, 62, 64, 65, 66, 75, 80, 99, 107, 110, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125.