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 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.

```
define random_reading ≡ true { should we skip around in the file? }
```

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, which goes to the label *final_end*.

```
  define final_end = 9999  { go here to wrap it up }
⟨ Compiler directives 9 ⟩
program DVI_copy(dvi_file, out_file, output);
  label final_end;
  const ⟨ Constants in the outer block 5 ⟩
  type ⟨ Types in the outer block 7 ⟩
  var ⟨ Globals in the outer block 17 ⟩
    ⟨ Error handling procedures 23 ⟩
  procedure initialize;  { this procedure gets things started properly }
    var ⟨ Local variables for initialization 16 ⟩
    begin print_ln(banner);
    print_ln(copyright); print_ln(´Distributed␣under␣terms␣of␣GNU␣General␣Public␣License´);
    ⟨ Set initial values 18 ⟩
    end;
```

4. The definition of *max_font_type* should be adapted to the number of font types used by the program; the first three values have a fixed meaning: *defined_font* = 0 indicates that a font has been defined, *loaded_font* = 1 indicates that the `TFM` file has been loaded but the font has not yet been used, and *vf_font_type* = 2 indicates a virtual font. Font type values $\geq$ *real_font* = 3 indicate real fonts and different font types are used to distinguish various kinds of font files (`GF` or `PK` or `PXL`). `DVIcopy` uses *out_font_type* = 3 for fonts that appear in the output `DVI` file.

```
  define defined_font = 0  { this font has been defined }
  define loaded_font = 1  { this font has been defined and loaded }
  define vf_font_type = 2  { this font is a virtual font }
  define real_font = 3  { smallest font type for real fonts }
  define out_font_type = 3  { this font appears in the output file }
  define max_font_type = 3
```

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 = 100;  { maximum number of distinct fonts }
  max_chars = 10000;  { maximum number of different characters among all fonts }
  max_widths = 3000;  { maximum number of different characters widths }
  max_packets = 5000;  { maximum number of different characters packets; must be less than 65536 }
  max_bytes = 30000;  { 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 = 150;
      { maximum number of characters input in a single line of input from the terminal }
  name_length = 50;  { a file name shouldn't be longer than this }
```

This code is used in section 3.

6. As mentioned above, `DVIcopy` has two chief purposes: (1) It produces a copy of the input `DVI` file with all references to characters from virtual fonts replaced by their expansion as specified in the character packets of `VF` files; 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.

In fact, a very large section of code (starting with the second chapter 'Introduction (continued)' and ending with the fifteenth chapter 'The main program') is used in identical form in `DVIcopy` and in `DVIprint`, a prototype `DVI`-driver. This has been made possible mostly by using several `WEB` coding tricks, such as not to make the resulting Pascal program inefficient in any way.

Parts of the program that are needed in `DVIprint` but not in `DVIcopy` are delimited by the code words '**device** . . . **ecived**'; these are mostly the pixel rounding algorithms used to convert the `DVI` units of a `DVI` file to the raster units of a real output device and have been copied more or less verbatim from `DVItype`.

define *device* ≡ `@{` { change this to '*device* ≡ ' when output for a real device is produced }
define *ecived* ≡ `@}` { change this to '*ecived* ≡ ' when output for a real device is produced }
format *device* ≡ *begin*
format *ecived* ≡ *end*

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 $0 \mathinner{..} 255$: 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 }

⟨ 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.

8. Some of this code is optional for use when debugging only; such material is enclosed between the delimiters **debug** and **gubed**. Other parts, delimited by **stat** and **tats**, are optionally included if statistics about `DVIcopy`'s memory usage are desired.

define *debug* ≡ `@{` { change this to '*debug* ≡ ' when debugging }
define *gubed* ≡ `@}` { change this to '*gubed* ≡ ' when debugging }
format *debug* ≡ *begin*
format *gubed* ≡ *end*

define *stat* ≡ `@{` { change this to '*stat* ≡ ' when gathering usage statistics }
define *tats* ≡ `@}` { change this to '*tats* ≡ ' when gathering usage statistics }
format *stat* ≡ *begin*
format *tats* ≡ *end*

9. The Pascal compiler used to develop this program has "compiler directives" that can appear in comments whose first character is a dollar sign. In production versions of `DVIcopy` these directives tell the compiler that it is safe to avoid range checks and to leave out the extra code it inserts for the Pascal debugger's benefit, although interrupts will occur if there is arithmetic overflow.

⟨ Compiler directives 9 ⟩ ≡
`@{@&$`$C-, A+, D-$`@}` { no range check, catch arithmetic overflow, no debug overhead }
debug `@{@&$`$C+, D+$`@}` **gubed** { but turn everything on when debugging }

This code is used in section 3.

10. Labels are given symbolic names by the following definitions. We insert the label '*exit*:' just before the '**end**' of a procedure in which we have used the '**return**' statement defined below; the label '*restart*' is occasionally used at the very beginning of a procedure; and the label '*reswitch*' is occasionally used just prior to a **case** statement in which some cases change the conditions and we wish to branch to the newly applicable case. Loops that are set up with the **loop** construction defined below are commonly exited by going to '*done*' or to '*found*' or to '*not_found*', and they are sometimes repeated by going to '*continue*'.

```
define exit = 10  { go here to leave a procedure }
define restart = 20  { go here to start a procedure again }
define reswitch = 21  { go here to start a case statement again }
define continue = 22  { go here to resume a loop }
define done = 30  { go here to exit a loop }
define found = 31  { go here when you've found it }
define not_found = 32  { go here when you've found something else }
```

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(output, #)
define print_ln(#) ≡ write_ln(output, #)
define new_line ≡ write_ln(output)  { 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
```

12. Here are some macros for common programming idioms.

```
define incr(#) ≡ # ← # + 1  { increase a variable by unity }
define decr(#) ≡ # ← # − 1  { decrease a variable by unity }
define Incr_Decr_end(#) ≡ #
define Incr(#) ≡ # ← # + Incr_Decr_end  { we use Incr(a)(b) to increase ... }
define Decr(#) ≡ # ← # − Incr_Decr_end
          { ... and Decr(a)(b) to decrease variable a by b; this can be optimized for some compilers }
define loop ≡ while true do  { repeat over and over until a goto happens }
define do_nothing ≡  { empty statement }
define return ≡ goto exit  { terminate a procedure call }
format return ≡ nil
format loop ≡ xclause
```

13. We assume that **case** statements may include a default case that applies if no matching label is found. Thus, we shall use constructions like

```
case $x$ of
1: ⟨ code for $x = 1$ ⟩;
3: ⟨ code for $x = 3$ ⟩;
othercases ⟨ code for $x \neq 1$ and $x \neq 3$ ⟩
endcases
```

since most Pascal compilers have plugged this hole in the language by incorporating some sort of default mechanism. For example, the compiler used to develop `WEB` and TeX allows '*others*:' as a default label, and other Pascals allow syntaxes like '**else**' or '**otherwise**' or '*otherwise*:', etc. The definitions of **othercases** and **endcases** should be changed to agree with local conventions. (Of course, if no default mechanism is available, the **case** statements of this program must be extended by listing all remaining cases. Donald E. Knuth, the author of the `WEB` system program `TANGLE`, would have taken the trouble to modify `TANGLE` so that such extensions were done automatically, if he had not wanted to encourage Pascal compiler writers to make this important change in Pascal, where it belongs.)

define *othercases* ≡ *others*: { default for cases not listed explicitly }
define *endcases* ≡ **end** { follows the default case in an extended **case** statement }
format *othercases* ≡ *else*
format *endcases* ≡ *end*

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

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

16. ⟨ Local variables for initialization 16 ⟩ ≡
i: *int_16*; { loop index for initializations }

See also section 39.

This code is used in section 3.

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

⟨ Globals in the outer block 17 ⟩ ≡
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [0 .. 255] **of** *text_char*; { specifies conversion of output characters }

See also sections 21, 32, 37, 46, 49, 62, 65, 71, 77, 80, 81, 84, 90, 92, 96, 100, 108, 117, 120, 122, 124, 125, 128, 134, 137, 142, 146, 157, 158, 173, 177, 183, 185, 193, 199, 220, 231, 244, 255, and 259.

This code is used in section 3.

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

⟨ Set initial values 18 ⟩ ≡

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

See also sections 19, 22, 38, 72, 78, 82, 85, 93, 118, 121, 123, 126, 129, 138, 147, 159, 174, 175, 184, 194, 221, 245, and 256.

This code is used in section 3.

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

⟨ Set initial values 18 ⟩ +≡

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

20. Reporting errors to the user. The `DVIcopy` processor does not verify that every single bit read from one of its binary input files is meaningful and consistent; there are other programs, e.g., `DVItype`, `TFtoPL`, and `VFtoPL`, specially designed for that purpose.

On the other hand, `DVIcopy` is designed to avoid unpredictable results due to undetected arithmetic overflow, or due to violation of integer subranges or array bounds under *all* circumstances. Thus a fair amount of checking is done when reading and analyzing the input data, even in cases where such checking reduces the efficiency of the program to some extent.

21. A global variable called *history* will contain one of four values at the end of every run: *spotless* means that no unusual messages were printed; *harmless_message* means that a message of possible interest was printed but no serious errors were detected; *error_message* means that at least one error was found; *fatal_message* means that the program terminated abnormally. The value of *history* does not influence the behavior of the program; it is simply computed for the convenience of systems that might want to use such information.

```
define spotless = 0   { history value for normal jobs }
define harmless_message = 1   { history value when non-serious info was printed }
define error_message = 2   { history value when an error was noted }
define fatal_message = 3   { history value when we had to stop prematurely }

define mark_harmless ≡ if history = spotless then history ← harmless_message
define mark_error ≡ history ← error_message
define mark_fatal ≡ history ← fatal_message
```

⟨ Globals in the outer block 17 ⟩ +≡
history: *spotless* . . *fatal_message*;   { how bad was this run? }

22. ⟨ Set initial values 18 ⟩ +≡
history ← *spotless*;

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. This procedure, which transfers control to the label *final_end* at the end of the program, contains the only non-local **goto** statement in `DVIcopy`. Some Pascal compilers do not implement non-local **goto** statements. In such cases the **goto** *final_end* in *jump_out* should simply be replaced by a call on some system procedure that quietly terminates the program.

```
define abort(#) ≡
        begin print_ln(´␣´,#,´.´); jump_out;
        end
```

⟨ Error handling procedures 23 ⟩ ≡
⟨ Basic printing procedures 48 ⟩

```
procedure close_files_and_terminate; forward;

  procedure jump_out;
    begin mark_fatal; close_files_and_terminate; goto final_end;
    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⟩ +≡
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⟩ +≡
procedure overflow(p : pckt_pointer; n : int_16u);
  begin print('␣!Sorry,␣', title, '␣capacity␣exceeded␣['); print_packet(p); print_ln('=', n : 1, '].');
  jump_out;
  end;
```

26. Binary data and binary files. A detailed description of the `DVI` file format can be found in the documentation of TeX, `DVItype`, or `GFtoDVI`; here we just define symbolic names for some of the `DVI` command bytes.

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

define *dvi_id* = 2 { identifies `DVI` files }
define *dvi_pad* = 223 { pad bytes at end of `DVI` file }

27. A `DVI`, `VF`, or `TFM` file is a sequence of 8-bit bytes. The bytes appear physically in what is called a '**packed file of** 0 . . 255' in Pascal lingo. One, two, three, or four consecutive bytes are often interpreted as (signed or unsigned) integers. We might as well define the corresponding data types.

⟨ Types in the outer block 7 ⟩ +≡
signed_byte = −"80 . . "7F; { signed one-byte quantity }
eight_bits = 0 . . "FF; { unsigned one-byte quantity }
signed_pair = −"8000 . . "7FFF; { signed two-byte quantity }
sixteen_bits = 0 . . "FFFF; { unsigned two-byte quantity }
signed_trio = −"800000 . . "7FFFFF; { signed three-byte quantity }
twentyfour_bits = 0 . . "FFFFFF; { unsigned three-byte quantity }
signed_quad = *int_32*; { signed four-byte quantity }

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

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

```
if $x \ge 0$ then $a \gets x$ div $'100000000$
else begin $x \gets (x + '10000000000) + '10000000000$; $a \gets x$ div $'100000000 + 128$;
   end
$x \gets x$ mod $'100000000$;
$b \gets x$ div $'200000$; $x \gets x$ mod $'200000$;
$c \gets x$ div $'400$; $d \gets x$ mod $'400$;
```

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

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

⟨ Types in the outer block 7 ⟩ +≡
  byte_file = **packed file of** *eight_bits*; { files that contain binary data }

30. For some operating systems it may be convenient or even necessary to close the input files.

  define *close_in*(#) ≡ *do_nothing* { close an input file }

31. Character packets extracted from `VF` files will be stored in a large array *byte_mem*. Other packets of bytes, e.g., character packets extracted from a `GF` or `PK` or `PXL` file could be stored in the same way. A '*pckt_pointer*' variable, which signifies a packet, is an index into another array *pckt_start*. The actual sequence of bytes in the packet pointed to by p appears in positions *pckt_start*[p] to *pckt_start*[$p + 1$] − 1, inclusive, in *byte_mem*.

Packets will also be used to store sequences of *ASCII_code*s; in this respect the *byte_mem* array is very similar to TeX's string pool and part of the following code has, in fact, been copied more or less verbatim from TeX.

In other respects the packets resemble the identifiers used by `TANGLE` and `WEAVE` (also stored in an array called *byte_mem*) since there is, in general, at most one packet with a given contents; thus part of the code below has been adapted from the corresponding code in these programs.

Some Pascal compilers won't pack integers into a single byte unless the integers lie in the range −128 . . 127. To accommodate such systems we access the array *byte_mem* only via macros that can easily be redefined.

  define *bi*(#) ≡ # { convert from *eight_bits* to *packed_byte* }
  define *bo*(#) ≡ # { convert from *packed_byte* to *eight_bits* }

⟨ Types in the outer block 7 ⟩ +≡
  packed_byte = *eight_bits*; { elements of *byte_mem* array }
  byte_pointer = 0 . . *max_bytes*; { an index into *byte_mem* }
  pckt_pointer = 0 . . *max_packets*; { an index into *pckt_start* }

32. The global variable *byte_ptr* points to the first unused location in *byte_mem* and *pckt_ptr* points to the first unused location in *pckt_start*.

⟨ Globals in the outer block 17 ⟩ +≡
byte_mem: **packed array** [*byte_pointer*] **of** *packed_byte*; { bytes of packets }
pckt_start: **array** [*pckt_pointer*] **of** *byte_pointer*; { directory into *byte_mem* }
byte_ptr: *byte_pointer*;
pckt_ptr: *pckt_pointer*;

33. Several of the elementary operations with packets are performed using `WEB` macros instead of Pascal procedures, because many of the operations are done quite frequently and we want to avoid the overhead of procedure calls. For example, here is a simple macro that computes the length of a packet.

define *pckt_length*(#) ≡ (*pckt_start*[# + 1] − *pckt_start*[#]) { the number of bytes in packet number # }

34. Packets are created by appending bytes to *byte_mem*. The *append_byte* macro, defined here, does not check to see if the value of *byte_ptr* has gotten too high; this test is supposed to be made before *append_byte* is used. There is also a *flush_byte* macro, which erases the last byte appended.

To test if there is room to append l more bytes to *byte_mem*, we shall write *pckt_room*(l), which aborts `DVIcopy` and gives an apologetic error message if there isn't enough room.

```
define append_byte(#) ≡ { put byte # at the end of byte_mem }
        begin byte_mem[byte_ptr] ← bi(#); incr(byte_ptr);
        end
define flush_byte ≡ decr(byte_ptr) { forget the last byte in byte_mem }
define pckt_room(#) ≡ { make sure that byte_mem hasn't overflowed }
        if max_bytes − byte_ptr < # then overflow(str_bytes, max_bytes)
define append_one(#) ≡
          begin pckt_room(1); append_byte(#);
          end
```

35. The length of the current packet is called *cur_pckt_length*:

define *cur_pckt_length* ≡ (*byte_ptr* − *pckt_start*[*pckt_ptr*])

36. Once a sequence of bytes has been appended to *byte_mem*, it officially becomes a packet when the *make_packet* function is called. This function returns as its value the identification number of either an existing packet with the same contents or, if no such packet exists, of the new packet. Thus two packets have the same contents if and only if they have the same identification number. In order to locate the packet with a given contents, or to find out that no such packet exists, we need a hash table. The hash table is kept by the method of simple chaining, where the heads of the individual lists appear in the *p_hash* array. If h is a hash code, the hash table list starts at *p_hash*[h] and proceeds through *p_link* pointers.

define *hash_size* = 353 { should be prime, must be > 256 }

⟨ Types in the outer block 7 ⟩ +≡
hash_code = 0 . . *hash_size*;

37. ⟨ Globals in the outer block 17 ⟩ +≡
p_link: **array** [*pckt_pointer*] **of** *pckt_pointer*; { hash table }
p_hash: **array** [*hash_code*] **of** *pckt_pointer*;

38. Initially *byte_mem* and all the hash lists are empty; *empty_packet* is the empty packet.

```
define empty_packet = 0   { the empty packet }
define invalid_packet ≡ max_packets   { used when there is no packet }
```

⟨ Set initial values 18 ⟩ +≡

```
  pckt_ptr ← 1; byte_ptr ← 1; pckt_start[0] ← 1; pckt_start[1] ← 1;
  for h ← 0 to hash_size − 1 do p_hash[h] ← 0;
```

39. ⟨ Local variables for initialization 16 ⟩ +≡

```
h: hash_code;   { index into hash-head arrays }
```

40. Here now is the *make_packet* function used to create packets (and strings).

```
function make_packet: pckt_pointer;
  label found;
  var i, k: byte_pointer;   { indices into byte_mem }
    h: hash_code;   { hash code }
    s, l: byte_pointer;   { start and length of the given packet }
    p: pckt_pointer;   { where the packet is being sought }
  begin s ← pckt_start[pckt_ptr]; l ← byte_ptr − s;   { compute start and length }
  if l = 0 then p ← empty_packet
  else begin ⟨ Compute the packet hash code h 41 ⟩;
    ⟨ Compute the packet location p 42 ⟩;
    if pckt_ptr = max_packets then overflow(str_packets, max_packets);
    incr(pckt_ptr); pckt_start[pckt_ptr] ← byte_ptr;
    end;
found: make_packet ← p;
  end;
```

41. A simple hash code is used: If the sequence of bytes is $b_1 b_2 \ldots b_n$, its hash value will be

$$(2^{n-1}b_1 + 2^{n-2}b_2 + \cdots + b_n) \bmod \mathit{hash\_size}.$$

⟨ Compute the packet hash code h 41 ⟩ ≡

```
  h ← bo(byte_mem[s]); i ← s + 1;
  while i < byte_ptr do
    begin h ← (h + h + bo(byte_mem[i])) mod hash_size; incr(i);
    end
```

This code is used in section 40.

42. If the packet is new, it will be placed in position $p = \mathit{pckt\_ptr}$, otherwise p will point to its existing location.

⟨ Compute the packet location p 42 ⟩ ≡

```
  p ← p_hash[h];
  while p ≠ 0 do
    begin if pckt_length(p) = l then ⟨ Compare packet p with current packet, goto found if equal 43 ⟩;
    p ← p_link[p];
    end;
  p ← pckt_ptr;   { the current packet is new }
  p_link[p] ← p_hash[h]; p_hash[h] ← p   { insert p at beginning of hash list }
```

This code is used in section 40.

43. ⟨ Compare packet p with current packet, **goto** *found* if equal 43 ⟩ ≡

```
begin $i \gets s$; $k \gets pckt\_start[p]$;
while $(i < byte\_ptr) \land (byte\_mem[i] = byte\_mem[k])$ do
  begin $incr(i)$; $incr(k)$;
  end;
if $i = byte\_ptr$ then   { all bytes agree }
  begin $byte\_ptr \gets pckt\_start[pckt\_ptr]$; goto found;
  end;
end
```

This code is used in section 42.

44. Some packets are initialized with predefined strings of *ASCII_code*s; a few macros permit us to do the initialization with a compact program. Since this initialization is done when *byte_mem* is still empty, and since *byte_mem* is supposed to be large enough for all the predefined strings, *pckt_room* is used only if we are debugging.

```
define $pid0(\#) \equiv \# \gets make\_packet$
define $pid1(\#) \equiv byte\_mem[byte\_ptr - 1] \gets bi(\#)$; $pid0$
define $pid2(\#) \equiv byte\_mem[byte\_ptr - 2] \gets bi(\#)$; $pid1$
define $pid3(\#) \equiv byte\_mem[byte\_ptr - 3] \gets bi(\#)$; $pid2$
define $pid4(\#) \equiv byte\_mem[byte\_ptr - 4] \gets bi(\#)$; $pid3$
define $pid5(\#) \equiv byte\_mem[byte\_ptr - 5] \gets bi(\#)$; $pid4$
define $pid6(\#) \equiv byte\_mem[byte\_ptr - 6] \gets bi(\#)$; $pid5$
define $pid7(\#) \equiv byte\_mem[byte\_ptr - 7] \gets bi(\#)$; $pid6$
define $pid8(\#) \equiv byte\_mem[byte\_ptr - 8] \gets bi(\#)$; $pid7$
define $pid9(\#) \equiv byte\_mem[byte\_ptr - 9] \gets bi(\#)$; $pid8$
define $pid10(\#) \equiv byte\_mem[byte\_ptr - 10] \gets bi(\#)$; $pid9$

define $pid\_init(\#) \equiv$
      debug $pckt\_room(\#)$; gubed
    $Incr(byte\_ptr)(\#)$

define $id1 \equiv pid\_init(1)$; $pid1$
define $id2 \equiv pid\_init(2)$; $pid2$
define $id3 \equiv pid\_init(3)$; $pid3$
define $id4 \equiv pid\_init(4)$; $pid4$
define $id5 \equiv pid\_init(5)$; $pid5$
define $id6 \equiv pid\_init(6)$; $pid6$
define $id7 \equiv pid\_init(7)$; $pid7$
define $id8 \equiv pid\_init(8)$; $pid8$
define $id9 \equiv pid\_init(9)$; $pid9$
define $id10 \equiv pid\_init(10)$; $pid10$
```

45. Here we initialize some strings used as argument of the *overflow* and *confusion* procedures.

⟨ Initialize predefined strings 45 ⟩ ≡

```
id5("f")("o")("n")("t")("s")(str_fonts); id5("c")("h")("a")("r")("s")(str_chars);
id6("w")("i")("d")("t")("h")("s")(str_widths); id7("p")("a")("c")("k")("e")("t")("s")(str_packets);
id5("b")("y")("t")("e")("s")(str_bytes);
id9("r")("e")("c")("u")("r")("s")("i")("o")("n")(str_recursion);
id5("s")("t")("a")("c")("k")(str_stack);
id10("n")("a")("m")("e")("l")("e")("n")("g")("t")("h")(str_name_length);
```

See also sections 91, 135, and 191.

This code is used in section 241.

46. ⟨ Globals in the outer block 17 ⟩ +≡

str_fonts, *str_chars*, *str_widths*, *str_packets*, *str_bytes*, *str_recursion*, *str_stack*, *str_name_length*: *pckt_pointer*;

47. Some packets, e.g., the preamble comments of `DVI` and `VF` files, are needed only temporarily. In such cases *new_packet* is used to create a packet (which might duplicate an existing packet) and *flush_packet* is used to discard it; the calls to *new_packet* and *flush_packet* must occur in balanced pairs, without any intervening calls to *make_packet*.

```
function new_packet: pckt_pointer;
  begin if pckt_ptr = max_packets then overflow(str_packets, max_packets);
  new_packet ← pckt_ptr; incr(pckt_ptr); pckt_start[pckt_ptr] ← byte_ptr;
  end;

procedure flush_packet;
  begin decr(pckt_ptr); byte_ptr ← pckt_start[pckt_ptr];
  end;
```

48. The *print_packet* procedure prints the contents of a packet; such a packet should, of course, consists of a sequence of *ASCII_code*s.

```
⟨ Basic printing procedures 48 ⟩ ≡
procedure print_packet(p : pckt_pointer);
  var k: byte_pointer;
  begin for k ← pckt_start[p] to pckt_start[p + 1] − 1 do print(xchr[bo(byte_mem[k])]);
  end;
```

See also sections 60, 61, and 181.

This code is used in section 23.

49. When we interpret a packet we will use two (global or local) variables: *cur_loc* will point to the byte to be used next, and *cur_limit* will point to the start of the next packet. The macro *pckt_extract* will be used to extract one byte; it should, however, never be used with *cur_loc* $\geq$ *cur_limit*.

```
  define pckt_extract(#) ≡
          debug if cur_loc ≥ cur_limit then confusion(str_packets) else
          gubed
        begin # ← bo(byte_mem[cur_loc]); incr(cur_loc); end

⟨ Globals in the outer block 17 ⟩ +≡
cur_pckt: pckt_pointer;   { the current packet }
cur_loc: byte_pointer;   { current location in a packet }
cur_limit: byte_pointer;   { start of next packet }
```

50. We will need routines to extract one, two, three, or four bytes from *byte_mem*, from the DVI file, or from a VF file and assemble them into (signed or unsigned) integers and these routines should be optimized for efficiency. Here we define WEB macros to be used for the body of these routines; thus the changes for system dependent optimization have to be applied only once.

In addition we demonstrates how these macros can be used to define functions that extract one, two, three, or four bytes from a character packet and assemble them into signed or unsigned integers (assuming that *cur_loc* and *cur_limit* are initialized suitably).

```
  define begin_byte(#) ≡
        var a: eight_bits;
        begin #(a)
  define comp_sbyte(#) ≡
          if $a < 128$ then # ← a else # ← $a - 256$
  define comp_ubyte(#) ≡ # ← a
  format begin_byte ≡ begin
function pckt_sbyte: int_8;   { returns the next byte, signed }
  begin_byte (pckt_extract); comp_sbyte(pckt_sbyte);
  end;
function pckt_ubyte: int_8u;   { returns the next byte, unsigned }
  begin_byte (pckt_extract); comp_ubyte(pckt_ubyte);
  end;
```

51.
```
  define begin_pair(#) ≡
        var a, b: eight_bits;
        begin #(a); #(b)
  define comp_spair(#) ≡
          if $a < 128$ then # ← $a * 256 + b$ else # ← $(a - 256) * 256 + b$
  define comp_upair(#) ≡ # ← $a * 256 + b$
  format begin_pair ≡ begin
function pckt_spair: int_16;   { returns the next two bytes, signed }
  begin_pair (pckt_extract); comp_spair(pckt_spair);
  end;
function pckt_upair: int_16u;   { returns the next two bytes, unsigned }
  begin_pair (pckt_extract); comp_upair(pckt_upair);
  end;
```

52.
```
  define begin_trio(#) ≡
        var a, b, c: eight_bits;
        begin #(a); #(b); #(c)
  define comp_strio(#) ≡
          if $a < 128$ then # ← $(a * 256 + b) * 256 + c$ else # ← $((a - 256) * 256 + b) * 256 + c$
  define comp_utrio(#) ≡ # ← $(a * 256 + b) * 256 + c$
  format begin_trio ≡ begin
function pckt_strio: int_24;   { returns the next three bytes, signed }
  begin_trio (pckt_extract); comp_strio(pckt_strio);
  end;
function pckt_utrio: int_24u;   { returns the next three bytes, unsigned }
  begin_trio (pckt_extract); comp_utrio(pckt_utrio);
  end;
```

53.
```
define begin_quad(#) ≡
      var a, b, c, d: eight_bits;
      begin #(a); #(b); #(c); #(d)
define comp_squad(#) ≡
        if $a < 128$ then # ← $((a * 256 + b) * 256 + c) * 256 + d$
        else # ← $(((a - 256) * 256 + b) * 256 + c) * 256 + d$
format begin_quad ≡ begin

function pckt_squad: int_32;   { returns the next four bytes, signed }
  begin_quad (pckt_extract); comp_squad(pckt_squad);
  end;
```

54. A similar set of routines is needed for the inverse task of decomposing a `DVI` command into a sequence of bytes to be appended to *byte_mem* or, in the case of `DVIcopy`, to be written to the output file. Again we define `WEB` macros to be used for the body of these routines; thus the changes for system dependent optimization have to be applied only once.

First, the *pckt_one* outputs one byte, negative values are represented in two's complement notation.

```
define begin_one ≡
      begin
define comp_one(#) ≡
        if $x < 0$ then Incr(x)(256);
      #(x)
format begin_one ≡ begin

device procedure pckt_one(x : int_32);   { output one byte }
begin_one ; pckt_room(1); comp_one(append_byte);
end;
ecived
```

55. The *pckt_two* outputs two bytes, negative values are represented in two's complement notation.

```
define begin_two ≡
      begin
define comp_two(#) ≡
        if $x < 0$ then Incr(x)("10000);
      #(x div "100); #(x mod "100)
format begin_two ≡ begin

device procedure pckt_two(x : int_32);   { output two byte }
begin_two ; pckt_room(2); comp_two(append_byte);
end;
ecived
```

56. The *pckt_four* procedure outputs four bytes in two's complement notation, without risking arithmetic overflow.

```
define begin_four ≡
        begin
define comp_four(#) ≡
          if $x \ge 0$ then #($x$ div ″1000000)
          else begin Incr(x)(″40000000); Incr(x)(″40000000); #(($x$ div ″1000000) + 128);
            end;
        $x \leftarrow x$ mod ″1000000; #($x$ div ″10000); $x \leftarrow x$ mod ″10000; #($x$ div ″100); #($x$ mod ″100)
format begin_four ≡ begin
procedure pckt_four(x : int_32);   { output four bytes }
  begin_four ; pckt_room(4); comp_four(append_byte);
  end;
```

57. Next, the *pckt_char* procedure outputs a *set_char* or *set* command or, if *upd* = *false*, a *put* command.

```
define begin_char ≡
        var o: eight_bits;   { set1 or put1 }
        begin
define comp_char(#) ≡
          if (¬upd) ∨ (res > 127) ∨ (ext ≠ 0) then
            begin o ← dvi_char_cmd[upd];   { set1 or put1 }
            if ext < 0 then Incr(ext)(″1000000);
            if ext = 0 then #(o) else
            begin if ext < ″100 then #(o + 1) else
            begin if ext < ″10000 then #(o + 2) else
            begin #(o + 3); #(ext div ″10000); ext ← ext mod ″10000;
            end; #(ext div ″100); ext ← ext mod ″100;
            end; #(ext);
            end;
            end;
        #(res)
format begin_char ≡ begin
procedure pckt_char(upd : boolean; ext : int_32; res : eight_bits);   { output set or put }
  begin_char ; pckt_room(5); comp_char(append_byte);
  end;
```

58. Then, the *pckt_unsigned* procedure outputs a *fnt* or *xxx* command with its first parameter (normally unsigned); a *fnt* command is converted into *fnt_num* whenever this is possible.

```
define begin_unsigned ≡
      begin
define comp_unsigned(#) ≡
        if (x < "100) ∧ (x ≥ 0) then
          if (o = fnt1) ∧ (x < 64) then Incr(x)(fnt_num_0) else #(o)
        else begin if (x < "10000) ∧ (x ≥ 0) then #(o + 1) else
          begin if (x < "1000000) ∧ (x ≥ 0) then #(o + 2) else
          begin #(o + 3);
          if x ≥ 0 then #(x div "1000000)
          else begin Incr(x)("40000000); Incr(x)("40000000); #((x div "1000000) + 128);
            end;
          x ← x mod "1000000;
          end; #(x div "10000); x ← x mod "10000;
          end; #(x div "100); x ← x mod "100;
          end;
      #(x)
format begin_unsigned ≡ begin
procedure pckt_unsigned(o : eight_bits; x : int_32);   { output fnt_num, fnt, or xxx }
  begin_unsigned ; pckt_room(5); comp_unsigned(append_byte);
  end;
```

59. Finally, the *pckt_signed* procedure outputs a movement (*right*, *w*, *x*, *down*, *y*, or *z*) command with its (signed) parameter.

```
  define begin_signed ≡
        var xx: int_31;  { 'absolute value' of x }
        begin
  define comp_signed(#) ≡
          if x ≥ 0 then xx ← x else xx ← −(x + 1);
        if xx < "80 then
          begin #(o); if x < 0 then Incr(x)("100);
          end
        else begin if xx < "8000 then
            begin #(o + 1); if x < 0 then Incr(x)("10000);
            end
          else begin if xx < "800000 then
              begin #(o + 2); if x < 0 then Incr(x)("1000000);
              end
            else begin #(o + 3);
              if x ≥ 0 then #(x div "1000000)
              else begin x ← "7FFFFFFF − xx; #((x div "1000000) + 128); end;
              x ← x mod "1000000;
              end;
            #(x div "10000); x ← x mod "10000;
            end;
          #(x div "100); x ← x mod "100;
          end;
        #(x)
  format begin_signed ≡ begin
procedure pckt_signed(o : eight_bits; x : int_32);   { output right, w, x, down, y, or z }
  begin_signed ; pckt_room(5); comp_signed(append_byte);
  end;
```

60. The *hex_packet* procedure prints the contents of a packet in hexadecimal form.

```
⟨ Basic printing procedures 48 ⟩ +≡
  debug procedure hex_packet(p : pckt_pointer);  { prints a packet in hex }
  var j, k, l: byte_pointer;  { indices into byte_mem }
    d: int_8u;
  begin j ← pckt_start[p] − 1; k ← pckt_start[p + 1] − 1;
  print_ln(´␣packet=´, p : 1, ´␣start=´, j + 1 : 1, ´␣length=´, k − j : 1);
  for l ← j + 1 to k do
    begin d ← (bo(byte_mem[l])) div 16;
    if d < 10 then print(xchr[d + "0"]) else print(xchr[d − 10 + "A"]);
    d ← (bo(byte_mem[l])) mod 16;
    if d < 10 then print(xchr[d + "0"]) else print(xchr[d − 10 + "A"]);
    if (l = k) ∨ (((l − j) mod 16) = 0) then new_line
    else if ((l − j) mod 4) = 0 then print(´␣␣´)
      else print(´␣´);
    end;
  end;
  gubed
```

61. File names. The structure of file names is different for different systems; therefore this part of the program will, in most cases, require system dependent modifications. Here we assume that a file name consists of three parts: an area or directory specifying where the file can be found, a name proper and an extension; `DVIcopy` assumes that these three parts appear in order stated above but this need not be true in all cases.

The font names extracted from `DVI` and `VF` files consist of an area part and a name proper; these are stored as packets consisting of the length of the area part followed by the area and the name proper. When we print an external font name we simple print the area and the name contained in the 'file name packet' without delimiter between them. This may need to be modified for some systems.

```
⟨ Basic printing procedures 48 ⟩ +≡
procedure print_font(f : font_number);
  var p: pckt_pointer;   { the font name packet }
    k: byte_pointer;   { index into byte_mem }
    m: int_31;   { font magnification }
  begin print(´␣=␣´); p ← font_name(f);
  for k ← pckt_start[p] + 1 to pckt_start[p + 1] − 1 do print(xchr[bo(byte_mem[k])]);
  m ← round((font_scaled(f)/font_design(f)) ∗ out_mag);
  if m ≠ 1000 then print(´␣scaled␣´, m : 1);
  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 17 ⟩ +≡
cur_name: packed array [1 .. name_length] of char;   { external name, with no lower case letters }
l_cur_name: int_15;   { this many characters are actually relevant in cur_name }
```

63. For `TFM` and `VF` files we just append the appropriate extension to the file name packet; in addition a system dependent area part (usually different for `TFM` and `VF` files) is prepended if the file name packet contains no area part.

```
  define append_to_name(#) ≡
        if l_cur_name < name_length then
          begin incr(l_cur_name); cur_name[l_cur_name] ← #;
          end
        else overflow(str_name_length, name_length)
  define make_font_name_end(#) ≡ append_to_name(#[l]); make_name
  define make_font_name(#) ≡ l_cur_name ← 0;
      for l ← 1 to # do make_font_name_end
```

64. For files with character raster data (e.g., `GF` or `PK` files) the extension and/or area part will in most cases depend on the resolution of the output device (corrected for font magnification). If the special character *res_char* occurs in the extension and/or default area, a character string representing the device resolution will be substituted.

```
define res_char ≡ ´?´   { character to be replaced by font resolution }
define res_ASCII = "?"   { xord[res_char] }
define append_res_to_name(#) ≡
        begin c ← #;
        device if c = res_char then
          for ll ← n_res_digits downto 1 do append_to_name(res_digits[ll])
        else
        ecived
        append_to_name(c);
        end
define make_font_res_end(#) ≡ append_res_to_name(#[l]); make_name
define make_font_res(#) ≡ make_res; l_cur_name ← 0;
      for l ← 1 to # do make_font_res_end
```

65. ⟨ Globals in the outer block 17 ⟩ +≡

```
  device f_res: int_16u;   { font resolution }
res_digits: array [1 .. 5] of char;
n_res_digits: int_7;   { number of significant characters in res_digits }
  ecived
```

66. The *make_res* procedure creates a sequence of characters representing to the font resolution *f_res*.

```
  device procedure make_res;
  var r: int_16u;
  begin n_res_digits ← 0; r ← f_res;
  repeat incr(n_res_digits); res_digits[n_res_digits] ← xchr["0" + (r mod 10)]; r ← r div 10;
  until r = 0;
  end;
  ecived
```

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
c: char;   { a character to be appended to cur_name }
  begin n ← font_name(cur_fnt); 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);
    if $(b \geq "a") \wedge (b \leq "z")$ then Decr(b)(("a" − "A"));   { convert to upper case }
    append_to_name(xchr[b]);
    end;
  cur_loc ← pckt_start[e]; cur_limit ← pckt_start[e + 1];
  while cur_loc < cur_limit do
    begin pckt_extract(b); append_res_to_name(xchr[b]);
    end;
  while l_cur_name < name_length do
    begin incr(l_cur_name); cur_name[l_cur_name] ← ´␣´;
    end;
  end;
```

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

The character-width data appears also in other files (e.g., in `VF` files or in `GF` and `PK` files that specify bit patterns for digitized characters); thus, it is usually possible for `DVI` reading programs to get by with accessing only one file per font. For `VF` reading programs there is, however, a problem: (1) when reading the character packets from a `VF` file the `TFM` width for its local fonts should be known in order to analyze and optimize the packets (e.g., determine if a packet must indeed be enclosed with *push* and *pop* as implied by the `VF` format); and (2) in order to avoid infinite recursion such programs must not try to read a `VF` file for a font before a character from that font is actually used. Thus `DVIcopy` reads the `TFM` file whenever a new font is encountered and delays the decision whether this is a virtual font or not.

69. First of all we need to know for each font f such things as its external name, design and scaled size, and the approximate size of inter-word spaces. In addition we need to know the range $bc \mathrel{..} ec$ of valid characters for this font, and for each character c in f we need to know if this character exists and if so what is the width of c. Depending on the font type of f we may want to know a few other things about character c in f such as the character packet from a `VF` file or the raster data from a `PK` file.

In `DVIcopy` we want to be able to handle the full range $-2^{31} \le c < 2^{31}$ of character codes; each character code is decomposed into a character residue $0 \le res < 256$ and character extension $-2^{23} \le ext < 2^{23}$ such that $c = 256 * ext + res$. At present `VFtoVP`, `VPtoVF`, and the standard version of TeX use only characters in the range $0 \le c < 256$ (i.e., $ext = 0$), there are, however, extensions of TeX which use characters with $ext \ne 0$. In any case characters with $ext \ne 0$ will be used rather infrequently and we want to handle this possibility without too much overhead.

Some of the data for each character c depend only on its residue: first of all its width and escapement; others, such as `VF` packets or raster data will also depend on its extension. The later will be stored as packets in *byte_mem*, and the packets for characters with the same residue but different extension will be chained.

Thus we have to maintain several variables for each character residue $bc \le res \le ec$ from each font f; we store each type of variable in a large array such that the array index $font\_chars(f) + res$ points to the value for characters with residue res from font f.

70. Quite often a particular width value is shared by several characters in a font or even by characters from different fonts; the later will probably occur in particular for virtual fonts and the local fonts used by them. Thus the array *widths* is used to store all different `TFM` width values of all legal characters in all fonts; a variable of type *width_pointer* is an index into *widths* or is zero if a characters does not exist.

In order to locate a given width value we use again a hash table with simple chaining; this time the heads of the individual lists appear in the *w_hash* array and the lists proceed through *w_link* pointers.

⟨ Types in the outer block 7 ⟩ +≡
  width_pointer = 0 .. *max_widths*; { an index into *widths* }

71. ⟨ Globals in the outer block 17 ⟩ +≡
widths: **array** [*width_pointer*] **of** *int_32*; { the different width values }
w_link: **array** [*width_pointer*] **of** *width_pointer*; { hash table }
w_hash: **array** [*hash_code*] **of** *width_pointer*;
n_widths: *width_pointer*; { first unoccupied position in *widths* }

72. Initially the *widths* array and all the hash lists are empty, except for one entry: the width value zero; in addition we set $widths[0] \leftarrow 0$.

```
define invalid_width = 0  { width pointer for invalid characters }
define zero_width = 1  { a width pointer to the value zero }
```

⟨ Set initial values 18 ⟩ +≡

```
  w_hash[0] ← 1; w_link[1] ← 0; widths[0] ← 0; widths[1] ← 0; n_widths ← 2;
  for h ← 1 to hash_size − 1 do w_hash[h] ← 0;
```

73. The *make_width* function returns an index into *widths* and, if necessary, adds a new width value; thus two characters will have the same *width_pointer* if and only if their widths agree.

```
function make_width(w : int_32): width_pointer;
  label found;
  var h: hash_code;  { hash code }
    p: width_pointer;  { where the identifier is being sought }
    x: int_16;  { intermediate value }
  begin widths[n_widths] ← w; ⟨ Compute the width hash code h 74 ⟩;
  ⟨ Compute the width location p, goto found unless the value is new 75 ⟩;
  if n_widths = max_widths then overflow(str_widths, max_widths);
  incr(n_widths);
found: make_width ← p;
  end;
```

74. A simple hash code is used: If the width value consists of the four bytes $b_0b_1b_2b_3$, its hash value will be

$$(8 * b_0 + 4 * b_1 + 2 * b_2 + b_3) \bmod hash\_size.$$

⟨ Compute the width hash code *h* 74 ⟩ ≡

```
  if w ≥ 0 then x ← w div ″1000000
  else begin w ← w + ″40000000; w ← w + ″40000000; x ← (w div ″1000000) + ″80;
    end;
  w ← w mod ″1000000; x ← x + x + (w div ″10000); w ← w mod ″10000; x ← x + x + (w div ″100);
  h ← (x + x + (w mod ″100)) mod hash_size
```

This code is used in section 73.

75. If the width is new, it has been placed into position $p = n\_widths$, otherwise p will point to its existing location.

⟨ Compute the width location *p*, **goto** found unless the value is new 75 ⟩ ≡

```
  p ← w_hash[h];
  while p ≠ 0 do
    begin if widths[p] = widths[n_widths] then goto found;
    p ← w_link[p];
    end;
  p ← n_widths;  { the current width is new }
  w_link[p] ← w_hash[h]; w_hash[h] ← p  { insert p at beginning of hash list }
```

This code is used in section 73.

76. The *char_widths* array is used to store the *width_pointer*s for all different characters among all fonts. The *char_packets* array is used to store the *pckt_pointer*s for all different characters among all fonts; they can point to character packets from `VF` files or, e.g., raster packets from `PK` files.

⟨ Types in the outer block 7 ⟩ +≡
  char_offset = −255 . . *max_chars*; { *char_pointer* offset for a font }
  char_pointer = 0 . . *max_chars*; { index into *char_widths* or similar arrays }

77. ⟨ Globals in the outer block 17 ⟩ +≡
char_widths: **array** [*char_pointer*] **of** *width_pointer*; { width pointers }
char_packets: **array** [*char_pointer*] **of** *pckt_pointer*; { packet pointers }
n_chars: *char_pointer*; { first unused position in *char_widths* }

78. ⟨ Set initial values 18 ⟩ +≡
  n_chars ← 0;

79. The current number of known fonts is *nf*; each known font has an internal number f, where $0 \le f < nf$. For the moment we need for each known font: *font_check*, *font_scaled*, *font_design*, *font_name*, *font_bc*, *font_ec*, *font_chars*, and *font_type*. Here *font_scaled* and *font_design* are measured in `DVI` units and *font_chars* is of type *char_offset*: the width pointer for character c of the font is stored in *char_widths*[*char_offset* + c] (for *font_bc* ≤ c ≤ *font_ec*). Later on we will need additional information depending on the font type: `VF` or real (`GF`, `PK`, or `PXL`).

⟨ Types in the outer block 7 ⟩ +≡
  f_type = *defined_font* . . *max_font_type*; { type of a font }
  font_number = 0 . . *max_fonts*;

80. ⟨ Globals in the outer block 17 ⟩ +≡
nf: *font_number*;

81. These data are stored in several arrays and we use WEB macros to access the various fields. Thus it would be simple to store the data in an array of record structures and adapt the WEB macros accordingly.

We will say, e.g., *font_name*(f) for the name field of font f, and *font_width*$(f)(c)$ for the width pointer of character c in font f and *font_packet*$(f)(c)$ for its character packet (this character exists provided *font_bc*$(f) \le c \le$ *font_ec*(f) and *font_width*$(f)(c) \ne$ *invalid_width*). The actual width of character c in font f is stored in *widths*[*font_width*$(f)(c)$].

define *font_check*(#) ≡ *fnt_check*[#] { checksum }
define *font_scaled*(#) ≡ *fnt_scaled*[#] { scaled or 'at' size }
define *font_design*(#) ≡ *fnt_design*[#] { design size }
define *font_name*(#) ≡ *fnt_name*[#] { area plus name packet }
define *font_bc*(#) ≡ *fnt_bc*[#] { first character }
define *font_ec*(#) ≡ *fnt_ec*[#] { last character }
define *font_chars*(#) ≡ *fnt_chars*[#] { character info offset }
define *font_type*(#) ≡ *fnt_type*[#] { type of this font }
define *font_font*(#) ≡ *fnt_font*[#] { use depends on *font_type* }

define *font_width_end*(#) ≡ #]
define *font_width*(#) ≡ *char_widths* [*font_chars*(#) + *font_width_end*
define *font_packet*(#) ≡ *char_packets* [*font_chars*(#) + *font_width_end*

⟨ Globals in the outer block 17 ⟩ +≡
fnt_check: **array** [*font_number*] **of** *int_32*; { checksum }
fnt_scaled: **array** [*font_number*] **of** *int_31*; { scaled size }
fnt_design: **array** [*font_number*] **of** *int_31*; { design size }
  device ⟨ Declare device dependent font data arrays 195 ⟩ **ecived**
fnt_name: **array** [*font_number*] **of** *pckt_pointer*; { pointer to area plus name packet }
fnt_bc: **array** [*font_number*] **of** *eight_bits*; { first character }
fnt_ec: **array** [*font_number*] **of** *eight_bits*; { last character }
fnt_chars: **array** [*font_number*] **of** *char_offset*; { character info offset }
fnt_type: **array** [*font_number*] **of** *f_type*; { type of font }
fnt_font: **array** [*font_number*] **of** *font_number*; { use depends on *font_type* }

82. **define** *invalid_font* ≡ *max_fonts* { used when there is no valid font }

⟨ Set initial values 18 ⟩ +≡
  device ⟨ Initialize device dependent font data 196 ⟩ **ecived**
  nf ← 0;

83. A VF, or GF, or PK file may contain information for several characters with the same residue but with different extension; all except the first of the corresponding packets in *byte_mem* will contain a pointer to the previous one and *font_packet*$(f)(res)$ identifies the last such packet.

A character packet in *byte_mem* starts with a flag byte

$$flag = {''}40 * ext\_flag + {''}20 * chain\_flag + type\_flag$$

with $0 \le ext\_flag \le 3$, $0 \le chain\_flag \le 1$, $0 \le type\_flag \le {''}1F$, followed by *ext_flag* bytes with the character extension for this packet and, if $chain\_flag = 1$, by a two byte packet pointer to the previous packet for the same font and character residue. The actual character packet follows after these header bytes and the interpretation of the *type_flag* depends on whether this is a VF packet or a packet for raster data.

The empty packet is interpreted as a special case of a packet with $flag = 0$.

define $ext\_flag = {''}40$
define $chain\_flag = {''}20$

⟨ Types in the outer block 7 ⟩ +≡
  $type\_flag = 0 \,..\, chain\_flag - 1$; { the range of values for the *type_flag* }

84. The global variable *cur_fnt* is the internal font number of the currently selected font, or equals *invalid_font* if no font has been selected; *cur_res* and *cur_ext* are the residue and extension part of the current character code. The type of a character packet located by the *find_packet* function defined below is *cur_type*. While building a character packet for a character, *pckt_ext* and *pckt_res* are the extension and residue of this character; *pckt_dup* indicates whether a packet for this extension exists already.

```
⟨ Globals in the outer block 17 ⟩ +≡
cur_fnt: font_number;  { the currently selected font }
cur_ext: int_24;  { the current character extension }
cur_res: int_8u;  { the current character residue }
cur_type: type_flag;  { type of the current character packet }
pckt_ext: int_24;  { character extension for the current character packet }
pckt_res: int_8u;  { character residue for the current character packet }
pckt_dup: boolean;  { is there a previous packet for the same extension? }
pckt_prev: pckt_pointer;  { a previous packet for the same extension }
pckt_m_msg, pckt_s_msg, pckt_d_msg: int_7;  { counts for various character packet error messages }
```

85. ⟨ Set initial values 18 ⟩ +≡

```
  cur_fnt ← invalid_font; pckt_m_msg ← 0; pckt_s_msg ← 0; pckt_d_msg ← 0;
```

86. The *find_packet* functions is used to locate the character packet for the character with residue *cur_res* and extension *cur_ext* from font *cur_fnt* and returns *false* if no packet exists for any extension; otherwise the result is *true* and the global variables *cur_packet*, *cur_type*, *cur_loc*, and *cur_limit* are initialized. In case none of the character packets has the correct extension, the last one in the chain (the one defined first) is used instead and *cur_ext* is changed accordingly.

```
function find_packet: boolean;
  label found, exit;
  var p, q: pckt_pointer;  { current and next packet }
    f: eight_bits;  { a flag byte }
    e: int_24;  { extension for a packet }
  begin q ← font_packet(cur_fnt)(cur_res);
  if q = invalid_packet then
    begin if pckt_m_msg < 10 then  { stop telling after first 10 times }
      begin
        print_ln('---missing␣character␣packet␣for␣character␣', cur_res : 1, '␣font␣', cur_fnt : 1);
      incr(pckt_m_msg); mark_error;
      if pckt_m_msg = 10 then print_ln('---further␣messages␣suppressed.');
      end;
    find_packet ← false; return;
    end;
  ⟨ Locate a character packet and goto found if found 87 ⟩;
  if pckt_s_msg < 10 then  { stop telling after first 10 times }
    begin print_ln('---substituted␣character␣packet␣with␣extension␣', e : 1, '␣instead␣of␣',
        cur_ext : 1, '␣for␣character␣', cur_res : 1, '␣font␣', cur_fnt : 1); incr(pckt_s_msg); mark_error;
    if pckt_s_msg = 10 then print_ln('---further␣messages␣suppressed.');
    end;
  cur_ext ← e;
found: cur_pckt ← p; cur_type ← f; find_packet ← true;
exit: end;
```

87. ⟨Locate a character packet and **goto** *found* if found 87⟩ ≡

```
repeat $p \gets q$; $q \gets invalid\_packet$; $cur\_loc \gets pckt\_start[p]$; $cur\_limit \gets pckt\_start[p+1]$;
  if $p = empty\_packet$ then
    begin $e \gets 0$; $f \gets 0$;
    end
  else begin $pckt\_extract(f)$;
    case $(f$ div $ext\_flag)$ of
    0: $e \gets 0$;
    1: $e \gets pckt\_ubyte$;
    2: $e \gets pckt\_upair$;
    othercases $e \gets pckt\_strio$;   { $f$ div $ext\_flag = 3$ }
    endcases;
    if $(f$ mod $ext\_flag) \ge chain\_flag$ then $q \gets pckt\_upair$;
    $f \gets f$ mod $chain\_flag$;
    end;
  if $e = cur\_ext$ then goto $found$;
until $q = invalid\_packet$
```

This code is used in sections 86 and 88.

88. The *start_packet* procedure is used to create the header bytes of a character packet for the character with residue *cur_res* and extension *cur_ext* from font *cur_fnt*; if a previous such packet exists, we try to build an exact duplicate, i.e., use the chain field of that previous packet.

```
procedure $start\_packet(t : type\_flag)$;
  label $found, not\_found$;
  var $p, q$: $pckt\_pointer$;   { current and next packet }
    $f$: $int\_8u$;   { a flag byte }
    $e$: $int\_32$;   { extension for a packet }
    $cur\_loc$: $byte\_pointer$;   { current location in a packet }
    $cur\_limit$: $byte\_pointer$;   { start of next packet }
  begin $q \gets font\_packet(cur\_fnt)(cur\_res)$;
  if $q \ne invalid\_packet$ then ⟨Locate a character packet and goto found if found 87⟩;
  $q \gets font\_packet(cur\_fnt)(cur\_res)$; $pckt\_dup \gets false$; goto $not\_found$;
$found$: $pckt\_dup \gets true$; $pckt\_prev \gets p$;
$not\_found$: $pckt\_ext \gets cur\_ext$; $pckt\_res \gets cur\_res$; $pckt\_room(6)$;
  debug if $byte\_ptr \ne pckt\_start[pckt\_ptr]$ then $confusion(str\_packets)$;
  gubed
  if $q = invalid\_packet$ then $f \gets t$ else $f \gets t + chain\_flag$;
  $e \gets cur\_ext$;
  if $e < 0$ then $Incr(e)(''1000000)$;
  if $e = 0$ then $append\_byte(f)$ else
  begin if $e < ''100$ then $append\_byte(f + ext\_flag)$ else
  begin if $e < ''10000$ then $append\_byte(f + ext\_flag + ext\_flag)$ else
  begin $append\_byte(f + ext\_flag + ext\_flag + ext\_flag)$; $append\_byte(e$ div $''10000)$; $e \gets e$ mod $''10000$;
  end; $append\_byte(e$ div $''100)$; $e \gets e$ mod $''100$;
  end; $append\_byte(e)$;
  end;
  if $q \ne invalid\_packet$ then
    begin $append\_byte(q$ div $''100)$; $append\_byte(q$ mod $''100)$;
    end;
  end;
```

89. The *build_packet* procedure is used to finish a character packet. If a previous packet for the same character extension exists, the new one is discarded; if the two packets are identical, as it occasionally occurs for raster files, this is done without an error message.

```
procedure build_packet;
  var k, l: byte_pointer;   { indices into byte_mem }
  begin if pckt_dup then
    begin k ← pckt_start[pckt_prev + 1]; l ← pckt_start[pckt_ptr];
    if (byte_ptr − l) ≠ (k − pckt_start[pckt_prev]) then pckt_dup ← false;
    while pckt_dup ∧ (byte_ptr > l) do
      begin flush_byte; decr(k);
      if byte_mem[byte_ptr] ≠ byte_mem[k] then pckt_dup ← false;
      end;
    if (¬pckt_dup) ∧ (pckt_d_msg < 10) then   { stop telling after first 10 times }
      begin print(´---duplicate␣packet␣for␣character␣´, pckt_res : 1);
      if pckt_ext ≠ 0 then print(´.´, pckt_ext : 1);
      print_ln(´␣font␣´, cur_fnt : 1); incr(pckt_d_msg); mark_error;
      if pckt_d_msg = 10 then print_ln(´---further␣messages␣suppressed.´);
      end;
    byte_ptr ← l;
    end
  else font_packet(cur_fnt)(pckt_res) ← make_packet;
  end;
```

90. Defining fonts. A detailed description of the `TFM` file format can be found in the documentation of TeX, METAFONT, or `TFtoPL`. In order to read `TFM` files the program uses the binary file variable *tfm_file*.

```
⟨Globals in the outer block 17⟩ +≡
tfm_file: byte_file;   { a TFM file }
tfm_ext: pckt_pointer;   { extension for TFM files }
```

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, `DVIcopy` is supposed to use the default `TFM` directory, which is a system-dependent place where the `TFM` files for standard fonts are kept. The string variable *TFM_default_area* contains the name of this area.

```
  define TFM_default_area_name ≡ ´TeXfonts:´   { change this to the correct name }
  define TFM_default_area_name_length = 9   { change this to the correct length }
⟨Globals in the outer block 17⟩ +≡
TFM_default_area: packed array [1 .. TFM_default_area_name_length] of char;
```

93. ⟨Set initial values 18⟩ +≡

```
  TFM_default_area ← TFM_default_area_name;
```

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⟩ +≡
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;
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: Open tfm_file 95⟩ ≡
  make_font_name(TFM_default_area_name_length)(TFM_default_area)(tfm_ext); reset(tfm_file, cur_name);
  if eof(tfm_file) then abort(´---not␣loaded,␣TFM␣file␣can´´t␣be␣opened!´)
```

This code is used in section 99.

96. It turns out to be convenient to read four bytes at a time, when we are inputting from `TFM` files. The input goes into global variables *tfm_b0*, *tfm_b1*, *tfm_b2*, and *tfm_b3*, with *tfm_b0* getting the first byte and *tfm_b3* the fourth.

```
⟨Globals in the outer block 17⟩ +≡
tfm_b0, tfm_b1, tfm_b2, tfm_b3: eight_bits;   { four bytes input at once }
```

97. Reading a `TFM` file should be done as efficient as possible for a particular system; on many systems this means that a large number of bytes from *tfm_file* is read into a buffer and will then be extracted from that buffer. In order to simplify such system dependent changes we use the `WEB` macro *tfm_byte* to extract the next `TFM` byte; this macro and *eof*(*tfm_file*) are used only in the *read_tfm_word* procedure which sets *tfm_b0* through *tfm_b3* to the next four bytes in the current `TFM` file. Here we give simple minded definitions in terms of standard Pascal.

```
  define tfm_byte(#) ≡ read(tfm_file, #)   { read next TFM byte }
procedure read_tfm_word;
  begin tfm_byte(tfm_b0); tfm_byte(tfm_b1); tfm_byte(tfm_b2); tfm_byte(tfm_b3);
  if eof(tfm_file) then bad_font;
  end;
```

98. Here are three procedures used to check the consistency of font files: First, the *check_check_sum* procedure compares two check sum values: a warning is given if they differ and are both non-zero; if the second value is not zero it may replace the first one. Next, the *check_design_size* procedure compares two design size values: a warning is given if they differ by more than a small amount. Finally, the *check_width* function compares the character width value for character *cur_res* read from a `VF` or raster file for font *cur_fnt* with the value previously read from the `TFM` file and returns the width pointer for that value; a warning is given if the two values differ.

```
procedure check_check_sum(c : int_32; u : boolean);   { compare font_check(cur_fnt) with c }
  begin if (c ≠ font_check(cur_fnt)) ∧ (c ≠ 0) then
    begin if font_check(cur_fnt) ≠ 0 then
      begin new_line;
      print_ln(´---beware:␣check␣sums␣do␣not␣agree!␣␣␣(´, c : 1, ´␣vs.␣´, font_check(cur_fnt) : 1, ´)´);
      mark_harmless;
      end;
    if u then font_check(cur_fnt) ← c;
    end;
  end;

procedure check_design_size(d : int_32);   { compare font_design(cur_fnt) with d }
  begin if abs(d − font_design(cur_fnt)) > 2 then
    begin new_line; print_ln(´---beware:␣design␣sizes␣do␣not␣agree!␣␣␣(´, d : 1, ´␣vs.␣´,
        font_design(cur_fnt) : 1, ´)´); mark_error;
    end;
  end;

function check_width(w : int_32): width_pointer;   { compare widths[font_width(cur_fnt)(cur_res)] with w }
  var wp: width_pointer;   { pointer to TFM width value }
  begin if (cur_res ≥ font_bc(cur_fnt)) ∧ (cur_res ≤ font_ec(cur_fnt)) then
    wp ← font_width(cur_fnt)(cur_res)
  else wp ← invalid_width;
  if wp = invalid_width then
    begin print_nl(´Bad␣char␣´, cur_res : 1);
    if cur_ext ≠ 0 then print(´.´, cur_ext : 1);
    print(´␣font␣´, cur_fnt : 1); print_font(cur_fnt); abort(´␣(compare␣TFM␣file)´);
    end;
  if w ≠ widths[wp] then
    begin new_line;
    print_ln(´---beware:␣char␣widths␣do␣not␣agree!␣␣␣(´, w : 1, ´␣vs.␣´, widths[wp] : 1, ´)´);
    mark_error;
    end;
  check_width ← wp;
  end;
```

99. The *load_font* procedure reads the `TFM` file for a font and puts the data extracted into position *cur_fnt* of the font data arrays.

```
procedure load_font;   { reads a TFM file }
  var l: int_16;   { loop index }
    p: char_pointer;   { index into char_widths }
    q: width_pointer;   { index into widths }
    bc, ec: int_15;   { first and last character in this font }
    lh: int_15;   { length of header in four byte words }
    nw: int_15;   { number of words in width table }
    w: int_32;   { a four byte integer }
    ⟨ Variables for scaling computation 103 ⟩
  begin print(´TFM:␣font␣´, cur_fnt : 1); print_font(cur_fnt); font_type(cur_fnt) ← loaded_font;
  ⟨ TFM: Open tfm_file 95 ⟩;
  ⟨ TFM: Read past the header data 101 ⟩;
  ⟨ TFM: Store character-width indices 102 ⟩;
  ⟨ TFM: Read and convert the width values 105 ⟩;
  ⟨ TFM: Convert character-width indices to character-width pointers 106 ⟩;
  close_in(tfm_file);
  device ⟨ Initialize device dependent data for a font 197 ⟩ ecived
  d_print(´␣loaded␣at␣´, font_scaled(cur_fnt) : 1, ´␣DVI␣units´); print_ln(´.´);
  end;
```

100. ⟨ Globals in the outer block 17 ⟩ +≡

```
tfm_conv: real;   { DVI units per absolute TFM unit }
```

101. We will use the following `WEB` macros to construct integers from two or four of the four bytes read by *read_tfm_word*.

```
define tfm_b01(#) ≡ { tfm_b0 .. tfm_b1 as non-negative integer }
        if tfm_b0 > 127 then bad_font
        else # ← tfm_b0 * 256 + tfm_b1
define tfm_b23(#) ≡ { tfm_b2 .. tfm_b3 as non-negative integer }
        if tfm_b2 > 127 then bad_font
        else # ← tfm_b2 * 256 + tfm_b3
define tfm_squad(#) ≡ { tfm_b0 .. tfm_b3 as signed integer }
        if tfm_b0 < 128 then # ← ((tfm_b0 * 256 + tfm_b1) * 256 + tfm_b2) * 256 + tfm_b3
        else # ← (((tfm_b0 − 256) * 256 + tfm_b1) * 256 + tfm_b2) * 256 + tfm_b3
define tfm_uquad ≡ { tfm_b0 .. tfm_b3 as unsigned integer }
        (((tfm_b0 * 256 + tfm_b1) * 256 + tfm_b2) * 256 + tfm_b3)
```

```
⟨ TFM: Read past the header data 101 ⟩ ≡
  read_tfm_word; tfm_b23(lh); read_tfm_word; tfm_b01(bc); tfm_b23(ec);
  if ec < bc then
    begin bc ← 1; ec ← 0;
    end
  else if ec > 255 then bad_font;
  read_tfm_word; tfm_b01(nw);
  if (nw = 0) ∨ (nw > 256) then bad_font;
  for l ← −2 to lh do
    begin read_tfm_word;
    if l = 1 then
      begin tfm_squad(w); check_check_sum(w, true);
      end
    else if l = 2 then
        begin if tfm_b0 > 127 then bad_font;
        check_design_size(round(tfm_conv * tfm_uquad));
        end;
    end
```

This code is used in section 99.

102. The width indices for the characters are stored in positions *n_chars* through $n\_chars - bc + ec$ of the *char_widths* array; if characters on either end of the range *bc* .. *ec* do not exist, they are ignored and the range is adjusted accordingly.

```
⟨ TFM: Store character-width indices 102 ⟩ ≡
  read_tfm_word;
  while (tfm_b0 = 0) ∧ (bc ≤ ec) do
    begin incr(bc); read_tfm_word;
    end;
  font_bc(cur_fnt) ← bc; font_chars(cur_fnt) ← n_chars − bc;
  if ec ≥ max_chars − font_chars(cur_fnt) then overflow(str_chars, max_chars);
  for l ← bc to ec do
    begin char_widths[n_chars] ← tfm_b0; incr(n_chars); read_tfm_word;
    end;
  while (char_widths[n_chars − 1] = 0) ∧ (ec ≥ bc) do
    begin decr(n_chars); decr(ec);
    end;
  font_ec(cur_fnt) ← ec
```

This code is used in section 99.

103. The most important part of *load_font* is the width computation, which involves multiplying the relative widths in the `TFM` file by the scaling factor in the `DVI` file. A similar computation is used for dimensions read from `VF` files. This fixed-point multiplication must be done with precisely the same accuracy by all `DVI`-reading programs, in order to validate the assumptions made by `DVI`-writing programs like TeX82.

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

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

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

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

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

```
define tfm_fix3u ≡ { convert tfm_b1 .. tfm_b3 to an unsigned scaled dimension }
        (((((tfm_b3 ∗ z) div ´400) + (tfm_b2 ∗ z)) div ´400) + (tfm_b1 ∗ z)) div beta
define tfm_fix4(#) ≡ { convert tfm_b0 .. tfm_b3 to a scaled dimension }
        # ← tfm_fix3u;
        if tfm_b0 > 0 then
          if tfm_b0 = 255 then Decr(#)(alpha)
          else bad_font
define tfm_fix3(#) ≡ { convert tfm_b1 .. tfm_b3 to a scaled dimension }
        # ← tfm_fix3u; if tfm_b1 > 127 then Decr(#)(alpha)
define tfm_fix2 ≡ { convert tfm_b2 .. tfm_b3 to a scaled dimension }
        if tfm_b2 > 127 then tfm_b1 ← 255
        else tfm_b1 ← 0;
        tfm_fix3
define tfm_fix1 ≡ { convert tfm_b3 to a scaled dimension }
        if tfm_b3 > 127 then tfm_b1 ← 255
        else tfm_b1 ← 0;
        tfm_b2 ← tfm_b1; tfm_fix3
```

⟨Variables for scaling computation 103⟩ ≡

```
z: int_32;  { multiplier }
alpha: int_32;  { correction for negative values }
beta: int_15;  { divisor }
```

This code is used in sections 99 and 142.

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

```
alpha ← 16;
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.

105. The first width value, which indicates that a character does not exist and which must vanish, is converted to *invalid_width*; the other width values are scaled by *font_scaled*(*cur_fnt*) and converted to width pointers by *make_width*. The resulting width pointers are stored temporarily in the *char_widths* array, following the with indices.

```
⟨ TFM: Read and convert the width values 105 ⟩ ≡
  if $nw - 1 > max\_chars - n\_chars$ then overflow(str_chars, max_chars);
  if $(tfm\_b0 \neq 0) \lor (tfm\_b1 \neq 0) \lor (tfm\_b2 \neq 0) \lor (tfm\_b3 \neq 0)$ then bad_font
  else char_widths[n_chars] ← invalid_width;
  z ← font_scaled(cur_fnt); ⟨ Replace $z$ by $z'$ and compute $\alpha, \beta$ 104 ⟩;
  for p ← n_chars + 1 to n_chars + nw − 1 do
    begin read_tfm_word; tfm_fix4(w); char_widths[p] ← make_width(w);
    end
```

This code is used in section 99.

106. We simply translate the width indices into width pointers. In addition we initialize the character packets with the invalid packet.

```
⟨ TFM: Convert character-width indices to character-width pointers 106 ⟩ ≡
  for p ← font_chars(cur_fnt) + bc to n_chars − 1 do
    begin q ← char_widths[n_chars + char_widths[p]]; char_widths[p] ← q;
    char_packets[p] ← invalid_packet;
    end
```

This code is used in section 99.

107. When processing a font definition we put the data extracted from the DVI or VF file into position *nf* of the font data arrays and call *define_font* to obtain the internal font number for this font. The parameter *load* is true if the TFM file should be loaded.

```
function define_font(load : boolean): font_number;
  var save_fnt: font_number;  { used to save cur_fnt }
  begin save_fnt ← cur_fnt;  { save }
  cur_fnt ← 0;
  while (font_name(cur_fnt) ≠ font_name(nf)) ∨ (font_scaled(cur_fnt) ≠ font_scaled(nf)) do
    incr(cur_fnt);
  d_print(´␣=>␣´, cur_fnt : 1); print_font(cur_fnt);
  if cur_fnt < nf then
    begin check_check_sum(font_check(nf), true); check_design_size(font_design(nf));
    debug if font_type(cur_fnt) = defined_font then print(´␣defined´)
    else print(´␣loaded´);
    print(´␣previously´);
    gubed
    end
  else begin if nf = max_fonts then overflow(str_fonts, max_fonts);
    incr(nf); font_font(cur_fnt) ← invalid_font; font_type(cur_fnt) ← defined_font; d_print(´␣defined´);
    end;
  print_ln(´.´);
  if load ∧ (font_type(cur_fnt) = defined_font) then load_font;
  define_font ← cur_fnt; cur_fnt ← save_fnt;  { restore }
  end;
```

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 17 ⟩ +≡
dvi_file: byte_file;   { the stuff we are DVIcopying }
dvi_loc: int_32;   { where we are about to look, in dvi_file }
```

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 ⟩ +≡
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 ⟩ ≡
  reset(dvi_file);   { prepares to read packed bytes from dvi_file }
  dvi_loc ← 0;
```

This code is used in section 241.

111. Reading the DVI file should be done as efficient as possible for a particular system; on many systems this means that a large number of bytes from *dvi_file* is read into a buffer and will then be extracted from that buffer. In order to simplify such system dependent changes we use a pair of WEB macros: *dvi_byte* extracts the next DVI byte and *dvi_eof* is *true* if we have reached the end of the DVI file. Here we give simple minded definitions for these macros in terms of standard Pascal.

```
  define dvi_eof ≡ eof(dvi_file)   { has the DVI file been exhausted? }
  define dvi_byte(#) ≡
          if dvi_eof then bad_dvi
          else read(dvi_file, #)   { obtain next DVI byte }
```

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 set_pos(dvi_file, −1); dvi_length ← cur_pos(dvi_file);
  end;

procedure dvi_move(n : int_32);
  begin set_pos(dvi_file, n); dvi_loc ← n;
  end;
```

113. We need seven simple functions to read the next byte or bytes from *dvi_file*.

```
function dvi_sbyte: int_8;  { returns the next byte, signed }
  begin_byte (dvi_byte); incr(dvi_loc); comp_sbyte(dvi_sbyte);
  end;
function dvi_ubyte: int_8u;  { returns the next byte, unsigned }
  begin_byte (dvi_byte); incr(dvi_loc); comp_ubyte(dvi_ubyte);
  end;
function dvi_spair: int_16;  { returns the next two bytes, signed }
  begin_pair (dvi_byte); Incr(dvi_loc)(2); comp_spair(dvi_spair);
  end;
function dvi_upair: int_16u;  { returns the next two bytes, unsigned }
  begin_pair (dvi_byte); Incr(dvi_loc)(2); comp_upair(dvi_upair);
  end;
function dvi_strio: int_24;  { returns the next three bytes, signed }
  begin_trio (dvi_byte); Incr(dvi_loc)(3); comp_strio(dvi_strio);
  end;
function dvi_utrio: int_24u;  { returns the next three bytes, unsigned }
  begin_trio (dvi_byte); Incr(dvi_loc)(3); comp_utrio(dvi_utrio);
  end;
function dvi_squad: int_32;  { returns the next four bytes, signed }
  begin_quad (dvi_byte); Incr(dvi_loc)(4); comp_squad(dvi_squad);
  end;
```

114. Three other functions are used in cases where a four byte integer (which is always signed) must have a non-negative value, a positive value, or is a pointer which must be either positive or $= -1$.

```
function dvi_uquad: int_31;  { result must be non-negative }
  var x: int_32;
  begin x ← dvi_squad;
  if x < 0 then bad_dvi
  else dvi_uquad ← x;
  end;
function dvi_pquad: int_31;  { result must be positive }
  var x: int_32;
  begin x ← dvi_squad;
  if x ≤ 0 then bad_dvi
  else dvi_pquad ← x;
  end;
function dvi_pointer: int_32;  { result must be positive or = −1 }
  var x: int_32;
  begin x ← dvi_squad;
  if (x ≤ 0) ∧ (x ≠ −1) then bad_dvi
  else dvi_pointer ← x;
  end;
```

115. Given the structure of the `DVI` commands it is fairly obvious that their interpretation consists of two steps: First zero to four bytes are read in order to obtain the value of the first parameter (e.g., zero bytes for *set_char_0*, four bytes for *set4*); then, depending on the command class, a specific action is performed (e.g., typeset a character but don't move the reference point for *put1* .. *put4*).

The `DVItype` program uses large case statements for both steps; unfortunately some Pascal compilers fail to implement large case statements efficiently – in particular those as the one used in the *first_par* function of `DVItype`. Here we use a pair of look up tables: *dvi_par* determines how to obtain the value of the first parameter, and *dvi_cl* determines the command class.

A slight complication arises from the fact that we want to decompose the character code of each character to be typeset into a residue $0 \le char\_res < 256$ and extension: $char\_code = char\_res + 256 * char\_ext$; the `TFM` widths as well as the pixel widths for a given resolution are the same for all characters in a font with the same residue.

define $two\_cases(\#) \equiv \#, \# + 1$
define $three\_cases(\#) \equiv \#, \# + 1, \# + 2$
define $five\_cases(\#) \equiv \#, \# + 1, \# + 2, \# + 3, \# + 4$

116. First we define the values used as array elements of *dvi_par*; we distinguish between pure numbers and dimensions because dimensions read from a `VF` file must be scaled.

define *char_par* = 0 { character for *set* and *put* }
define *no_par* = 1 { no parameter }
define *dim1_par* = 2 { one-byte signed dimension }
define *num1_par* = 3 { one-byte unsigned number }
define *dim2_par* = 4 { two-byte signed dimension }
define *num2_par* = 5 { two-byte unsigned number }
define *dim3_par* = 6 { three-byte signed dimension }
define *num3_par* = 7 { three-byte unsigned number }
define *dim4_par* = 8 { four-byte signed dimension }
define *num4_par* = 9 { four-byte signed number }
define *numu_par* = 10 { four-byte non-negative number }
define *rule_par* = 11 { dimensions for *set_rule* and *put_rule* }
define *fnt_par* = 12 { font for *fnt_num* commands }
define *max_par* = 12 { largest possible value }

⟨ Types in the outer block 7 ⟩ +≡
cmd_par = *char_par* .. *max_par*;

117. Here we declare the array *dvi_par*.

⟨ Globals in the outer block 17 ⟩ +≡
dvi_par: **packed array** [*eight_bits*] **of** *cmd_par*;

118. And here we initialize it.

```
⟨Set initial values 18⟩ +≡
  for i ← 0 to put1 + 3 do dvi_par[i] ← char_par;
  for i ← nop to 255 do dvi_par[i] ← no_par;
  dvi_par[set_rule] ← rule_par; dvi_par[put_rule] ← rule_par;
  dvi_par[right1] ← dim1_par; dvi_par[right1 + 1] ← dim2_par; dvi_par[right1 + 2] ← dim3_par;
  dvi_par[right1 + 3] ← dim4_par;
  for i ← fnt_num_0 to fnt_num_0 + 63 do dvi_par[i] ← fnt_par;
  dvi_par[fnt1] ← num1_par; dvi_par[fnt1 + 1] ← num2_par; dvi_par[fnt1 + 2] ← num3_par;
  dvi_par[fnt1 + 3] ← num4_par;
  dvi_par[xxx1] ← num1_par; dvi_par[xxx1 + 1] ← num2_par; dvi_par[xxx1 + 2] ← num3_par;
  dvi_par[xxx1 + 3] ← numu_par;
  for i ← 0 to 3 do
    begin dvi_par[i + w1] ← dvi_par[i + right1]; dvi_par[i + x1] ← dvi_par[i + right1];
    dvi_par[i + down1] ← dvi_par[i + right1]; dvi_par[i + y1] ← dvi_par[i + right1];
    dvi_par[i + z1] ← dvi_par[i + right1]; dvi_par[i + fnt_def1] ← dvi_par[i + fnt1];
    end;
```

119. Next we define the values used as array elements of *dvi_cl*; several DVI commands (e.g., *nop*, *bop*, *eop*, *pre*, *post*) will always be treated separately and are therefore assigned to the invalid class here.

```
  define char_cl = 0
  define rule_cl = char_cl + 1
  define xxx_cl = char_cl + 2
  define push_cl = 3
  define pop_cl = 4
  define w0_cl = 5
  define x0_cl = w0_cl + 1
  define right_cl = w0_cl + 2
  define w_cl = w0_cl + 3
  define x_cl = w0_cl + 4
  define y0_cl = 10
  define z0_cl = y0_cl + 1
  define down_cl = y0_cl + 2
  define y_cl = y0_cl + 3
  define z_cl = y0_cl + 4
  define fnt_cl = 15
  define fnt_def_cl = 16
  define invalid_cl = 17
  define max_cl = invalid_cl   { largest possible value }

⟨Types in the outer block 7⟩ +≡
  cmd_cl = char_cl .. max_cl;
```

120. Here we declare the array *dvi_cl*.

```
⟨Globals in the outer block 17⟩ +≡
dvi_cl: packed array [eight_bits] of cmd_cl;
```

121. And here we initialize it.

```
⟨Set initial values 18⟩ +≡
  for $i \leftarrow set\_char\_0$ to $put1 + 3$ do $dvi\_cl[i] \leftarrow char\_cl$;
  $dvi\_cl[set\_rule] \leftarrow rule\_cl$; $dvi\_cl[put\_rule] \leftarrow rule\_cl$;
  $dvi\_cl[nop] \leftarrow invalid\_cl$; $dvi\_cl[bop] \leftarrow invalid\_cl$; $dvi\_cl[eop] \leftarrow invalid\_cl$;
  $dvi\_cl[push] \leftarrow push\_cl$; $dvi\_cl[pop] \leftarrow pop\_cl$;
  $dvi\_cl[w0] \leftarrow w0\_cl$; $dvi\_cl[x0] \leftarrow x0\_cl$;
  $dvi\_cl[y0] \leftarrow y0\_cl$; $dvi\_cl[z0] \leftarrow z0\_cl$;
  for $i \leftarrow 0$ to 3 do
    begin $dvi\_cl[i + right1] \leftarrow right\_cl$; $dvi\_cl[i + w1] \leftarrow w\_cl$; $dvi\_cl[i + x1] \leftarrow x\_cl$;
    $dvi\_cl[i + down1] \leftarrow down\_cl$; $dvi\_cl[i + y1] \leftarrow y\_cl$; $dvi\_cl[i + z1] \leftarrow z\_cl$;
    $dvi\_cl[i + xxx1] \leftarrow xxx\_cl$; $dvi\_cl[i + fnt\_def1] \leftarrow fnt\_def\_cl$;
    end;
  for $i \leftarrow fnt\_num\_0$ to $fnt1 + 3$ do $dvi\_cl[i] \leftarrow fnt\_cl$;
  for $i \leftarrow pre$ to 255 do $dvi\_cl[i] \leftarrow invalid\_cl$;
```

122. A few small arrays are used to generate `DVI` commands.

```
⟨Globals in the outer block 17⟩ +≡
dvi_char_cmd: array [boolean] of eight_bits;  { put1 and set1 }
dvi_rule_cmd: array [boolean] of eight_bits;  { put_rule and set_rule }
dvi_right_cmd: array [right_cl .. x_cl] of eight_bits;  { right1, w1, and x1 }
dvi_down_cmd: array [down_cl .. z_cl] of eight_bits;  { down1, y1, and z1 }
```

123. ⟨Set initial values 18⟩ +≡

```
  dvi_char_cmd[false] ← put1; dvi_char_cmd[true] ← set1;
  dvi_rule_cmd[false] ← put_rule; dvi_rule_cmd[true] ← set_rule;
  dvi_right_cmd[right_cl] ← right1; dvi_right_cmd[w_cl] ← w1; dvi_right_cmd[x_cl] ← x1;
  dvi_down_cmd[down_cl] ← down1; dvi_down_cmd[y_cl] ← y1; dvi_down_cmd[z_cl] ← z1;
```

124. The global variables *cur_cmd*, *cur_parm*, and *cur_class* are used for the current `DVI` command, its first parameter (if any), and its command class respectively.

```
⟨Globals in the outer block 17⟩ +≡
cur_cmd: eight_bits;  { current DVI command byte }
cur_parm: int_32;  { its first parameter (if any) }
cur_class: cmd_cl;  { its class }
```

125. When typesetting a character or rule, the boolean variable *cur_upd* is *true* for *set* commands, *false* for *put* commands.

```
⟨Globals in the outer block 17⟩ +≡
cur_cp: char_pointer;  { char_widths index for the current character }
cur_wp: width_pointer;  { width pointer of the current character }
cur_upd: boolean;  { is this a set or set_rule command ? }
cur_v_dimen: int_32;  { a vertical dimension }
cur_h_dimen: int_32;  { a horizontal dimension }
```

126. ⟨Set initial values 18⟩ +≡

```
  cur_cp ← 0; cur_wp ← invalid_width;  { so they can be saved and restored! }
```

127. The *dvi_first_par* procedure first reads DVI command bytes into *cur_cmd* until $cur\_cmd \neq nop$; then *cur_parm* is set to the value of the first parameter (if any) and *cur_class* to the command class.

```
  define set_cur_char(#) ≡ { set up cur_res, cur_ext, and cur_upd }
          begin cur_ext ← 0;
          if cur_cmd < set1 then
             begin cur_res ← cur_cmd; cur_upd ← true
             end
          else begin cur_res ← #; cur_upd ← (cur_cmd < put1); Decr(cur_cmd)(dvi_char_cmd[cur_upd]);
             while cur_cmd > 0 do
                begin if cur_cmd = 3 then
                   if cur_res > 127 then cur_ext ← −1;
                cur_ext ← cur_ext ∗ 256 + cur_res; cur_res ← #; decr(cur_cmd);
                end;
             end;
          end
procedure dvi_first_par;
  begin repeat cur_cmd ← dvi_ubyte;
  until cur_cmd ≠ nop; { skip over nops }
  case dvi_par[cur_cmd] of
  char_par: set_cur_char(dvi_ubyte);
  no_par: do_nothing;
  dim1_par: cur_parm ← dvi_sbyte;
  num1_par: cur_parm ← dvi_ubyte;
  dim2_par: cur_parm ← dvi_spair;
  num2_par: cur_parm ← dvi_upair;
  dim3_par: cur_parm ← dvi_strio;
  num3_par: cur_parm ← dvi_utrio;
  two_cases(dim4_par): cur_parm ← dvi_squad; { dim4_par and num4_par }
  numu_par: cur_parm ← dvi_uquad;
  rule_par: begin cur_v_dimen ← dvi_squad; cur_h_dimen ← dvi_squad; cur_upd ← (cur_cmd = set_rule);
     end;
  fnt_par: cur_parm ← cur_cmd − fnt_num_0;
  othercases abort(´internal␣error´);
  endcases; cur_class ← dvi_cl[cur_cmd];
  end;
```

128. The global variable *dvi_nf* is used for the number of different DVI fonts defined so far; their external font numbers (as extracted from the DVI file) are stored in the array *dvi_e_fnts*, the corresponding internal font numbers used internally by DVIcopy are stored in the array *dvi_i_fnts*.

```
⟨ Globals in the outer block 17 ⟩ +≡
dvi_e_fnts: array [font_number] of int_32; { external font numbers }
dvi_i_fnts: array [font_number] of font_number; { corresponding internal font numbers }
dvi_nf: font_number; { number of DVI fonts defined so far }
```

129. ⟨ Set initial values 18 ⟩ +≡

```
  dvi_nf ← 0;
```

130. The *dvi_font* procedure sets *cur_fnt* to the internal font number corresponding to the external font number *cur_parm* (or aborts the program if such a font was never defined).

```
procedure dvi_font;  { computes cur_fnt corresponding to cur_parm }
  var f: font_number;  { where the font is sought }
  begin ⟨ DVI: Locate font cur_parm 131 ⟩;
  if f = dvi_nf then bad_dvi;
  cur_fnt ← dvi_i_fnts[f];
  if font_type(cur_fnt) = defined_font then load_font;
  end;
```

131. ⟨ DVI: Locate font *cur_parm* 131 ⟩ ≡

```
  f ← 0; dvi_e_fnts[dvi_nf] ← cur_parm;
  while cur_parm ≠ dvi_e_fnts[f] do incr(f)
```

This code is used in sections 130 and 132.

132. Finally the *dvi_do_font* procedure is called when one of the commands *fnt_def1* .. *fnt_def4* and its first parameter have been read from the `DVI` file; the argument indicates whether this should be the second definition of the font (*true*) or not (*false*).

```
procedure dvi_do_font(second : boolean);
  var f: font_number;  { where the font is sought }
    k: int_15;  { general purpose variable }
  begin print(´DVI:␣font␣´, cur_parm : 1); ⟨ DVI: Locate font cur_parm 131 ⟩;
  if (f = dvi_nf) = second then bad_dvi;
  font_check(nf) ← dvi_squad; font_scaled(nf) ← dvi_pquad; font_design(nf) ← dvi_pquad; k ← dvi_ubyte;
  pckt_room(1); append_byte(k); Incr(k)(dvi_ubyte); pckt_room(k);
  while k > 0 do
    begin append_byte(dvi_ubyte); decr(k);
    end;
  font_name(nf) ← make_packet;  { the font area plus name }
  dvi_i_fnts[dvi_nf] ← define_font(false);
  if ¬second then
    begin if dvi_nf = max_fonts then overflow(str_fonts, max_fonts);
    incr(dvi_nf);
    end
  else if dvi_i_fnts[f] ≠ dvi_i_fnts[dvi_nf] then bad_dvi;
  end;
```

133. Low-level VF input routines. A detailed description of the `VF` file format can be found in the documentation of `VFtoVP`; here we just define symbolic names for some of the `VF` command bytes.

define *long_char* = 242 { `VF` command for general character packet }
define *vf_id* = 202 { identifies `VF` files }

134. The program uses the binary file variable *vf_file* for input from `VF` files; *vf_loc* is the number of the byte about to be read next from *vf_file*.

⟨ Globals in the outer block 17 ⟩ +≡
vf_file: *byte_file*; { a `VF` file }
vf_loc: *int_32*; { where we are about to look, in *vf_file* }
vf_limit: *int_32*; { value of *vf_loc* at end of a character packet }
vf_ext: *pckt_pointer*; { extension for `VF` files }
vf_cur_fnt: *font_number*; { current font number in a `VF` file }

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

136. If a `VF` file is badly malformed, we say *bad_font*; this procedure gives an error message which refers the user to `VFtoVP` and `VPtoVF`, and terminates `DVIcopy`.

⟨ Cases for *bad_font* 136 ⟩ ≡
vf_font_type: **begin** *print*(´Bad␣VF␣file´); *print_font*(*cur_fnt*); *print_ln*(´␣loc=´, *vf_loc* : 1);
abort(´Use␣VFtoVP/VPtoVF␣to␣diagnose␣and␣correct␣the␣problem´);
end;

This code is used in section 94.

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. The string variable *VF_default_area* contains the name of this area.

define *VF_default_area_name* ≡ ´TeXvfonts:´ { change this to the correct name }
define *VF_default_area_name_length* = 10 { change this to the correct length }

⟨ Globals in the outer block 17 ⟩ +≡
VF_default_area: **packed array** [1 .. *VF_default_area_name_length*] **of** *char*;

138. ⟨ Set initial values 18 ⟩ +≡
VF_default_area ← *VF_default_area_name*;

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

⟨ VF: Open *vf_file* or **goto** *not_found* 139 ⟩ ≡
make_font_name(*VF_default_area_name_length*)(*VF_default_area*)(*vf_ext*); *reset*(*vf_file*, *cur_name*);
if *eof*(*vf_file*) **then goto** *not_found*;
vf_loc ← 0

This code is used in section 151.

140. Reading a `VF` file should be done as efficient as possible for a particular system; on many systems this means that a large number of bytes from *vf_file* is read into a buffer and will then be extracted from that buffer. In order to simplify such system dependent changes we use a pair of `WEB` macros: *vf_byte* extracts the next `VF` byte and *vf_eof* is *true* if we have reached the end of the `VF` file. Here we give simple minded definitions for these macros in terms of standard Pascal.

```
define vf_eof ≡ eof(vf_file)   { has the VF file been exhausted? }
define vf_byte(#) ≡
        if vf_eof then bad_font
        else read(vf_file, #)   { obtain next VF byte }
```

141. We need several simple functions to read the next byte or bytes from *vf_file*.

```
function vf_ubyte: int_8u;   { returns the next byte, unsigned }
  begin_byte (vf_byte); incr(vf_loc); comp_ubyte(vf_ubyte);
  end;
function vf_upair: int_16u;   { returns the next two bytes, unsigned }
  begin_pair (vf_byte); Incr(vf_loc)(2); comp_upair(vf_upair);
  end;
function vf_strio: int_24;   { returns the next three bytes, signed }
  begin_trio (vf_byte); Incr(vf_loc)(3); comp_strio(vf_strio);
  end;
function vf_utrio: int_24u;   { returns the next three bytes, unsigned }
  begin_trio (vf_byte); Incr(vf_loc)(3); comp_utrio(vf_utrio);
  end;
function vf_squad: int_32;   { returns the next four bytes, signed }
  begin_quad (vf_byte); Incr(vf_loc)(4); comp_squad(vf_squad);
  end;
```

142. All dimensions in a `VF` file, except the design sizes of a virtual font and its local fonts, are *fix_word*s that must be scaled in exactly the same way as the character widths from a `TFM` file; we can use the same code, but this time z, *alpha*, and *beta* are global variables.

```
⟨ Globals in the outer block 17 ⟩ +≡
  ⟨ Variables for scaling computation 103 ⟩
```

143. We need five functions to read the next byte or bytes and convert a *fix_word* to a scaled dimension.

```
function vf_fix1: int_32;  { returns the next byte as scaled value }
  var x: int_32;  { accumulator }
  begin vf_byte(tfm_b3); incr(vf_loc); tfm_fix1(x); vf_fix1 ← x;
  end;
function vf_fix2: int_32;  { returns the next two bytes as scaled value }
  var x: int_32;  { accumulator }
  begin vf_byte(tfm_b2); vf_byte(tfm_b3); Incr(vf_loc)(2); tfm_fix2(x); vf_fix2 ← x;
  end;
function vf_fix3: int_32;  { returns the next three bytes as scaled value }
  var x: int_32;  { accumulator }
  begin vf_byte(tfm_b1); vf_byte(tfm_b2); vf_byte(tfm_b3); Incr(vf_loc)(3);
  tfm_fix3(x); vf_fix3 ← x;
  end;
function vf_fix3u: int_32;  { returns the next three bytes as scaled value }
  begin vf_byte(tfm_b1); vf_byte(tfm_b2); vf_byte(tfm_b3); Incr(vf_loc)(3);
  vf_fix3u ← tfm_fix3u;
  end;
function vf_fix4: int_32;  { returns the next four bytes as scaled value }
  var x: int_32;  { accumulator }
  begin vf_byte(tfm_b0); vf_byte(tfm_b1); vf_byte(tfm_b2); vf_byte(tfm_b3); Incr(vf_loc)(4);
  tfm_fix4(x); vf_fix4 ← x;
  end;
```

144. Three other functions are used in cases where the result must have a non-negative value or a positive value.

```
function vf_uquad: int_31;  { result must be non-negative }
  var x: int_32;
  begin x ← vf_squad;
  if x < 0 then bad_font else vf_uquad ← x;
  end;
function vf_pquad: int_31;  { result must be positive }
  var x: int_32;
  begin x ← vf_squad;
  if x ≤ 0 then bad_font else vf_pquad ← x;
  end;
function vf_fixp: int_31;  { result must be positive }
  begin vf_byte(tfm_b0); vf_byte(tfm_b1); vf_byte(tfm_b2); vf_byte(tfm_b3); Incr(vf_loc)(4);
  if tfm_b0 > 0 then bad_font;
  vf_fixp ← tfm_fix3u;
  end;
```

145. The *vf_first_par* procedure first reads a `VF` command byte into *cur_cmd*; then *cur_parm* is set to the value of the first parameter (if any) and *cur_class* to the command class.

```
  define set_cur_wp_end(#) ≡
            if cur_wp = invalid_width then #
  define set_cur_wp(#) ≡ { set cur_wp to the char's width pointer }
         cur_wp ← invalid_width;
         if # ≠ invalid_font then
            if (cur_res ≥ font_bc(#)) ∧ (cur_res ≤ font_ec(#)) then
               begin cur_cp ← font_chars(#) + cur_res; cur_wp ← char_widths[cur_cp];
               end;
         set_cur_wp_end
procedure vf_first_par;
  begin cur_cmd ← vf_ubyte;
  case dvi_par[cur_cmd] of
  char_par: begin set_cur_char(vf_ubyte); set_cur_wp(vf_cur_fnt)(bad_font);
    end;
  no_par: do_nothing;
  dim1_par: cur_parm ← vf_fix1;
  num1_par: cur_parm ← vf_ubyte;
  dim2_par: cur_parm ← vf_fix2;
  num2_par: cur_parm ← vf_upair;
  dim3_par: cur_parm ← vf_fix3;
  num3_par: cur_parm ← vf_utrio;
  dim4_par: cur_parm ← vf_fix4;
  num4_par: cur_parm ← vf_squad;
  numu_par: cur_parm ← vf_uquad;
  rule_par: begin cur_v_dimen ← vf_fix4; cur_h_dimen ← vf_fix4; cur_upd ← (cur_cmd = set_rule);
    end;
  fnt_par: cur_parm ← cur_cmd − fnt_num_0;
  othercases abort('internal␣error');
  endcases; cur_class ← dvi_cl[cur_cmd];
  end;
```

146. For a virtual font we set *font_type*(*f*) ← *vf_font_type*; in this case *font_font*(*f*) is the default font for character packets from virtual font *f*.

The global variable *vf_nf* is used for the number of different local fonts defined in a `VF` file so far; their external font numbers (as extracted from the `VF` file) are stored in the array *vf_e_fnts*, the corresponding internal font numbers used internally by `DVIcopy` are stored in the array *vf_i_fnts*.

```
⟨Globals in the outer block 17⟩ +≡
vf_e_fnts: array [font_number] of int_32; { external font numbers }
vf_i_fnts: array [font_number] of font_number; { corresponding internal font numbers }
vf_nf: font_number; { number of local fonts defined so far }
lcl_nf: font_number; { largest vf_nf value for any VF file }
```

147. ⟨Set initial values 18⟩ +≡

```
  lcl_nf ← 0;
```

148. The *vf_font* procedure sets *vf_cur_fnt* to the internal font number corresponding to the external font number *cur_parm* (or aborts the program if such a font was never defined).

```
procedure vf_font; { computes vf_cur_fnt corresponding to cur_parm }
  var f: font_number; { where the font is sought }
  begin ⟨ VF: Locate font cur_parm 149 ⟩;
  if f = vf_nf then bad_font;
  vf_cur_fnt ← vf_i_fnts[f];
  end;
```

149. ⟨ VF: Locate font *cur_parm* 149 ⟩ ≡

```
  f ← 0; vf_e_fnts[vf_nf] ← cur_parm;
  while cur_parm ≠ vf_e_fnts[f] do incr(f)
```

This code is used in sections 148 and 150.

150. Finally the *vf_do_font* procedure is called when one of the commands *fnt_def1* . . *fnt_def4* and its first parameter have been read from the VF file.

```
procedure vf_do_font;
  var f: font_number; { where the font is sought }
    k: int_15; { general purpose variable }
  begin print(´VF:␣font␣´, cur_parm : 1);
  ⟨ VF: Locate font cur_parm 149 ⟩;
  if f ≠ vf_nf then bad_font;
  font_check(nf) ← vf_squad; font_scaled(nf) ← vf_fixp; font_design(nf) ← round(tfm_conv ∗ vf_pquad);
  k ← vf_ubyte; pckt_room(1); append_byte(k); Incr(k)(vf_ubyte); pckt_room(k);
  while k > 0 do
    begin append_byte(vf_ubyte); decr(k);
    end;
  font_name(nf) ← make_packet; { the font area plus name }
  vf_i_fnts[vf_nf] ← define_font(true);
  if vf_nf = lcl_nf then
    if lcl_nf = max_fonts then overflow(str_fonts, max_fonts)
    else incr(lcl_nf);
  incr(vf_nf);
  end;
```

151. Reading VF files. The *do_vf* function attempts to read the `VF` file for a font and returns *false* if the `VF` file could not be found; otherwise the font type is changed to *vf_font_type*.

```
function do_vf: boolean;   { read a VF file }
  label reswitch, done, not_found, exit;
  var temp_byte: int_8u;   { byte for temporary variables }
    k: byte_pointer;   { index into byte_mem }
    l: int_15;   { general purpose variable }
    save_ext: int_24;   { used to save cur_ext }
    save_res: int_8u;   { used to save cur_res }
    save_cp: width_pointer;   { used to save cur_cp }
    save_wp: width_pointer;   { used to save cur_wp }
    save_upd: boolean;   { used to save cur_upd }
    vf_wp: width_pointer;   { width pointer for the current character packet }
    vf_fnt: font_number;   { current font in the current character packet }
    move_zero: boolean;   { true if rule 1 is used }
    last_pop: boolean;   { true if final pop has been manufactured }
  begin ⟨VF: Open vf_file or goto not_found 139⟩;
  save_ext ← cur_ext; save_res ← cur_res; save_cp ← cur_cp; save_wp ← cur_wp; save_upd ← cur_upd;
      { save }
  font_type(cur_fnt) ← vf_font_type;
  ⟨VF: Process the preamble 152⟩;
  ⟨VF: Process the font definitions 153⟩;
  while cur_cmd ≤ long_char do ⟨VF: Build a character packet 160⟩;
  if cur_cmd ≠ post then bad_font;
  debug print(´VF␣file␣for␣font␣´, cur_fnt : 1); print_font(cur_fnt); print_ln(´␣loaded.´);
  gubed
  close_in(vf_file); cur_ext ← save_ext; cur_res ← save_res; cur_cp ← save_cp; cur_wp ← save_wp;
  cur_upd ← save_upd;   { restore }
  do_vf ← true; return;
not_found: do_vf ← false;
exit: end;
```

152. ⟨VF: Process the preamble 152⟩ ≡

```
  if vf_ubyte ≠ pre then bad_font;
  if vf_ubyte ≠ vf_id then bad_font;
  temp_byte ← vf_ubyte; pckt_room(temp_byte);
  for l ← 1 to temp_byte do append_byte(vf_ubyte);
  print(´VF␣file:␣´´´); print_packet(new_packet); print(´´´,´); flush_packet;
  check_check_sum(vf_squad, false); check_design_size(round(tfm_conv * vf_pquad));
  z ← font_scaled(cur_fnt); ⟨Replace z by z′ and compute α, β 104⟩;
  print_nl(´␣␣␣for␣font␣´, cur_fnt : 1); print_font(cur_fnt); print_ln(´.´)
```

This code is used in section 151.

153. ⟨ VF: Process the font definitions 153 ⟩ ≡

```
vf_i_fnts[0] ← invalid_font; vf_nf ← 0;
cur_cmd ← vf_ubyte;
while (cur_cmd ≥ fnt_def1) ∧ (cur_cmd ≤ fnt_def1 + 3) do
  begin case cur_cmd − fnt_def1 of
  0: cur_parm ← vf_ubyte;
  1: cur_parm ← vf_upair;
  2: cur_parm ← vf_utrio;
  3: cur_parm ← vf_squad;
  end; { there are no other cases }
  vf_do_font; cur_cmd ← vf_ubyte;
  end;
font_font(cur_fnt) ← vf_i_fnts[0]
```

This code is used in section 151.

154. The VF format specifies that the interpretation of each packet begins with $w = x = y = z = 0$; any *w0*, *x0*, *y0*, or *z0* command using these initial values will be ignored.

⟨ Types in the outer block 7 ⟩ +≡

```
vf_state = array [0 . . 1, 0 . . 1] of boolean; { state of w, x, y, and z }
```

155. As implied by the VF format the DVI commands read from the VF file are enclosed by *push* and *pop*; as we read DVI commands and append them to *byte_mem*, we perform a set of transformations in order to simplify the resulting packet: Let *zero* be any of the commands *put*, *put_rule*, *fnt_num*, *fnt*, or *xxx* which all leave the current position on the page unchanged, let *move* be any of the horizontal or vertical movement commands *right1* . . *z4*, and let *any* be any sequence of commands containing *push* and *pop* in properly nested pairs; whenever possible we apply one of the following transformation rules:

1:	*push zero* → *zero push*
2:	*move pop* → *pop*
3:	*push pop* →
4a:	*push set_char pop* → *put*
4b:	*push set pop* → *put*
4c:	*push set_rule pop* → *put_rule*
5:	*push push any pop* → *push any pop push*
6:	*push any pop pop* → *any pop*

156. In order to perform these transformations we need a stack which is indexed by *vf_ptr*, the number of *push* commands without corresponding *pop* in the packet we are building; the *vf_push_loc* array contains the locations in *byte_mem* following such *push* commands. In view of rule 5 consecutive *push* commands are never stored, the *vf_push_num* array is used to count them. The *vf_last* array indicates the type of the last non-discardable item: a character, a rule, or a group enclosed by *push* and *pop*; the *vf_last_end* array points to the ending locations and, if *vf_last* $\neq$ *vf_other*, the *vf_last_loc* array points to the starting locations of these items.

```
    define vf_set = 0   { vf_set = char_cl, last item is a set_char or set }
    define vf_rule = 1   { vf_rule = rule_cl, last item is a set_rule }
    define vf_group = 2   { last item is a group enclosed by push and pop }
    define vf_put = 3   { last item is a put }
    define vf_other = 4   { last item (if any) is none of the above }
⟨ Types in the outer block 7 ⟩ +≡
    vf_type = vf_set . . vf_other;
```

157. ⟨ Globals in the outer block 17 ⟩ +≡

```
vf_move: array [stack_pointer] of vf_state;   { state of w, x, y, and z }
vf_push_loc: array [stack_pointer] of byte_pointer;   { end of a push }
vf_last_loc: array [stack_pointer] of byte_pointer;   { start of an item }
vf_last_end: array [stack_pointer] of byte_pointer;   { end of an item }
vf_push_num: array [stack_pointer] of eight_bits;   { push count }
vf_last: array [stack_pointer] of vf_type;   { type of last item }
vf_ptr: stack_pointer;   { current number of unfinished groups }
stack_used: stack_pointer;   { largest vf_ptr or stack_ptr value }
```

158. We use two small arrays to determine the item type of a character or a rule.

```
⟨ Globals in the outer block 17 ⟩ +≡
vf_char_type: array [boolean] of vf_type;
vf_rule_type: array [boolean] of vf_type;
```

159. ⟨ Set initial values 18 ⟩ +≡

```
    vf_move[0][0][0] ← false; vf_move[0][0][1] ← false; vf_move[0][1][0] ← false; vf_move[0][1][1] ← false;
    stack_used ← 0;
    vf_char_type[false] ← vf_put; vf_char_type[true] ← vf_set;
    vf_rule_type[false] ← vf_other; vf_rule_type[true] ← vf_rule;
```

160. Here we read the first bytes of a character packet from the VF file and initialize the packet being built in *byte_mem*; the start of the whole packet is stored in *vf_push_loc*[0]. When the character packet is finished, a type is assigned to it: *vf_simple* if the packet ends with a character of the correct width, or *vf_complex* otherwise. Moreover, if such a packet for a character with extension zero consists of just one character with extension zero and the same residue, and if there is no previous packet, the whole packet is replaced by the empty packet.

```
  define vf_simple = 0   { the packet ends with a character of the correct width }
  define vf_complex = vf_simple + 1   { otherwise }
⟨ VF: Build a character packet 160 ⟩ ≡
  begin if cur_cmd < long_char then
    begin vf_limit ← cur_cmd; cur_ext ← 0; cur_res ← vf_ubyte; vf_wp ← check_width(vf_fix3u);
    end
  else begin vf_limit ← vf_uquad; cur_ext ← vf_strio; cur_res ← vf_ubyte; vf_wp ← check_width(vf_fix4);
    end;
  Incr(vf_limit)(vf_loc); vf_push_loc[0] ← byte_ptr; vf_last_end[0] ← byte_ptr; vf_last[0] ← vf_other;
  vf_ptr ← 0;
  start_packet(vf_complex); ⟨ VF: Append DVI commands to the character packet 161 ⟩;
  k ← pckt_start[pckt_ptr];
  if vf_last[0] = vf_put then
    if cur_wp = vf_wp then
      begin decr(byte_mem[k]);   { change vf_complex into vf_simple }
      if (byte_mem[k] = bi(0)) ∧ (vf_push_loc[0] = vf_last_loc[0]) ∧ (cur_ext = 0) ∧ (cur_res = pckt_res)
          then byte_ptr ← k;
      end;
  build_packet; cur_cmd ← vf_ubyte;
  end
```

This code is used in section 151.

161. For every `DVI` command read from the `VF` file some action is performed; in addition the initial *push* and the final *pop* are manufactured here.

```
⟨VF: Append DVI commands to the character packet 161⟩ ≡
  vf_cur_fnt ← font_font(cur_fnt); vf_fnt ← vf_cur_fnt;
  last_pop ← false; cur_class ← push_cl;   { initial push }
  loop
    begin reswitch: case cur_class of
    three_cases(char_cl): ⟨VF: Do a char, rule, or xxx 164⟩;
    push_cl: ⟨VF: Do a push 162⟩;
    pop_cl: ⟨VF: Do a pop 168⟩;
    two_cases(w0_cl): if vf_move[vf_ptr][0][cur_class − w0_cl] then append_one(cur_cmd);
    three_cases(right_cl): begin pckt_signed(dvi_right_cmd[cur_class], cur_parm);
      if cur_class ≥ w_cl then vf_move[vf_ptr][0][cur_class − w_cl] ← true;
      end;
    two_cases(y0_cl): if vf_move[vf_ptr][1][cur_class − y0_cl] then append_one(cur_cmd);
    three_cases(down_cl): begin pckt_signed(dvi_down_cmd[cur_class], cur_parm);
      if cur_class ≥ y_cl then vf_move[vf_ptr][1][cur_class − y_cl] ← true;
      end;
    fnt_cl: vf_font;
    fnt_def_cl: bad_font;
    invalid_cl: if cur_cmd ≠ nop then bad_font;
      othercases abort('internal␣error');
    endcases;
    if vf_loc < vf_limit then vf_first_par
    else if last_pop then goto done
      else begin cur_class ← pop_cl; last_pop ← true;   { final pop }
        end;
    end;
done: if (vf_ptr ≠ 0) ∨ (vf_loc ≠ vf_limit) then bad_font
```

This code is used in section 160.

162. For a *push* we either increase *vf_push_num* or start a new level and append a *push*.

```
  define incr_stack(#) ≡
          if # = stack_used then
            if stack_used = stack_size then overflow(str_stack, stack_size)
            else incr(stack_used);
        incr(#)

⟨VF: Do a push 162⟩ ≡
  if (vf_ptr > 0) ∧ (vf_push_loc[vf_ptr] = byte_ptr) then
    begin if vf_push_num[vf_ptr] = 255 then overflow(str_stack, 255);
    incr(vf_push_num[vf_ptr]);
    end
  else begin incr_stack(vf_ptr); ⟨VF: Start a new level 163⟩;
    vf_push_num[vf_ptr] ← 0;
    end
```

This code is used in section 161.

163.
```
⟨VF: Start a new level 163⟩ ≡
  append_one(push); vf_move[vf_ptr] ← vf_move[vf_ptr − 1]; 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.

164. When a character, a rule, or an *xxx* is appended, transformation rule 1 might be applicable.

```
⟨ VF: Do a char, rule, or xxx 164 ⟩ ≡
  begin if $(vf\_ptr = 0) \lor (byte\_ptr > vf\_push\_loc[vf\_ptr])$ then $move\_zero \gets false$
  else case cur_class of
    char_cl: $move\_zero \gets (\lnot cur\_upd) \lor (vf\_cur\_fnt \neq vf\_fnt)$;
    rule_cl: $move\_zero \gets \lnot cur\_upd$;
    xxx_cl: $move\_zero \gets true$;
    othercases abort(´internal␣error´);
    endcases;
  if move_zero then
    begin decr(byte_ptr); decr(vf_ptr);
    end;
  case cur_class of
  char_cl: ⟨ VF: Do a fnt, a char, or both 165 ⟩;
  rule_cl: ⟨ VF: Do a rule 166 ⟩;
  xxx_cl: ⟨ VF: Do an xxx 167 ⟩;
  end; { there are no other cases }
  $vf\_last\_end[vf\_ptr] \gets byte\_ptr$;
  if move_zero then
    begin incr(vf_ptr); append_one(push); $vf\_push\_loc[vf\_ptr] \gets byte\_ptr$; $vf\_last\_end[vf\_ptr] \gets byte\_ptr$;
    if $cur\_class = char\_cl$ then
      if cur_upd then goto reswitch;
    end;
  end
```

This code is used in section 161.

165. A special situation arises if transformation rule 1 is applied to a *fnt_num* of *fnt* command, but not to the *set_char* or *set* command following it; in this case *cur_upd* and *move_zero* are both *true* and the *set_char* or *set* command will be appended later.

```
⟨ VF: Do a fnt, a char, or both 165 ⟩ ≡
  begin if $vf\_cur\_fnt \neq vf\_fnt$ then
    begin $vf\_last[vf\_ptr] \gets vf\_other$; pckt_unsigned(fnt1, vf_cur_fnt); $vf\_fnt \gets vf\_cur\_fnt$;
    end;
  if $(\lnot move\_zero) \lor (\lnot cur\_upd)$ then
    begin $vf\_last[vf\_ptr] \gets vf\_char\_type[cur\_upd]$; $vf\_last\_loc[vf\_ptr] \gets byte\_ptr$;
    pckt_char(cur_upd, cur_ext, cur_res);
    end;
  end
```

This code is used in section 164.

166.

```
⟨ VF: Do a rule 166 ⟩ ≡
  begin $vf\_last[vf\_ptr] \gets vf\_rule\_type[cur\_upd]$; $vf\_last\_loc[vf\_ptr] \gets byte\_ptr$;
  append_one(dvi_rule_cmd[cur_upd]); pckt_four(cur_v_dimen); pckt_four(cur_h_dimen);
  end
```

This code is used in section 164.

167. ⟨VF: Do an *xxx* 167⟩ ≡

```
begin vf_last[vf_ptr] ← vf_other; pckt_unsigned(xxx1, cur_parm); pckt_room(cur_parm);
while cur_parm > 0 do
  begin append_byte(vf_ubyte); decr(cur_parm);
  end;
end
```

This code is used in section 164.

168. Transformation rules 2–6 are triggered by a *pop*, either read from the VF file or manufactured at the end of the packet.

⟨VF: Do a *pop* 168⟩ ≡

```
begin if vf_ptr < 1 then bad_font;
byte_ptr ← vf_last_end[vf_ptr];   { this is rule 2 }
if vf_last[vf_ptr] ≤ vf_rule then
  if vf_last_loc[vf_ptr] = vf_push_loc[vf_ptr] then ⟨VF: Prepare for rule 4 169⟩;
if byte_ptr = vf_push_loc[vf_ptr] then ⟨VF: Apply rule 3 or 4 170⟩
else begin if vf_last[vf_ptr] = vf_group then ⟨VF: Apply rule 6 171⟩;
  append_one(pop); decr(vf_ptr); vf_last[vf_ptr] ← vf_group;
  vf_last_loc[vf_ptr] ← vf_push_loc[vf_ptr + 1] − 1; vf_last_end[vf_ptr] ← byte_ptr;
  if vf_push_num[vf_ptr + 1] > 0 then ⟨VF: Apply rule 5 172⟩;
  end;
end
```

This code is used in section 161.

169. In order to implement transformation rule 4, we cancel the *set_char*, *set*, or *set_rule*, append a *pop*, and insert a *put* or *put_rule* with the old parameters.

⟨VF: Prepare for rule 4 169⟩ ≡

```
begin cur_class ← vf_last[vf_ptr]; cur_upd ← false; byte_ptr ← vf_push_loc[vf_ptr];
end
```

This code is used in section 168.

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[vf_ptr] ← vf_move[vf_ptr − 1];
  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.

171. ⟨VF: Apply rule 6 171⟩ ≡

```
begin Decr(byte_ptr)(2);
for k ← vf_last_loc[vf_ptr] + 1 to byte_ptr do byte_mem[k − 1] ← byte_mem[k];
vf_last[vf_ptr] ← vf_other; vf_last_end[vf_ptr] ← byte_ptr;
end
```

This code is used in section 168.

172. ⟨ VF: Apply rule 5 172 ⟩ ≡

```
begin incr(vf_ptr); ⟨ VF: Start a new level 163 ⟩;
decr(vf_push_num[vf_ptr]);
end
```

This code is used in section 168.

173. The `VF` format specifies that after a character packet invoked by a *set_char* or *set* command, "*h* is increased by the `TFM` width (properly scaled)—just as if a simple character had been typeset"; for *vf_simple* packets this is achieved by changing the final *put* command into *set_char* or *set*, but for *vf_complex* packets an explicit movement must be done. This poses a problem for programs, such as `DVIcopy`, which write a new `DVI` file with all references to characters from virtual fonts replaced by their character packets: The `DVItype` program specifies that the horizontal movements after a *set_char* or *set* command, after a *set_rule* command, and after one of the commands *right1* .. *x4*, are all treated differently when `DVI` units are converted to pixels.

Thus we introduce a slight extension of `DVItype`'s pixel rounding algorithm and hope that this extension will become part of the standard `DVItype` program in the near future: If a `DVI` file contains a *set_rule* command for a rule with the negative height *width_dimen*, then this rule shall be treated in exactly the same way as a fictitious character whose width is the width of that rule; as value of *width_dimen* we choose -2^{31}, the smallest signed 32-bit integer.

⟨ Globals in the outer block 17 ⟩ +≡

```
width_dimen: int_32;   { vertical dimension of special rules }
```

174. When initializing *width_dimen* we are careful to avoid arithmetic overflow.

⟨ Set initial values 18 ⟩ +≡

```
width_dimen ← −"40000000; Decr(width_dimen)("40000000);
```

175. Terminal communication. When `DVIcopy` begins, it engages the user in a brief dialog so that various options may be specified. This part of `DVIcopy` requires nonstandard Pascal constructions to handle the online interaction; so it may be preferable in some cases to omit the dialog and simply to stick to the default options. On other hand, the system-dependent routines that are needed are not complicated, so it will not be terribly difficult to introduce them; furthermore they are similar to those in `DVItype`.

It may be desirable to (optionally) specify all the options in the command line and skip the dialog with the user, provided the operating system permits this. Here we just define the system-independent part of the code required for this possibility. Since a complete option (a keyword possibly followed by one or several parameters) may have embedded blanks it might be necessary to replace these blanks by some other separator, e.g., by a '/'. Using, e.g., `UNIX` style options one might then say

```
DVIcopy -mag/2000 -sel/17.3/5 -sel/47 ...
```

to override the magnification factor that is stated in the `DVI` file, and to select five pages starting with the page numbered 17.3 as well as all remaining pages starting with the one numbered 47; alternatively one might simply say

```
DVIcopy - ...
```

to skip the dialog and use the default options.

The system-dependent initialization code should set the *n_opt* variable to the number of options found in the command line. If $n\_opt = 0$ the *input_ln* procedure defined below will prompt the user for options. If $n\_opt > 0$ the *k_opt* variable will be incremented and another piece of system-dependent code is invoked instead of the dialog; that code should place the value of command line option number *k_opt* as temporary string into the $byte - mem$ array. This process will be repeated until $k\_opt = n\_opt$, indicating that all command line options have been processed.

```
define opt_separator = "/"   { acts as blank when scanning (command line) options }
⟨ Set initial values 18 ⟩ +≡
  n_opt ← 0;   { change this to indicate the presence of command line options }
  k_opt ← 0;   { just in case }
```

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 ≡ break(output)   { 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 if n_opt = 0 then
    begin print(´Enter␣option:␣´); update_terminal; reset(input);
    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[input↑]); incr(k); get(input);
      end;
    end
  else if k_opt < n_opt then
      begin incr(k_opt);   { Copy command line option number k_opt into byte_mem array! }
      end;
  end;
```

See also sections 178, 179, and 189.

This code is used in section 180.

177. The global variable *scan_ptr* is used while scanning the temporary packet; it points to the next byte in *byte_mem* to be examined.

⟨ Globals in the outer block 17 ⟩ +≡

```
n_opt: int_16;   { number of options found in command line }
k_opt: int_16;   { number of command line options processed }
scan_ptr: byte_pointer;   { pointer to next byte to be examined }
sep_char: text_char;   { ´␣´ or xchr[opt_separator] }
```

178. The *scan_keyword* function is used to test for keywords in a character string stored as temporary packet in *byte_mem*; the result is *true* (and *scan_ptr* is updated) if the characters starting at position *scan_ptr* are an abbreviation of a given keyword followed by at least one blank.

```
⟨ Action procedures for dialog 176 ⟩ +≡
function scan_keyword(p : pckt_pointer; l : int_7): boolean;
  var i, j, k: byte_pointer;   { indices into byte_mem }
  begin i ← pckt_start[p]; j ← pckt_start[p + 1]; k ← scan_ptr;
  while (i < j) ∧ ((byte_mem[k] = byte_mem[i]) ∨ (byte_mem[k] = byte_mem[i] − "a" + "A")) do
    begin incr(i); incr(k);
    end;
  if scan_blank(k) ∧ (i − pckt_start[p] ≥ l) then
    begin scan_ptr ← k; scan_skip; scan_keyword ← true;
    end
  else scan_keyword ← false;
  end;
```

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

```
⟨ Action procedures for dialog 176 ⟩ +≡
function scan_int: int_32;
  var x: int_32;   { accumulates the value }
    negative: boolean;   { should the value be negated? }
  begin if byte_mem[scan_ptr] = "-" then
    begin negative ← true; incr(scan_ptr);
    end
  else negative ← false;
  x ← 0;
  while (byte_mem[scan_ptr] ≥ "0") ∧ (byte_mem[scan_ptr] ≤ "9") do
    begin x ← 10 ∗ x + byte_mem[scan_ptr] − "0"; incr(scan_ptr);
    end;
  scan_skip;
  if negative then scan_int ← −x else scan_int ← x;
  end;
```

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

```
  ⟨Action procedures for dialog 176⟩
procedure dialog;
  label exit;
  var p: pckt_pointer;   { packet being created }
  begin ⟨Initialize options 187⟩
  loop
    begin input_ln; p ← new_packet; scan_init;
    if scan_ptr = byte_ptr then
      begin flush_packet; return;
      end
    ⟨Cases for options 190⟩
  else begin if n_opt = 0 then sep_char ← '␣'
    else sep_char ← xchr[opt_separator];
    print_options;
    if n_opt > 0 then
      begin print('Bad␣command␣line␣option:␣'); print_packet(p); abort('---run␣terminated');
      end;
    end; flush_packet;
    end;
exit: end;
```

181. The *print_options* procedure might be used in a ‘Usage message’ displaying the command line syntax.

```
⟨Basic printing procedures 48⟩ +≡
procedure print_options;
  begin print_ln('Valid␣options␣are:'); ⟨Print valid options 188⟩
  end;
```

182. Subroutines for typesetting commands. This is the central part of the whole `DVIcopy` program: When a typesetting command from the `DVI` file or from a `VF` packet has been decoded, one of the typesetting routines defined below is invoked to execute the command; apart from the necessary book keeping, these routines invoke device dependent code defined later.

```
  ⟨ Declare typesetting procedures 250 ⟩
```

183. These typesetting routines communicate with the rest of the program through global variables.

```
⟨ Globals in the outer block 17 ⟩ +≡
type_setting: boolean;   { true while typesetting a page }
```

184. ⟨ Set initial values 18 ⟩ +≡

```
  type_setting ← false;
```

185. The user may select up to *max_select* ranges of consecutive pages to be processed. Each 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. The number of pages is recorded in *max_pages*; a non positive value indicates that there is no limit.

```
  define start_count ≡ select_count[cur_select]   { count values to select starting page }
  define start_there ≡ select_there[cur_select]   { is the start_count value relevant? }
  define start_vals ≡ select_vals[cur_select]   { the last count considered significant }
  define max_pages ≡ select_max[cur_select]   { at most this many bop .. eop pages will be printed }

⟨ Globals in the outer block 17 ⟩ +≡
select_count: array [0 .. max_select − 1, 0 .. 9] of int_32;
select_there: array [0 .. max_select − 1, 0 .. 9] of boolean;
select_vals: array [0 .. max_select − 1] of 0 .. 9;
select_max: array [0 .. max_select − 1] of int_32;
out_mag: int_32;   { output magnification }
count: array [0 .. 9] of int_32;   { the count values on the current page }
num_select: 0 .. max_select;   { number of page selection ranges specified }
cur_select: 0 .. max_select;   { current page selection range }
selected: boolean;   { has starting page been found? }
all_done: boolean;   { have all selected pages been processed? }
str_mag, str_select: pckt_pointer;
```

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

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

187. ⟨ Initialize options 187 ⟩ ≡

```
  out_mag ← 0; cur_select ← 0; max_pages ← 0; selected ← true;
```

This code is used in section 180.

188. ⟨ Print valid options 188 ⟩ ≡

```
print_ln(´␣␣mag´, sep_char, ´<new_mag>´); print_ln(´␣␣select´, sep_char, ´<start_count>´, sep_char,
    ´[<max_pages>]␣␣(up␣to␣´, max_select : 1, ´␣ranges)´);
```

This code is used in section 181.

189. ⟨ Action procedures for *dialog* 176 ⟩ +≡

```
procedure scan_count; { scan a start_count value }
  begin if byte_mem[scan_ptr] = bi("*") then
    begin start_there[start_vals] ← false; incr(scan_ptr); scan_skip;
    end
  else begin start_there[start_vals] ← true; start_count[start_vals] ← scan_int;
    if cur_select = 0 then selected ← false; { don't start at first page }
    end;
  end;
```

190. ⟨ Cases for options 190 ⟩ ≡

```
else if scan_keyword(str_mag, 3) then out_mag ← scan_int
  else if scan_keyword(str_select, 3) then
      if cur_select = max_select then print_ln(´Too␣many␣page␣selections´)
      else begin start_vals ← 0; scan_count;
        while (start_vals < 9) ∧ (byte_mem[scan_ptr] = bi(".")) do
          begin incr(start_vals); incr(scan_ptr); scan_count;
          end;
        max_pages ← scan_int; incr(cur_select);
        end
```

This code is used in section 180.

191. ⟨ Initialize predefined strings 45 ⟩ +≡

```
id3("m")("a")("g")(str_mag); id6("s")("e")("l")("e")("c")("t")(str_select);
```

192. A stack is used to keep track of the current horizontal and vertical position, h and v, and the four registers w, x, y, and z; the register pairs (w, x) and (y, z) are maintained as arrays.

⟨ Types in the outer block 7 ⟩ +≡

```
device ⟨ Declare device dependent types 198 ⟩ ecived
stack_pointer = 0 .. stack_size;
stack_index = 1 .. stack_size;
pair_32 = array [0 .. 1] of int_32; { a pair of int_32 variables }
stack_record = record h_field: int_32; { horizontal position h }
  v_field: int_32; { vertical position v }
  w_x_field: pair_32; { w and x register for horizontal movements }
  y_z_field: pair_32; { y and z register for vertical movements }
    device ⟨ Device dependent stack record fields 200 ⟩ ecived
  end;
```

193. The current values are kept in *cur_stack*; they are pushed onto and popped from *stack*. We use `WEB` macros to access the current values.

```
  define cur_h ≡ cur_stack.h_field   { the current h value }
  define cur_v ≡ cur_stack.v_field   { the current v value }
  define cur_w_x ≡ cur_stack.w_x_field   { the current w and x value }
  define cur_y_z ≡ cur_stack.y_z_field   { the current y and z value }
⟨ Globals in the outer block 17 ⟩ +≡
stack: array [stack_index] of stack_record;   { the pushed values }
cur_stack: stack_record;   { the current values }
zero_stack: stack_record;   { initial values }
stack_ptr: stack_pointer;   { last used position in stack }
```

194. ⟨ Set initial values 18 ⟩ +≡

```
  zero_stack.h_field ← 0; zero_stack.v_field ← 0;
  for i ← 0 to 1 do
    begin zero_stack.w_x_field[i] ← 0; zero_stack.y_z_field[i] ← 0;
    end;
  device ⟨ Initialize device dependent stack record fields 201 ⟩ ecived
```

195. When typesetting for a real device we must convert the current position from `DVI` units to pixels, i.e., *cur_h* and *cur_v* into *cur_hh* and *cur_vv*. This might be a good place to collect everything related to the conversion from `DVI` units to pixels and in particular all the pixel rounding algorithms.

```
  define font_space(#) ≡ fnt_space[#]   { boundary between "small" and "large" spaces }
⟨ Declare device dependent font data arrays 195 ⟩ ≡
fnt_space: array [font_number] of int_32;   { boundary between "small" and "large" spaces }
```

This code is used in section 81.

196. ⟨ Initialize device dependent font data 196 ⟩ ≡

```
  font_space(invalid_font) ← 0;
```

This code is used in section 82.

197. ⟨ Initialize device dependent data for a font 197 ⟩ ≡

```
  font_space(cur_fnt) ← font_scaled(cur_fnt) div 6;   { this is a 3-unit "thin space" }
```

This code is used in section 99.

198. The *char_pixels* array is used to store the horizontal character escapements: for `PK` or `GF` files we use the values given there, otherwise we must convert the character widths to (horizontal) pixels. The horizontal escapement of character c in font f is given by *font_pixel*$(f)(c)$.

```
  define font_pixel(#) ≡ char_pixels [ font_chars(#) + font_width_end
  define max_pix_value ≡ "7FFF
         { largest allowed pixel value; this range may not suffice for high resolution output devices }
⟨ Declare device dependent types 198 ⟩ ≡
  pix_value = −max_pix_value . . max_pix_value;   { a pixel coordinate or displacement }
```

This code is used in section 192.

199. ⟨ Globals in the outer block 17 ⟩ +≡

```
  device char_pixels: array [char_pointer] of pix_value;   { character escapements }
h_pixels: pix_value;   { a horizontal dimension in pixels }
v_pixels: pix_value;   { a vertical dimension in pixels }
temp_pix: pix_value;   { temporary value for pixel rounding }
  ecived
```

200.

```
define cur_hh ≡ cur_stack.hh_field   { the current hh value }
  define cur_vv ≡ cur_stack.vv_field   { the current vv value }
```

⟨ Device dependent stack record fields 200 ⟩ ≡

```
hh_field: pix_value;   { horizontal pixel position hh }
vv_field: pix_value;   { vertical pixel position vv }
```

This code is used in section 192.

201. ⟨ Initialize device dependent stack record fields 201 ⟩ ≡

```
  zero_stack.hh_field ← 0; zero_stack.vv_field ← 0;
```

This code is used in section 194.

202. For small movements we round the increment in position, for large movements we round the incremented position. The same applies to rule dimensions with the only difference that they will always be rounded towards larger values. For characters we increment the horizontal position by the escapement values obtained, e.g., from a PK file or by the TFM width converted to pixels.

```
  define h_pixel_round(#) ≡ round(h_conv * (#))
  define v_pixel_round(#) ≡ round(v_conv * (#))
  define large_h_space(#) ≡ (# ≥ font_space(cur_fnt)) ∨ (# ≤ −4 * font_space(cur_fnt))
           { is this a "large" horizontal distance? }
  define large_v_space(#) ≡ (abs(#) ≥ 5 * font_space(cur_fnt))   { is this a "large" vertical distance? }
  define h_rule_pixels ≡   { converts the rule width cur_h_dimen to pixels }
        device if large_h_space(cur_h_dimen) then
          begin h_pixels ← h_pixel_round(cur_h + cur_h_dimen) − cur_hh;
          if h_pixels ≤ 0 then
            if cur_h_dimen > 0 then h_pixels ← 1;
          end
        else begin h_pixels ← trunc(h_conv * cur_h_dimen);
          if h_pixels < h_conv * cur_h_dimen then incr(h_pixels);
          end;
        ecived
  define v_rule_pixels ≡   { converts the rule height cur_v_dimen to pixels }
        device if large_v_space(cur_v_dimen) then
          begin v_pixels ← cur_vv − v_pixel_round(cur_v − cur_v_dimen);
          if v_pixels ≤ 0 then v_pixels ← 1;   { used only for cur_v_dimen > 0 }
          end
        else begin v_pixels ← trunc(v_conv * cur_v_dimen);
          if v_pixels < v_conv * cur_v_dimen then incr(v_pixels);
          end;
        ecived
```

203. A sequence of consecutive rules, or consecutive characters in a fixed-width font whose width is not an integer number of pixels, can cause *hh* to drift far away from a correctly rounded value. `DVIcopy` ensures that the amount of drift will never exceed *max_h_drift* pixels; similarly *vv* shall never drift away from the correctly rounded value by more than *max_v_drift* pixels.

```
define h_upd_end(#) ≡ { check for proper horizontal pixel rounding }
      begin Incr(cur_hh)(#); temp_pix ← h_pixel_round(cur_h);
      if abs(temp_pix − cur_hh) > max_h_drift then
         if temp_pix > cur_hh then cur_hh ← temp_pix − max_h_drift
         else cur_hh ← temp_pix + max_h_drift;
      end ecived
define h_upd_char(#) ≡ Incr(cur_h)(#)
      device ; h_upd_end
define h_upd_move(#) ≡ Incr(cur_h)(#)
      device ;
      if large_h_space(#) then cur_hh ← h_pixel_round(cur_h)
      else h_upd_end
define v_upd_end(#) ≡ { check for proper vertical pixel rounding }
      begin Incr(cur_vv)(#); temp_pix ← v_pixel_round(cur_v);
      if abs(temp_pix − cur_vv) > max_v_drift then
         if temp_pix > cur_vv then cur_vv ← temp_pix − max_v_drift
         else cur_vv ← temp_pix + max_v_drift;
      end ecived
define v_upd_move(#) ≡ Incr(cur_v)(#)
      device ;
      if large_v_space(#) then cur_vv ← v_pixel_round(cur_v)
      else v_upd_end
```

204. The routines defined below use sections named 'Declare local variables (if any) for ...' or 'Declare additional local variables for ...'; the former may declare variables (including the keyword **var**), whereas the later must at least contain the keyword **var**. In general, both may start with the declaration of labels, constants, and/or types.

Let us start with the simple cases: The *do_pre* procedure is called when the preamble has been read from the `DVI` file; the preamble comment has just been converted into a temporary packet with the *new_packet* procedure.

```
procedure do_pre;
   ⟨ OUT: Declare local variables (if any) for do_pre 260 ⟩
 begin all_done ← false; num_select ← cur_select; cur_select ← 0;
 if num_select = 0 then max_pages ← 0;
 device h_conv ← (dvi_num/254000.0) ∗ (h_resolution/dvi_den) ∗ (out_mag/1000.0);
 v_conv ← (dvi_num/254000.0) ∗ (v_resolution/dvi_den) ∗ (out_mag/1000.0);
 ecived
 ⟨ OUT: Process the pre 261 ⟩
 end;
```

205. The *do_bop* procedure is called when a *bop* has been read. This routine determines whether a page shall be processed or skipped and sets the variable *type_setting* accordingly.

```
procedure do_bop;
    ⟨ OUT: Declare additional local variables do_bop 262 ⟩
  i, j: 0 . . 9;   { indices into count }
  begin ⟨ Determine whether this page should be processed or skipped 206 ⟩;
  print(´DVI:␣´);
  if type_setting then print(´process´) else print(´skipp´);
  print(´ing␣page␣´, count[0] : 1); j ← 9;
  while (j > 0) ∧ (count[j] = 0) do decr(j);
  for i ← 1 to j do print(´.´, count[i] : 1);
  d_print(´␣at␣´, dvi_loc − 45 : 1); print_ln(´.´);
  if type_setting then
    begin stack_ptr ← 0; cur_stack ← zero_stack; cur_fnt ← invalid_font;
    ⟨ OUT: Process a bop 263 ⟩
    end;
  end;
```

206. Note that the device dependent code 'OUT: Process a *bop*' may choose to set *type_setting* to false even if *selected* is true.

```
⟨ Determine whether this page should be processed or skipped 206 ⟩ ≡
  if ¬selected then selected ← start_match;
  type_setting ← selected
```

This code is used in section 205.

207. The *do_eop* procedure is called in order to process an *eop*; the stack should be empty.

```
procedure do_eop;
    ⟨ OUT: Declare local variables (if any) for do_eop 264 ⟩
  begin if stack_ptr ≠ 0 then bad_dvi;
  ⟨ OUT: Process an eop 265 ⟩
  if max_pages > 0 then
    begin decr(max_pages);
    if max_pages = 0 then
      begin selected ← false; incr(cur_select);
      if cur_select = num_select then all_done ← true;
      end;
    end;
  type_setting ← false;
  end;
```

208. The procedures *do_push* and *do_pop* are called in order to process *push* and *pop* commands; *do_push* must check for stack overflow, *do_pop* should never be called when the stack is empty.

```
procedure do_push; { push onto stack }
    ⟨ OUT: Declare local variables (if any) for do_push 266 ⟩
  begin incr_stack(stack_ptr); stack[stack_ptr] ← cur_stack;
  ⟨ OUT: Process a push 267 ⟩
  end;

procedure do_pop; { pop from stack }
    ⟨ OUT: Declare local variables (if any) for do_pop 268 ⟩
  begin if stack_ptr = 0 then bad_dvi;
  cur_stack ← stack[stack_ptr]; decr(stack_ptr); ⟨ OUT: Process a pop 269 ⟩
  end;
```

209. The *do_xxx* procedure is called in order to process a special command. The bytes of the special string have been put into *byte_mem* as the current string. They are converted to a temporary packet and discarded again.

```
procedure do_xxx;
    ⟨ OUT: Declare additional local variables for do_xxx 270 ⟩
  p: pckt_pointer; { temporary packet }
  begin p ← new_packet;
  ⟨ OUT: Process an xxx 271 ⟩
  flush_packet;
  end;
```

210. Next are the movement commands: The *do_right* procedure is called in order to process the horizontal movement commands *right*, *w*, and *x*.

```
procedure do_right;
    ⟨ OUT: Declare local variables (if any) for do_right 272 ⟩
  begin if cur_class ≥ w_cl then cur_w_x[cur_class − w_cl] ← cur_parm
  else if cur_class < right_cl then cur_parm ← cur_w_x[cur_class − w0_cl];
  ⟨ OUT: Process a right or w or x 273 ⟩
  h_upd_move(cur_parm)(h_pixel_round(cur_parm)); ⟨ OUT: Move right 274 ⟩
  end;
```

211. The *do_down* procedure is called in order to process the vertical movement commands *down*, *y*, and *z*.

```
procedure do_down;
    ⟨ OUT: Declare local variables (if any) for do_down 275 ⟩
  begin if cur_class ≥ y_cl then cur_y_z[cur_class − y_cl] ← cur_parm
  else if cur_class < down_cl then cur_parm ← cur_y_z[cur_class − y0_cl];
  ⟨ OUT: Process a down or y or z 276 ⟩
  v_upd_move(cur_parm)(v_pixel_round(cur_parm)); ⟨ OUT: Move down 277 ⟩
  end;
```

212. The *do_width* procedure, or actually the *do_a_width* macro, is called in order to increase the current horizontal position *cur_h* by *cur_h_dimen* in exactly the same way as if a character of width *cur_h_dimen* had been typeset.

```
  define do_a_width(#) ≡
          begin device h_pixels ← #; ecived do_width;
          end
procedure do_width;
    ⟨OUT: Declare local variables (if any) for do_width 278⟩
  begin ⟨OUT: Typeset a width 279⟩
  h_upd_char(cur_h_dimen)(h_pixels); ⟨OUT: Move right 274⟩
  end;
```

213. Finally we have the commands for the typesetting of rules and characters; the global variable *cur_upd* is *true* if the horizontal position shall be updated (*set* commands).

The *do_rule* procedure is called in order to typeset a rule.

```
procedure do_rule;
    ⟨OUT: Declare additional local variables do_rule 280⟩
  visible: boolean;
  begin h_rule_pixels
  if $(cur\_h\_dimen > 0) \land (cur\_v\_dimen > 0)$ then
    begin visible ← true; v_rule_pixels ⟨OUT: Typeset a visible rule 281⟩
    end
  else begin visible ← false; ⟨OUT: Typeset an invisible rule 282⟩
    end;
  if cur_upd then
    begin h_upd_move(cur_h_dimen)(h_pixels); ⟨OUT: Move right 274⟩
    end;
  end;
```

214. Last not least the *do_char* procedure is called in order to typeset character *cur_res* with extension *cur_ext* from the real font *cur_fnt*.

```
procedure do_char;
    ⟨OUT: Declare local variables (if any) for do_char 287⟩
  begin ⟨OUT: Typeset a char 288⟩
  if cur_upd then
    begin h_upd_char(widths[cur_wp])(char_pixels[cur_cp]); ⟨OUT: Move right 274⟩
    end;
  end;
```

215. If the program terminates abnormally, the following code may be invoked in the middle of a page.

```
⟨Finish output file(s) 215⟩ ≡
  begin if type_setting then ⟨OUT: Finish incomplete page 289⟩;
  ⟨OUT: Finish output file(s) 290⟩
  end
```

This code is used in section 240.

216. When the first character of font *cur_fnt* is about to be typeset, the *do_font* procedure is called in order to decide whether this is a virtual font or a real font.

One step in this decision is the attempt to find and read the `VF` file for this font; other attempts to locate a font file may be performed before and after that, depending on the nature of the output device and on the structure of the file system at a particular installation. For a real device we convert the character widths to (horizontal) pixels.

In any case *do_font* must change *font_type*(*cur_fnt*) to a value > *defined_font*; as a last resort one might use the `TFM` width data and draw boxes or leave blank spaces in the output.

```
procedure do_font;
  label done;
    ⟨OUT: Declare additional local variables for do_font 283⟩
  p: char_pointer;  {index into char_widths and char_pixels}
  begin debug if font_type(cur_fnt) = defined_font then confusion(str_fonts);
  gubed p ← 0;  {such that p is used}
  device for p ← font_chars(cur_fnt) + font_bc(cur_fnt) to font_chars(cur_fnt) + font_ec(cur_fnt) do
    char_pixels[p] ← h_pixel_round(widths[char_widths[p]]);
  ecived ⟨OUT: Look for a font file before trying to read the VF file; if found goto done 284⟩
  if do_vf then goto done;  {try to read the VF file}
  ⟨OUT: Look for a font file after trying to read the VF file 285⟩
done: debug if font_type(cur_fnt) ≤ loaded_font then confusion(str_fonts);
  gubed
  end;
```

217. Before a character of font *cur_fnt* is typeset the following piece of code ensures that the font is ready to be used.

```
⟨Prepare to use font cur_fnt 217⟩ ≡
  ⟨OUT: Prepare to use font cur_fnt 286⟩
  if font_type(cur_fnt) ≤ loaded_font then do_font  {cur_fnt was not yet used}
```

This code is used in sections 226 and 238.

218. Interpreting VF packets. The *pckt_first_par* procedure first reads a `DVI` command byte from the packet into *cur_cmd*; then *cur_parm* is set to the value of the first parameter (if any) and *cur_class* to the command class.

```
procedure pckt_first_par;
  begin cur_cmd ← pckt_ubyte;
  case dvi_par[cur_cmd] of
  char_par: set_cur_char(pckt_ubyte);
  no_par: do_nothing;
  dim1_par: cur_parm ← pckt_sbyte;
  num1_par: cur_parm ← pckt_ubyte;
  dim2_par: cur_parm ← pckt_spair;
  num2_par: cur_parm ← pckt_upair;
  dim3_par: cur_parm ← pckt_strio;
  num3_par: cur_parm ← pckt_utrio;
  three_cases(dim4_par): cur_parm ← pckt_squad;  { dim4, num4, or numu }
  rule_par: begin cur_v_dimen ← pckt_squad; cur_h_dimen ← pckt_squad;
    cur_upd ← (cur_cmd = set_rule);
    end;
  fnt_par: cur_parm ← cur_cmd − fnt_num_0;
  othercases abort('internal␣error');
  endcases; cur_class ← dvi_cl[cur_cmd];
  end;
```

219. The *do_vf_packet* procedure is called in order to interpret the character packet for a virtual character. Such a packet may contain the instruction to typeset a character from the same or an other virtual font; in such cases *do_vf_packet* calls itself recursively. The recursion level, i.e., the number of times this has happened, is kept in the global variable *n_recur* and should not exceed *max_recursion*.

```
⟨ Types in the outer block 7 ⟩ +≡
  recur_pointer = 0 .. max_recursion;
```

220. The `DVIcopy` processor should detect an infinite recursion caused by bad `VF` files; thus a new recursion level is entered even in cases where this could be avoided without difficulty.

If the recursion level exceeds the allowed maximum, we want to give a traceback how this has happened; thus some of the global variables used in different invocations of *do_vf_packet* are saved in a stack, others are saved as local variables of *do_vf_packet*.

```
⟨ Globals in the outer block 17 ⟩ +≡
recur_fnt: array [recur_pointer] of font_number;  { this packet's font }
recur_ext: array [recur_pointer] of int_24;  { this packet's extension }
recur_res: array [recur_pointer] of eight_bits;  { this packet's residue }
recur_pckt: array [recur_pointer] of pckt_pointer;  { the packet }
recur_loc: array [recur_pointer] of byte_pointer;  { next byte of packet }
n_recur: recur_pointer;  { current recursion level }
recur_used: recur_pointer;  { highest recursion level used so far }
```

221. ⟨ Set initial values 18 ⟩ +≡

```
  n_recur ← 0; recur_used ← 0;
```

222. Here now is the *do_vf_packet* procedure.

```
procedure do_vf_packet;
  label continue, found, done;
  var k: recur_pointer;  { loop index }
    f: int_8u;  { packet type flag }
    save_upd: boolean;  { used to save cur_upd }
    save_cp: width_pointer;  { used to save cur_cp }
    save_wp: width_pointer;  { used to save cur_wp }
    save_limit: byte_pointer;  { used to save cur_limit }
  begin ⟨ VF: Save values on entry to do_vf_packet 223 ⟩;
  ⟨ VF: Interpret the DVI commands in the packet 225 ⟩
  if save_upd then
    begin cur_h_dimen ← widths[save_wp]; do_a_width(char_pixels[save_cp]);
    end;
  ⟨ VF: Restore values on exit from do_vf_packet 224 ⟩;
  end;
```

223. On entry to *do_vf_packet* several values must be saved.

```
⟨ VF: Save values on entry to do_vf_packet 223 ⟩ ≡
  save_upd ← cur_upd; save_cp ← cur_cp; save_wp ← cur_wp;
  recur_fnt[n_recur] ← cur_fnt; recur_ext[n_recur] ← cur_ext; recur_res[n_recur] ← cur_res
```

This code is used in section 222.

224. Some of these values must be restored on exit from *do_vf_packet*.

```
⟨ VF: Restore values on exit from do_vf_packet 224 ⟩ ≡
  cur_fnt ← recur_fnt[n_recur]
```

This code is used in section 222.

225. If *cur_pckt* is the empty packet, we manufacture a *put* command; otherwise we read and interpret DVI commands from the packet.

```
⟨VF: Interpret the DVI commands in the packet 225⟩ ≡
  if find_packet then f ← cur_type else goto done;
  recur_pckt[n_recur] ← cur_pckt; save_limit ← cur_limit; cur_fnt ← font_font(cur_fnt);
  if cur_pckt = empty_packet then
    begin cur_class ← char_cl; goto found;
    end;
  if cur_loc ≥ cur_limit then goto done;
continue: pckt_first_par;
found: case cur_class of
  char_cl: ⟨VF: Typeset a char 226⟩;
  rule_cl: do_rule;
  xxx_cl: begin pckt_room(cur_parm);
    while cur_parm > 0 do
      begin append_byte(pckt_ubyte); decr(cur_parm);
      end;
    do_xxx;
    end;
  push_cl: do_push;
  pop_cl: do_pop;
  five_cases(w0_cl): do_right;   { right, w, or x }
  five_cases(y0_cl): do_down;   { down, y, or z }
  fnt_cl: cur_fnt ← cur_parm;
  othercases confusion(str_packets);   { font definition or invalid }
  endcases;
  if cur_loc < cur_limit then goto continue;
done:
```

This code is used in section 222.

226. The final *put* of a simple packet may be changed into *set_char* or *set*.

```
⟨VF: Typeset a char 226⟩ ≡
  begin ⟨Prepare to use font cur_fnt 217⟩;
  cur_cp ← font_chars(cur_fnt) + cur_res; cur_wp ← char_widths[cur_cp];
  if (cur_loc = cur_limit) ∧ (f = vf_simple) ∧ save_upd then
    begin save_upd ← false; cur_upd ← true;
    end;
  if font_type(cur_fnt) = vf_font_type then ⟨VF: Enter a new recursion level 227⟩
  else do_char;
  end
```

This code is used in section 225.

227. Before entering a new recursion level we must test for overflow; in addition a few variables must be saved and restored. A *set_char* or *set* followed by *pop* is changed into *put*.

```
⟨ VF: Enter a new recursion level 227 ⟩ ≡
  begin recur_loc[n_recur] ← cur_loc;   { save }
  if cur_loc < cur_limit then
    if byte_mem[cur_loc] = bi(pop) then cur_upd ← false;
  if n_recur = recur_used then
    if recur_used = max_recursion then ⟨ VF: Display the recursion traceback and terminate 228 ⟩
    else incr(recur_used);
  incr(n_recur); do_vf_packet; decr(n_recur);   { recurse }
  cur_loc ← recur_loc[n_recur]; cur_limit ← save_limit;   { restore }
  end
```

This code is used in section 226.

228. ⟨ VF: Display the recursion traceback and terminate 228 ⟩ ≡

```
  begin print_ln(´␣!Infinite␣VF␣recursion?´);
  for k ← max_recursion downto 0 do
    begin print(´level=´, k : 1, ´␣font´); d_print(´=´, recur_fnt[k] : 1); print_font(recur_fnt[k]);
    print(´␣char=´, recur_res[k] : 1);
    if recur_ext[k] ≠ 0 then print(´.´, recur_ext[k] : 1);
    new_line;
    debug hex_packet(recur_pckt[k]); print_ln(´loc=´, recur_loc[k] : 1);
    gubed
    end;
  overflow(str_recursion, max_recursion);
  end
```

This code is used in section 227.

229. Interpreting the DVI file. The *do_dvi* procedure reads the entire `DVI` file and initiates whatever actions may be necessary.

```
procedure do_dvi;
  label done, exit;
  var temp_byte: int_8u;   { byte for temporary variables }
    temp_int: int_32;   { integer for temporary variables }
    dvi_start: int_32;   { starting location }
    dvi_bop_post: int_32;   { location of bop or post }
    dvi_back: int_32;   { a back pointer }
    k: int_15;   { general purpose variable }
  begin ⟨DVI: Process the preamble 230⟩;
  if random_reading then ⟨DVI: Process the postamble 232⟩;
  repeat dvi_first_par;
    while cur_class = fnt_def_cl do
      begin dvi_do_font(random_reading); dvi_first_par;
      end;
    if cur_cmd = bop then ⟨DVI: Process one page 235⟩;
  until cur_cmd ≠ eop;
  if cur_cmd ≠ post then bad_dvi;
exit: end;
```

230. ⟨DVI: Process the preamble 230⟩ ≡

```
  if dvi_ubyte ≠ pre then bad_dvi;
  if dvi_ubyte ≠ dvi_id then bad_dvi;
  dvi_num ← dvi_pquad; dvi_den ← dvi_pquad; dvi_mag ← dvi_pquad;
  tfm_conv ← (25400000.0/dvi_num) ∗ (dvi_den/473628672)/16.0; temp_byte ← dvi_ubyte;
  pckt_room(temp_byte);
  for k ← 1 to temp_byte do append_byte(dvi_ubyte);
  print(´DVI␣file:␣´´´); print_packet(new_packet); print_ln(´´´,´);
  print(´␣␣␣num=´, dvi_num : 1, ´,␣den=´, dvi_den : 1, ´,␣mag=´, dvi_mag : 1);
  if out_mag ≤ 0 then out_mag ← dvi_mag else print(´␣=>␣´, out_mag : 1);
  print_ln(´.´); do_pre; flush_packet
```

This code is used in section 229.

231. ⟨Globals in the outer block 17⟩ +≡

```
dvi_num: int_31;   { numerator }
dvi_den: int_31;   { denominator }
dvi_mag: int_31;   { magnification }
```

232. ⟨ DVI: Process the postamble 232 ⟩ ≡

```
begin dvi_start ← dvi_loc;   { remember start of first page }
⟨ DVI: Find the postamble 233 ⟩;
d_print_ln(´DVI:␣postamble␣at␣´, dvi_bop_post : 1); dvi_back ← dvi_pointer;
if dvi_num ≠ dvi_pquad then bad_dvi;
if dvi_den ≠ dvi_pquad then bad_dvi;
if dvi_mag ≠ dvi_pquad then bad_dvi;
temp_int ← dvi_squad; temp_int ← dvi_squad;
if stack_size < dvi_upair then overflow(str_stack, stack_size);
temp_int ← dvi_upair; dvi_first_par;
while cur_class = fnt_def_cl do
  begin dvi_do_font(false); dvi_first_par;
  end;
if cur_cmd ≠ post_post then bad_dvi;
if ¬selected then ⟨ DVI: Find the starting page 234 ⟩;
dvi_move(dvi_start);   { go to first or starting page }
end
```

This code is used in section 229.

233. ⟨ DVI: Find the postamble 233 ⟩ ≡

```
temp_int ← dvi_length − 5;
repeat if temp_int < 49 then bad_dvi;
  dvi_move(temp_int); temp_byte ← dvi_ubyte; decr(temp_int);
until temp_byte ≠ dvi_pad;
if temp_byte ≠ dvi_id then bad_dvi;
dvi_move(temp_int − 4);
if dvi_ubyte ≠ post_post then bad_dvi;
dvi_bop_post ← dvi_pointer;
if (dvi_bop_post < 15) ∨ (dvi_bop_post > dvi_loc − 34) then bad_dvi;
dvi_move(dvi_bop_post);
if dvi_ubyte ≠ post then bad_dvi
```

This code is used in section 232.

234. ⟨ DVI: Find the starting page 234 ⟩ ≡

```
begin dvi_start ← dvi_bop_post;   { just in case }
while dvi_back ≠ −1 do
  begin if (dvi_back < 15) ∨ (dvi_back > dvi_bop_post − 46) then bad_dvi;
  dvi_bop_post ← dvi_back; dvi_move(dvi_back);
  if dvi_ubyte ≠ bop then bad_dvi;
  for k ← 0 to 9 do count[k] ← dvi_squad;
  if start_match then dvi_start ← dvi_bop_post;
  dvi_back ← dvi_pointer;
  end;
end
```

This code is used in section 232.

235. When a *bop* has been read, the `DVI` commands for one page are interpreted until an *eop* is found.

```
⟨DVI: Process one page 235⟩ ≡
  begin for $k \leftarrow 0$ to 9 do $count[k] \leftarrow dvi\_squad$;
  $temp\_int \leftarrow dvi\_pointer$; $do\_bop$; $dvi\_first\_par$;
  if $type\_setting$ then ⟨DVI: Process a page; then goto $done$ 236⟩
  else ⟨DVI: Skip a page; then goto $done$ 237⟩;
done: if $cur\_cmd \neq eop$ then $bad\_dvi$;
  if $selected$ then
    begin $do\_eop$;
    if $all\_done$ then return;
    end;
  end
```

This code is used in section 229.

236. All `DVI` commands are processed, as long as $cur\_class \neq invalid\_cl$; then we should have found an *eop*.

```
⟨DVI: Process a page; then goto $done$ 236⟩ ≡
  loop
    begin case $cur\_class$ of
    $char\_cl$: ⟨DVI: Typeset a $char$ 238⟩;
    $rule\_cl$: if $cur\_upd \land (cur\_v\_dimen = width\_dimen)$ then $do\_a\_width(h\_pixel\_round(cur\_h\_dimen))$
      else $do\_rule$;
    $xxx\_cl$: begin $pckt\_room(cur\_parm)$;
      while $cur\_parm > 0$ do
        begin $append\_byte(dvi\_ubyte)$; $decr(cur\_parm)$;
        end;
      $do\_xxx$;
      end;
    $push\_cl$: $do\_push$;
    $pop\_cl$: $do\_pop$;
    $five\_cases(w0\_cl)$: $do\_right$;   { $right$, $w$, or $x$ }
    $five\_cases(y0\_cl)$: $do\_down$;   { $down$, $y$, or $z$ }
    $fnt\_cl$: $dvi\_font$;
    $fnt\_def\_cl$: $dvi\_do\_font(random\_reading)$;
    $invalid\_cl$: goto $done$;
    othercases abort('internal␣error');
    endcases; $dvi\_first\_par$;   { get the next command }
    end
```

This code is used in section 235.

237. While skipping a page all commands other than font definitions are ignored.

```
⟨DVI: Skip a page; then goto done 237⟩ ≡
  loop
    begin case cur_class of
    xxx_cl: while cur_parm > 0 do
        begin temp_byte ← dvi_ubyte; decr(cur_parm);
        end;
    fnt_def_cl: dvi_do_font(random_reading);
    invalid_cl: goto done;
    othercases do_nothing;
    endcases; dvi_first_par;  { get the next command }
    end
```

This code is used in section 235.

238. ⟨DVI: Typeset a *char* 238⟩ ≡

```
  begin ⟨Prepare to use font cur_fnt 217⟩;
  set_cur_wp(cur_fnt)(bad_dvi);
  if font_type(cur_fnt) = vf_font_type then do_vf_packet else do_char;
  end
```

This code is used in section 236.

239. The main program. The code for real devices is still rather incomplete. Moreover several branches of the program have not been tested because they are never used with DVI files made by TEX and VF files made by VPtoVF.

240. At the end of the program the output file(s) have to be finished and on some systems it may be necessary to close input and/or output files.

```
procedure close_files_and_terminate;
  var k: int_15;  { general purpose index }
  begin close_in(dvi_file);
  if history < fatal_message then ⟨ Finish output file(s) 215 ⟩;
  stat ⟨ Print memory usage statistics 242 ⟩; tats
  ⟨ Close output file(s) 247 ⟩
  ⟨ Print the job history 243 ⟩;
  end;
```

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 ⟩
  dialog;  { get options }
  ⟨ Open input file(s) 110 ⟩
  ⟨ Open output file(s) 246 ⟩
  do_dvi;  { process the entire DVI file }
  close_files_and_terminate;
final_end: end.
```

242. ⟨ Print memory usage statistics 242 ⟩ ≡

```
  print_ln('Memory␣usage␣statistics:'); print(dvi_nf : 1, '␣dvi,␣', lcl_nf : 1, '␣local,␣');
  ⟨ Print more font usage statistics 257 ⟩
  print_ln('and␣', nf : 1, '␣internal␣fonts␣of␣', max_fonts : 1); print_ln(n_widths : 1, '␣widths␣of␣',
      max_widths : 1, '␣for␣', n_chars : 1, '␣characters␣of␣', max_chars : 1); print_ln(pckt_ptr : 1,
      '␣byte␣packets␣of␣', max_packets : 1, '␣with␣', byte_ptr : 1, '␣bytes␣of␣', max_bytes : 1);
  ⟨ Print more memory usage statistics 292 ⟩
  print_ln(stack_used : 1, '␣of␣', stack_size : 1, '␣stack␣and␣', recur_used : 1, '␣of␣', max_recursion : 1,
      '␣recursion␣levels.')
```

This code is used in section 240.

243. Some implementations may wish to pass the *history* value to the operating system so that it can be used to govern whether or not other programs are started. Here we simply report the history to the user.

```
⟨ Print the job history 243 ⟩ ≡
  case history of
  spotless: print_ln('(No␣errors␣were␣found.)');
  harmless_message: print_ln('(Did␣you␣see␣the␣warning␣message␣above?)');
  error_message: print_ln('(Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong.)');
  fatal_message: print_ln('(That␣was␣a␣fatal␣error,␣my␣friend.)');
  end  { there are no other cases }
```

This code is used in section 240.

244. Low-level output routines. The program uses the binary file variable *out_file* for its main output file; *out_loc* is the number of the byte about to be written next on *out_file*.

```
⟨ Globals in the outer block 17 ⟩ +≡
out_file: byte_file;   { the DVI file we are writing }
out_loc: int_32;   { where we are about to write, in out_file }
out_back: int_32;   { a back pointer }
out_max_v: int_31;   { maximum v value so far }
out_max_h: int_31;   { maximum h value so far }
out_stack: int_16u;   { maximum stack depth }
out_pages: int_16u;   { total number of pages }
```

245. ⟨ Set initial values 18 ⟩ +≡

```
  out_loc ← 0; out_back ← −1; out_max_v ← 0; out_max_h ← 0; out_stack ← 0; out_pages ← 0;
```

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

```
⟨ Open output file(s) 246 ⟩ ≡
  rewrite(out_file);   { prepares to write packed bytes to out_file }
```

This code is used in section 241.

247. For some operating systems it may be necessary to close *out_file*.

⟨ Close output file(s) 247 ⟩ ≡

This code is used in section 240.

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(#) ≡ write(out_file, #)   { write next DVI byte }
```

249. The WEB macro *out_one* is used to write one byte and to update *out_loc*.

```
  define out_one(#) ≡
            begin out_byte(#); incr(out_loc); end
```

250. First the *out_packet* procedure copies a packet to *out_file*.

```
⟨ Declare typesetting procedures 250 ⟩ ≡
procedure out_packet(p : pckt_pointer);
  var k: byte_pointer;   { index into byte_mem }
  begin Incr(out_loc)(pckt_length(p));
  for k ← pckt_start[p] to pckt_start[p + 1] − 1 do out_byte(bo(byte_mem[k]));
  end;
```

See also sections 251, 252, 253, 254, and 258.

This code is used in section 182.

251. Next are the procedures used to write integer numbers or even complete DVI commands to *out_file*; they all keep *out_loc* up to date.

The *out_four* procedure outputs four bytes in two's complement notation, without risking arithmetic overflow.

⟨ Declare typesetting procedures 250 ⟩ +≡
```
procedure out_four(x : int_32);   { output four bytes }
  begin_four ; comp_four(out_byte); Incr(out_loc)(4);
  end;
```

252. The *out_char* procedure outputs a *set_char* or *set* command or, if $upd = false$, a *put* command.

⟨ Declare typesetting procedures 250 ⟩ +≡
```
procedure out_char(upd : boolean; ext : int_32; res : eight_bits);   { output set or put }
  begin_char ; comp_char(out_one);
  end;
```

253. The *out_unsigned* procedure outputs a *fnt*, *xxx*, or *fnt_def* command with its first parameter (normally unsigned); a *fnt* command is converted into *fnt_num* whenever this is possible.

⟨ Declare typesetting procedures 250 ⟩ +≡
```
procedure out_unsigned(o : eight_bits; x : int_32);   { output fnt_num, fnt, xxx, or fnt_def }
  begin_unsigned ; comp_unsigned(out_one);
  end;
```

254. The *out_signed* procedure outputs a movement (*right*, *w*, *x*, *down*, *y*, or *z*) command with its (signed) parameter.

⟨ Declare typesetting procedures 250 ⟩ +≡
```
procedure out_signed(o : eight_bits; x : int_32);   { output right, w, x, down, y, or z }
  begin_signed ; comp_signed(out_one);
  end;
```

255. For an output font we set $font\_type(f) \leftarrow out\_font\_type$; in this case $font\_font(f)$ is the font number used for font f in *out_file*.

The global variable *out_nf* is the number of fonts already used in *out_file* and the array *out_fnts* contains their internal font numbers; the current font in *out_file* is called *out_fnt*.

⟨ Globals in the outer block 17 ⟩ +≡
```
out_fnts: array [font_number] of font_number;   { internal font numbers }
out_nf: font_number;   { number of fonts used in out_file }
out_fnt: font_number;   { internal font number of current output font }
```

256. ⟨ Set initial values 18 ⟩ +≡
```
  out_nf ← 0;
```

257. ⟨ Print more font usage statistics 257 ⟩ ≡
```
  print(out_nf : 1, ´␣out,␣´);
```
This code is used in section 242.

258. The *out_fnt_def* procedure outputs a complete font definition command.

```
⟨ Declare typesetting procedures 250 ⟩ +≡
procedure out_fnt_def(f : font_number);
  var p: pckt_pointer;   { the font name packet }
    k, l: byte_pointer;   { indices into byte_mem }
    a: eight_bits;   { length of area part }
  begin out_unsigned(fnt_def1, font_font(f)); out_four(font_check(f)); out_four(font_scaled(f));
  out_four(font_design(f));
  $p \gets$ font_name(f); $k \gets$ pckt_start[p]; $l \gets$ pckt_start[$p + 1$] $- 1$; $a \gets$ bo(byte_mem[k]);
  Incr(out_loc)($l - k + 2$); out_byte(a); out_byte($l - k - a$);
  while $k < l$ do
    begin incr(k); out_byte(bo(byte_mem[k]));
    end;
  end;
```

259. Writing the output file. Here we define the device dependent parts of the typesetting routines described earlier in this program.

First we define a few quantities required by the device dependent code for a real output device in order to demonstrate how they might be defined and in order to be able to compile `DVIcopy` with the device dependent code included.

```
define h_resolution ≡ 300   { horizontal resolution in pixels per inch (dpi) }
define v_resolution ≡ 300   { vertical resolution in pixels per inch (dpi) }
define max_h_drift ≡ 2   { we insist that $abs(hh - h\_pixel\_round(h)) \le max\_h\_drift$ }
define max_v_drift ≡ 2   { we insist that $abs(vv - v\_pixel\_round(v)) \le max\_v\_drift$ }
```

⟨ Globals in the outer block 17 ⟩ +≡

```
  device h_conv: real;   { converts DVI units to horizontal pixels }
v_conv: real;   { converts DVI units to vertical pixels }
  ecived
```

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: packed array [1 .. comm_length] of char;   { 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 ← 1 to comm_length do append_byte(xord[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.

262. These are the additional local variables (if any) needed for *do_bop*; the variables *i* and *j* are already declared.

⟨ OUT: Declare additional local variables *do_bop* 262 ⟩ ≡

```
var
```

This code is used in section 205.

263. And here is the device dependent code for *do_bop*.

```
⟨ OUT: Process a bop 263 ⟩ ≡
   out_one(bop); incr(out_pages);
   for i ← 0 to 9 do out_four(count[i]);
   out_four(out_back); out_back ← out_loc − 45; out_fnt ← invalid_font;
```

This code is used in section 205.

264. These are the local variables (if any) needed for *do_eop*.

⟨ OUT: Declare local variables (if any) for *do_eop* 264 ⟩ ≡

This code is used in section 207.

265. And here is the device dependent code for *do_eop*.

```
⟨ OUT: Process an eop 265 ⟩ ≡
   out_one(eop);
```

This code is used in section 207.

266. These are the local variables (if any) needed for *do_push*.

⟨ OUT: Declare local variables (if any) for *do_push* 266 ⟩ ≡

This code is used in section 208.

267. And here is the device dependent code for *do_push*.

```
⟨ OUT: Process a push 267 ⟩ ≡
   if stack_ptr > out_stack then out_stack ← stack_ptr;
   out_one(push);
```

This code is used in section 208.

268. These are the local variables (if any) needed for *do_pop*.

⟨ OUT: Declare local variables (if any) for *do_pop* 268 ⟩ ≡

This code is used in section 208.

269. And here is the device dependent code for *do_pop*.

```
⟨ OUT: Process a pop 269 ⟩ ≡
   out_one(pop);
```

This code is used in section 208.

270. These are the additional local variables (if any) needed for *do_xxx*; the variable p, the pointer to the packet containing the special string, is already declared.

```
⟨ OUT: Declare additional local variables for do_xxx 270 ⟩ ≡
var
```

This code is used in section 209.

271. And here is the device dependent code for *do_xxx*.

```
⟨ OUT: Process an xxx 271 ⟩ ≡
   out_unsigned(xxx1, pckt_length(p)); out_packet(p);
```

This code is used in section 209.

272. These are the local variables (if any) needed for *do_right*.

⟨ OUT: Declare local variables (if any) for *do_right* 272 ⟩ ≡

This code is used in section 210.

273. And here is the device dependent code for *do_right*.

```
⟨ OUT: Process a right or w or x 273 ⟩ ≡
  if cur_class < right_cl then out_one(cur_cmd)   { w0 or x0 }
  else out_signed(dvi_right_cmd[cur_class], cur_parm);   { right, w, or x }
```

This code is used in section 210.

274. Here we update the *out_max_h* value.

```
⟨ OUT: Move right 274 ⟩ ≡
  if abs(cur_h) > out_max_h then out_max_h ← abs(cur_h);
```

This code is used in sections 210, 212, 213, and 214.

275. These are the local variables (if any) needed for *do_down*.

```
⟨ OUT: Declare local variables (if any) for do_down 275 ⟩ ≡
```

This code is used in section 211.

276. And here is the device dependent code for *do_down*.

```
⟨ OUT: Process a down or y or z 276 ⟩ ≡
  if cur_class < down_cl then out_one(cur_cmd)   { y0 or z0 }
  else out_signed(dvi_down_cmd[cur_class], cur_parm);   { down, y, or z }
```

This code is used in section 211.

277. Here we update the *out_max_v* value.

```
⟨ OUT: Move down 277 ⟩ ≡
  if abs(cur_v) > out_max_v then out_max_v ← abs(cur_v);
```

This code is used in section 211.

278. These are the local variables (if any) needed for *do_width*.

```
⟨ OUT: Declare local variables (if any) for do_width 278 ⟩ ≡
```

This code is used in section 212.

279. And here is the device dependent code for *do_width*.

```
⟨ OUT: Typeset a width 279 ⟩ ≡
  out_one(set_rule); out_four(width_dimen); out_four(cur_h_dimen);
```

This code is used in section 212.

280. These are the additional local variables (if any) needed for *do_rule*; the variable *visible* is already declared.

```
⟨ OUT: Declare additional local variables do_rule 280 ⟩ ≡
var
```

This code is used in section 213.

281. And here is the device dependent code for *do_rule*.

```
⟨ OUT: Typeset a visible rule 281 ⟩ ≡
  out_one(dvi_rule_cmd[cur_upd]); out_four(cur_v_dimen); out_four(cur_h_dimen);
```

This code is used in sections 213 and 282.

282.
```
⟨ OUT: Typeset an invisible rule 282 ⟩ ≡
  ⟨ OUT: Typeset a visible rule 281 ⟩
```

This code is used in section 213.

283. These are the additional local variables (if any) needed for *do_font*; the variable p is already declared.

⟨ OUT: Declare additional local variables for *do_font* 283 ⟩ ≡
var

This code is used in section 216.

284. And here is the device dependent code for *do_font*; if the `VF` file for a font could not be found, we simply assume this must be a real font.

⟨ OUT: Look for a font file before trying to read the `VF` file; if found **goto** *done* 284 ⟩ ≡

This code is used in section 216.

285. ⟨ OUT: Look for a font file after trying to read the `VF` file 285 ⟩ ≡

```
if (out_nf ≥ max_fonts) then overflow(str_fonts, max_fonts);
print(´OUT:␣font␣´, cur_fnt : 1); d_print(´␣=>␣´, out_nf : 1); print_font(cur_fnt);
d_print(´␣at␣´, font_scaled(cur_fnt) : 1, ´␣DVI␣units´); print_ln(´.´);
font_type(cur_fnt) ← out_font_type; font_font(cur_fnt) ← out_nf; out_fnts[out_nf] ← cur_fnt;
incr(out_nf); out_fnt_def(cur_fnt);
```

This code is used in section 216.

286. And here is some device dependent code used before each character.

⟨ OUT: Prepare to use font *cur_fnt* 286 ⟩ ≡

This code is used in section 217.

287. These are the local variables (if any) needed for *do_char*.

⟨ OUT: Declare local variables (if any) for *do_char* 287 ⟩ ≡

This code is used in section 214.

288. And here is the device dependent code for *do_char*.

⟨ OUT: Typeset a *char* 288 ⟩ ≡

```
debug if font_type(cur_fnt) ≠ out_font_type then confusion(str_fonts);
gubed
if cur_fnt ≠ out_fnt then
   begin out_unsigned(fnt1, font_font(cur_fnt)); out_fnt ← cur_fnt;
   end;
out_char(cur_upd, cur_ext, cur_res);
```

This code is used in section 214.

289. If the program terminates in the middle of a page, we write as many *pop*s as necessary and one *eop*.

⟨ OUT: Finish incomplete page 289 ⟩ ≡

```
begin while stack_ptr > 0 do
   begin out_one(pop); decr(stack_ptr);
   end;
out_one(eop);
end
```

This code is used in section 215.

290. If the output file has been started, we write the postamble; in addition we print the number of bytes and pages written to *out_file*.

```
⟨OUT: Finish output file(s) 290⟩ ≡
  if out_loc > 0 then
    begin ⟨OUT: Write the postamble 291⟩;
    k ← 7 − ((out_loc − 1) mod 4);   { the number of dvi_pad bytes }
    while k > 0 do
      begin out_one(dvi_pad); decr(k);
      end;
    print(´OUT␣file:␣´, out_loc : 1, ´␣bytes,␣´, out_pages : 1, ´␣page´);
    if out_pages ≠ 1 then print(´s´);
    end
  else print(´OUT␣file:␣no␣output´);
  print_ln(´␣written.´);
  if out_pages = 0 then mark_harmless;
```

This code is used in section 215.

291. Here we simply write the values accumulated during the DVI output.

```
⟨OUT: Write the postamble 291⟩ ≡
  out_one(post); out_four(out_back); out_back ← out_loc − 5;
  out_four(dvi_num); out_four(dvi_den); out_four(out_mag);
  out_four(out_max_v); out_four(out_max_h);
  out_one(out_stack div "100); out_one(out_stack mod "100);
  out_one(out_pages div "100); out_one(out_pages mod "100);
  k ← out_nf;
  while k > 0 do
    begin decr(k); out_fnt_def(out_fnts[k]);
    end;
  out_one(post_post); out_four(out_back);
  out_one(dvi_id)
```

This code is used in section 290.

292. Here we could print more memory usage statistics; this possibility is, however, not used for DVIcopy.

⟨Print more memory usage statistics 292⟩ ≡

This code is used in section 242.

293. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make `DVIcopy` work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

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

⟨ 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.