

2* This program is written entirely in standard Pascal, except that it occasionally has lower case letters in strings that are output. Such letters can be converted to upper case if necessary. The input is read from *vf_file* and *tfm_file*; the output is written on *vpl_file*. Error messages and other remarks are written on the *output* file, which the user may choose to assign to the terminal if the system permits it.

The term *print* is used instead of *write* when this program writes on the *output* file, so that all such output can be easily deflected.

```

define print(#)  $\equiv$  write(stderr, #)
define print_ln(#)  $\equiv$  write_ln(stderr, #)
define print_real(#)  $\equiv$  fprint_real(stderr, #)
program VFtoVP(vf_file, tfm_file, vpl_file, output);
label <Labels in the outer block 3>
const <Constants in the outer block 4*>
type <Types in the outer block 5>
var <Globals in the outer block 7>
    <Define parse_arguments 136*>
procedure initialize; { this procedure gets things started properly }
    var k: integer; { all-purpose index for initialization }
    begin kpse.set_progname(argv[0]); kpse.init_prog(^VFtoVP^, 0, nil, nil); parse_arguments;
    <Set initial values 11*>
    end;

```

4* The following parameters can be changed at compile time to extend or reduce *VFtoVP*'s capacity.

```

define class  $\equiv$  class_var
<Constants in the outer block 4*>  $\equiv$ 
    tfm_size = 150000; { maximum length of tfm data, in bytes }
    vf_size = 100000; { maximum length of vf data, in bytes }
    max_fonts = 300; { maximum number of local fonts in the vf file }
    lig_size = 32510; { maximum length of lig_kern program, in words }
    hash_size = 32579;
    { preferably a prime number, a bit larger than the number of character pairs in lig/kern steps }
    max_stack = 100; { maximum depth of DVI stack in character packets }

```

See also section 144*.

This code is used in section 2*.

11*: We don't have to do anything special to read a packed file of bytes, but we do want to use environment variables to find the input files.

```

⟨Set initial values 11*⟩ ≡
  { See comments at kpse_find_vf in kpathsea/tex-file.h for why we don't use it. }
  vf_file ← kpse_open_file(vf_name, kpse_vf_format); tfm_file ← kpse_open_file(tfm_name, kpse_tfm_format);
  if verbose then
    begin print(banner); print_ln(version_string);
    end;

```

See also sections 21*, 51*, 56, 69, and 87.

This code is used in section 2*.

21*: If an explicit filename isn't given, we write to *stdout*.

```

⟨Set initial values 11*⟩ +≡
  if optind + 3 > argc then
    begin vpl_file ← stdout;
    end
  else begin vpl_name ← extend_filename(cmdline(optind + 2), 'vpl'); rewrite(vpl_file, vpl_name);
  end;

```

22* **Unpacking the TFM file.** The first thing VFtoVP does is read the entire *tfm_file* into an array of bytes, *tfm*[0 .. (4 * *lf* - 1)].

```

define index  $\equiv$  index.type
⟨Types in the outer block 5⟩  $\equiv$ 
  index = 0 .. tfm.size; { address of a byte in tfm }

```

24* The input may, of course, be all screwed up and not a TFM file at all. So we begin cautiously.

```

define abort(#)  $\equiv$ 
  begin print_ln(#);
  print_ln('Sorry, but I can't go on; are you sure this is a TFM?'); uexit(1);
  end

⟨Read the whole TFM file 24*⟩  $\equiv$ 
  read(tfm_file, tfm[0]);
  if tfm[0] > 127 then abort('The first byte of the input file exceeds 127!');
  if eof(tfm_file) then abort('The input file is only one byte long!');
  read(tfm_file, tfm[1]); lf  $\leftarrow$  tfm[0] * 400 + tfm[1];
  if lf = 0 then abort('The file claims to have length zero, but that's impossible!');
  if 4 * lf - 1 > tfm.size then abort('The file is bigger than I can handle!');
  for tfm_ptr  $\leftarrow$  2 to 4 * lf - 1 do
    begin if eof(tfm_file) then abort('The file has fewer bytes than it claims!');
    read(tfm_file, tfm[tfm_ptr]);
    end;
  if  $\neg$ eof(tfm_file) then
    begin print_ln('There's some extra junk at the end of the TFM file,');
    print_ln('but I'll proceed as if it weren't there. ');
    end

```

This code is used in section 132*.

25* After the file has been read successfully, we look at the subfile sizes to see if they check out.

```

define eval_two_bytes(#)  $\equiv$ 
  begin if tfm[tfm_ptr] > 127 then abort('One of the subfile sizes is negative!');
  #  $\leftarrow$  tfm[tfm_ptr] * 400 + tfm[tfm_ptr + 1]; tfm_ptr  $\leftarrow$  tfm_ptr + 2;
  end

⟨Set subfile sizes lh, bc, ..., np 25*⟩  $\equiv$ 
  begin tfm_ptr  $\leftarrow$  2;
  eval_two_bytes(lh); eval_two_bytes(bc); eval_two_bytes(ec); eval_two_bytes(nw); eval_two_bytes(nh);
  eval_two_bytes(nd); eval_two_bytes(ni); eval_two_bytes(nl); eval_two_bytes(nk); eval_two_bytes(ne);
  eval_two_bytes(np);
  if lh < 2 then abort('The header length is only', lh : 1, '!');
  if nl > lig_size then abort('The lig/kern program is longer than I can handle!');
  if (bc > ec + 1)  $\vee$  (ec > 255) then
    abort('The character code range', bc : 1, '...', ec : 1, 'is illegal!');
  if (nw = 0)  $\vee$  (nh = 0)  $\vee$  (nd = 0)  $\vee$  (ni = 0) then
    abort('Incomplete subfiles for character dimensions!');
  if ne > 256 then abort('There are', ne : 1, 'extensible recipes!');
  if lf  $\neq$  6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np then
    abort('Subfile sizes don't add up to the stated total!');
  end

```

This code is used in section 132*.

31* Again we cautiously verify that we've been given decent data.

```

define read_vf(#)  $\equiv$  read(vf_file, #)
define vf_abort(#)  $\equiv$ 
    begin print_ln(#); print_ln(`Sorry, but I can't go on; are you sure this is a VF?`);
    uexit(1);
    end

⟨ Read the whole VF file 31* ⟩  $\equiv$ 
    read_vf(temp_byte);
    if temp_byte  $\neq$  pre then vf_abort(`The first byte isn't pre`!`);
    ⟨ Read the preamble command 32* ⟩;
    ⟨ Read and store the font definitions and character packets 33 ⟩;
    ⟨ Read and verify the postamble 34 ⟩

```

This code is used in section 132*.

```

32* define vf_store(#)  $\equiv$ 
    if vf_ptr + #  $\geq$  vf_size then vf_abort(`The file is bigger than I can handle!`);
    for k  $\leftarrow$  vf_ptr to vf_ptr + # - 1 do
        begin if eof(vf_file) then vf_abort(`The file ended prematurely!`);
        read_vf(vf[k]);
        end;
    vf_count  $\leftarrow$  vf_count + #; vf_ptr  $\leftarrow$  vf_ptr + #

⟨ Read the preamble command 32* ⟩  $\equiv$ 
    if eof(vf_file) then vf_abort(`The input file is only one byte long!`);
    read_vf(temp_byte);
    if temp_byte  $\neq$  id_byte then vf_abort(`Wrong VF version number in second byte!`);
    if eof(vf_file) then vf_abort(`The input file is only two bytes long!`);
    read_vf(temp_byte); { read the length of introductory comment }
    vf_count  $\leftarrow$  11; vf_ptr  $\leftarrow$  0; vf_store(temp_byte);
    if verbose then
        begin for k  $\leftarrow$  0 to vf_ptr - 1 do print(xchr[vf[k]]);
        print_ln(` `);
        end;
    count  $\leftarrow$  0;
    for k  $\leftarrow$  0 to 7 do
        begin if eof(vf_file) then vf_abort(`The file ended prematurely!`);
        read_vf(temp_byte);
        if temp_byte = tfm[check_sum + k] then incr(count);
        end;
    real_dsize  $\leftarrow$  (((tfm[design_size] * 256 + tfm[design_size + 1]) * 256 + tfm[design_size + 2]) * 256 +
        tfm[design_size + 3]) / 4000000;
    if count  $\neq$  8 then
        begin print_ln(`Check sum and/or design size mismatch.`);
        print_ln(`Data from TFM file will be assumed correct.`);
        end

```

This code is used in section 31*.

```

35*  ⟨ Read and store a font definition 35* ⟩ ≡
  begin if packet_found ∨ (temp_byte ≥ pre) then
    vf_abort('Illegal_byte', temp_byte : 1, 'at_beginning_of_character_packet!');
    font_number[font_ptr] ← vf_read(temp_byte - fnt_def1 + 1);
    if font_ptr = max_fonts then vf_abort('I_can_t_handle_that_many_fonts!');
    vf_store(14); { c[4] s[4] d[4] a[1] l[1] }
    if vf[vf_ptr - 10] > 0 then { s is negative or exceeds 224 - 1 }
      vf_abort('Mapped_font_size_is_too_big!');
    a ← vf[vf_ptr - 2]; l ← vf[vf_ptr - 1]; vf_store(a + l); { n[a + l] }
    if verbose then
      begin ⟨ Print the name of the local font 36* ⟩;
      end;
    ⟨ Read the local font's TFM file and record the characters it contains 39* ⟩;
    incr(font_ptr); font_start[font_ptr] ← vf_ptr;
  end

```

This code is used in section 33.

36* The font area may need to be separated from the font name on some systems. Here we simply reproduce the font area and font name (with no space or punctuation between them).

```

⟨ Print the name of the local font 36* ⟩ ≡
  print('MAPFONT', font_ptr : 1, ':');
  for k ← font_start[font_ptr] + 14 to vf_ptr - 1 do print(xchr[vf[k]]);
  k ← font_start[font_ptr] + 5; print('at');
  print_real((((vf[k] * 256 + vf[k + 1]) * 256 + vf[k + 2]) / '4000000') * real_dsize, 2, 2); print_ln('pt')

```

This code is used in section 35*.

37* Now we must read in another TFM file. But this time we needn't be so careful, because we merely want to discover which characters are present. The next few sections of the program are copied pretty much verbatim from *DVItype*, so that system-dependent modifications can be copied from existing software.

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

```

⟨ Globals in the outer block 7 ⟩ +≡
a: integer; { length of the area/directory spec }
l: integer; { length of the font name proper }
cur_name: ↑char; { external tfm name }
b0, b1, b2, b3: byte; { four bytes input at once }
font_lh: 0 .. '777777; { header length of current local font }
font_bc, font_ec: 0 .. '777777; { character range of current local font }

```

39* We use the *vf* array to store a list of all valid characters in the local font, beginning at location *font_chars[f]*.

```

⟨ Read the local font's TFM file and record the characters it contains 39* ⟩ ≡
  font_chars[font_ptr] ← vf_ptr; ⟨ Move font name into the cur_name string 44* ⟩;
  tfm_name ← kpse_find_tfm(cur_name);
  if ¬tfm_name then
    print_ln('---not_loaded, TFM_file', stringcast(cur_name), 'can't be opened!')
  else begin resetbin(tfm_file, tfm_name); font_bc ← 0; font_ec ← 256;
    { will cause error if not modified soon }
    read_tfm_word;
    if b2 < 128 then
      begin font_lh ← b2 * 256 + b3; read_tfm_word;
      if (b0 < 128) ∧ (b2 < 128) then
        begin font_bc ← b0 * 256 + b1; font_ec ← b2 * 256 + b3;
        end;
      end;
    if font_bc ≤ font_ec then
      if font_ec > 255 then print_ln('---not_loaded, bad_TFM_file', stringcast(tfm_name), '!');
      else begin for k ← 0 to 3 + font_lh do
        begin read_tfm_word;
        if k = 4 then ⟨ Check the check sum 40* ⟩;
        if k = 5 then ⟨ Check the design size 41 ⟩;
        end;
        for k ← font_bc to font_ec do
          begin read_tfm_word;
          if b0 > 0 then { character k exists in the font }
            begin vf[vf_ptr] ← k; incr(vf_ptr);
            if vf_ptr = vf_size then vf_abort('I'm out of VF memory!');
            end;
          end;
        end;
      if eof(tfm_file) then
        print_ln('---trouble_is_brewing, TFM_file', stringcast(tfm_name), 'ended too soon!');
        free(tfm_name);
      end;
      free(cur_name); incr(vf_ptr) { leave space for character search later }
    end;
  end;

```

This code is used in section 35*.

40* ⟨ Check the check sum 40* ⟩ ≡

```

  if b0 + b1 + b2 + b3 > 0 then
    if (b0 ≠ vf[font_start[font_ptr]]) ∨ (b1 ≠ vf[font_start[font_ptr] + 1]) ∨
      (b2 ≠ vf[font_start[font_ptr] + 2]) ∨ (b3 ≠ vf[font_start[font_ptr] + 3]) then
      begin if verbose then print_ln('Check_sum_in_VF_file_being_replaced_by_TFM_check_sum');
        vf[font_start[font_ptr]] ← b0; vf[font_start[font_ptr] + 1] ← b1; vf[font_start[font_ptr] + 2] ← b2;
        vf[font_start[font_ptr] + 3] ← b3;
      end
    end

```

This code is used in section 39*.

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

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

⟨ Move font name into the *cur_name* string 44* ⟩ ≡

See also section 45*.

This code is used in section 39*.

45* The string *cur_name* is supposed to be set to the external name of the TFM file for the current font. We do not impose an arbitrary limit on the filename length.

define *name_start* ≡ (*font_start*[*font_ptr*] + 14)

define *name_end* ≡ *vf_ptr*

⟨ Move font name into the *cur_name* string 44* ⟩ +≡

r ← *name_end* − *name_start*; *cur_name* ← *xmalloc_array*(*char*, *r*);

{ *strncpy* might be faster, but it’s probably a good idea to keep the *xchr* translation. }

for *k* ← *name_start* **to** *name_end* **do**

begin *cur_name*[*k* − *name_start*] ← *xchr*[*vf*[*k*]];

end;

cur_name[*r*] ← 0; { Append null byte since this is C. }

50* In order to stick to standard Pascal, we use an *xchr* array to do appropriate conversion of ASCII codes. Three other little strings are used to produce *face* codes like MIE.

⟨Globals in the outer block 7⟩ +≡

ASCII_04, *ASCII_10*, *ASCII_14*: *const_c_string*; { strings for output in the user's external character set }

xchr: **packed array** [0 .. 255] **of** *char*;

MBL_string, *RI_string*, *RCE_string*: *const_c_string*; { handy string constants for *face* codes }

51* ⟨Set initial values 11*⟩ +≡

ASCII_04 ← '␣␣! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? ' ;

ASCII_10 ← '␣@ABCDEFGHIJKLMN O P Q R S T U V W X Y Z [\] ^ _ ' ;

ASCII_14 ← '␣`abcdefghijklmnopqrstuvwxyz{|}~?' ;

for *k* ← 0 **to** 255 **do** *xchr*[*k*] ← '?' ;

for *k* ← 0 **to** '37' **do**

begin *xchr*[*k* + '40'] ← *ASCII_04*[*k* + 1]; *xchr*[*k* + '100'] ← *ASCII_10*[*k* + 1];

xchr[*k* + '140'] ← *ASCII_14*[*k* + 1];

end;

MBL_string ← '␣MBL'; *RI_string* ← '␣RI␣'; *RCE_string* ← '␣RCE';

61* The property value may be a character, which is output in octal unless it is a letter or a digit.

procedure *out_char*(*c* : *byte*); { outputs a character }

begin if (*font_type* > *vanilla*) ∨ (*charcode_format* = *charcode_octal*) **then**

begin *tfm*[0] ← *c*; *out_octal*(0, 1)

end

else if (*charcode_format* = *charcode_ascii*) ∧ (*c* > "␣") ∧ (*c* ≤ "~") ∧ (*c* ≠ "(") ∧ (*c* ≠ ")") **then**

out('␣C␣', *xchr*[*c*]) { default case, use C only for letters and digits }

else if ((*c* ≥ "0") ∧ (*c* ≤ "9")) ∨ ((*c* ≥ "A") ∧ (*c* ≤ "Z")) ∨ ((*c* ≥ "a") ∧ (*c* ≤ "z")) **then**

out('␣C␣', *xchr*[*c*])

else begin *tfm*[0] ← *c*; *out_octal*(0, 1);

end;

end;

62* The property value might be a "face" byte, which is output in the curious code mentioned earlier, provided that it is less than 18.

procedure *out_face*(*k* : *index*); { outputs a *face* }

var *s*: 0 .. 1; { the slope }

b: 0 .. 8; { the weight and expansion }

begin if *tfm*[*k*] ≥ 18 **then** *out_octal*(*k*, 1)

else begin *out*('␣F␣', { specify face-code format }

s ← *tfm*[*k*] **mod** 2; *b* ← *tfm*[*k*] **div** 2; *put_byte*(*MBL_string*[1 + (*b mod* 3)], *vpl_file*);

put_byte(*RI_string*[1 + *s*], *vpl_file*); *put_byte*(*RCE_string*[1 + (*b div* 3)], *vpl_file*);

end;

end;

63* And finally, the value might be a *fix_word*, which is output in decimal notation with just enough decimal places for **VPtoVF** to recover every bit of the given *fix_word*.

All of the numbers involved in the intermediate calculations of this procedure will be nonnegative and less than $10 \cdot 2^{24}$.

```

procedure out_fix(k : index); { outputs a fix_word }
  var a: 0 .. '7777; { accumulator for the integer part }
      f: integer; { accumulator for the fraction part }
      j: 0 .. 12; { index into dig }
      delta: integer; { amount if allowable inaccuracy }
  begin out('R'); { specify real format }
  a ← (tfm[k] * 16) + (tfm[k + 1] div 16);
  f ← ((tfm[k + 1] mod 16) * intcast('400) + tfm[k + 2]) * '400 + tfm[k + 3];
  if a > '3777 then ⟨ Reduce negative to positive 66⟩;
  ⟨ Output the integer part, a, in decimal notation 64⟩;
  ⟨ Output the fraction part, f/220, in decimal notation 65⟩;
end;

```

101*: The last thing on **VFtoVP**'s agenda is to go through the list of *char_info* and spew out the information about each individual character.

```

⟨Do the characters 101*⟩ ≡
  sort_ptr ← 0; { this will suppress 'STOP' lines in ligature comments }
  for c ← bc to ec do
    if width_index(c) > 0 then
      begin if chars_on_line = 8 then
        begin print_ln('␣'); chars_on_line ← 1;
        end
      else begin if chars_on_line > 0 then print('␣');
        if verbose then incr(chars_on_line); { keep chars_on_line = 0 }
        end;
      if verbose then print_octal(c); { progress report }
      left; out('CHARACTER'); out_char(c); out_ln; ⟨Output the character's width 102⟩;
      if height_index(c) > 0 then ⟨Output the character's height 103⟩;
      if depth_index(c) > 0 then ⟨Output the character's depth 104⟩;
      if italic_index(c) > 0 then ⟨Output the italic correction 105⟩;
      case tag(c) of
        no_tag: do_nothing;
        lig_tag: ⟨Output the applicable part of the ligature/kern program as a comment 106⟩;
        list_tag: ⟨Output the character link unless there is a problem 107⟩;
        ext_tag: ⟨Output an extensible character recipe 108⟩;
      end;
      if ¬do_map(c) then goto final_end;
      right;
    end
  end

```

This code is used in section 134*.

```

113*  ⟨ Check for ligature cycles 113* ⟩ ≡
    hash_ptr ← 0; y_lig_cycle ← 256;
    for hh ← 0 to hash_size do hash[hh] ← 0; { clear the hash table }
    for c ← bc to ec do
        if tag(c) = lig_tag then
            begin i ← remainder(c);
                if tfm[lig_step(i)] > stop_flag then i ← 256 * tfm[lig_step(i) + 2] + tfm[lig_step(i) + 3];
                ⟨ Enter data for character c starting at location i in the hash table 114 ⟩;
            end;
        if bchar_label < nl then
            begin c ← 256; i ← bchar_label;
                ⟨ Enter data for character c starting at location i in the hash table 114 ⟩;
            end;
        if hash_ptr = hash_size then
            begin print_ln(‘Sorry, I haven’t room for so many ligature/kern pairs!’); uexit(1);
            end;
        for hh ← 1 to hash_ptr do
            begin r ← hash_list[hh];
                if class[r] > simple then { make sure f is defined }
                    r ← lig_f(r, (hash[r] - 1) div 256, (hash[r] - 1) mod 256);
                end;
            if y_lig_cycle < 256 then
                begin print(‘Infinite ligature loop starting with’);
                    if x_lig_cycle = 256 then print(‘boundary’) else print_octal(x_lig_cycle);
                    print(‘ and ’); print_octal(y_lig_cycle); print_ln(‘!’);
                    out(‘(INFINITE LIGATURE LOOP MUST BE BROKEN!)’); uexit(1);
                end
            end
    end

```

This code is used in section 89.

117* Evaluation of $f(x, y)$ is handled by two mutually recursive procedures. Kind of a neat algorithm, generalizing a depth-first search.

```

    ifdef(‘notdef’)
        function lig_f(h, x, y : index): index;
            begin end;
            { compute f for arguments known to be in hash[h] }
    endif(‘notdef’)
    function eval(x, y : index): index; { compute f(x, y) with hashtable lookup }
        var key: integer; { value sought in hash table }
        begin key ← 256 * x + y + 1; h ← (1009 * key) mod hash_size;
            while hash[h] > key do
                if h > 0 then decr(h) else h ← hash_size;
            if hash[h] < key then eval ← y { not in ordered hash table }
            else eval ← lig_f(h, x, y);
        end;
    end;

```

118* Pascal's beastly convention for *forward* declarations prevents us from saying **function** $f(h, x, y : index) : index$ here.

```

function lig-f(h, x, y : index): index;
  begin case class[h] of
    simple: do_nothing;
    left_z: begin class[h]  $\leftarrow$  pending; lig-z[h]  $\leftarrow$  eval(lig-z[h], y); class[h]  $\leftarrow$  simple;
      end;
    right_z: begin class[h]  $\leftarrow$  pending; lig-z[h]  $\leftarrow$  eval(x, lig-z[h]); class[h]  $\leftarrow$  simple;
      end;
    both_z: begin class[h]  $\leftarrow$  pending; lig-z[h]  $\leftarrow$  eval(eval(x, lig-z[h]), y); class[h]  $\leftarrow$  simple;
      end;
    pending: begin x_lig-cycle  $\leftarrow$  x; y_lig-cycle  $\leftarrow$  y; lig-z[h]  $\leftarrow$  257; class[h]  $\leftarrow$  simple;
      end; { the value 257 will break all cycles, since it's not in hash }
  end; { there are no other cases }
  lig-f  $\leftarrow$  lig-z[h];
end;

```

```

125*  ⟨ Do the packet for character c 125* ⟩ ≡
  if packet_start[c] = vf_size then bad_vf(Missing_packet_for_character, c:1)
  else begin left; out(MAP); out_ln; top ← 0; wstack[0] ← 0; xstack[0] ← 0; ystack[0] ← 0;
    zstack[0] ← 0; vf_ptr ← packet_start[c]; vf_limit ← packet_end[c] + 1; f ← 0;
    while vf_ptr < vf_limit do
      begin o ← vf[vf_ptr]; incr(vf_ptr);
      if (o ≤ set1 + 3) ∨ ((o ≥ put1) ∧ (o ≤ put1 + 3)) then
        ⟨ Special cases of DVI instructions to typeset characters 130* ⟩
      else case o of
        ⟨ Cases of DVI instructions that can appear in character packets 127* ⟩
        improper_DVI_for_VF: bad_vf(Illegal_DVI_code, o:1, will_be_ignored);
      end; { there are no other cases }
    end;
  if top > 0 then
    begin bad_vf(More_pushes_than_pops!);
    repeat out(POP); decr(top); until top = 0;
    end;
  right;
end

```

This code is used in section 134*.

126* A procedure called *get_bytes* helps fetch the parameters of DVI commands.

```

  define signed ≡ is_signed { signed is a reserved word in ANSI C }
function get_bytes(k : integer; signed : boolean): integer;
  var a: integer; { accumulator }
  begin if vf_ptr + k > vf_limit then
    begin bad_vf(Packet_ended_prematurely); k ← vf_limit - vf_ptr;
    end;
  a ← vf[vf_ptr];
  if (k = 4) ∨ signed then
    if a ≥ 128 then a ← a - 256;
  incr(vf_ptr);
  while k > 1 do
    begin a ← a * 256 + vf[vf_ptr]; incr(vf_ptr); decr(k);
    end;
  get_bytes ← a;
end;

```

127* Let's look at the simplest cases first, in order to get some experience.

```

define four_cases(#)  $\equiv$  #, # + 1, # + 2, # + 3
define eight_cases(#)  $\equiv$  four_cases(#), four_cases(# + 4)
define sixteen_cases(#)  $\equiv$  eight_cases(#), eight_cases(# + 8)
define thirty_two_cases(#)  $\equiv$  sixteen_cases(#), sixteen_cases(# + 16)
define sixty_four_cases(#)  $\equiv$  thirty_two_cases(#), thirty_two_cases(# + 32)

⟨ Cases of DVI instructions that can appear in character packets 127* ⟩  $\equiv$ 
nop: do_nothing;
push: begin if top = max_stack then
    begin print_ln(^Stack_overflow!^); uexit(1);
    end;
    incr(top); wstack[top]  $\leftarrow$  wstack[top - 1]; xstack[top]  $\leftarrow$  xstack[top - 1]; ystack[top]  $\leftarrow$  ystack[top - 1];
    zstack[top]  $\leftarrow$  zstack[top - 1]; out(^(PUSH)^); out_ln;
    end;
pop: if top = 0 then bad_vf(^More_pops_than_pushes!^)
    else begin decr(top); out(^(POP)^); out_ln;
    end;
set_rule, put_rule: begin if o = put_rule then out(^(PUSH)^);
    left; out(^SETRULE^); out_as_fix(get_bytes(4, true)); out_as_fix(get_bytes(4, true));
    if o = put_rule then out(^(POP)^);
    right;
    end;

```

See also sections 128, 129, and 131.

This code is used in section 125*.

130* Before we typeset a character we make sure that it exists.

```

⟨ Special cases of DVI instructions to typeset characters 130* ⟩  $\equiv$ 
begin if o  $\geq$  set1 then
    if o  $\geq$  put1 then k  $\leftarrow$  get_bytes(o - put1 + 1, false)
    else k  $\leftarrow$  get_bytes(o - set1 + 1, false)
else k  $\leftarrow$  o;
    c  $\leftarrow$  k;
    if (k < 0)  $\vee$  (k > 255) then bad_vf(^Character_, k : 1, ^is_out_of_range_and_will_be_ignored^)
    else if f = font_ptr then bad_vf(^Character_, c : 1, ^in_undeclared_font_will_be_ignored^)
    else begin vf[font_start[f + 1] - 1]  $\leftarrow$  c; { store c in the "hole" we left }
        k  $\leftarrow$  font_chars[f]; while vf[k]  $\neq$  c do incr(k);
        if k = font_start[f + 1] - 1 then
            bad_vf(^Character_, c : 1, ^in_font_, f : 1, ^will_be_ignored^)
        else begin if o  $\geq$  put1 then out(^(PUSH)^);
            left; out(^SETCHAR^); out_char(c);
            if o  $\geq$  put1 then out(^(POP)^);
            right;
            end;
        end;
    end;
end

```

This code is used in section 125*.

132* The main program. The routines sketched out so far need to be packaged into separate procedures, on some systems, since some Pascal compilers place a strict limit on the size of a routine. The packaging is done here in an attempt to avoid some system-dependent changes.

First come the *vf_input* and *organize* procedures, which read the input data and get ready for subsequent events. If something goes wrong, the routines return *false*.

```
function vf_input: boolean;
  var vf_ptr: 0 .. vf_size; { an index into vf }
      k: integer; { all-purpose index }
      c: integer; { character code }
  begin { Read the whole VF file 31* };
    vf_input ← true;
  end;

function organize: boolean;
  var tfm_ptr: index; { an index into tfm }
  begin { Read the whole TFM file 24* };
    { Set subfile sizes lh, bc, ..., np 25* };
    { Compute the base addresses 27 };
    organize ← vf_input;
  end;
```

134* And then there's a routine for individual characters.

```
function do_map(c: byte): boolean;
  var k: integer; f: 0 .. vf_size; { current font number }
  begin { Do the packet for character c 125* };
    do_map ← true;
  end;

function do_characters: boolean;
  label final_end, exit;
  var c: byte; { character being done }
      k: index; { a random index }
      ai: 0 .. lig_size; { index into activity }
  begin { Do the characters 101* };
    do_characters ← true; return;
  final_end: do_characters ← false;
  exit: end;
```

135* Here is where VFtoVP begins and ends.

```
begin initialize;
if ¬organize then goto final_end;
do_simple_things;
{ Do the ligatures and kerns 89 };
{ Check the extensible recipes 110 };
if ¬do_characters then goto final_end;
if verbose then print_ln(' ');
if level ≠ 0 then print_ln('This program isn't working! ');
if ¬perfect then
  begin out(' (COMMENT THE TFM AND/OR VF FILE WAS BAD, ');
    out(' SO THE DATA HAS BEEN CHANGED! ) '); write_ln(vpl_file);
  end;
final_end: end.
```

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

```

    define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
  (Define parse_arguments 136*) ≡
  procedure parse_arguments;
    const n_options = 4; { Pascal won't count array lengths for us. }
    var long_options: array [0 .. n_options] of getopt_struct;
        getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options;
    begin { Initialize the option variables 141* };
    (Define the option table 137*);
    repeat getopt_return_val ← getopt_long_only(argc, argv, ^, long_options, address_of(option_index));
      if getopt_return_val = -1 then
        begin { End of arguments; we exit the loop below. }
          ;
        end
      else if getopt_return_val = "?" then
        begin usage(^vftovp^);
        end
      else if argument_is(^help^) then
        begin usage_help(VFTOVP_HELP, nil);
        end
      else if argument_is(^version^) then
        begin print_version_and_exit(banner, nil, ^D.E. Knuth^, nil);
        end
      else if argument_is(^charcode-format^) then
        begin if strcmp(optarg, ^ascii^) = 0 then charcode_format ← charcode_ascii
              else if strcmp(optarg, ^octal^) = 0 then charcode_format ← charcode_octal
              else print_ln(^Bad character code format^, optarg, ^.^);
              end; { Else it was a flag; getopt has already done the assignment. }
        until getopt_return_val = -1; { Now optind is the index of first non-option on the command line. We
          must have one two three remaining arguments. }
        if (optind + 1 ≠ argc) ∧ (optind + 2 ≠ argc) ∧ (optind + 3 ≠ argc) then
          begin print_ln(^vftovp: Need one to three file arguments.^); usage(^vftovp^);
          end;
        vf_name ← cmdline(optind);
        if optind + 2 ≤ argc then
          begin tfm_name ← cmdline(optind + 1); { The user specified the TFM name. }
          end
        else begin { User did not specify TFM name; default it from the VF name. }
          tfm_name ← basename_change_suffix(vf_name, ^.vf^, ^.tfm^);
          end;
        end;
    end;
  end;

```

This code is used in section 2*.

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

```

  (Define the option table 137*) ≡
    current_option ← 0; long_options[current_option].name ← ^help^;
    long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
    long_options[current_option].val ← 0; incr(current_option);

```

See also sections 138*, 139*, 142*, and 147*.

This code is used in section 136*.

138* Another of the standard options.

```
⟨ Define the option table 137* ⟩ +≡
  long_options[current_option].name ← `version`; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

139* Print progress information?

```
⟨ Define the option table 137* ⟩ +≡
  long_options[current_option].name ← `verbose`; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← address_of(verbose); long_options[current_option].val ← 1;
  incr(current_option);
```

140* The global variable *verbose* determines whether or not we print progress information.

```
⟨ Globals in the outer block 7 ⟩ +≡
  verbose: c_int_type;
```

141* It starts off *false*.

```
⟨ Initialize the option variables 141* ⟩ ≡
  verbose ← false;
```

See also section 146*.

This code is used in section 136*.

142* Here is an option to change how we output character codes.

```
⟨ Define the option table 137* ⟩ +≡
  long_options[current_option].name ← `charcode-format`; long_options[current_option].has_arg ← 1;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

143* We use an “enumerated” type to store the information.

```
⟨ Types in the outer block 5 ⟩ +≡
  charcode_format_type = charcode_ascii .. charcode_default;
```

144*

```
⟨ Constants in the outer block 4* ⟩ +≡
  charcode_ascii = 0; charcode_octal = 1; charcode_default = 2;
```

145*

```
⟨ Globals in the outer block 7 ⟩ +≡
  charcode_format: charcode_format_type;
```

146* It starts off as the default, that is, we output letters and digits as ASCII characters, everything else in octal.

```
⟨ Initialize the option variables 141* ⟩ +≡
  charcode_format ← charcode_default;
```

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

```
⟨ Define the option table 137* ⟩ +≡
  long_options[current_option].name ← 0; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0;
```

148* Global filenames.

```
⟨ Globals in the outer block 7 ⟩ +≡
  vf_name, tfm_name, vpl_name: c_string;
```

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

The following sections were changed by the change file: 2, 4, 11, 21, 22, 24, 25, 31, 32, 35, 36, 37, 39, 40, 43, 44, 45, 50, 51, 61, 62, 63, 101, 113, 117, 118, 125, 126, 127, 130, 132, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149.

-charcode-format: 142*
 -help: 137*
 -verbose: 139*
 -version: 138*
 a: 37*, 46, 48, 59, 63*, 126*
 abort: 24*, 25*
 abs: 121.
 accessible: 88, 91, 92, 93, 98.
 acti: 88, 94.
 activity: 88, 89, 90, 91, 92, 93, 94, 96, 98, 134*
 address_of: 136*, 139*
 ai: 88, 89, 93, 98, 134*
 argc: 21*, 136*
 argument_is: 136*
 argv: 2*, 136*
 ASCII_04: 50*, 51*
 ASCII_10: 50*, 51*
 ASCII_14: 50*, 51*
 axis_height: 19.
 b: 46, 59, 62*
 bad: 70, 73, 75, 83, 85, 93, 97, 99, 107.
 Bad TFM file: 70.
 bad TFM file: 39*
 Bad VF file: 120.
 bad_char: 70, 107, 110.
 bad_char_tail: 70.
 bad_design: 73, 74.
 bad_vf: 120, 121, 123, 125*, 126*, 127*, 129, 130*, 131.
 bal: 119.
 banner: 1, 11*, 136*
 basename_change_suffix: 136*
 bc: 12, 13, 15, 17, 25*, 27, 28, 70, 90, 101*, 113*
 bchar_label: 86, 87, 92, 113*
 big_op_spacing1: 19.
 big_op_spacing5: 19.
 boolean: 30, 68, 119, 126*, 132*, 134*
 bop: 8.
 bot: 18.
 both_z: 112, 115, 116, 118*
 boundary_char: 86, 87, 92, 99, 100.
 byte: 5, 7, 10, 23, 30, 37*, 46, 48, 54, 61*, 75, 124, 134*
 b0: 37*, 38, 39*, 40*, 41.
 b1: 37*, 38, 39*, 40*, 41.
 b2: 37*, 38, 39*, 40*, 41.
 b3: 37*, 38, 39*, 40*, 41.
 c: 48, 61*, 70, 75, 132*, 134*
 c_int_type: 136*, 140*
 c_string: 148*
 cc: 8, 47, 86, 91, 92, 95, 115, 116.
 char: 37*, 42, 45*, 50*
 char_base: 26, 27, 28.
 char_info: 15, 26, 28, 101*
 char_info_word: 13, 15, 16.
 Character c does not exist: 47.
 Character list link...: 107.
 Character...will be ignored: 130*
 charcode_ascii: 61*, 136*, 143*, 144*
 charcode_default: 143*, 144*, 146*
 charcode_format: 61*, 136*, 145*, 146*
 charcode_format_type: 143*, 145*
 charcode_octal: 61*, 136*, 144*
 chars_on_line: 68, 69, 70, 101*, 120.
 check sum: 14.
 Check sum...mismatch: 32*
 Check sum...replaced...: 40*
 check_BCPL: 75, 76, 78.
 check_fix: 83, 85.
 check_fix_tail: 83.
 check_sum: 28, 32*, 72, 79.
 class: 4*, 112, 113*, 115, 118*
 class_var: 4*
 cmdline: 21*, 136*
 coding scheme: 14.
 const_c_string: 50*
 correct_bad_char: 70, 99, 100.
 correct_bad_char_tail: 70.
 count: 30, 32*, 47, 98.
 cs: 7.
 cur_name: 37*, 39*, 44*, 45*
 current_option: 136*, 137*, 138*, 139*, 142*, 147*
 Cycle in a character list: 107.
 d: 70.
 decr: 5, 46, 53, 57, 58, 60, 66, 91, 115, 117*, 119, 125*, 126*, 127*, 131.
 default_directory: 42.
 default_directory_name: 42.
 default_directory_name_length: 42.
 default_rule_thickness: 19.
 delim1: 19.
 delim2: 19.
 delta: 63*, 65.
 denom1: 19.
 denom2: 19.

depth: 15, 28, 104.
 Depth index for char: 104.
 Depth n is too big: 85.
depth_base: 26, 27, 28, 85.
depth_index: 15, 28, 101*, 104.
 design size: 14.
 Design size wrong: 73.
 Design size...replaced...: 41.
design_size: 28, 32*, 74, 121.
 DESIGNSIZE IS IN POINTS: 74.
dig: 52, 53, 54, 59, 60, 63*, 64.
 Discarding earlier packet...: 47.
do_characters: 134*, 135*.
do_map: 101*, 134*.
do_nothing: 5, 101*, 116, 118*, 127*.
do_simple_things: 133, 135*.
down1: 8, 128.
ds: 7.
dvi: 8.
ec: 12, 13, 15, 17, 25*, 27, 28, 90, 101*, 113*.
eight_cases: 127*.
endif: 117*.
eof: 24*, 32*, 33, 34, 38, 39*, 46.
eop: 8.
eval: 117*, 118*.
eval_two_bytes: 25*.
exit: 5, 115, 119, 134*.
ext_tag: 16, 101*.
exten: 16, 28, 109.
exten_base: 26, 27, 28, 110.
extend_filename: 21*.
 Extensible index for char: 108.
 Extensible recipe involves...: 110.
extensible_recipe: 13, 18.
extra_space: 19.
f: 63*, 118*, 133, 134*.
face: 14, 50*, 62*.
false: 33, 70, 90, 92, 119, 120, 129, 130*, 131, 132*, 134*, 141*.
family: 28, 78.
 family name: 14.
 File ended without a postamble: 33.
final_end: 3, 101*, 134*, 135*.
fix_word: 7, 8, 13, 14, 19, 28, 63*, 83, 85, 121.
flag: 137*, 138*, 139*, 142*, 147*.
fnt_def1: 7, 8, 35*.
fnt_def2: 7.
fnt_def3: 7.
fnt_def4: 7.
fnt_num_0: 8, 129.
fnt_num0: 8.
fnt1: 8, 129.

fnt4: 8.
 font identifier: 14.
font_bc: 37*, 39*.
font_chars: 30, 39*, 130*.
font_ec: 37*, 39*.
font_lh: 37*, 39*.
font_number: 30, 35*, 129.
font_ptr: 30, 33, 35*, 36*, 39*, 40*, 41, 45*, 122, 129, 130*.
font_start: 30, 33, 35*, 36*, 40*, 41, 45*, 120, 122, 123, 130*.
font_type: 29, 61*, 71, 76, 82, 84.
forward: 118*.
four_cases: 127*, 128, 129, 131.
fprint_real: 2*.
free: 39*.
 Fuchs, David Raymond: 1.
get_bytes: 126*, 127*, 128, 129, 130*, 131.
get_vf: 46.
getopt: 136*.
getopt_long_only: 136*.
getopt_return_val: 136*.
getopt_struct: 136*.
h: 112, 117*, 118*.
has_arg: 137*, 138*, 139*, 142*, 147*.
hash: 112, 113*, 115, 117*, 118*.
hash_input: 114, 115.
hash_list: 112, 113*, 115.
hash_ptr: 112, 113*, 115.
hash_size: 4*, 112, 113*, 115, 117*.
header: 14.
height: 15, 28, 103.
 Height index for char...: 103.
 Height n is too big: 85.
height_base: 26, 27, 28, 85.
height_index: 15, 28, 101*, 103.
hh: 112, 113*.
i: 70, 133.
 I can't handle that many fonts: 35*.
 I'm out of VF memory: 39*.
id_byte: 7, 32*.
ifdef: 117*.
 Illegal byte...: 35*.
 Improper font area: 123.
 Improper font name: 123.
improper_DVI_for_VF: 8, 125*.
 Incomplete subfiles...: 25*.
 Incorrect TFM width...: 47.
incr: 5, 32*, 33, 34, 35*, 39*, 57, 58, 59, 60, 64, 91, 95, 98, 101*, 115, 119, 125*, 126*, 127*, 129, 130*, 131, 137*, 138*, 139*, 142*.

index: 22*, 48, 58, 59, 62*, 63*, 70, 75, 117*,
 118*, 132*, 134*
index_type: 22*
 Infinite ligature loop...: 113*
initialize: 2*, 135*
intcast: 63*
integer: 2*, 26, 30, 37*, 46, 48, 53, 63*, 115, 117*, 119,
 121, 124, 126*, 132*, 133, 134*, 136*
is_signed: 126*
italic: 15, 28, 105.
 Italic correction index for char...: 105.
 Italic correction n is too big: 85.
italic_base: 26, 27, 28, 85.
italic_index: 15, 28, 101*, 105.
j: 54, 59, 63*, 75, 119.
k: 2*, 46, 48, 58, 59, 62*, 63*, 70, 75, 119, 121,
126*, 132*, 133, 134*
kern: 17, 28, 85, 99.
 Kern index too large: 99.
 Kern n is too big: 85.
 Kern step for nonexistent...: 99.
kern_base: 26, 27, 28.
kern_flag: 17, 97, 99, 116.
key: 115, 117*
kpse_find_tfm: 39*
kpse_find_vf: 11*
kpse_init_prog: 2*
kpse_open_file: 11*
kpse_set_progname: 2*
kpse_tfm_format: 11*
kpse_vf_format: 11*
l: 37*, 57, 58, 59, 75, 119.
label_ptr: 86, 87, 90, 91, 92.
label_table: 86, 87, 90, 91, 92, 95.
left: 57, 72, 74, 77, 78, 79, 80, 81, 83, 89, 92, 95,
 96, 99, 100, 101*, 102, 103, 104, 105, 106, 107,
 108, 109, 120, 122, 123, 125*, 127*, 130*, 131.
left_z: 112, 116, 118*
level: 55, 56, 57, 94, 96, 97, 135*
lf: 12, 22*, 24*, 25*
lh: 12, 13, 25*, 27, 71, 79, 80.
 Lig...skips too far: 93.
lig_f: 113*, 117*, 118*
lig_kern: 4*, 16, 17.
lig_kern_base: 26, 27, 28.
lig_kern_command: 13, 17.
lig_size: 4*, 25*, 86, 88, 90, 134*
lig_step: 28, 90, 92, 93, 97, 106, 113*, 114, 116.
lig_tag: 16, 86, 90, 101*, 113*
lig_z: 112, 115, 118*
 Ligature step for nonexistent...: 100.
 Ligature step produces...: 100.

Ligature unconditional stop...: 97.
 Ligature/kern starting index...: 90, 92.
list_tag: 16, 101*, 107.
long_char: 8, 33, 47.
long_options: 136*, 137*, 138*, 139*, 142*, 147*
 Mapped font size...big: 35*
mathex: 29, 76, 82, 84.
mathsy: 29, 76, 82, 84.
max_fonts: 4*, 30, 35*
max_stack: 4*, 124, 127*
MBL_string: 50*, 51*, 62*
mid: 18.
 Missing packet: 125*
 More pops than pushes: 127*
 More pushes than pops: 125*
n_options: 136*
name: 136*, 137*, 138*, 139*, 142*, 147*
name_end: 45*
name_start: 45*
nd: 12, 13, 25*, 27, 85, 104.
ne: 12, 13, 25*, 27, 108, 110.
 Negative packet length: 47.
next_char: 17.
nh: 12, 13, 25*, 27, 85, 103.
ni: 12, 13, 25*, 27, 85, 105.
nil: 5.
nk: 12, 13, 25*, 27, 85, 99.
nl: 12, 13, 17, 25*, 27, 89, 90, 92, 93, 94, 97,
 106, 113*, 114.
no_tag: 16, 28, 101*
nonexistent: 28, 47, 99, 100, 107, 109, 110.
 Nonstandard ASCII code...: 75.
nonzero_fix: 85.
nop: 8, 127*
not_found: 5, 119.
np: 12, 13, 25*, 81, 82.
num1: 19.
num2: 19.
num3: 19.
nw: 12, 13, 25*, 27, 85, 102.
o: 124.
odd: 100.
 One of the subfile sizes...: 25*
op_byte: 17.
optarg: 136*
optind: 21*, 136*
option_index: 136*
organize: 132*, 135*
out: 49, 53, 57, 58, 59, 61*, 62*, 63*, 65, 66, 72,
 73, 74, 77, 78, 79, 80, 81, 83, 84, 89, 92, 95,
 96, 98, 99, 100, 101*, 102, 103, 104, 105, 106,

- 107, 108, 109, 113*, 120, 122, 123, 125*, 127*, 128, 129, 130*, 131, 135*.
- out_as_fix*: 121, 127*, 128.
- out_BCPL*: 58, 77, 78.
- out_char*: 61*, 92, 95, 99, 100, 101*, 107, 109, 130*.
- out_digs*: 53, 59, 64.
- out_face*: 62*, 79.
- out_fix*: 63*, 74, 83, 99, 102, 103, 104, 105, 121, 122.
- out_hex*: 131.
- out_ln*: 57, 74, 81, 89, 96, 98, 101*, 106, 108, 122, 125*, 127*, 128, 129, 131.
- out_octal*: 59, 61*, 62*, 72, 79, 122.
- output*: 2*.
- Oversize dimension...**: 121.
- packet_end*: 30, 47, 125*.
- packet_found*: 30, 33, 35*, 47.
- packet_start*: 30, 33, 47, 125*.
- param*: 14, 19, 28, 83.
- param_base*: 26, 27, 28.
- Parameter n is too big**: 83.
- Parenthesis...changed to slash**: 75.
- parse_arguments*: 2*, 136*.
- pass_through*: 88, 90, 92, 94.
- pending*: 112, 118*.
- perfect*: 68, 69, 70, 90, 92, 120, 135*.
- pl*: 8, 30, 47.
- pop*: 8, 127*.
- post*: 8, 9, 33, 34.
- pre*: 7, 8, 31*, 35*.
- print*: 2*, 11*, 32*, 36*, 53, 54, 70, 90, 92, 101*, 107, 113*.
- print_digs*: 53, 54.
- print_ln*: 2*, 11*, 24*, 31*, 32*, 33, 34, 36*, 39*, 40*, 41, 47, 70, 73, 82, 83, 90, 92, 93, 100, 101*, 107, 113*, 120, 127*, 135*, 136*.
- print_octal*: 54, 70, 90, 101*, 107, 113*.
- print_real*: 2*, 36*.
- print_version_and_exit*: 136*.
- push*: 8, 127*.
- put_byte*: 62*.
- put_rule*: 8, 127*.
- put1*: 8, 125*, 130*.
- quad*: 19.
- r*: 70.
- random_word*: 28, 79, 80.
- range_error*: 70, 102, 103, 104, 105, 108.
- RCE_string*: 50*, 51*, 62*.
- read*: 24*, 31*, 38.
- read_tfm*: 38.
- read_tfm_word*: 38, 39*.
- read_vf*: 31*, 32*, 33, 34, 46.
- real*: 30.
- real_dsize*: 30, 32*, 36*.
- remainder*: 15, 16, 17, 28, 90, 106, 107, 108, 113*.
- rep*: 18.
- reset_tag*: 28, 90, 107, 108.
- resetbin*: 39*.
- return**: 5.
- rewrite*: 21*.
- RI_string*: 50*, 51*, 62*.
- right*: 57, 72, 74, 77, 78, 79, 80, 81, 83, 89, 92, 94, 95, 96, 99, 100, 101*, 102, 103, 104, 105, 106, 107, 108, 109, 120, 122, 123, 125*, 127*, 130*, 131.
- right_z*: 112, 116, 118*.
- right1*: 8, 128.
- rr*: 86, 87, 90, 91, 92, 95.
- s*: 62*.
- scheme*: 28, 76, 77.
- set_char_0*: 8.
- set_rule*: 8, 127*.
- set1*: 8, 125*, 130*.
- seven_bit_safe_flag*: 14, 80.
- short_char0*: 8.
- short_char241*: 8.
- should be zero**: 85.
- signed*: 126*.
- simple*: 112, 113*, 115, 116, 118*.
- sixteen_cases*: 127*.
- sixty_four_cases*: 127*, 129.
- skip_byte*: 17.
- slant*: 19.
- Sorry, I haven't room...**: 113*.
- sort_ptr*: 86, 91, 94, 95, 101*.
- space*: 19.
- space_shrink*: 19.
- space_stretch*: 19.
- Stack overflow**: 127*.
- stderr*: 2*.
- stdout*: 21*.
- stop_flag*: 17, 90, 93, 97, 98, 106, 113*, 114.
- strcmp*: 136*.
- String is too long...**: 75.
- string_balance*: 119, 120, 123, 131.
- stringcast*: 39*.
- strncpy*: 45*.
- stuff*: 13.
- subdrop*: 19.
- Subfile sizes don't add up...**: 25*.
- sub1*: 19.
- sub2*: 19.
- supdrop*: 19.
- sup1*: 19.
- sup2*: 19.
- sup3*: 19.

system dependencies: 2* 11* 36* 38, 42, 44* 45* 61*
t: 115.
tag: 15, 16, 28, 86, 90, 101* 107, 113*
temp_byte: 30, 31* 32* 33, 34, 35* 47.
text: 20.
tfm: 4* 8, 22* 23, 24* 25* 26, 28, 32* 47, 48, 58,
 59, 60, 61* 62* 63* 70, 74, 75, 76, 80, 83, 85,
 90, 92, 93, 97, 98, 99, 100, 106, 109, 110, 113*
 114, 116, 119, 121, 122, 132*
 TFM file can't be opened: 39*
tfm_file: 2* 10, 11* 22* 24* 38, 39*
tfm_name: 11* 39* 136* 148*
tfm_ptr: 24* 25* 132*
tfm_size: 4* 22* 23, 24*
tfm_width: 47, 48.
 The character code range...: 25*
 The file claims...: 24*
 The file ended prematurely: 32*
 The file has fewer bytes...: 24*
 The file is bigger...: 24* 32*
 The first byte...: 24* 31*
 The header length...: 25*
 The input...one byte long: 24* 32*
 The lig/kern program...: 25*
 THE TFM AND/OR VF FILE WAS BAD...: 135*
 There are ... recipes: 25*
 There's some extra junk...: 24* 34.
thirty_two_cases: 127*
 This program isn't working: 135*
 Title is not balanced: 120.
top: 18, 124, 125* 127* 128.
 trouble is brewing...: 39*
true: 47, 69, 119, 127* 128, 132* 134*
uexit: 24* 31* 113* 127*
 Undeclared font selected: 129.
unreachable: 88, 89, 90, 96.
 Unusual number of fontdimen...: 82.
usage: 136*
usage_help: 136*
val: 137* 138* 139* 142* 147*
vanilla: 29, 61* 71, 76.
verbose: 11* 32* 35* 40* 101* 135* 139* 140* 141*
version_string: 11*
vf: 4* 30, 32* 35* 36* 39* 40* 41, 45* 119, 120, 122,
 123, 125* 126* 130* 131, 132*
 VF data not a multiple of 4 bytes: 34.
vf_abort: 31* 32* 35* 39* 47.
vf_count: 30, 32* 33, 34, 46.
vf_file: 2* 7, 11* 30, 31* 32* 33, 34, 46.
vf_input: 132*
vf_limit: 124, 125* 126* 131.
vf_name: 11* 136* 148*
vf_ptr: 30, 32* 33, 35* 36* 39* 45* 47, 125*
 126* 131, 132*
vf_read: 35* 46, 47.
vf_size: 4* 30, 32* 33, 39* 47, 124, 125* 132*
 133, 134*
vf_store: 32* 35* 47.
VFtoVP: 2*
VFTOVP_HELP: 136*
vpl_file: 2* 20, 21* 49, 57, 62* 135*
vpl_name: 21* 148*
width: 15, 28, 48, 85, 102.
 Width n is too big: 85.
width_base: 26, 27, 28, 85.
width_index: 15, 28, 101* 102.
write: 2* 49.
write_ln: 2* 57, 135*
 Wrong VF version...: 32*
wstack: 124, 125* 127* 128.
w0: 8, 128.
w1: 8, 128.
x: 117* 118* 121.
x_height: 19.
x_lig_cycle: 112, 113* 118*
xchr: 32* 36* 45* 50* 51* 58, 61* 120, 123, 131.
xmalloc_array: 45*
xstack: 124, 125* 127* 128.
xxx1: 8, 131.
xxx4: 8.
x0: 8, 128.
x1: 8, 128.
y: 115, 117* 118*
y_lig_cycle: 112, 113* 118*
ystack: 124, 125* 127* 128.
y0: 8, 128.
y1: 8, 128.
zstack: 124, 125* 127* 128.
zz: 115, 116.
z0: 8, 128.
z1: 8, 128.

- ⟨Build the label table 90⟩ Used in section 89.
- ⟨Cases of DVI instructions that can appear in character packets 127*, 128, 129, 131⟩ Used in section 125*.
- ⟨Check and output the *i*th parameter 83⟩ Used in section 81.
- ⟨Check for a boundary char 92⟩ Used in section 89.
- ⟨Check for ligature cycles 113*⟩ Used in section 89.
- ⟨Check the check sum 40*⟩ Used in section 39*.
- ⟨Check the design size 41⟩ Used in section 39*.
- ⟨Check the extensible recipes 110⟩ Used in section 135*.
- ⟨Check the *fix.word* entries 85⟩ Used in section 133.
- ⟨Check to see if *np* is complete for this font type 82⟩ Used in section 81.
- ⟨Compute the base addresses 27⟩ Used in section 132*.
- ⟨Compute the command parameters *y*, *cc*, and *zz* 116⟩ Used in section 115.
- ⟨Compute the *activity* array 93⟩ Used in section 89.
- ⟨Constants in the outer block 4*, 144*⟩ Used in section 2*.
- ⟨Define the option table 137*, 138*, 139*, 142*, 147*⟩ Used in section 136*.
- ⟨Define *parse_arguments* 136*⟩ Used in section 2*.
- ⟨Do the characters 101*⟩ Used in section 134*.
- ⟨Do the header 71⟩ Used in section 133.
- ⟨Do the ligatures and kerns 89⟩ Used in section 135*.
- ⟨Do the local fonts 122⟩ Used in section 133.
- ⟨Do the packet for character *c* 125*⟩ Used in section 134*.
- ⟨Do the parameters 81⟩ Used in section 133.
- ⟨Do the virtual font title 120⟩ Used in section 133.
- ⟨Enter data for character *c* starting at location *i* in the hash table 114⟩ Used in sections 113* and 113*.
- ⟨Globals in the outer block 7, 10, 12, 20, 23, 26, 29, 30, 37*, 42, 50*, 52, 55, 68, 70, 86, 88, 112, 124, 140*, 145*, 148*⟩
Used in section 2*.
- ⟨Initialize the option variables 141*, 146*⟩ Used in section 136*.
- ⟨Insert (*c*, *r*) into *label.table* 91⟩ Used in section 90.
- ⟨Labels in the outer block 3⟩ Used in section 2*.
- ⟨Move font name into the *cur_name* string 44*, 45*⟩ Used in section 39*.
- ⟨Output a kern step 99⟩ Used in section 97.
- ⟨Output a ligature step 100⟩ Used in section 97.
- ⟨Output an extensible character recipe 108⟩ Used in section 101*.
- ⟨Output and correct the ligature/kern program 94⟩ Used in section 89.
- ⟨Output any labels for step *i* 95⟩ Used in section 94.
- ⟨Output either **SKIP** or **STOP** 98⟩ Used in section 97.
- ⟨Output step *i* of the ligature/kern program 97⟩ Used in sections 94 and 106.
- ⟨Output the applicable part of the ligature/kern program as a comment 106⟩ Used in section 101*.
- ⟨Output the character coding scheme 77⟩ Used in section 71.
- ⟨Output the character link unless there is a problem 107⟩ Used in section 101*.
- ⟨Output the character's depth 104⟩ Used in section 101*.
- ⟨Output the character's height 103⟩ Used in section 101*.
- ⟨Output the character's width 102⟩ Used in section 101*.
- ⟨Output the check sum 72⟩ Used in section 71.
- ⟨Output the design size 74⟩ Used in section 71.
- ⟨Output the extensible pieces that exist 109⟩ Used in section 108.
- ⟨Output the family name 78⟩ Used in section 71.
- ⟨Output the font area and name 123⟩ Used in section 122.
- ⟨Output the fraction part, $f/2^{20}$, in decimal notation 65⟩ Used in section 63*.
- ⟨Output the integer part, *a*, in decimal notation 64⟩ Used in section 63*.
- ⟨Output the italic correction 105⟩ Used in section 101*.
- ⟨Output the name of parameter *i* 84⟩ Used in section 83.

- ⟨Output the rest of the header 79⟩ Used in section 71.
- ⟨Output the *seven_bit_safe_flag* 80⟩ Used in section 71.
- ⟨Print the name of the local font 36*⟩ Used in section 35*.
- ⟨Read and store a character packet 47⟩ Used in section 33.
- ⟨Read and store a font definition 35*⟩ Used in section 33.
- ⟨Read and store the font definitions and character packets 33⟩ Used in section 31*.
- ⟨Read and verify the postamble 34⟩ Used in section 31*.
- ⟨Read the local font's TFM file and record the characters it contains 39*⟩ Used in section 35*.
- ⟨Read the preamble command 32*⟩ Used in section 31*.
- ⟨Read the whole TFM file 24*⟩ Used in section 132*.
- ⟨Read the whole VF file 31*⟩ Used in section 132*.
- ⟨Reduce l by one, preserving the invariants 60⟩ Used in section 59.
- ⟨Reduce negative to positive 66⟩ Used in section 63*.
- ⟨Set initial values 11*, 21*, 51*, 56, 69, 87⟩ Used in section 2*.
- ⟨Set subfile sizes lh , bc , ..., np 25*⟩ Used in section 132*.
- ⟨Set the true *font_type* 76⟩ Used in section 71.
- ⟨Special cases of DVI instructions to typeset characters 130*⟩ Used in section 125*.
- ⟨Take care of commenting out unreachable steps 96⟩ Used in section 94.
- ⟨Types in the outer block 5, 22*, 143*⟩ Used in section 2*.

The VFtoVP processor

(Version 1.3, December 2002)

	Section	Page
Introduction	1	102
Virtual fonts	6	103
Font metric data	10	103
Unpacking the TFM file	22	104
Unpacking the VF file	30	105
Basic output subroutines	49	109
Outputting the TFM info	67	111
Checking for ligature loops	111	112
Outputting the VF info	119	114
The main program	132	116
System-dependent changes	136	117
Index	149	119

The preparation of this program was supported in part by the National Science Foundation and by the System Development Foundation. ‘TEX’ is a trademark of the American Mathematical Society.