2* `MFT` uses a few features of the local Pascal compiler that may need to be changed in other installations:

1) Case statements have a default.
2) Input-output routines may need to be adapted for use with a particular character set and/or for printing messages on the user's terminal.

These features are also present in the Pascal version of TEX, where they are used in a similar (but more complex) way. System-dependent portions of `MFT` can be identified by looking at the entries for 'system dependencies' in the index below.

The "banner line" defined here should be changed whenever `MFT` is modified.

define *banner* ≡ `'This␣is␣MFT,␣C␣Version␣2.0'`

3* The program begins with a fairly normal header, made up of pieces that will mostly be filled in later. The `MF` input comes from files *mf_file*, *change_file*, and *style_file*; the TEX output goes to file *tex_file*.

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 *end_of_MFT*.

```
⟨ Compiler directives 4* ⟩
program MFT;
  const ⟨ Constants in the outer block 8 ⟩
  type ⟨ Types in the outer block 12 ⟩
  var ⟨ Globals in the outer block 9 ⟩
    ⟨ Error handling procedures 29 ⟩
    ⟨ scan_args procedure 115* ⟩
  procedure initialize;
    var ⟨ Local variables for initialization 14 ⟩
    begin ⟨ Set initial values 10 ⟩
    end;
```

4* The Pascal compiler used to develop this system has "compiler directives" that can appear in comments whose first character is a dollar sign. In our case these directives tell the compiler to detect things that are out of range.

⟨ Compiler directives 4* ⟩ ≡

This code is used in section 3*.

13:* The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lowercase letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, especially in a program for font design; so the present specification of MFT has been written under the assumption that the Pascal compiler and run-time system permit the use of text files with more than 64 distinguishable characters. More precisely, we assume that the character set contains at least the letters and symbols associated with ASCII codes *´40* through *´176*. If additional characters are present, MFT can be configured to work with them too.

Since we are dealing with more characters than were present in the first Pascal compilers, we have to decide what to call the associated data type. Some Pascals use the original name *char* for the characters in text files, even though there now are more than 64 such characters, 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 that are converted to and from *ASCII_code* when they are input and output. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

```
define text_char ≡ ASCII_code  { the data type of characters in text files }
define first_text_char = 0  { ordinal number of the smallest element of text_char }
define last_text_char = 255  { ordinal number of the largest element of text_char }
⟨ Types in the outer block 12 ⟩ +≡
  text_file = packed file of text_char;
```

17:* The ASCII code is "standard" only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. If MFT is being used on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, it doesn't really matter what codes are specified in *xchr*[0 .. *´37*], but the safest policy is to blank everything out by using the code shown below.

However, other settings of *xchr* will make MFT more friendly on computers that have an extended character set, so that users can type things like '≠' instead of '<>', and so that MFT can echo the page breaks found in its input. People with extended character sets can assign codes arbitrarily, giving an *xchr* equivalent to whatever characters the users of MFT are allowed to have in their input files. Appropriate changes to MFT's *char_class* table should then be made. (Unlike TEX, each installation of METAFONT has a fixed assignment of category codes, called the *char_class*.) Such changes make portability of programs more difficult, so they should be introduced cautiously if at all.

```
⟨ Set initial values 10 ⟩ +≡
  for i ← 1 to ´37 do xchr[i] ← chr(i);
  for i ← ´177 to ´377 do xchr[i] ← chr(i);
```

20:* Terminal output is done by writing on file *term_out*, which is assumed to consist of characters of type *text_char*:

```
define term_out ≡ stdout
define print(#) ≡ write(term_out, #)   { 'print' means write on the terminal }
define print_ln(#) ≡ write_ln(term_out, #)   { 'print' and then start new line }
define new_line ≡ write_ln(term_out)   { start new line on the terminal }
define print_nl(#) ≡   { print information starting on a new line }
        begin new_line; print(#);
        end
```

21:* Different systems have different ways of specifying that the output on a certain file will appear on the user's terminal.

```
⟨ Set initial values 10 ⟩ +≡
    { nothing need be done }
```

22:* The *update_terminal* procedure is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent.

```
define update_terminal ≡ flush(term_out)   { empty the terminal output buffer }
```

24:* The following code opens the input files. This is called after *scan_args* has set the file name variables appropriately.

```
procedure open_input;   { prepare to read inputs }
  begin if test_read_access(mf_file_name, MF_INPUT_PATH) then reset(mf_file, mf_file_name)
  else begin print_pascal_string(mf_file_name); print('':␣Metafont␣source␣file␣not␣found'); uexit(1);
    end;
  reset(change_file, change_file_name);
  if test_read_access(style_file_name, TEX_INPUT_PATH) then reset(style_file, style_file_name)
  else begin print_pascal_string(style_file_name); print('':␣Style␣file␣not␣found'); uexit(1);
    end;
  end;
```

26:* The following code opens *tex_file*. The *scan_args* procedure is used to set up *tex_file_name* as required.

```
⟨ Set initial values 10 ⟩ +≡
  scan_args; rewrite(tex_file, tex_file_name);
```

28.* The *input_ln* procedure brings the next line of input from the specified file into the *buffer* array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false*. The conventions of TEX are followed; i.e., *ASCII_code* numbers representing the next line of the file are input into *buffer*[0], *buffer*[1], ..., *buffer*[*limit* − 1]; trailing blanks are ignored; and the global variable *limit* is set to the length of the line. The value of *limit* must be strictly less than *buf_size*.

```
function input_ln(var f : text_file): boolean;  { inputs a line or returns false }
  var final_limit: 0 .. buf_size;  { limit without trailing blanks }
  begin limit ← 0; final_limit ← 0;
  if eof(f) then input_ln ← false
  else begin while ¬eoln(f) do
      begin buffer[limit] ← xord[getc(f)]; incr(limit);
      if buffer[limit − 1] ≠ "␣" then final_limit ← limit;
      if limit = buf_size then
        begin while ¬eoln(f) do vgetc(f);
        decr(limit);  { keep buffer[buf_size] empty }
        if final_limit > limit then final_limit ← limit;
        print_nl('!␣Input␣line␣too␣long'); loc ← 0; error;
        end;
      end;
    read_ln(f); limit ← final_limit; input_ln ← true;
    end;
  end;
```

31.* The *jump_out* procedure cleans up, prints appropriate messages, and exits back to the operating system.

```
define fatal_error(#) ≡
        begin new_line; print(#); error; mark_fatal; jump_out;
        end
⟨ Error handling procedures 29 ⟩ +≡
procedure jump_out;
  begin{here files should be closed if the operating system requires it}
  ⟨ Print the job history 113 ⟩;
  new_line;
  if (history ≠ spotless) ∧ (history ≠ harmless_message) then uexit(1)
  else uexit(0);
  end;
```

112.* The main program. Let's put it all together now: `MFT` starts and ends here.

```
begin initialize; { beginning of the main program }
print_ln(banner); { print a "banner line" }
⟨ Store all the primitives 65 ⟩;
⟨ Store all the translations 73 ⟩;
⟨ Initialize the input system 44 ⟩;
do_the_translation; ⟨ Check that all changes have been read 49 ⟩;
⟨ Print the job history 113 ⟩;
new_line;
if (history ≠ spotless) ∧ (history ≠ harmless_message) then uexit(1)
else uexit(0);
end.
```

114.* System-dependent changes.

The user calls `MFT` with arguments on the command line. These are either file names or flags (beginning with '`-`'). The following globals are for communicating the user's desires to the rest of the program. The various *file_name* variables contain strings with the full names of those files, as UNIX knows them.

The flags that affect `MFT` are `-c` and `-s`, whose statuses is kept in *no_change* and *no_style*, respectively.

⟨ Globals in the outer block 9 ⟩ +≡

mf_file_name, *change_file_name*, *style_file_name*, *tex_file_name*: **packed array** [1 .. *FILENAMESIZE*] **of** *char*;

no_change, *no_style*: *boolean*;

115.* The *scan_args* procedure looks at the command line arguments and sets the *file_name* variables accordingly.

At least one file name must be present as the first argument: the `mf` file. It may have an extension, or it may omit it to get `´.mf´` added. If there is only one file name, the output file name is formed by replacing the `mf` file name extension by `´.tex´`. Thus, the command line `mf foo` implies the use of the `METAFONT` input file `foo.mf` and the output file `foo.tex`. If this style of command line, with only one argument is used, the default style file, *plain.mft*, will be used to provide basic formatting.

An argument beginning with a hyphen is a flag. Any letters following the minus sign may cause global flag variables to be set. Currently, a `c` means that there is a change file, and an `s` means that there is a style file. The auxiliary files must of course appear in the same order as the flags. For example, the flag `-sc` must be followed by the name of the *style_file* first, and then the name of the *change_file*.

```
⟨ scan_args procedure 115* ⟩ ≡
procedure scan_args;
  var dot_pos, slash_pos, i, a: integer;  { indices }
    which_file_next: char; fname: array [1 .. FILENAMESIZE] of char;  { temporary argument holder }
    found_mf, found_change, found_style: boolean;  { true when those file names have been seen }
  begin  { get style file path from the environment variable TEXINPUTS }
  set_paths(MF_INPUT_PATH_BIT + TEX_INPUT_PATH_BIT); found_mf ← false;
  ⟨ Set up null change file 119* ⟩;
  found_change ← true;  { default to no change file }
  ⟨ Set up plain style file 120* ⟩;
  found_style ← true;  { default to plain style file }
  for a ← 1 to argc − 1 do
    begin argv(a, fname);  { put argument number a into fname }
    if fname[1] ≠ ´-´ then
      begin if ¬found_mf then ⟨ Get mf_file_name from fname, and set up tex_file_name 116* ⟩
      else if ¬found_change then
          begin if which_file_next ≠ ´s´ then
            begin ⟨ Get change_file_name from fname 117* ⟩;
            which_file_next ← ´s´
            end
          else ⟨ Get style_file_name from fname 118* ⟩
          end
        else if ¬found_style then
            begin if which_file_next = ´s´ then
              begin ⟨ Get style_file_name from fname 118* ⟩;
              which_file_next ← ´c´
              end;
            end
          else ⟨ Print usage error message and quit 122* ⟩;
      end
    else ⟨ Handle flag argument in fname 121* ⟩;
    end;
  if ¬found_mf then ⟨ Print usage error message and quit 122* ⟩;
  end;
```

This code is used in section 3*.

116.* Use all of *fname* for the *mf_file_name* if there is a ´.´ in it, otherwise add ´.mf´. The TEX file name comes from adding things after the dot. The *argv* procedure will not put more than *FILENAMESIZE* − 5 characters into *fname*, and this leaves enough room in the *file_name* variables to add the extensions. We declared *fname* to be of size *FILENAMESIZE* instead of *FILENAMESIZE* − 5 because the latter implies *FILENAMESIZE* must be a numeric constant—and we don't want to repeat the value from `site.h` just to save five bytes of memory.

The end of a file name is marked with a ´␣´, the convention assumed by the *reset* and *rewrite* procedures.

```
⟨ Get mf_file_name from fname, and set up tex_file_name 116* ⟩ ≡
  begin dot_pos ← −1; slash_pos ← −1; i ← 1;
  while (fname[i] ≠ ´␣´) ∧ (i ≤ FILENAMESIZE − 5) do
    begin mf_file_name[i] ← fname[i];
    if fname[i] = ´.´ then dot_pos ← i;
    if fname[i] = ´/´ then slash_pos ← i;
    incr(i);
    end;
  if (dot_pos = −1)|(dot_pos < slash_pos) then
    begin dot_pos ← i; mf_file_name[dot_pos] ← ´.´; mf_file_name[dot_pos + 1] ← ´m´;
    mf_file_name[dot_pos + 2] ← ´f´; mf_file_name[dot_pos + 3] ← ´␣´;
    end;
  for i ← 1 to dot_pos do
    begin tex_file_name[i] ← mf_file_name[i];
    end;
  tex_file_name[dot_pos + 1] ← ´t´; tex_file_name[dot_pos + 2] ← ´e´; tex_file_name[dot_pos + 3] ← ´x´;
  tex_file_name[dot_pos + 4] ← ´␣´; which_file_next ← ´z´; found_mf ← true;
  end
```

This code is used in section 115*.

117.* Use all of *fname* for the *change_file_name* if there is a ´.´ in it, otherwise add ´.ch´.

```
⟨ Get change_file_name from fname 117* ⟩ ≡
  begin dot_pos ← −1; slash_pos ← −1; i ← 1;
  while (fname[i] ≠ ´␣´) ∧ (i ≤ FILENAMESIZE − 5) do
    begin change_file_name[i] ← fname[i];
    if fname[i] = ´.´ then dot_pos ← i;
    if fname[i] = ´/´ then slash_pos ← i;
    incr(i);
    end;
  if (dot_pos = −1)|(dot_pos < slash_pos) then
    begin dot_pos ← i; change_file_name[dot_pos] ← ´.´; change_file_name[dot_pos + 1] ← ´c´;
    change_file_name[dot_pos + 2] ← ´h´; change_file_name[dot_pos + 3] ← ´␣´;
    end;
  which_file_next ← ´z´; found_change ← true;
  end
```

This code is used in section 115*.

118.* Use all of *fname* for the *style_file_name* if there is a ´.´ in it; otherwise, add ´.mft´.

```
⟨ Get style_file_name from fname 118* ⟩ ≡
  begin dot_pos ← −1; slash_pos ← −1; i ← 1;
  while (fname[i] ≠ ´␣´) ∧ (i ≤ FILENAMESIZE − 5) do
    begin style_file_name[i] ← fname[i];
    if fname[i] = ´.´ then dot_pos ← i;
    if fname[i] = ´/´ then slash_pos ← i;
    incr(i);
    end;
  if (dot_pos = −1)|(dot_pos < slash_pos) then
    begin dot_pos ← i; style_file_name[dot_pos] ← ´.´; style_file_name[dot_pos + 1] ← ´m´;
    style_file_name[dot_pos + 2] ← ´f´; style_file_name[dot_pos + 3] ← ´t´;
    style_file_name[dot_pos + 4] ← ´␣´;
    end;
  which_file_next ← ´z´; found_style ← true;
  end
```

This code is used in sections 115* and 115*.

119.*

```
⟨ Set up null change file 119* ⟩ ≡
  begin change_file_name[1] ← ´/´; change_file_name[2] ← ´d´; change_file_name[3] ← ´e´;
  change_file_name[4] ← ´v´; change_file_name[5] ← ´/´; change_file_name[6] ← ´n´;
  change_file_name[7] ← ´u´; change_file_name[8] ← ´l´; change_file_name[9] ← ´l´;
  change_file_name[10] ← ´␣´;
  end
```

This code is used in section 115*.

120.*

```
⟨ Set up plain style file 120* ⟩ ≡
  begin style_file_name[1] ← ´p´; style_file_name[2] ← ´l´; style_file_name[3] ← ´a´;
  style_file_name[4] ← ´i´; style_file_name[5] ← ´n´; style_file_name[6] ← ´.´; style_file_name[7] ← ´m´;
  style_file_name[8] ← ´f´; style_file_name[9] ← ´t´; style_file_name[10] ← ´␣´;
  end
```

This code is used in section 115*.

121.* ⟨ Handle flag argument in *fname* 121* ⟩ ≡

```
begin i ← 2;
while (fname[i] ≠ ´␣´) ∧ (i ≤ FILENAMESIZE − 5) do
  begin if fname[i] = ´c´ then
    begin found_change ← false;
    if which_file_next ≠ ´s´ then which_file_next ← ´c´
    end
  else if fname[i] = ´s´ then
      begin found_style ← false;
      if which_file_next ≠ ´c´ then which_file_next ← ´s´
      end
    else print_nl(´Invalid␣flag␣´, xchr[xord[fname[i]]], ´.´);
  incr(i);
  end;
end
```

This code is used in section 115*.

122.* ⟨ Print usage error message and quit 122* ⟩ ≡

```
begin print_ln(´Usage:␣mft␣file[.mf]␣[-cs]␣[change[.ch]]␣[style[.mft]].´); uexit(1);
end
```

This code is used in sections 115* and 115*.

123* Index.

The following sections were changed by the change file: 2, 3, 4, 13, 17, 20, 21, 22, 24, 26, 28, 31, 112, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123.

The MFT processor

(Version 2.0, October 1989)

The preparation of this report was supported in part by the National Science Foundation under grants IST-8201926, MCS-8300984, and CCR-8610181, and by the System Development Foundation.