Notes on reading Adobe font files

by Bill Casselman

Adobe Type 1 font files are special types of PostScript files which are read by their own special PostScript interpreter. There are two major parts to one of these, the first containing the **character encoding** and the second the **character descriptions**. In practice, these files almost always come with two extensions, `.pfa` and `.pfb`. The major difference between these two is that a certain part of the font specification, known as the **binary section**, is stored as raw bytes in `.pfb` files, but as bytes in the form of pairs of hex characters in `.pfa` files. Most fonts are stored on a system in `.pfb` format because these files are about half the size of the equivalent `.pfa` file, but using a Type 1 font in a PostScript program requires translation to the longer `.pfa` format before importing it.

There are several reasons why one might wish to analyze one of them:

- convert a `.pfb` file to a `.pfa` file in order to load the font into a PostScript program;
- extract from one of them the references to only a subset of characters in order to include the shortened font in a PostScript program;
- extract character path descriptions to use in a graphics program that is not necessarily PostScript.

All three of these are relevant to me, since my graphics program PiScript does a lot of manipulation of Type 1 fonts.

My principal references for this note have been Adobe's manual for Type 1 Fonts, and the source code for the `t1utils` package (look at the discussion at the end of this note).

1. The basic structure

A Type 1 font file has three segments. In order:

(a) an initial ascii part
(b) the middle part
(c) a final ascii part

Parts (a) and (c) are always in printable ascii characters, but the middle part (b), which is basically in a binary format, differs in `.pfa` and `.pfb` files. In the `.pfb` format the data in this middle section is stored as raw bytes, whereas in the `.pfa` version these bytes are stored in hexadecimal ascii format. Since raw bytes are not so amenable to parsing, `.pfb` files have in addition certain **segment headers** which specify the lengths of segments in the file. There are four of these.

The binary sections in all Type 1 font files contain the instructions for actually drawing the characters of the font. The entire Type 1 file is created from an original ascii file by two or possibly three stages of encryption. Two of these are open—i.e. the algorithms for encryption and decryption have been published by Adobe Systems. It is not clear to me what the purpose of these top levels of encryption is. A document by Adobe says that they discourage casual inspection, but it is not hard to write a program that deciphers them. I am not aware that fonts are enciphered in a truly secret manner. Certainly many publicly available fonts, in particular all those derived from Donald Knuth's original cm fonts, are completely readable.

It is extremely easy to convert back and forth between `.pfb` and `.pfa` format. A `.pfb` file is not easy to parse, so it contains three segment headers to help navigate. Each begins with a binary segment header of 6 bytes. The first byte is always the hex form 80 of 128. The second tells the type of the header, which is 1 for the initial header. The next four bytes specify the length of the first ascii segment, in low-endian order. Thus the initial segment header of `cmr10.pfb` is (in decimal notation) 128:1:27:14:0:0, which tells that the length of the first segment is $27 + 14 \cdot 256 = 3611$ characters. What this means is that the next header is byte number $6 + 3611 = 3617$ of the file, and that bytes in the range $[6, 3617)$ make up the initial ascii segment. Similarly, there are two more segment headers in the file, each one starting just at the end of the segment marked by the previous one. There is one final last 'header' of two bytes, always $[128 : 3]$, which functions as an end-of-file marker. It is presumably there to flag file corruption.

The corresponding `.pfa` file is easy to parse with standard text tools, and hence has no segment headers. It is exactly the same as the `.pfb` file, except that each byte of the binary (second) segment in a `.pfb` file is expressed as a pair of hex 'digits' "0123456789abcdef" in the `.pfa` file. So to understand these files, we may as well assume we are working with a `.pfa` file.

It will help to have in view an example. Here follow the three segments from `cmr10.pfa`. First the initial ascii segment:

```
%!PS-AdobeFont-1.1:  CMR10 1.00B
%%CreationDate:  1992 Feb 19 19:54:52
% Copyright (C) 1997 American Mathematical Society.    All Rights Reserved.
11 dict begin
/FontInfo 7 dict dup begin
/version (1.00B) readonly def
/Notice (Copyright (C) 1997 American Mathematical Society) readonly def
/FullName (CMR10) readonly def
/FamilyName (Computer Modern) readonly def
/Weight (Medium) readonly def
/ItalicAngle 0 def
/isFixedPitch false def
end readonly def
/FontName /CMR10 def
/PaintType 0 def
/FontType 1 def
/FontMatrix [0.001 0 0 0.001 0 0] readonly def
/Encoding 256 array
0 1 255 {1 index exch /.notdef put} for
dup 161 /Gamma put
dup 162 /Delta put
dup 163 /Theta put
...
dup 0 /Gamma put
...
dup 160 /space put
readonly def
/FontBBox{-251 -250 1009 969}readonly def
/UniqueID 5000793 def
currentdict end
currentfile eexec
```

Next the binary part (which is actually very, very long):

```
8053514d28ec28da1 ...
...  72a7d2c89178fb92f
```

And then the final ascii part:

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
cleartomark
```

The final ascii part of every `.pfa` file looks exactly the same, including 512 zeros mingled with space characters, so all the interesting stuff is in the earlier parts.

2. The first ascii segment

Let's look at the initial ascii segment. The first significant thing is that the file (once converted to `.pfa` format) must begin with `%!`, as all PostScript files do. Next, the second word of the first line is the name of the font as it is used in a PostScript file. Here it is `/CMR10`. To load the file `cmr10.pfa` into a PostScript program and then use the font in it, you would put into the program

```
(cmr10.pfa) run
/CMR10 findfont
```

This leaves the font on the operator stack. I repeat: the `.pfb` font files are not runnable PostScript files, as are the `.pfa` files.

The first ascii segment contains a few PostScript dictionaries, and an **encoding array**. Among the information in the dictionaries is the font's name as it would be used in a PostScript file (so it is specified twice, it seems, in practice). It also contains the `/FontMatrix`, which in this example is `[0.001 0 0 0.001 0 0]`. This PostScript matrix specifies the affine transformation by which coordinates in the character path descriptions are changed to real-world coordinates. The character path coordinates are always specified in integers, but in a very small unit. In this case they must be scaled by $1/1000$ to obtain the coordinates in points.

The font encoding is an array of length equal to the nominal number of characters in the font. the entry at the i-th place in the array is a PostScript name for the character (for example `\Gamma`), by which it is referred to in the character description dictionary. In the example, `/Gamma` corresponds to characters indexed by either 0 or 161. There are many reasons for having this double layer of character references, but among other things it means the encoding can be changed trivially without reading the binary segment. Some, maybe most, fonts use a standard encoding specified in Adobe's manuals, but all the TEX fonts I know of have their own encoding, which is specified explicitly in the font file. In `cmr10`, for example, the space has index 160 rather than the standard 32. In the encoding, at least one character must be undefined, with name `/.notdef`. In the example, this is done by the initialization of the encoding array. Also in this section, the font is usually assigned a `UniqueID`, one which distinguishes this font from all others. Adobe maintains a registry of `UniqueID`s assigned to all publicly available fonts.

There is also in this section a font's bounding box, which tells the dimensions of the smallest box that would contain all the characters in the font. In the example, you can see confirmed that the character coordinates are given in integers roughly in a box of width 260 and height 1219 units. (Remember these numbers are to be scaled by the font matrix before use.)

As I have mentioned at the beginning, one reason for taking apart a Type 1 font file is to extract only what is needed for a subset of characters. In doing this extraction and reassembly, the `/Encoding` section will be abbreviated, and the `/UniqueID` assignment can be removed.

The first ascii segment *always* ends with `currentfile eexec`, which tells the font manipulation program what to do with the next binary section—namely, apply `eexec` to it it.

3. The binary segment

The first step in interpreting the binary segment is to translate it from hex form to an array of bytes—that is to say, an array of integers in the range $[0, 256)$. Thus `8053514d28ec28da1 ...` is read in pairs `80 53 51 4d 28 ec 28 da \dots` and then interpreted as the array $[128, 83, 81, 77, 40, 236, 40, 218, \ldots]$.

This array must be decrypted—turned into a more readable string of bytes. This is to be done with a fixed encryption procedure specified once and for all by Adobe. We start with three constants $R = 55665$, $c_1 = 52845$, $c_2 = 22719$. Then we apply to the entire binary array `c[i]` of length n the decryption procedure:

```
for in [0, n):
    p[i] = c[i]^(R >> 8)
    R = ((c[i] + R)*c1 + c2) & ((1 << 16) - 1)
```

Here ˆ means `xor` addition, in which one interprets the bits modulo 2. The encryption key R changes as the procedure is carried out.

What one gets from `cmr10` after this decryption process has been applied looks like this:

```
YANYdup/Private 16 dict dup begin
/RD{string currentfile exch readstring pop}executeonly def
/ND{noaccess def}executeonly def
/NP{noaccess put}executeonly def
/BlueValues [-22 0 683 705 431 448 666 677] def
...
/Subrs 39 array
dup 0 15 RD R...
P.k......f NP
dup 1 9 RD 0..^?.S[.  NP
dup 2 9 RD 0.(a.._..  NP
...
2 index /CharStrings 130 dict dup begin
/Gamma 97 RD t.....(M.&3.V..i=...
NB.NF....8..b......A.G...R41x....J.,...VTxd.f....T...*c..Ae.......#y%D&...\| ND
...
/.notdef 12 RD tA...^.ev.." ND
/space 12 RD 0..{^.w/0...  ND
end
end
readonly put
noaccess put
dup/FontName get exch definefont pop
mark currentfile closefile
```

The dots, except the ellipses, stand for unprintable binary characters. In other words, after transforming the binary segment into supposedly readable form, we are still faced with some binary data! The next section will explain this.

This segment begins with 4 characters that are meaningless to PostScript, and are discarded by it. Here the string `YANY` probably signifies that these fonts were produced by the now defunct font foundry Y & Y (but, I believe, originally designed for use in Macintosh computers by Bluesky Research). After that follows the `/Private` dictionary and the `/CharString` dictionary. The first defines a number of technical terms involving character construction, and contains also an array of subroutines used in character paths. The second contains the character descriptions themselves. Both the subroutines and the character descriptions are yet again encrypted in a fashion similar to the entire binary segment, but now with an initial value of $R = 4330$ instead of 55665. In order to figure out exactly how characters are drawn these also have to be decrypted, but if all one wants to do is remove unused characters from a font file, it is not necessary. As I have already mentioned, I'll say more in the next section about reading the subroutines and character path descriptions—they involve special variants of the usual PostScript operators.

In order to re-encrypt the binary section, one applies the inverse to decryption:

```
for in [0, n):
    c[i] = p[i]^(R >> 8)
    R = ((c[i] + R)*c1 + c2) & ((1 << 16) - 1)
```

It is not quite obtained by swapping $p[i]$ and $c[i]$.

4. Character strings and subroutines

Both character strings and subroutines are doubly or even in some sense triply encrypted. First of all, the binary sequences that are exposed by decrypting the whole binary section must be decrypted using a very similar procedure to that used on the entire segment, except that the initial value of R is 4330 instead of 55665. But then, what you encounter, instead of a string of PostScript operators and data, but an array of bytes which are a coded form of special Type 1 font operators. Some of these encoding layers are there for compression, but some are presumably remnants of the early days before Adobe had opened up the Type 1 font format to public inspection.

The character strings section begins with a phrase in the form `2 index /CharStrings 130 dict dup begin`, where 130 is the number of strings defined. Each character is represented in the form `/A n RD <binary string> ND` where `/A` is the name specified in the character encoding array and n is the length of the binary part. The words `RD` and `ND` are short forms of PostScript operators defined in the `/Private` dictionary. The dictionary must have an entry for `/.notdef`. The binary part of the character definition must be decrypted first by the CharStrings cipher scheme, but then the bytes you get must be interpreted according to the Type 1 operator byte code, which I now say a few words about.

Each character string, when decrypted, is an array of bytes, which make up a compressed list of integers and operators. The operators are slightly modified versions of the standard PostScript operators such as `rmoveto` and `rlineto`, but including also what Adobe calls 'hints' such as `hstem`. These hints improve the appearance of characters at low resolution.

So for `CMR10` the character `/Gamma`, when decrypted, becomes the string of bytes:

```
100, 101, 114, 110,
172, 249, 5, 13, 139, 170, 1, 249, 29, 170, 1, 242, 228, 3, 248, 158, 166, 3,
248, 157, 249, 60, 21, 252, 157, 6, 108, 7, 163, 6, 216, 141, 128, 103, 31,
252, 160, 7, 103, 137, 128, 62, 30, 115, 6, 108, 7,
174, 142, 217, 139, 178, 139, 8, 180, 139, 230, 139, 175, 136, 8, 170, 7,
106, 6, 44, 139, 152, 174, 31, 248, 166, 7, 172, 141, 146, 186, 30,
242, 6, 247, 35, 139, 160, 80, 155, 251, 27, 8, 164, 6, 9, 14
```

After we remove the first 4 characters, which for reasons beyond my ken spell "dern", then interpret this according to the binary code, and finally make it human-readable, this becomes:

```
33 625 hsbw 0 31 hstem 649 31 hstem 103 89 vstem 522 27 vstem
521 680 moveto -521 hlineto -31 vlineto 24 hlineto 77 2 -11 -36 hvcurveto
-524 vlineto -36 -2 -11 -77 vhcurveto -24 hlineto -31 vlineto
35 3 78 0 39 0 rrcurveto 41 0 91 0 36 -3 rrcurveto 31 vlineto
-33 hlineto -95 0 13 35 hvcurveto 530 vlineto 33 2 7 47 vhcurveto
103 hlineto 143 0 21 -59 16 -135 rrcurveto 25 hlineto closepath endchar
```

The other major component of the binary section is the array of subroutines. They are read exactly as the character strings are, and are occasionally called by commands encountered in the character strings.

For details of the interpretation of character strings and subroutines, see the file `BuildChar.py`.

5. To be done . . .

Other computer fonts are in `.ttf` or `.gsf` format. The first were first developed by Microsoft, I suspect with no other goal in mind except to be different from Adobe. The chief difference is technical—character strings in `.ttf` use quadratic Bezier curves whereas Adobe uses cubic ones. The `.gsf` fonts come with `ghostscript`. The `.ttf` fonts have to be converted to Type 1 format before being used in PostScript programs, but there are many utilities available to do this. The `.gsf` fonts can be read by any PostScript interpreter. I'll say more about both of these, and also discuss how one can assemble a useful Type 1 font collection, in a later version of this document. I'll also say something about font metric files (both Adobe `.afm` and TEX `.tfm`) required to use these elegantly, and also something about how to read the `.dvi` files produced by TEX that use fonts.

6. References

The primary reference is the manual **Adobe Type 1 Font Format**, available at

`http://partners.adobe.com/public/developer/en/font/T1_SPEC.PDF`

There are a number of programs available on the Internet for manipulating Type 1 font files. One common program is `pfbtopfa` and its inverse, which come with the `ghostscript` interpreter at

```
http://pages.cs.wisc.edu/~ghost/
```

but a larger collection of helpful tools is found in the `t1utils` package:

```
http://www.ctan.org/tex-archive/help/Catalogue/entries/t1utils.html
```

The program `t1disasm` is particularly interesting, and its source code is worth scanning.

The Y & Y Type 1 versions of Knuth's CM fonts were contributed for public use to the CTAN project. Information about the company can be found at

```
http://www.microsoft.com/typography/links/vendor.aspx?VID=Y%26Y
```