Coding of Outline Fonts: PFR Specification (Version 1.2)

Introduction

This document defines the Bitstream portable font resource (PFR), which is a compact, platform independent format for representing scalable outline fonts. Several independent organizations responsible for setting digital TV standards have adopted the PFR font format as their de facto, standard font format. These organizations include Digital Audio Visual Council (DAVIC), Digital Video Broadcasting (DVB), and Digital TV Group (DTG). DAVIC sets multimedia standards for international broadcasting. DVB is a Swiss-based industry organization representing one standard for digital TV, which has been adopted extensively in Europe. DTG coordinates standards for digital TV broadcasting in the United Kingdom.

Bitstream is making the PFR font format public to help anyone who wants to adopt these digital TV standards. Bitstream allows unrestricted use of the information in this document to make and use software that renders images from outlines coded in the PFR format, provided such rendering is used with one or more of these digital TV standards.

Using Bitstream PFRMaker, anyone can create fonts in the PFR outline font format from fonts installed on his or her system. PFRMaker is a simple, command-line utility that anyone can evaluate and license from Bitstream.

Revision History

This document is based on and is compatible with the font format specification published by DAVIC as Version 1.4.1, Part 9, Annex A.

Version 1.2, whose changes are highlighted in blue and underlined, includes information on PFRMaker, as stated above. PFRMaker replaces WebFont Wizard. Bitstream no longer sells WebFont Wizard. Unfortunately, we do not build the browsers, nor can we control how they handle fonts, so we can no longer support WebFont Wizard for every release of every browser. We would like to support Netscape Navigator 6 and 7 (and future versions), as well as future versions of Microsoft Internet Explorer, but we cannot. Bitstream suggests you contact Microsoft and AOL and encourage them to support dynamic fonts in the new releases of their browsers. We are sorry about any inconvenience this may cause, but after careful evaluation, we believe that it is only fair to retire this retail product line. We will, however, continue to support PFRs.

Version 1.1 contains editorial updates that are intended to correct typographical errors and to make this specification easier to understand. Two areas, in particular, that were found to be confusing or contained inconsistent information were corrected as follows.

1. offsets. The original specification contained a statement in section 1 to the effect that all offsets except for offsets to glyph program strings were relative to the start of the PFR header. With the addition of support for bitmap representation of glyphs that use offsets relative to other bases, this global statement caused considerable confusion and has therefore been removed. The base reference for each offset is now defined explicitly.

2. Bitmap row order. Several implementors were confused by the vague, and in one case, incorrect drafting of the bitmap image specifications. This has been carefully redrafted. To avoid problems with possibly incorrect implementations of renderers, it is now strongly recommended that the header field pfrInvertBitmap be always set to zero in all new PFRs.

3. Confusion between physical font record and physical font section. A physical font section consists of a physical font record which may, if bitmaps are included, be followed by bitmap character records. The syntax definition of Physical Font Section has been updated to be consistent with the accompanying text.

This document includes the specifications for adding kerning data. This is extracted directly from the DAVIC specification and appears in section 10 of this specification. Both pair and track kerning is supported.

1 Font Format Specification

The scaleable outline format representation will be referred to as a portable font resource (PFR) that can be stored statically in ROM or hard disks, or moved dynamically within a network. This dynamic aspect is the reason the font resource is often referred to as portable. The file representation of the PFR is designed with two, sometimes conflicting, goals in mind. One is to minimize the size of the file representation; the other is to provide the information in a way that optimizes rendering performance even if the amount of memory is limited at playback time.

A Portable Font Resource consists of the following sections in order:

- PFR header
- Logical font directory
- Logical font section
- Physical font section
- Glyph program strings
- PFR trailer

The PFR header contains global information about the PFR and the fonts contained within it.

The logical font directory consists of a table of pointers to the logical fonts contained within the PFR.

The logical font section contains the logical font records themselves. Each logical font record defines the transformation (size, oblique effect, condense, expand) to be applied to a physical font. It therefore represents an instance of a physical font.

The physical font section consists of a set of physical font records. Each physical font record contains information about one physical font contained within the PFR including a table of character codes defined for that physical font. A physical font record may optionally be immediately followed by bitmap size and bitmap character table records associated with that physical font.

The glyph program strings section contains the definition of the shapes of each of the characters defined within the font. Both outline and bitmap image shapes are defined by glyph program strings. Glyph program strings are shared across all physical fonts within a PFR.

All integers are written most-significant bit first.

Many of the concepts used in this specification are based on the Adobe Type 1 font format, version 1.1 (Addison-Wesley Publishing Company, Inc., 1991).

2 PFR Header

The PFR header is the first block of data in a Portable Font Resource. It contains global information about the PFR and its constituent fonts.

The size of the PFR header is a fixed 58 bytes. Its structure is as follows:

Table A-1. PFR Header

Syntax	*Number of bits*	*Mnemonic*
pfrHeader() {		
pfrHeaderSig	32	bslbf
pfrVersion	16	uimsbf
pfrHeaderSig2	16	bslbf
pfrHeaderSize	16	uimsbf
logFontDirSize	16	uimsbf
logFontDirOffset	16	uimsbf
logFontMaxSize	16	uimsbf
logFontSectionSize	24	uimsbf
logFontSectionOffset	24	uimsbf
physFontMaxSize	16	uimsbf
physFontSectionSize	24	uimsbf
physFontSectionOffset	24	uimsbf
gpsMaxSize	16	uimsbf
gpsSectionSize	24	uimsbf
gpsSectionOffset	24	uimsbf
maxBlueValues	8	uimsbf
maxXorus	8	uimsbf
maxYorus	8	uimsbf
physFontMaxSizeHighByte	8	uimsbf
zeros	6	bslbf
pfrInvertBitmap	1	bslbf
pfrBlackPixel	1	bslbf
bctMaxSize	24	uimsbf
bctSetMaxSize	24	uimsbf
pftBctSetMaxSize	24	uimsbf
nPhysFonts	16	uimsbf
maxStemSnapVsize	8	uimsbf
maxStemSnapHsize	8	uimsbf
maxChars	16	uimsbf
}		

pfrHeaderSig: A byte string which indicates the file type and format. This field shall be set to the constant value 0x50465230 representing the ASCII string “PFR0”.

pfrVersion: An unsigned integer indicating the PFR format version number. This field shall be set to the constant value 4.

pfrHeaderSig2: A byte string which further confirms the integrity of the PFR. This field shall be set to the constant value 0x0d0a representing the ASCII characters carriage return and line feed.

pfrHeaderSize: An unsigned integer indicating the number of bytes in the PFR header. This field shall be set to the constant value 58.

logFontDirSize: An unsigned integer indicating the total size of the logical font directory in bytes.

logFontDirOffset: An unsigned integer indicating the byte offset of the first byte of the logical font directory relative to the first byte of the PFR header.

logFontMaxSize: An unsigned integer indicating the size in bytes of the largest logical font record. This may be used to allocate a buffer capable of holding any logical font record.

logFontSectionSize: An unsigned integer indicating the size in bytes of the entire set of logical font records. This may be used to allocate a buffer capable of holding the entire logical font section.

logFontSectionOffset: An unsigned integer indicating the byte offset of the first byte of the first logical font record in the logical font section. The offset is relative to the first byte of the PFR header.

physFontMaxSize: An unsigned integer indicating the size in bytes of the largest physical font record. This may be used to allocate a buffer capable of holding any physical font record.

physFontSectionSize: An unsigned integer indicating the size in bytes of the entire set of physical font record including any bitmap character tables that may follow each physical font record. This may be used to allocate a buffer capable of holding the entire physical font section (including any bitmap character tables that might follow each physical font record).

physFontSectionOffset: An unsigned integer indicating the byte offset of the first byte of the first physical font in the physical font section. The offset is relative to the first byte of the PFR header.

gpsMaxSize: An unsigned integer indicating the size in bytes of the largest glyph program string. In the case of compound glyphs, the size must include the total size of its component glyphs. This may be used to allocate a buffer capable of holding any glyph program string.

gpsSectionSize: An unsigned integer indicating the size in bytes of the entire set of glyph program strings. This may be used to allocate a buffer capable of holding the entire set of glyph program strings.

gpsSectionOffset: An unsigned integer indicating the byte offset to the first byte of the first glyph program string in the glyph program string section. The offset is relative to the first byte of the PFR header.

maxBlueValues: An unsigned integer indicating the maximum number of vertical alignment zones defined in any physical font record.

maxXorus: An unsigned integer indicating the maximum number of controlled X coordinates in any glyph program string. The number of controlled X coordinates in a glyph program string includes primary stroke edges, secondary stroke edges and secondary edges.

maxYorus: An unsigned integer indicating the maximum number of controlled Y coordinates in any glyph program string. The number of controlled Y coordinates in a glyph program string includes primary stroke edges, secondary stroke edges and secondary edges.

physFontMaxSizeHighByte: An unsigned number indicating the number of times 65536 should be added to the specified value of physFontMaxSize. This provides a means of handling physical font records whose size exceeds 64K bytes.

zeros: A concatenation of bits that shall all be set to zero.

pfrInvertBitmap: A bit flag that indicates, if set, that the row order of image data in bitmap glyph program strings is top-to-bottom rather than the standard bottom-to-top row order. It is strongly recommended that this field be set to zero and that image data in bitmap glyph program strings be always stored in the standard botton-to-top row order.

pfrBlackPixel: A bit flag that indicates the bit value used to represent black in image data contained in bitmap glyph program strings.

bctMaxSize: An unsigned integer that indicates the maximum size in bytes of any bitmap character table in any physical font record. This may be used to allocate a buffer capable of holding any bitmap character table.

bctSetMaxSize: An unsigned integer that indicates the maximum size in bytes of any complete set of bitmap character tables in any physical font. This may be used to allocate a buffer capable of holding the set of bitmap character tables for any physical font.

pftBctSetMaxSize: An unsigned integer that indicates the maximum size in bytes of any physical font record together with all of the bitmap character tables associated with it. This may be used to allocate a buffer capable of holding any physical font record and its associated bitmap character tables.

nPhysFonts: An unsigned integer that indicates the number of physical font records in the physical font section.

maxStemSnapVsize: An unsigned integer that indicates the number of values contained in the largest vertical stem snap table in any physical font record.

maxStemSnapHsize: An unsigned integer that indicates the number of values contained in the largest horizontal stem snap table in any physical font record.

maxChars: An unsigned integer that indicates the maximum number of characters in any of the physical font records.

3 Logical font directory

The logical font directory consists of a table of pointers to all of the logical font records contained with the PFR. The table is indexed by the logical font code. The structure of the logical font directory is as follows.

Table A-2. Logical Font Directory

Syntax	*Number of bits*	*Mnemonic*
logFontDir() {		
nLogFonts	16	uimsbf
for (i = 0; i < nLogFonts; i++){		
logFontSize	16	uimsbf
logFontOffset	24	uimsbf
}		
}		

nLogFonts: An unsigned integer indicating the total number of logical fonts contained in the logical font directory.

logFontSize: An unsigned integer indicating the size in bytes of one logical font record.

logFontOffset: An unsigned integer indicating the byte offset to the first byte of that logical font record. The offset is relative to the first byte of the PFR header.

4 Logical font section

The logical font section consists of zero or more logical font records. Each logical record contains information about one logical font. It is accessed via the logFontOffset value in the appropriate entry in the logical font directory. The structure of the logical font section is as follows.

Table A-3. Logical Font Section

Syntax	*Number of bits*	*Mnemonic*
logFontSection() {		
for (i = 0; i < nLogFonts; i++){		
logFontRecord()		
}		
}		

The structure of each logical font record is as follows.

Table A-4. Logical Font Record

Syntax	*Number of bits*	*Mnemonic*
logFontRecord() {		
fontMatrix[0]	24	tcimsbf
fontMatrix[1]	24	tcimsbf
fontMatrix[2]	24	tcimsbf
fontMatrix[3]	24	tcimsbf
zero	1	bslbf
extraItemsPresent	1	bslbf
twoByteBoldThicknessValue	1	bslbf
boldFlag	1	bslbf
twoByteStrokeThicknessValue	1	bslbf
strokeFlag	1	bslbf
lineJoinType	2	bslbf
if (strokeFlag){		
if (twoByteStrokeThicknessValue)		
strokeThickness	16	tcimsbf
else		
strokeThickness	8	uimsbf
if (lineJoinType == MITERLINEJOIN)		
miterLimit	24	tcimsbf
}		
else if (boldFlag){		
if (twoByteBoldThicknessValue)		
boldThickness	16	tcimsbf
else		
boldThickness	8	uimsbf
}		
if (extraItemsPresent){		
nExtraItems	8	uimsbf
for (i = 0; i < nExtraItems; i++){		
extraItemSize	8	uimsbf
extraItemType	8	uimsbf
for (j = 0; j < extraItemSize; j++){		
extraItemData	8	uimsbf
}		
}		
}		
physFontSize	16	uimsbf
physFontOffset	24	uimsbf
if (pfrHeader.physFontMaxSizeHighByte){		
physFontSizeIncrement	8	uimsbf
}		
}		

fontMatrix[]: An array of four signed integers representing the coefficients of the transformation matrix (in units of 1/256) from the font's coordinate system to the document coordinate system. The defined transformation is:

x' = (fontMatrix[0] * x + fontMatrix[2] * y) / (256 * outlineResolution)

y' = (fontMatrix[1] * x + fontMatrix[3] * y) / (256 * outlineResolution)

where (x, y) is a point in the character outline resolution unit coordinate system, outlineResolution is the resolution of the associated physical font and (x', y') is the corresponding point in the (scaled) logical font.

zero: A bit flag that shall be set to zero.

extraItemsPresent: A bit flag that indicates that the logical font contains extra data items. This should be set to zero for the current version as no extra item types are defined for logical font records.

twoByteBoldThicknessValue: A bit flag that indicates that the bold thickness value is expressed as a signed 16-bit integer rather than as an unsigned 8-bit integer.

boldFlag: A bit flag that indicates that emboldening should be enabled when rendering this logical font.

twoByteStrokeThicknessValue: A bit flag that indicates that the stroke thickness value is expressed as a signed 16-bit integer rather than as an unsigned 8-bit integer.

strokeFlag: A bit flag that indicates that this logical font should be rendered by drawing a stroke with the specified thickness around the outline rather than by conventionally filling the interior of the outline. Note that if this flag is set, it overrides the bold flag.

lineJoinType: A two-bit field that indicates how convex corners should be handled during stroked rendering. Its values are the standard PostScript definitions:

0: MITER_LINE_JOIN

1: ROUND_LINE_JOIN

2: BEVEL_LINE_JOIN

3: Undefined

strokeThickness: This is a signed integer that indicates the thickness of the stroke to be used to render the character in stroke mode. The units are character coordinates (outline resolution units). If twoByteStrokeThicknessValue is equal to zero, this value is represented by an unsigned 8-bit field. If twoByteStrokeThicknessValue is equal to one, this value is represented by an signed 16-bit field. The effect of using a negative value for stroke thickness is undefined.

miterLimit: A signed integer representing the limit beyond which mitered corners should be rendered as beveled corners. It is represented by the standard PostScript value for miterLimit. The value represents the maximum value of the miter ratio for any mitered corner in units of 1/65536. The miter ratio is the distance of the mitered corner from the outline corner divided by half the bold or stroke thickness.

boldThickness: This is a signed integer that indicates the total amount by which rendered characters should be emboldened. The units are character coordinates (outline resolution units). Thus, for example, a 100 by 200 square emboldened by 10 units would be rendered as if the square were 110 by 210 character units. If twoByteBoldThicknessValue is equal to zero, this value is represented by an unsigned 8-bit field. If twoByteBoldThicknessValue is equal to one, this value is represented by an signed 16-bit field. A negative value for boldThickness may be used to specify a reduced boldness for a character. This should be used with caution as an excessively negative value for boldThickness can cause thin parts of a character shape to turn inside out.

nExtraItems: An unsigned integer that indicates the number of extra data items present. Extra data items added to a logical font contain data that will be ignored by earlier versions of the PFR interpreter and used by later versions of the PFR interpreter. This field is not used in the current version.

extraItemSize: An unsigned integer indicating the size in bytes of one extra data item. The size includes only the extra item data following the extraItemType field. This field is not used in the current version

extraItemType: An unsigned integer indicating the type of extra item data present. No extra data item types have been defined for logical font records at this time.

extraItemData: One byte of extra item data. This data is interpreted in accordance with the extraItemType defined for logical font records. All undefined extra item types will be ignored. This field is not used in the current version

physFontSize: An unsigned integer that defines the size in bytes of the associated physical font record.

physFontOffset: An unsigned integer that defines the offset in bytes of the first byte of the associated physical font record. The offset is relative to the first byte of the PFR header.

physFontSizeIncrement: An unsigned integer that allows physical font sizes in excess of 64K bytes to be supported. If the physFontMaxSizeHighByte field in the PFR header is non-zero, the value of physFontSizeIncrement is multiplied by 65535 and added to physFontSize. In other words, it provides a high byte for the physical font size.

5 Physical font section

The physical font section contains information about each of the physical fonts contained in the PFR. Each physical font is represented by a physical font record containing information about one physical font. It is accessed via the physFontOffset value in the parent logical font record. In the case that bitmaps are associated with a physical font, the bitmap size records and bitmap character tables form part of the physical font section and should appear immediately following the parent physical font record. The structure of the physical font section is as follows.

Table A-5. Physical Font Section

Syntax	*Number of bits*	*Mnemonic*
physFontSection() {		
for (i = 0; i < nPhysFonts; i++)		
{		
physFontRecord()		
for (j = 0; j < nBitmapSizes; j++) {		
for (k = 0; k < nBmapChars; k++) {		
bmapCharRecord();		
}		
}		
}		
}		

The structure of each physical font record in the physical font section is as follows.

Table A-6. Physical Font Record

Syntax	*Number of bits*	*Mnemonic*
physFontRecord() {		
fontRefNumber	16	uimsbf
outlineResolution	16	uimsbf
metricsResolution	16	uimsbf
xMin	16	tcimsbf
yMin	16	tcimsbf
xMax	16	tcimsbf
yMax	16	tcimsbf
extraItemsPresent	1	bslbf
zero	1	bslbf
threeByteGpsOffset	1	bslbf
twoByteGpsSize	1	bslbf
asciiCodeSpecified	1	bslbf
proportionalEscapement	1	bslbf
twoByteCharCode	1	bslbf
verticalEscapement	1	bslbf
if (!proportionalEscapement)		
standardSetWidth	16	tcimsbf
if (extraItemsPresent){		

```
            nExtraItems                                     8           uimsbf
            for (i = 0; i < nExtraItems; i++){
                extraItemSize                               8           uimsbf
                extraItemType                               8           uimsbf
                switch(extraItemType){
                case 1:
                    bitmapInfo()
                    break;
                case 2:
                    fontID()
                    break;
                case 3:
                    stemSnapTables()
                    break;
                default:
                    for (j = 0; j < extraItemSize; j++){
                    extraItemData                           8           uimsbf
                       }
                    break;
                    }
                }
            }
        nAuxBytes                                          24           uimsbf
        for(I=0; I<nAuxBytes; I++)
            {
            auxData                                         8           uimsbf
            }
        nBlueValues                                         8           uimsbf
        for(i = 0; i < nBlueValues; i++)
            {
            blueValue[i]                                   16          tcimsbf
            }
        blueFuzz                                            8           uimsbf
        blueScale                                           8           uimsbf
        stdVW                                              16           uimsbf
        stdHW                                              16           uimsbf
        nCharacters                                        16           uimsbf
        for (i = 0; i < nCharacters; i++)
            {
            charRecord()
            }
```

fontRefNumber: An unsigned integer that uniquely defines the physical font record within the PFR. Conventionally, physical fonts are numbered in sequence starting at 0.

outlineResolution: A signed integer that defines the resolution of the coordinate system of the character outlines. The value represents the number of units in one em.

metricsResolution; A signed integer that defines the resolution of the coordinate system of the character set width values. The value represents the number of units in one em.

xMin: A signed integer whose value is the smallest value of any X-coordinate of any point in the outline representation of any character in the physical font.

yMin: A signed integer whose value is the smallest value of any Y-coordinate of any point in the outline representation of any character in the physical font.

xMax: A signed integer whose value is the largest value of any X-coordinate of any point in the outline representation of any character in the physical font.

yMax: A signed integer whose value is the largest value of any Y-coordinate of any point in the outline representation of any character in the physical font.

extraItemsPresent: A bit flag that indicates that the physical font contains extra data items. This field is normally set to one because the fontID extra item is required to be present.

zero: A bit flag that shall be set to zero.

threeByteGpsOffset: A bit flag that indicates that the value of gpsOffset is represented by a three-byte rather than by a two-byte integer.

twoByteGpsSize: A bit flag that indicates that the value of gpsSize is represented by a two-byte rather than by a single-byte integer.

asciiCodeSpecified: Obsolete; set to zero.

proportionalEscapement: A bit flag that indicates that the set width is specified for each character rather than for all characters.

twoByteCharCode: A bit flag that indicates that the value of charCode is represented by a two-byte rather than by a single byte integer.

verticalEscapement: A bit flag that indicates that set width value should be interpreted as a vertical rather than horizontal escapement value.

standardSetWidth: A signed integer whose value is the set width of all characters in the font.

nExtraItems: An unsigned integer that indicates the number of extra data items present. Extra data items added to a logical font contain data that will be ignored by earlier versions of the PFR interpreter and used by later versions of the PFR interpreter. This field normally has a value of at least one because the fontID extra item is required in the current version.

extraItemSize: An unsigned integer indicating the size in bytes of one extra data item. The size includes only the extra item data following the extraItemType field.

extraItemType: An unsigned integer indicating the type of extra item data present. Three extra data item types (values 1 - 3) have been defined for physical font records at this time.

extraItemData: One byte of extra item data. This data is interpreted in accordance with the values of extraItemType defined for physical font records. All undefined extra item types will be ignored.

nAuxBytes: An unsigned integer defining the number of bytes of auxiliary data that follow.

auxData: nAuxBytes bytes of arbitrary data.

nBlueValues: An unsigned integer defining the number of vertical alignment edges. The number of alignment edges shall always be an even number.

blueValue: A signed integer defining the Y-coordinate of one edge of a blue zone. The contained blue values must be in ascending order. Each succeeding pair of blue values defines one blue zone. See the Adobe Type 1 Font Format specification for details on the effect of defining blue zones.

blueFuzz: An unsigned integer defining the value of blueFuzz. See the Adobe Type 1 Font Format specification for details on the effect of this value.

blueScale: An unsigned integer defining the value of blueScale See the Adobe Type 1 Font Format specification for details on the effect of this value.

stdVW: An unsigned integer defining the value of stdVW See the Adobe Type 1 Font Format specification for details on the effect of this value.

stdHW: An unsigned integer defining the value of stdHW See the Adobe Type 1 Font Format specification for details on the effect of this value.

nCharacters: An unsigned integer defining the number of character records present. Character records following this field must be in ascending order of charCode.

The format of each character record is as follows.

Table A-7. Character Record

Syntax	*Number of bits*	*Mnemonic*
charRecord() {		
if (twoByteCharCode)		
charCode	16	uimsbf
else		
charCode	8	uimsbf
if (proportionalEscapement)		
charSetWidth	16	tcimsbf
if (asciiCodeSpecified)		
asciiCodeValue	8	uimsbf
if (twoByteGpsSize)		
gpsSize	16	uimsbf
else		
gpsSize	8	uimsbf
if (threeByteGpsOffset)		
gpsOffset	24	uimsbf
else		
gpsOffset	16	uimsbf
}		

charCode: An unsigned integer defining the character code value.

charSetWidth: A signed integer defining the set width of the character. This determines the horizontal or vertical distance from the origin of the current character to the origin of the immediately following character.

asciiCodeValue: This field if present should be ignored.

gpsSize: An unsigned integer indicating the size in bytes of the glyph program string containing the outline representation of the character.

gpsOffset: An unsigned integer indicating the byte offset of the first byte of the glyph program string containing the outline representation of the character. The offset is relative to the first byte of the first glyph program string in the glyph program string section of the PFR.

5.1 Bitmap Information

Optional bitmap information, contained in one or more extra data items in a physical font record, associates a set of bitmap character tables with that physical font record. These bitmap character tables must be written immediately following the parent physical font record. As these bitmap character tables are individually accessed via bitmap size records their order is arbitrary. Each bitmap character table contains bitmap character records for a single bitmap size. Bitmap size is measured in pixels per em. The horizontal size of a bitmap image may be different from its vertical size. The bitmap information record consists of a header followed by one or more bitmap size records. The structure of the bitmap information record is as follows.

Table A-8. Bitmap Information Extra Data Item

Syntax	*Number of bits*	*Mnemonic*
`bitmapInfo() {`		
`  fontBctSize`	`24`	`tcimsbf`
`  zeros`	`3`	`bslbf`
`  twoByteNBmapChars`	`1`	`bslbf`
`  threeByteBctOffset`	`1`	`bslbf`
`  threeByteBctSize`	`1`	`bslbf`
`  twoByteYppm`	`1`	`bslbf`
`  twoByteXppm`	`1`	`bslbf`
`  nBitmapSizes`	`8`	`uimsbf`
`  for (i = 0; i < nBitmapSizes; i++){`		
`  bmapSizeRecord()`		
`  }`		
`}`		

fontBctSize: A signed integer that represents the total size in bytes of all bitmap character tables associated with this physical font record. Note that if there are multiple bitmap information records associated with a physical font record, all must have the same value of fontBctSize.

zeros: A concatenation of bits that shall all be set to zero.

twoByteNBmapChars: A bit flag that is set to indicate that the nBmapChars field in each bitmap size record within this bitmap information record is represented by two bytes rather than by a single byte field.

threeByteBctOffset: A bit flag that is set to indicate that the bctOffset field in each bitmap size record within this bitmap information record is represented by three bytes rather than by a two byte field.

threeByteBctSize: A bit flag that is set to indicate that the bctSize field in each bitmap size record within this bitmap information record is represented by three bytes rather than by a two byte field.

twoByteYppm: A bit flag that is set to indicate that the yppm field in each bitmap size record within this bitmap information record is represented by two bytes rather than by a single byte field.

twoByteXppm: A bit flag that is set to indicate that the xppm field in each bitmap size record within this bitmap information record is represented by a two bytes rather than by a single byte field.

nBitmapSizes: An unsigned integer indicating the number of bitmap size records that appear in the remainder of the bitmap information record.

The number of bitmap size records that can fit in one extra data item is limited by the 256 byte limit on the total size of any extra data item. Multiple extra data items may be used to get around this limitation. Each bitmap size record contains

information about one bitmap character table. Within each extra data item, bitmap size records must be in ascending order of Y pixels per em (X pixels per em is a secondary sort key in the event of duplicate values of Y pixels per em). The format of each bitmap size record is as follows.

Table A-9. Bitmap Size Record

Syntax	*Number of bits*	*Mnemonic*
bmapSizeRecord() {		
if (twoByteXppm)		
xppm	16	uimsbf
else		
xppm	8	uimsbf
if (twoByteYppm)		
yppm	16	uimsbf
else		
yppm	8	uimsbf
zeros	5	bslbf
threeByteGpsOffset	1	bslbf
twoByteGpsSize	1	bslbf
twoByteCharCode	1	bslbf
if (threeByteBctSize)		
bctSize	24	uimsbf
else		
bctsize	16	uimsbf
if (threeByteBctOffset)		
bctOffset	24	uimsbf
else		
bctOffset	16	uimsbf
if (twoByteNBmapChars)		
nBmapChars	16	uimsbf
else		
nBmapChars	8	uimsbf
}		

xppm: An unsigned integer that represents the number of pixels per em in the X dimension

yppm: An unsigned integer that represents the number of pixels per em in the Y dimension

zeros: A concatenation of bits that shall all be set to zero.

threeByteGpsOffset: A bit flag that is set to indicate that the value of gpsOffset is represented by a three-byte rather than by a two-byte integer.

twoByteGpsSize: A bit flag that is set to indicate that the value of gpsSize is represented by a two-byte integer rather than by a single-byte integer.

twoByteCharCode: A bit flag that is set to indicate that the value of charCode is represented by a two-byte flag rather than by a single-byte flag.

bctSize: An unsigned integer that represents the total size in bytes of the bitmap character table for the specified values of xppm and yppm.

bctOffset: An unsigned integer that represents the offset in bytes of the first byte of the bitmap character table for the specified values of xppm and yppm. The offset is relative to the byte immediately after the parent physical font record.

nBmapChars: An unsigned integer indicating the number of bitmap character records provided in the bitmap character table.

5.2 Font ID

The font ID provides a means of uniquely identifying the physical font. It is structured as a type 2 extra data item for physical font records. Its data consists of 1 to 254 non-null bytes followed by a null byte. This extra data item must be present.

The structure of the fontID record is as follows:

Table A-10. Font ID Extra Data Item

Syntax	*Number of bits*	*Mnemonic*

`fontID() {`
`  for (i = 0; ; i++){`
`  character[i]`
`  if (character[i] == 0)`
`  break`
`  }`
`  }`
	8	uimsbf

character[]: An unsigned integer representing each character in the name of the physical font. Although the preferred coding system is ASCII, any 8-bit coding system can be used as long as it is consistent with the manner in which the font is referred to.

5.3 Stem snap tables

Stem snap tables may be specified to enhance stem weight consistency during rendering by providing values of secondary stem weights (other than the primary values of stdVW and stdHW) which can be used for dynamic stem weight regularization. See the Adobe Type 1 font format specification for details on the behavior of stem snap tables. The vertical stem snap table contains zero or more values of vertical stem sizes measured in character outline resolution units. The horizontal stem snap table contains zero or more values of horizontal stem sizes measured in character outline resolution units. Stem snap tables are structured as a type 3 extra data item for a physical font. The format of the stem snap table data is as follows.

Table A-11. Stem Snap Tables

Syntax	*Number of bits*	*Mnemonic*
`stemSnapTables() {`		
`  sshSize`	`4`	`uimsbf`
`  ssvSize`	`4`	`uimsbf`
`  for (i = 0; i < ssvSize; i++){`		
`  stemSnapV[i]`	`16`	`tcimsbf`
`  }`		
`  for (i = 0; i < sshSize; i++){`		
`  stemSnapH[i]`	`16`	`tcimsbf`
`  }`		
`  }`		

The values in each of the stem snap tables (vertical and horizontal) must be in ascending order.

sshSize: An unsigned integer indicating the number of horizontal stem snap values provided.

ssvSize: An unsigned integer indicating the number of vertical stem snap values provided.

stemSnapV: A signed integer representing one secondary vertical stem weight in character outline units.

stemSnapH: A signed integer representing one secondary horizontal stem weight in character outline units.

5.4 Bitmap character tables

Bitmap character tables provide a means of finding an optional bitmap image associated with a character code. A bitmap character table consists of a list of character codes and for each character code a pointer to the bitmap glyph program string containing the character image.

Bitmap character tables are written immediately following the physical font record with which they are associated. Each bitmap character table consists of one or more bitmap character records arranged in increasing order of character code.

The format of each bitmap character record in a bitmap character table is as follows.

Table A-12. Bitmap Character Record

Syntax	*Number of bits*	*Mnemonic*
`bmapCharRecord() {`		
`  if (twoByteCharCode)`		
`  charCode`	16	uimsbf
`  else`		
`  charCode`	8	uimsbf
`  if (twoByteGpsSize)`		
`  gpsSize`	16	uimsbf
`  else`		
`  gpsSize`	8	uimsbf
`  if (ThreeByteGpsOffset)`		
`  gpsOffset`	24	uimsbf
`  else`		
`  gpsOffset`	16	uimsbf
`  }`		

Note that because twoByteCharCode, twoByteGpsSize and ThreeByteGpsOffset are defined in the parent bitmap size record, they apply to all bitmap character records in a given bitmap character table. This ensures that the size in bytes of every record in a bitmap character table is the same.

charCode: An unsigned integer defining the bitmap character code value.

gpsSize: An unsigned integer indicating the size of the glyph program string containing the bitmap image of the character.

gpsOffset: An unsigned integer indicating the byte offset of the first byte of the glyph program string containing the bitmap image of the character. The offset is relative to the first byte of the first glyph program string in the glyph program string section of the PFR.

6 Glyph program strings

Glyph program strings define character shapes and images. There are three kinds of glyph program strings supported:

- Simple glyph program strings that encode a scaleable glyph shape
- Compound glyph program strings that define a scaleable glyph in terms of one or more simple glyph program strings.
- Bitmap glyph program strings that encode a bitmap image.

7 Simple glyph program strings

A simple glyph program string defines the shape of a character as zero or more outline contours. The points defining the outline are in character outline resolution units based on the value of outlineResolution in the parent physical font record. The structure of a glyph program string is as follows.

Table A-13. Simple Glyph Program String

Syntax	*Number of bits*	*Mnemonic*
simpleGps() {		
isCompoundGlyph	1	bslbf
zeros	3	bslbf
extraItemsPresent	1	bslbf
oneByteXYCoordCount	1	bslbf
controlledYCoords	1	bslbf
controlledXCoords	1	bslbf
if (oneByteXYCoordCount)		
{		
nYorus	4	uimsbf
nXorus	4	uimsbf
}		
else		
{		
if (controlledXCoords)		
{		
nXorus	8	uimsbf
}		
if (controlledYCoords)		
{		
nYorus	8	uimsbf
}		
}		
for (i = 0; i < (nXorus + nYorus); i++)		
{		
if (i & 7 == 0)		
{		
twoByteCoord[7]	1	bslbf
twoByteCoord[6]	1	bslbf
twoByteCoord[5]	1	bslbf
twoByteCoord[4]	1	bslbf
twoByteCoord[3]	1	bslbf
twoByteCoord[2]	1	bslbf
twoByteCoord[1]	1	bslbf
twoByteCoord[0]	1	bslbf
}		
if (twoByteCoord[i & 7])		
oruValue	16	tcimsbf
else		
oruValue	8	uimsbf
}		

```
        if (extraItemsPresent)
            {
            nExtraItems                                 8           uimsbf
            for (i = 0; i < nExtraItems; i++){
                extraItemSize                           8           uimsbf
                extraItemType                           8           uimsbf
                switch(extraItemType){
                case 1:
                    secondaryStrokeInfo()
                    break;
                case 2:
                    secondaryEdgeInfo()
                    break;
                default:
                    for (j = 0; j < extraItemSize, j++){
                    extraItemData                       8           uimsbf
                       }
                    break;
                    }
                }
            }
        do
            {
            glyphOutlineRecord()
            } while (!endGlyph)
        }
```

isCompoundGlyph: A bit flag that indicates that the glyph is compound. This flag is always clear for a simple glyph program string.

zeros: A concatenation of bits that shall all be set to zero.

extraItemsPresent: A bit flag that indicates extra data items are present.

oneByteXYCoordCount: A bit flag indicating that the values of nXorus and nYorus are packed into a single byte.

controlledYCoords: A bit flag indicating that there are one or more controlled Y coordinates.

controlledXCoords: A bit flag indicating that there are one or more controlled X coordinates.

nXorus: An unsigned integer indicating the number of controlled X coordinates

nYorus: An unsigned integer indicating the number of controlled Y coordinates.

twoByteCoord[]: A bit flag indicating the format and method of interpreting a controlled coordinate value. If this bit flag is clear, the controlled coordinate value is represented by one byte which is interpreted as an unsigned coordinate value in outline resolution units relative to the preceding coordinate value. In the case of the first controlled coordinate value, the preceding value is deemed to be at the origin. If this bit flag is set, the controlled coordinate value is represented by two bytes

whose integer value is interpreted as an absolute signed coordinate value in outline resolution units.

oruValue: A signed integer representing a controlled coordinate value in X or Y. Controlled coordinate values must be in ascending order within each dimension. Controlled X coordinates are listed first, controlled Y coordinates are listed second.

nExtraItems: An unsigned integer that indicates the number of extra data items present. Extra data items added to a simple glyph program string contain data that will be ignored by earlier versions of the PFR interpreter and used by later versions of the PFR interpreter.

extraItemSize: An unsigned integer indicating the size in bytes of one extra data item. The size includes only the extra item data following the extraItemType field.

extraItemType: An unsigned integer indicating the type of extra item data present. Two extra data item types (values 1 - 2) have been defined for simple glyph program strings at this time.

extraItemData: One byte of extra item data. This data is interpreted in accordance with the values of extraItemType defined for simple glyph program strings. All undefined extra item types will be ignored.

secondaryStrokeInfo(): A block of data representing one or more secondary stroke definitions.

secondaryEdgeInfo(): A block of data representing one or more secondary edge definitions.

glyphOutlineRecord(): A block of data representing one segment of an outline definition.

Four standard types of glyph outline records are defined:

moveTo()

lineTo()

curveTo()

endGlyph()

The first glyph record must be a moveTo record. This defines the start point of the first contour. The shape of a contour is defined by a sequence of lineTo and curveTo records in any order. The outline shape defining a contour should not be self-intersecting. Each successive moveTo record terminates the preceding contour and starts a new one. If the end point of the previous contour is not coincident with its start point, the contour is closed as if a lineTo record back to the start point of the contour had been included. The last contour must be terminated by an endGlyph record.

Glyph outline records make use of the concept of an argument format that defines one of four possible formats for specifying an X or Y coordinate. Argument formats have the following meaning:

Table A-14. X - Coordinate Argument Format

Syntax	*Number of bits*	*Mnemonic*
xArg(xArgFormat) {		
switch (xArgFormat){		
case 0: xIndex	8	uimsbf
break		
case 1: xValue	16	tcimsbf
break		
case 2: xIncrement	8	tcimsbf
break		
case 3:		
}		
}		

xIndex: An unsigned integer representing the index in the controlled X coordinate table at which the value is found. Note that only controlled coordinates representing the edges of primary strokes may be used in this manner.

xValue: A signed integer representing the X coordinate of the point.

xIncrement: A signed integer representing the change in the X coordinate value relative to the X coordinate of the preceding point.

Table A-15. Y - Coordinate Argument Format

Syntax	*Number of bits*	*Mnemonic*
yArg(yArgFormat) {		
switch (yArgFormat){		
case 0: yIndex	8	uimsbf
break		
case 1: yValue	16	tcimsbf
break		
case 2: yIncrement	8	tcimsbf
break		
case 3:		
}		
}		

yIndex: An unsigned integer representing the index in the controlled Y coordinate table at which the value of the Y coordinate may be found. Note that only controlled coordinates representing the edges of primary horizontal strokes may be used in this manner.

yValue: A signed integer representing the Y coordinate of the point.

yIncrement: A signed integer representing the change in the Y coordinate value relative to the Y coordinate of the preceding point.

7.1 MoveTo glyph record

The moveTo glyph record starts a new contour at a specified point.

Its structure is:

Table A-16. Move To Glyph Record

Syntax	*Number of bits*	*Mnemonic*
moveTo() {		
moveOp	3	uimsbf
isOutsideContour	1	bslbf
yArgFormat	2	uimsbf
xArgFormat	2	uimsbf
xArg(xArgFormat)		
yArg(yArgFormat)		
}		

moveOp: An unsigned integer constant with value 2; this field provides, together with the subsequent isOutsideContour field, a unique identification of a moveTo glyph record.

isOutsideContour: A bit flag that is set to indicate that the contour is an outside contour whose direction is counterclockwise

yArgFormat: An unsigned integer that defines the encoding format of a Y coordinate value. In the case of the first move in a glyph program string, the preceding Y coordinate value is deemed to have a value of 0.

xArgFormat: An unsigned integer that defines the encoding format of a X coordinate value. In the case of the first move in a glyph program string, the preceding X coordinate value is deemed to have a value of 0.

xArg: The X coordinate of the start point of the contour as defined above.

yArg: The Y coordinate of the start point of the contour as defined above.

7.2 LineTo glyph record

The lineTo glyph record continues a contour from the current point in a straight line to a specified point. Its structure is as follows.

Table A-17. Line To Glyph Record

Syntax	*Number of bits*	*Mnemonic*
lineTo() {		
lineOp	4	uimsbf
yArgFormat	2	uimsbf
xArgFormat	2	uimsbf
xArg(xArgFormat)		
yArg(yArgFormat)		
}		

lineOp: An unsigned integer constant with value 1

yArgFormat: An unsigned integer that defines the encoding format of a Y coordinate value.

xArgFormat: An unsigned integer that defines the encoding format of a X coordinate value.

xArg: The X coordinate of the end point of the line as defined above.

yArg: The Y coordinate of the end point of the line as defined above.

7.3 CurveTo glyph record

The curveTo glyph record continues a contour from the current point in a curved outline to a specified point. The shape of the intervening curve is a cubic bezier and is defined by a pair of curve control points. Its structure is as follows.

Table A-18. Curve To Glyph Record

Syntax	*Number of bits*	*Mnemonic*
curveTo() {		
curveOp	1	uimsbf
curveDepth	3	uimsbf
y1ArgFormat	2	uimsbf
x1ArgFormat	2	uimsbf
xArg(x1ArgFormat)		
yArg(y1ArgFormat)		
y3ArgFormat	2	uimsbf
x3ArgFormat	2	uimsbf
y2ArgFormat	2	uimsbf
x1ArgFormat	2	uimsbf
xArg(x2ArgFormat)		
yArg(y2ArgFormat)		
xArg(x3ArgFormat)		
yArg(y3ArgFormat)		
}		

curveOp: An unsigned integer constant with value 1; this field provides, together with the subsequent curveDepth field, a unique identification of a curveTo glyph record.

curveDepth: An unsigned integer indicating the number of recursive subdivisions required to result in a polygonal representation with an error less than one half of an outline resolution unit.

y1ArgFormat: An unsigned integer that defines the encoding format of the Y coordinate of the first curve control point.

x1ArgFormat: An unsigned integer that defines the encoding format of the X coordinate of the first curve control point.

xArg(x1ArgFormat): The X coordinate of the first curve control point.

yArg(y1ArgFormat): The Y coordinate of the first curve control point.

y3ArgFormat: An unsigned integer that defines the encoding format of the Y coordinate of the curve end point.

x3ArgFormat: An unsigned integer that defines the encoding format of the X coordinate of the curve end point.

y2ArgFormat: An unsigned integer that defines the encoding format of the Y coordinate of the second curve control point.

x2ArgFormat: An unsigned integer that defines the encoding format of the X coordinate of the second curve control point.

xArg(x2ArgFormat): The X coordinate of the second curve control point.

yArg(y2ArgFormat): The Y coordinate of the second curve control point.

xArg(x3ArgFormat): The X coordinate of the curve end point.

yArg(y3ArgFormat): The Y coordinate of the curve end point.

7.4 EndGlyph record

The endGlyph record terminates a contour with a line back to the start point of the contour. Its structure is as follows.

Table A-19. End Glyph Record

Syntax	*Number of bits*	*Mnemonic*
endGlyph() {		
endGlyphOp	8	uimsbf
}		

endGlyphOp: An unsigned integer constant with value 0

7.5 Short-form glyph records

In addition to these standard glyph outline records, there are several special-case versions to provide more compact representations of common shapes:

7.6 hLineTo glyph record

For horizontal straight lines that end on a X controlled coordinate, the hLineTo glyph outline record is provided. Its structure is as follows.

Table A-20. Horizontal Line To Glyph Outline Record

Syntax	*Number of bits*	*Mnemonic*
hLineTo() {		
hLineOp	4	uimsbf
xIndex	4	uimsbf
}		

hLineop: an unsigned integer constant with value 2.

xIndex: an unsigned integer indicating the index into the table of controlled X coordinates at which the X coordinate of the end point of the line is found. The first entry of this table has index value zero.

7.7 vLineTo glyph record

For vertical straight lines that end on a controlled X coordinate, the vLineTo glyph outline record is provided. Its structure is as follows.

Table A-21. Vertical Line To Glyph Outline Record

Syntax	*Number of bits*	*Mnemonic*
vLineTo() {		
vLineOp	4	uimsbf
yIndex	4	uimsbf
}		

vLineOp: an unsigned integer constant with value 3.

yIndex: an unsigned integer indicating the index into the table of controlled Y coordinates at which the Y coordinate of the end point of the line is found. The first entry of this table has index value zero.

7.8 hvCurveTo glyph record

For curves that start in a horizontal direction and end in a vertical direction along a controlled X coordinate, the hvCurveTo glyph outline record is provided. Its structure is as follows.

Table A-22. Horizontal to Vertical Curve Glyph Outline Record

Syntax	*Number of bits*	*Mnemonic*
hvCurveTo() {		
hvCurveOp	4	uimsbf
zero	1	bslbf
curveDepth	3	uimsbf
x1Increment	8	tcimsbf
xIndex	4	uimsbf
y2Increment	8	tcimsbf
y3Increment	8	tcimsbf
}		

hvCurveOp: an unsigned integer constant with value 6

zero: A bit flag that shall be set to zero.

curveDepth: An unsigned integer indicating the number of recursive subdivisions required to result in a polygonal representation with an error less than one half of an outline resolution unit.

x1Increment: A signed integer representing the X coordinate of the first curve control point relative to the start point of the curve.

xIndex: an unsigned integer indicating the index into the table of controlled X coordinates at which the X coordinate of the second control point and the end point of the curve is found. The first entry of this table has index value zero.

y2Increment: A signed integer representing the Y coordinate of the second curve control point relative to the first curve control point.

y3Increment: A signed integer representing the Y coordinate of the end point of the curve relative to the second curve control point.

7.9 vhCurveTo glyph record

For curves that start in a vertical direction and end in a horizontal direction along a controlled Y coordinate, the vhCurveTo glyph outline record is provided. Its structure is as follows.

Table A-23. Vertical to Horizontal Curve Glyph Outline Record

Syntax	*Number of bits*	*Mnemonic*
vhCurveTo() {		
vhCurveOp	4	uimsbf
zero	1	bslbf
curveDepth	3	uimsbf
y1Increment	8	tcimsbf
x2Increment	8	tcimsbf
yIndex	4	uimsbf
x3Increment	8	tcimsbf
}		

vhCurveOp: an unsigned integer constant with value 7.

zero: A bit flag that shall be set to zero.

curveDepth: An unsigned integer indicating the number of recursive subdivisions required to result in a polygonal representation with an error less than one half of an outline resolution unit.

y1Increment: A signed integer representing the Y coordinate of the first curve control point relative to the start point of the curve.

x2Increment: A signed integer representing the X coordinate of the second curve control point relative to the first curve control point.

yIndex: an unsigned integer indicating the index into the table of controlled Y coordinates at which the Y coordinate of the second control point and the end point of the curve is found.

x3Increment: A signed integer representing the X coordinate of the end point of the curve relative to the second curve control point.

7.10 Secondary stroke definitions

Primary strokes cannot be mutually overlapping. Secondary strokes that overlap primary strokes may be other secondary strokes that are encoded into secondary stroke information. Secondary strokes that overlap a primary stroke are positioned relative to the primary stroke after the primary stroke has been positioned. Secondary stroke information is structured as a type 1 extra data item. The format for a secondary stroke information is as follows.

Table A-24. Secondary Stroke Information Extra Data Item

Syntax	*Number of bits*	*Mnemonic*
secondaryStrokeInfo() {		
nVertSecStrokes	8	uimsbf
for (i = 0; i < nVertSecStrokes; i++){		
leftEdge[i]	16	tcimsbf
rightEdge[i]	16	tcimsbf
}		
nHorizSecStrokes	8	uimsbf
for (i = 0; i < nHorizSecStrokes; i++)		
{		
bottomEdge[i]	16	tcimsbf
topEdge[i]	16	tcimsbf
}		
}		

nVertSecStrokes: An unsigned integer representing the number of secondary vertical strokes defined for the current simple glyph.

leftEdge[]: A signed integer representing the X coordinate of the left edge of a secondary vertical stroke in outline resolution units.

rightEdge[]: A signed integer representing the X coordinate of the right edge of a secondary vertical stroke in outline resolution units.

nHorizSecStrokes: An unsigned integer representing the number of secondary horizontal strokes defined for the current simple glyph.

bottomEdge[]: A signed integer representing the Y coordinate of the lower edge of a secondary horizontal stroke in outline resolution units.

topEdge[]: A signed integer representing the Y coordinate of the upper edge of a secondary horizontal stroke in outline resolution units.

Because the maximum size of a secondary stroke definition item is 255 bytes, the maximum number of secondary strokes that may be defined in one extra data item is 63. Secondary vertical strokes must be in increasing order of their left edge. Secondary horizontal strokes must be in increasing order of their lower edge.

7.11 Secondary edge definitions

When the edge of a stroke is represented by a shallow curve or other irregularity, it is often desirable to straighten the outline at small sizes and low resolutions. A secondary edge may be defined relative to any stroke edge (its parent). At small sizes and low resolutions, the secondary edge is snapped to the position of its parent. This has the effect of squeezing outline points between the parent edge and the secondary edge onto the primary edge thus resulting in a locally straightened outline. Either edge of any primary or secondary stroke may have one or two secondary edges associated with it. Two edges allow the squeezing operation to take place from both sides of the parent edge. A secondary edge is a generalization of the flex mechanism used in Type 1 fonts which is restricted to certain specific curve structures. Secondary edges may be used with any shape that should be flattened at small sizes. Secondary edge information is structured as a type 2 extra data item. The format for secondary edge information is as follows.

Table A-25. Secondary Edge Information Extra Data Item

Syntax	*Number of bits*	*Mnemonic*
`secondaryEdgeInfo() {`		
`  nVertSecEdges`	`8`	`uimsbf`
`  for (i = 0; i < nVertSecEdges; i++){`		
`  secEdgeDef()`		
`  }`		
`  nHorizSecEdges`	`8`	`uimsbf`
`  for (i = 0; i < nHorizSecEdges; i++){`		
`  secEdgeDef()`		
`  }`		
`  }`		

nVertSecEdges: An unsigned integer indicating the number of vertical secondary edge definitions provided.

nHorizSecEdges: An unsigned integer indicating the number of horizontal secondary edge definitions provided.

The briefest format for a secondary edge definition (either horizontal or vertical) is as follows.

Table A-26. Simplified Secondary Edge Definition

Syntax	*Number of bits*	*Mnemonic*
`secEdgeDef() {`		
`  secEdgeFormat`	`1`	`bslbf`
`  deltaIndex`	`3`	`uimsbf`
`  deltaOrus`	`4`	`tcimsbf`
`  }`		

secEdgeFormat: A bit flag with a constant value of 0

deltaIndex: An unsigned integer in the range 0 to 7 representing the index of the parent edge relative to the index of the parent edge of the immediately preceding secondary edge. In the case of the first edge in each dimension, deltaIndex is interpreted absolutely as the index of the parent edge.

deltaOrus: A signed integer in the range -8 to +7 representing the position of the secondary edge relative to its parent edge in units of character outline resolution units.

In this format, a standard secondary edge snap threshold of 1 pixel is assumed.

A more general (and longer) format for a secondary edge definition is as follows.

Table A-27. General Secondary Edge Definition

Syntax	*Number of bits*	*Mnemonic*
`secEdgeDef() {`		
`  secEdgeFormat`	`1`	`bslbf`
`  threshFlag`	`1`	`bslbf`
`  index`	`6`	`uimsbf`
`  if (threshFlag == 0)`		
`    thresh`	`8`	`uimsbf`
`  deltaOrus`	`8`	`tcimsbf`
`  if (deltaOrus == 0)`		
`    deltaOrus`	`16`	`tcimsbf`
`  }`		

secEdgeFormat: A bit flag with a constant value of 1

threshFlag: A bit flag that indicates how the threshold value is represented. If set, a standard threshold value of 1 pixel is assumed. If clear, an explicit value is provided.

index: An unsigned integer in the range 0 to 63. A value of zero indicates that the index of the parent coordinate is explicitly specified. Any other value indicates that the index of the parent coordinate is index - 1.

thresh: An unsigned integer representing the threshold at which the secondary edge should be snapped to its parent. The units are 1/16 pixel.

deltaOrus: A signed integer representing the position of the secondary edge relative to its parent in units of character outline resolution units.

8 Compound glyph program strings

A compound glyph program string is constructed out of one or more simple glyph program strings. Each of the elements may be independently scaled and positioned in the process of constructing the compound glyph.

The structure of a compound glyph program string is as follows.

Table A-28. Compound Glyph Program String

Syntax	*Number of bits*	*Mnemonic*
compoundGps() {		
isCompoundGlyph	1	bslbf
extraItemsPresent		
1 bslbf		
nElements	6	uimsbf
if (extraItemsPresent)		
{		
nExtraItems	8	uimsbf
for (i = 0; i < nExtraItems; i++)		
{		
extraItemSize	8	uimsbf
extraItemType	8	uimsbf
switch(extraItemType){		
default:		
for (j = 0; j < extraItemSize; j++){		
extraItemData	8	uimsbf
}		
break;		
}		
}		
for (i = 0; i < nElements; i++){		
threeByteGpsOffset	1	bslbf
twoByteGpsSize	1	bslbf
yScalePresent	1	bslbf
xScalePresent	1	bslbf
yPosFormat	2	uimsbf
xPosFormat	2	uimsbf
if (xScalePresent)		
xScale	16	tcimsbf
if (yScalePresent)		
yScale	16	tcimsbf
switch(xPosFormat)		
{		
case 1: xPos	16	tcimsbf
break		
case 2: xPos	8	tcimsbf
break		
}		
switch(yPosFormat)		
{		
case 1: yPos	16	tcimsbf

```
                    break
                case 2: yPos                                8       tcimsbf
                    break
                    }
                if (twoByteGpsSize)
                    gpsSize                                16        uimsbf
                else
                    gpsSize                                 8        uimsbf
                if (threeByteGpsOffset)
                    gpsOffset                              24        uimsbf
                else
                    gpsOffset                              16        uimsbf
                }
            }
        }
```

isCompoundGlyph: A bit flag with a constant value to 1 to indicate that the glyph program string should be interpreted as a compound glyph program string.

extraItemsPresent: A bit flag that indicates extra data items are present. This should be set to zero for the current version.

nElements: An unsigned integer indicating the number of elements in the compound character.

nExtraItems: An unsigned integer that indicates the number of extra data items present. Extra data items added to a compound glyph program string contain data that will be ignored by earlier versions of the PFR interpreter and may be used by later versions of the PFR interpreter. This field is not used in the current version.

extraItemSize: An unsigned integer indicating the size in bytes of one extra data item. The size includes only the extra item data following the extraItemType field. This field is not used in the current version.

extraItemType: An unsigned integer indicating the type of extra item data present. No extra data item types have been defined for compound glyph program strings at this time

extraItemData: One byte of extra item data. This data is interpreted in accordance with the values of extraItemType defined for compound glyph program strings. All undefined extra item types will be ignored.

threeByteGpsOffset: A bit flag that indicates, if set, that the gpsOffset value is defined as a 3-byte integer rather than by 2-byte integer

twoByteGpsSize: A bit flag that indicates, if set, that the value of gpsSize is defined as a 2-byte integer rather than as a single-byte integer.

yScalePresent: A bit flag that indicates, if set, that an explicit value of xScale is defined.

xScalePresent: A bit flag that indicates, if set, that an explicit value of yScale is defined.

yPosFormat: An unsigned integer that indicates how the value of yPos is defined. A value of 1 indicates that it is defined as a 2-byte absolute value; a value of 2 indicates that it is defined as a single-byte value relative to the previous value of yPos; a value of 3 indicates that it is identical to the previous value of yPos.

xPosFormat: An unsigned integer that indicates how the value of xPos is defined. A value of 1 indicates that it is defined as a 2-byte absolute value; a value of 2 indicates that it is defined as a single-byte value relative to the previous value of xPos; a value of 3 indicates that it is identical to the previous value of xPos.

xScale: A signed integer representing the scale factor to be applied to the glyph element in the X dimension. This field is in units of 1/4096.

yScale: A signed integer representing the scale factor to be applied to the glyph element in the Y dimension. This field is in units of 1/4096.

xPos: A signed integer representing the amount by which the glyph element should be shifted in the X dimension. This field is in units of character outline resolution units.

yPos: A signed integer representing the amount by which the glyph element should be shifted in the Y dimension. This field is in units of character outline resolution units.

gpsSize: An unsigned integer representing the size in bytes of the glyph program string defining the glyph element.

gpsOffset: An unsigned integer representing the byte offset of the first byte of the glyph program string that defines the glyph element. The offset is relative to the first byte of the first glyph program string in the glyph program string section.

8.1 Bitmap glyph program string

A bitmap glyph program string defines the image of a glyph in the form of a bitmap. Its structure is as follows.

Table A-29. Bitmap Glyph Program String

Syntax	*Number of bits*	*Mnemonic*
`bitmapGps() {`		
`  imageFormat`	2	uimsbf
`  escapementFormat`	2	uimsbf
`  sizeFormat`	2	uimsbf
`  positionFormat`	2	uimsbf
`  switch(positionFormat)`		
`  {`		
`  case 0:`		

```
            xPos                                          4       tcimsbf
            yPos                                          4       tcimsbf
            break
        case 1:
            xPos                                          8       tcimsbf
            yPos                                          8       tcimsbf
            break
        case 2:
            xPos                                         16       tcimsbf
            yPos                                         16       tcimsbf
            break
        case 3:
            xPos                                         24       tcimsbf
            yPos                                         24       tcimsbf
            break
            }
        switch(sizeFormat)
            {
        case 0:
            break
        case 1:
            xSize                                         4        uimsbf
            ySize                                         4        uimsbf
            break;
        case 2:
            xSize                                         8        uimsbf
            ySize                                         8        uimsbf
            break;
        case 3:
            xSize                                        16        uimsbf
            ySize                                        16        uimsbf
            break;
            }
        switch(escapementFormat)
            {
        case 0:
            break:
        case 1:
            setWidth                                      8       tcimsbf
            break;
        case 2:
            setWidth                                     16       tcimsbf
            break;
        case 3:
            setWidth                                     24       tcimsbf
            break;
            }
        imageData                                  variable
        }
```

imageFormat: An unsigned integer that indicates how the bitmap image is represented.

A value of 0 indicates that the image is stored directly as a bitmap fully packed with no padding between rows.

A value of 1 indicates that the image is run-length encoded in which each byte specifies the unsigned number of white bits in the most significant 4 bits and the unsigned number of following black bits in the least significant 4 bits. A run of more than 15 bits of the same color is handled by multiple bytes Adjacent rows are encoded together without regard to the end of each row. Trailing white bits must be encoded.

A value of 2 indicates that the image is run-length encoded in which each pair of bytes specifies the unsigned number of white bits in the first byte and the unsigned number of following black bits in the second byte. A run of more than 255 bits of the same color is handled by multiple pairs of bytes. Adjacent rows are encoded together without regard to the end of each row. Trailing white bits must be encoded.

A value of 3 is undefined.

escapementFormat: An unsigned integer that indicates how the escapement value is represented.

A value of 0 indicates that no bitmap escapement data is included and that the linearly scaled outline width should be used without rounding.

A value of 1 indicates that the bitmap escapement is represented by a signed single-byte value in units of whole pixels.

A value of 2 indicates that the bitmap escapement is represented by a signed 2-byte value in units of 1/256 pixels.

A value of 3 indicates that the bitmap escapement is represented by a signed 3-byte value in units of 1/256 pixels.

sizeFormat: An unsigned integer that indicates how the dimensions of the bitmap image are represented.

A value of 0 indicates that that bitmap image is blank and no image data is present.

A value of 1 indicates that the width and the height of the bitmap image are each represented by an unsigned 4-bit value in units of whole pixels.

A value of 2 indicates that the width and the height of the bitmap image are each represented by an unsigned 8-bit value in units of whole pixels.

A value of 3 indicates that the width and the height of the bitmap image are each represented by an unsigned 2-byte value in units of whole pixels.

positionFormat: An unsigned integer that indicates how the (x, y) position of the first pixel in the bitmap image is represented.

A value of 0 indicates that the X and the Y coordinates are each represented by a signed 4-bit value in units of whole pixels.

A value of 1 indicates that the X and the Y coordinates are each represented by a signed single-byte value in units of whole pixels.

A value of 2 indicates that the X and the Y coordinates are each represented by a signed 2-byte value in units of 1/256 pixels.

A value of 3 indicates that the X and the Y coordinates are each represented by a signed 3-byte value in units of 1/256 pixels.

xPos: A signed integer representing the horizontal position of the left edge of the bitmap image relative to the character origin. The units are pixels. A negative value indicates that the left edge of the bitmap image is to the left of the character origin.

yPos: A signed integer representing the vertical position of the bottom edge of the bitmap image relative to the character origin (baseline). The units are pixels. A negative value indicates that the bottom edge of the bitmap image is below the baseline.

xSize: An unsigned integer representing the width of the bitmap image in pixels.

ySize: An unsigned integer representing the height of the bitmap image in pixels.

setWidth: A signed integer representing the distance in pixels (or 1/256 pixels depending upon the value of escapementFormat) the current rendering position should be moved by prior to imaging the next character. If the value of verticalEscapement in the parent physical font record is 1, the direction of the escapement vector is vertical. Otherwise, it is horizontal.

imageData: This data is interpreted depending upon the value of imageFormat.

9 Portable font resource trailer

The PFR trailer block shall be the last block of data in the Portable Font Resource. Its primary use it to facilitate the location of the start of a PFR that ends at the end of a file. Its structure is:

Table A-30. Portable Font Resource Trailer

Syntax	*Number of bits*	*Mnemonic*
`pfrTrailer() {`		
`  pfrSize`	`24`	`uimsbf`
`  pfrTrailerSig`	`40`	`bslbf`
`  }`		

pfrSize: An unsigned integer representing the total size of the PFR in bytes.

pfrTrailerSig: A bit pattern used as a PFR trailer signature. It shall have the constant value 0x2450465224 representing the string “PFR”.

10 Updates for kerning data

Kerning data for a physical font is stored as one or more extra data items attached to the physical font for which the kerning data applies. Track and pair kerning data are stored in separate types of extra data items.

10.1 Pair kerning Data

The format of a pair kerning data block is as follows:

Syntax	*Number of bits*	*Mnemonic*
pairKernData() {		
extraItemSize	8	uimsbf
extraItemType	8	uimsbf
nKernPairs	8	uimsbf
baseAdjustment	16	imsbf
reserved	6	
twoByteAdjValues	1	
twoByteCharCodes	1	
for (i = 0; i < nKernPairs; i++){		
if (twoByteCharCodes) {		
charCode1	16	uimsbf
charCode2	16	uimsbf
}		
else {		
charCode1	8	uimsbf
charCode2	8	uimsbf
}		
if (twoByteAdjustment)		
adjustment	16	imsbf
else		
adjustment	8	uimsbf
}		
}		

extraItemSize: This is the number of bytes of data in the extra data item. This does not include the two bytes for the extraItemType and extraItemSize.

extraItemType: This is a constant with a value of 4. It identifies the extra data item as kerning pair data.

nKernPairs: The number of kerning pairs included in the table.

baseAdjustment: The base value of the adjustment, in metrics resolution units, relative to which all adjustment values are encoded. It is primarily intended to facilitate compaction from the use of the single byte adjustment values

Reserved: These unused 6 bits must be set to zero.

twoByteAdjValues: A bit flag defining how all kerning adjustment values are encoded. A zero indicates that every kerning adjustment value is encoded as an unsigned byte relative to the base adjustment. A one indicates that every kerning adjustment value is encoded as a signed 2-byte word relative to the base adjustment.

twoByteCharCodes: A bit flag defining how all character codes are encoded. A zero indicates that each character code is encoded as an unsigned byte. A one indicates that each character code is encoded as unsigned 2-byte words.

charCode1: The character code for the left character of each kerning pair.

charCode2: The character code for the right character of each kerning pair.

adjustment: The amount by which the escapement is to be adjusted between the left and right characters of the kerning pair in metrics resolution units. The adjustment is positive to increase the spacing, negative to reduce the spacing. The adjustment is relative to the value of baseAdjustment for the block of kerning data.

The order of the kerning pair records is required to be in increasing order of charCode1. Groups of records with a common value of charCode1 are required to be in increasing order of charCode2.

Because the maximum number of bytes in an extra data item is limited to 255, there is a limit on the number of kerning pairs that may be included in one extra data item. Multiple extra data items may be used to overcome this limit. The order of such multiple items must be in ascending order of character pair codes. This allows the search for a specific character code pair to scan the first entry in each type 4 extra data item to determine which block contains the pair.

10.2 Track kerning data

The format of a track kerning data block is as follows:

Syntax	*Number of bits*	*Mnemonic*
trackKernData() {		
extraItemType	8	uimsbf
extraItemSize	8	uimsbf
nKernTracks	8	uimsbf
for (i = 0; i < nKernTracks; i++){		
degree	16	uimsbf
minPointSize	16	uimsbf
minAdjust	16	imsbf
maxPointSize	16	uimsbf
maxAdjust	16	imsbf
}		
}		

extraItemType: This is a constant with a value of 5. It identifies the extra data item as kerning track data.

extraItemSize: This is the number of bytes of data in the extra data item. This does not include the two bytes for the extraItemType and extraItemSize.

nKernTracks: The number of track kerning entries,

degree: This identifies the amount of track kerning. Standard values are –1 for light kerning, -2 for medium kerning, and –3 for tight kerning.

minPointSize: This is the minimum point size at which the track kerning takes place for the current track. Its value is in units of points.

minAdjust: This is the spacing adjustment to be applied between each pair of characters at the minimum point size. Its value is in units of 1/256 points. A positive value indicates an increase in spacing; a negative value indicates a decrease in spacing.

maxPointSize: This is the minimum point size at which the track kerning takes place for the current track. Its value is in units of points.

maxAdjust: This is the spacing adjustment to be applied between each pair of characters at the maximum point size. Its value is in units of 1/256 points. A positive value indicates an increase in spacing; a negative value indicates a decrease in spacing.

It is not expected that the 255-byte limit on the size of an extra data item will be significant as this allows about 25 kerning tracks to be included.

Authors: John Collins (jcollins@bitstream.com) and
Bob Thomas (bthomas@bitstream.com)