The TrueType Font File

A TrueType font file contains data, in table format, that comprises an outline font. The rasterizer uses combinations of data from different tables to render the glyph data in the font.

The rasterizer has a much easier time traversing tables if they are padded so that each table begins on a 4-byte boundary. It is highly recommended that all tables be long aligned and padded with zeroes.

Data Types

The following data types are used in the TrueType font file. All TrueType fonts use Motorola-style byte ordering (Big Endian):

Data type	Description
BYTE	8-bit unsigned integer.
CHAR	8-bit signed integer.
USHORT	16-bit unsigned integer.
SHORT	16-bit signed integer.
ULONG	32-bit unsigned integer.
LONG	32-bit signed integer.
FIXED	32-bit signed fixed-point number (16.16)
FUNIT	Smallest measurable distance in the em space.
FWORD	16-bit signed integer (SHORT) that describes a quantity in FUnits.
UFWORD	Unsigned 16-bit integer (USHORT) that describes a quantity in FUnits.
F2DOT14	16-bit signed fixed number with the low 14 bits of fraction (2.14).

Most tables have version numbers, and the version number for the entire font is contained in the Table Directory (see below). Note that there are two different version number types, each with its own numbering scheme. USHORT version numbers always start at zero (0). Fixed version numbers always start at one (1.0 or 0x00010000).

The Fixed point format consists of a signed, 2's complement mantissa and an unsigned fraction. To compute the actual value, take the mantissa and add the fraction. Examples of 2.14 values are:

Decimal Value	Hex Value	Mantissa	Fraction
1.999939	0x7fff	1	16383/16384
1.75	0x7000	1	0/16384
0.000061	0x0001	0	1/16384
0.0	0x0000	0	0/16384
-0.000061	0xffff	-1	16383/16384
-2.0	0x8000	-2	0/16384

The Table Directory

The TrueType font file begins at byte 0 with the Offset Table.

Type	Name	Description
Fixed	sfnt version	0x00010000 for version 1.0.
USHORT	numTables	Number of tables.
USHORT	searchRange	(Maximum power of 2 ≤ numTables) x 16.
USHORT	entrySelector	Log_2(maximum power of 2 ≤ numTables).
USHORT	rangeShift	NumTables x 16-searchRange.

This is followed at byte 12 by the Table Directory entries. Entries in the Table Directory must be sorted in ascending order by tag.

Type	Name	Description
ULONG	tag	4 -byte identifier.
ULONG	checkSum	CheckSum for this table.
ULONG	offset	Offset from beginning of TrueType font file.
ULONG	length	Length of this table.

The Table Directory makes it possible for a given font to contain only those tables it actually needs. As a result there is no standard value for numTables.

Tags are the names given to tables in the TrueType font file. At present, all tag names consist of four characters, though this need not be the case. Names with less than four letters are allowed if followed by the necessary trailing spaces. A list of the currently defined tags follows.

Required Tables

Tag	Name
cmap	character to glyph mapping
glyf	glyph data
head	font header
hhea	horizontal header
hmtx	horizontal metrics
loca	index to location
maxp	maximum profile
name	naming table
post	PostScript information
OS/2	OS/2 and Windows specific metrics

Optional Tables

Tag	Name
cvt	Control Value Table
EBDT	Embedded bitmap data
EBLC	Embedded bitmap location data
EBSC	Embedded bitmap scaling data
fpgm	font program
gasp	grid-fitting and scan conversion procedure (grayscale)
hdmx	horizontal device metrics
kern	kerning
LTSH	Linear threshold table
prep	CVT Program
PCLT	PCL5
VDMX	Vertical Device Metrics table
vhea	Vertical Metrics header
vmtx	Vertical Metrics

Other tables may be defined for other platforms and for future expansion. Note that these tables will not have any effect on the scan converter. Tags for these tables must be registered with Apple Developer Technical Support. Tag names consisting of all lower case letters are reserved for Apple’s use. The number 0 is never a valid tag name.

Table checksums are the unsigned sum of the longs of a given table. In C, the following function can be used to determine a checksum:

```
ULONG
CalcTableChecksum(ULONG *Table, ULONG Length)
{
ULONG Sum = 0L;
ULONG *Endptr = Table+((Length+3) & ~3) / sizeof(ULONG);

while (Table < EndPtr)
  Sum += *Table++;
return Sum;
}
```

Note: This function implies that the length of a table must be a multiple of four bytes. While this is not a requirement for the TrueType scaler itself, it is suggested that all tables begin on four byte boundries, and pad any remaining space between tables with zeros. The length of all tables should be recorded in the table directory with their actual length.

Note that the offset in the Table Directory is measured from the start of the TrueType font file.

''cmap - Character To Glyph Index Mapping Table

This table defines the mapping of character codes to the glyph index values used in the font. It may contain more than one subtable, in order to support more than one character encoding scheme. Character codes that do not correspond to any glyph in the font should be mapped to glyph index 0. The glyph at this location must be a special glyph representing a missing character.

The table header indicates the character encodings for which subtables are present. Each subtable is in one of four possible formats and begins with a format code indicating the format used.

The platform ID and platform-specific encoding ID are used to specify the subtable; this means that each platform ID/platform-specific encoding ID pair may only appear once in the cmap table. Each subtable can specify a different character encoding. (*See the 'name' table section).* The entries must be sorted first by platform ID and then by platform-specific encoding ID.

When building a Unicode font for Windows, the platform ID should be 3 and the encoding ID should be 1. When building a symbol font for Windows, the platform ID should be 3 and the encoding ID should be 0. When building a font that will be used on the Macintosh, the platform ID should be 1 and the encoding ID should be 0.

All Microsoft Unicode encodings (Platform ID = 3, Encoding ID = 1) must use Format 4 for their 'cmap' subtable. Microsoft **strongly** recommends using a Unicode 'cmap' for all fonts. However, some other encodings that appear in current fonts follow:

Platform ID	Encoding ID	Description
3	0	Symbol
3	1	Unicode
3	2	ShiftJIS
3	3	Big5
3	4	PRC
3	5	Wansung
3	6	Johab

The Character To Glyph Index Mapping Table is organized as follows:

Type	Description
USHORT	Table version number (0).
USHORT	Number of encoding tables, *n.*

This is followed by an entry for each of the *n* encoding table specifying the particular encoding, and the offset to the actual subtable:

Type	Description
USHORT	Platform ID.
USHORT	Platform-specific encoding ID.
ULONG	Byte offset from beginning of table to the subtable for this encoding.

Format 0: Byte encoding table

This is the Apple standard character to glyph index mapping table.

Type	Name	Description
USHORT	format	Format number is set to 0.
USHORT	length	This is the length in bytes of the subtable.
USHORT	version	Version number (starts at 0).
BYTE	glyphIdArray[256]	An array that maps character codes to glyph index values.

This is a simple 1 to 1 mapping of character codes to glyph indices. The glyph set is limited to 256. Note that if this format is used to index into a larger glyph set, only the first 256 glyphs will be accessible.

Format 2: High-byte mapping through table

This subtable is useful for the national character code standards used for Japanese, Chinese, and Korean characters. These code standards use a mixed 8/16-bit encoding, in which certain byte values signal the first byte of a 2-byte character (but these values are also legal as the second byte of a 2-byte character). Character codes are always 1-byte. The glyph set is limited to 256.

In addition, even for the 2-byte characters, the mapping of character codes to glyph index values depends heavily on the first byte. Consequently, the table begins with an array that maps the first byte to a 4-word subHeader. For 2-byte character codes, the subHeader is used to map the second byte's value through a subArray, as described below. When processing mixed 8/16-bit text, subHeader 0 is special: it is used for single-byte character codes. When subHeader zero is used, a second byte is not needed; the single byte value is mapped through the subArray.

Type	Name	Description
USHORT	format	Format number is set to 2.
USHORT	length	Length in bytes.
USHORT	version	Version number (starts at 0)
USHORT	subHeaderKeys[256]	Array that maps high bytes to subHeaders: value is subHeader index * 8.
4 words struct	subHeaders[ ]	Variable-length array of subHeader structures.
4 words-struct	subHeaders[ ]	
USHORT	glyphIndexArray[ ]	Variable-length array containing subarrays used for mapping the low byte of 2-byte characters.

A subHeader is structured as follows:

Type	Name	Description
USHORT	firstCode	First valid low byte for this subHeader.
USHORT	entryCount	Number of valid low bytes for this subHeader.
SHORT	idDelta	See text below.
USHORT	idRangeOffset	See text below.

The firstCode and entryCount values specify a subrange that begins at firstCode and has a length equal to the value of entryCount. This subrange stays within the 0–255 range of the byte being mapped. Bytes outside of this subrange are mapped to glyph index 0 (missing glyph).The offset of the byte within this subrange is then used as index into a corresponding subarray of glyphIndexArray. This subarray is also of length entryCount. The value of the idRangeOffset is the number of bytes past the actual location of the idRangeOffset word where the glyphIndexArray element corresponding to firstCode appears.

Finally, if the value obtained from the subarray is not 0 (which indicates the missing glyph), you should add idDelta to it in order to get the glyphIndex. The value idDelta permits the same subarray to be used for several different subheaders. The idDelta arithmetic is modulo 65536.

Format 4: Segment mapping to delta values

This is the Microsoft standard character to glyph index mapping table.

This format is used when the character codes for the characters represented by a font fall into several contiguous ranges, possibly with holes in some or all of the ranges (that is, some of the codes in a range may not have a representation in the font). The format-dependent data is divided into three parts, which must occur in the following order:

1. A four-word header gives parameters for an optimized search of the segment list;
2. Four parallel arrays describe the segments (one segment for each contiguous range of codes);
3. A variable-length array of glyph IDs (unsigned words).

Type	Name	Description
USHORT	format	Format number is set to 4.
USHORT	length	Length in bytes.
USHORT	version	Version number (starts at 0).
USHORT	segCountX2	2 x segCount.
USHORT	searchRange	2 x (2**floor($\log_2$(segCount)))
USHORT	entrySelector	$\log_2$(searchRange/2)
USHORT	rangeShift	2 x segCount - searchRange
USHORT	endCount[segCount]	End characterCode for each segment, last =0xFFFF.
USHORT	reservedPad	Set to 0.
USHORT	startCount[segCount]	Start character code for each segment.
USHORT	idDelta[segCount]	Delta for all character codes in segment.
USHORT	idRangeOffset[segCount]	Offsets into glyphIdArray or 0
USHORT	glyphIdArray[ ]	Glyph index array (arbitrary length)

The number of segments is specified by segCount, which is not explicitly in the header; however, all of the header parameters are derived from it. The searchRange value is twice the largest power of 2 that is less than or equal to segCount. For example, if segCount=39, we have the following:

segCountX2	78	
searchRange	64	(2 * largest power of 2 ≤ 39)
entrySelector	5	$\log_2$(32)
rangeShift	14	2 x 39 - 64

Each segment is described by a startCode and endCode, along with an idDelta and an idRangeOffset, which are used for mapping the character codes in the segment. The segments are sorted in order of increasing endCode values, and the segment values are specified in four parallel arrays. You search for the first endCode that is greater than or equal to the character code you want to map. If the corresponding startCode is less than or equal to the character code, then you use the corresponding idDelta and idRangeOffset to map the character code to a glyph index (otherwise, the missingGlyph is returned). For the search to terminate, the final endCode value must be 0xFFFF. This segment need not contain any valid mappings. (It can just map the single character code 0xFFFF to missingGlyph). However, the segment must be present.

If the idRangeOffset value for the segment is not 0, the mapping of character codes relies on glyphIdArray. The character code offset from startCode is added to the idRangeOffset value. This sum is used as an offset from the current location within idRangeOffset itself to index out the correct glyphIdArray value. This obscure indexing trick works because glyphIdArray immediately follows idRangeOffset in the font file. The C expression that yields the glyph index is:

```
*(idRangeOffset[i]/2 + (c - startCount[i]) + &idRangeOffset[i])
```

The value *c* is the character code in question, and *i* is the segment index in which *c* appears. If the value obtained from the indexing operation is not 0 (which indicates missingGlyph), idDelta[i] is added to it to get the glyph index. The idDelta arithmetic is modulo 65536.

If the idRangeOffset is 0, the idDelta value is added directly to the character code offset (i.e. `idDelta[i] + c`) to get the corresponding glyph index. Again, the idDelta arithmetic is modulo 65536.

As an example, the variant part of the table to map characters 10–20, 30–90, and 100–153 onto a contiguous range of glyph indices may look like this:

segCountX2:	8			
searchRange:	8			
entrySelector:	4			
rangeShift:	0			
endCode:	20	90	153	0xFFFF
reservedPad:	0			
startCode:	10	30	100	0xFFFF
idDelta:	-9	-18	-27	1
idRangeOffset:	0	0	0	0

This table performs the following mappings:

10 –> 10 – 9 = 1

20 –> 20 – 9 = 11

30 –> 30 – 18 = 12

90 –> 90 – 18 = 72

...and so on.

Note that the delta values could be reworked so as to reorder the segments.

Format 6: Trimmed table mapping

Type	Name	Description
USHORT	format	Format number is set to 6.
USHORT	length	Length in bytes.
USHORT	version	Version number (starts at 0)
USHORT	firstCode	First character code of subrange.
USHORT	entryCount	Number of character codes in subrange.
USHORT	glyphIdArray [entryCount]	Array of glyph index values for character codes in the range.

The firstCode and entryCount values specify a subrange (beginning at firstCode,length = entryCount) within the range of possible character codes. Codes outside of this subrange are mapped to glyph index 0. The offset of the code (from the first code) within this subrange is used as index to the glyphIdArray, which provides the glyph index value.

cvt - Control Value Table

This table contains a list of values that can be referenced by instructions. They can be used, among other things, to control characteristics for different glyphs.

Type	Description
FWORD[*n*]	List of *n* values referenceable by instructions.

EBDT - Embedded Bitmap Data Table

Three new tables are used to embed bitmaps in TrueType fonts. They are the ‘EBLC’ table for embedded bitmap locators, the ‘EBDT’ table for embedded bitmap data, and the ‘EBSC’ table for embedded bitmap scaling information.

TrueType embedded bitmaps are also called ‘sbits’ (for “scaler bitmaps”). A set of bitmaps for a face at a given size is called a strike.

The ‘EBLC’ table identifies the sizes and glyph ranges of the sbits, and keeps offsets to glyph bitmap data in indexSubTables. The ‘EBDT’ table then stores the glyph bitmap data, in a number of different possible formats. Glyph metrics information may be stored in either the ‘EBLC’ or ‘EBDT’ table, depending upon the indexSubTable and glyph bitmap data formats. The ‘EBSC’ table identifies sizes that will be handled by scaling up or scaling down other sbit sizes.

The ‘EBDT’ table uses the same format as Apple has defined for the QuickDraw GX ‘bdat’ table.

The ‘EBDT’ table begins with a header containing simply the table version number.

Type	Name	Description
FIXED	version	Initially defined as 0x00020000

The rest of the ‘EBDT’ table is a simply a collection of bitmap data. The data can be in a number of possible formats, indicated by information in the ‘EBLC’ table. Some of the formats contain metric information plus image data, and other formats contain only the image data. Long word alignment is not required for these sub tables; byte alignment is sufficient.

There are also two different formats for glyph metrics: big glyph metrics and small glyph metrics. Big glyph metrics define metrics information for both horizontal and vertical layouts. This is important in fonts (such as Kanji) where both types of layout may be used. Small glyph metrics define metrics information for one layout direction only. Which direction applies, horizontal or vertical, is determined by the ‘flags’ field in the bitmapSizeTable field of the ‘EBLC’ table.

bigGlyphMetrics

Type	Name
BYTE	height
BYTE	width
CHAR	horiBearingX
CHAR	horiBearingY
BYTE	horiAdvance
CHAR	vertBearingX
CHAR	vertBearingY
BYTE	vertAdvance

smallGlyphMetrics

Type	Name
BYTE	height
BYTE	width
CHAR	BearingX
CHAR	BearingY
BYTE	Advance

The nine different formats currently defined for glyph bitmap data are listed and described below. Different formats are better for different purposes. Apple ‘bdat’ tables support only formats 1 through 7.

Format 1: small metrics, byte-aligned data

Type	Name	Description
smallGlyphMetrics	smallMetrics	Metrics information for the glyph
VARIABLE	image data	Byte-aligned bitmap data

Glyph bitmap format 1 consists of small metrics records (either horizontal or vertical depending on the bitmapSizeTable ‘flag’ value in the ‘EBLC’ table) followed by byte aligned bitmap data. The bitmap data begins with the most significant bit of the first byte corresponding to the top-left pixel of the bounding box, proceeding through succeeding bits moving left to right. The data for each row is padded to a byte boundary, so the next row begins with the most significant bit of a new byte. 1 bits correspond to black, and 0 bits to white.

Format 2: small metrics, bit-aligned data

Type	Name	Description
smallGlyphMetrics	small Metrics	Metrics information for the glyph
VARIABLE	image data	Bit-aligned bitmap data

Glyph bitmap format 2 is the same as format 1 except that the bitmap data is bit aligned. This means that the data for a new row will begin with the bit immediately following the last bit of the previous row. The start of each glyph must be byte aligned, so the last row of a glyph may require padding. This format takes a little more time to parse, but saves file space compared to format 1.

Format 3: (obsolete)

Format 4: (not supported) metrics in EBLC, compressed data

Glyph bitmap format 4 is a compressed format used by Apple in some of their Far East fonts. MS has not implemented it in our rasterizer.

Format 5: metrics in EBLC, bit-aligned image data only

Type	Name	Description
VARIABLE	image data	Bit-aligned bitmap data

Glyph bitmap format 5 is similar to format 2 except that no metrics information is included, just the bit aligned data. This format is for use with ‘EBLC’ indexSubTable format 2 or format 5, which will contain the metrics information for all glyphs. It works well for Kanji fonts.

The rasterizer recalculates sbit metrics for Format 5 bitmap data, allowing Windows to report correct ABC widths, even if the bitmaps have white space on either side of the bitmap image. This allows fonts to store monospaced bitmap glyphs in the efficient Format 5 without breaking Windows GetABCWidths call.

Format 6: big metrics, byte-aligned data

Type	Name	Description
bigGlyphMetrics	bigMetrics	Metrics information for the glyph
VARIABLE	image data	Byte-aligned bitmap data

Glyph bitmap format 6 is the same as format 1 except that is uses big glyph metrics instead of small.

Format 7: big metrics, bit-aligned data

Type	Name	Description
bigGlyphMetrics	bigMetrics	Metrics information for the glyph
VARIABLE	image data	Bit-aligned bitmap data

Glyph bitmap format 7 is the same as format 2 except that is uses big glyph metrics instead of small.

ebdtComponent; array used by Formats 8 and 9

Type	Name	Description
USHORT	glyphCode	Component glyph code
CHAR	xOffset	Position of component left
CHAR	yOffset	Position of component top

The component array, used by Formats 8 and 9, contains the glyph code of the component, which can be looked up in the 'EBLC' table, as well as xOffset and yOffset values which tell where to position the top-left corner of the component in the composite. Nested composites (a composite of composites) are allowed, and the number of nesting levels is determined by implementation stack space.

Format 8: small metrics, component data

Type	Name	Description
smallGlyphMetrics	smallMetrics	Metrics information for the glyph
BYTE	pad	Pad to short boundary
USHORT	numComponents	Number of components
ebdtComponent	componentArray[n]	Glyph code, offset array

Format 9: big metrics, component data

Type	Name	Description
bigGlyphMetrics	bigMetrics	Metrics information for the glyph
USHORT	numComponents	Number of components
ebdtComponent	componentArray[n]	Glyph code, offset array

Glyph bitmap formats 8 and 9 are used for composite bitmaps. For accented characters and other composite glyphs it may be more efficient to store a copy of each component separately, and then use a composite description to construct the finished glyph. The composite formats allow for any number of components, and allow the components to be positioned anywhere in the finished glyph. Format 8 uses small metrics, and format 9 uses big metrics.

EBLC - Embedded Bitmap Location Table

Three new tables are used to embed bitmaps in TrueType fonts. They are the 'EBLC' table for embedded bitmap locators, the 'EBDT' table for embedded bitmap data, and the 'EBSC' table for embedded bitmap scaling information. TrueType embedded bitmaps are called 'sbits' (for "scaler bitmaps"). A set of bitmaps for a face at a given size is called a strike.

The 'EBLC' table identifies the sizes and glyph ranges of the sbits, and keeps offsets to glyph bitmap data in indexSubTables. The 'EBDT' table then stores the glyph bitmap data, also in a number of different possible formats. Glyph metrics information may be stored in either the 'EBLC' or 'EBDT' table, depending upon the indexSubTable and glyph bitmap formats. The 'EBSC' table identifies sizes that will be handled by scaling up or scaling down other sbit sizes.

The 'EBLC' table uses the same format as the Apple QuickDraw GX 'bloc' table.

The 'EBLC' table begins with a header containing the table version and number of strikes. A TrueType font may have one or more strikes embedded in the 'EBDT' table.

eblcHeader

Type	Name	Description
FIXED	version	initially defined as 0x00020000
ULONG	numSizes	Number of bitmapSizeTables

The eblcHeader is followed immediately by the bitmapSizeTable array(s). The numSizes in the eblcHeader indicates the number of bitmapSizeTables in the array. Each strike is defined by one bitmapSizeTable.

bitmapSizeTable

Type	Name	Description
ULONG	indexSubTableArrayOffset	offset to index subtable from beginning of EBLC.
ULONG	indexTablesSize	number of bytes in corresponding index subtables and array
ULONG	numberOfIndexSubTables	an index subtable for each range or format change
ULONG	colorRef	not used; set to 0.
sbitLineMetrics	hori	line metrics for text rendered horizontally
sbitLineMetrics	vert	line metrics for text rendered vertically
USHORT	startGlyphIndex	lowest glyph index for this size
USHORT	endGlyphIndex	highest glyph index for this size
BYTE	ppemX	horizontal pixels per Em
BYTE	ppemY	vertical pixels per Em
BYTE	bitDepth	set to 1 for now
CHAR	flags	vertical or horizontal (see bitmapFlags)

The indexSubTableArrayOffset is the offset from the beginning of the ‘EBLC’ table to the indexSubTableArray. Each strike has one of these arrays to support various formats and discontiguous ranges of bitmaps. The indexTablesSize is the total number of bytes in the indexSubTableArray and the associated indexSubTables. The numberOfIndexSubTables is a count of the indexSubTables for this strike.

The horizontal and vertical line metrics contain the ascender, descender, linegap, and advance information for the strike. The line metrics format is described in the following table:

sbitLineMetrics

Type	Name
CHAR	ascender
CHAR	descender
BYTE	widthMax
CHAR	caretSlopeNumerator
CHAR	caretSlopeDenominator
CHAR	caretOffset
CHAR	minOriginSB
CHAR	minAdvanceSB
CHAR	maxBeforeBL
CHAR	minAfterBL
CHAR	pad1
CHAR	pad2

The caret slope determines the angle at which the caret is drawn, and the offset is the number of pixels (+ or -) to move the caret. This is a signed char since we are dealing with integer metrics. The minOriginSB, minAdvanceSB , maxBeforeBL, and minAfterBL are described in the diagrams below. The main need for these numbers is for scalers that may need to pre-allocate memory and/or need more metric information to position glyphs. All of the line metrics are one byte in length. The line metrics are not used directly by the rasterizer, but are available to clients who want to parse the 'EBLC' table.

The startGlyphIndex and endGlyphIndex describe the minimum and maximum glyph codes in the strike, but a strike does not necessarily contain bitmaps for all glyph codes in this range. The indexSubTables determine which glyphs are actually present in the 'EBDT' table.

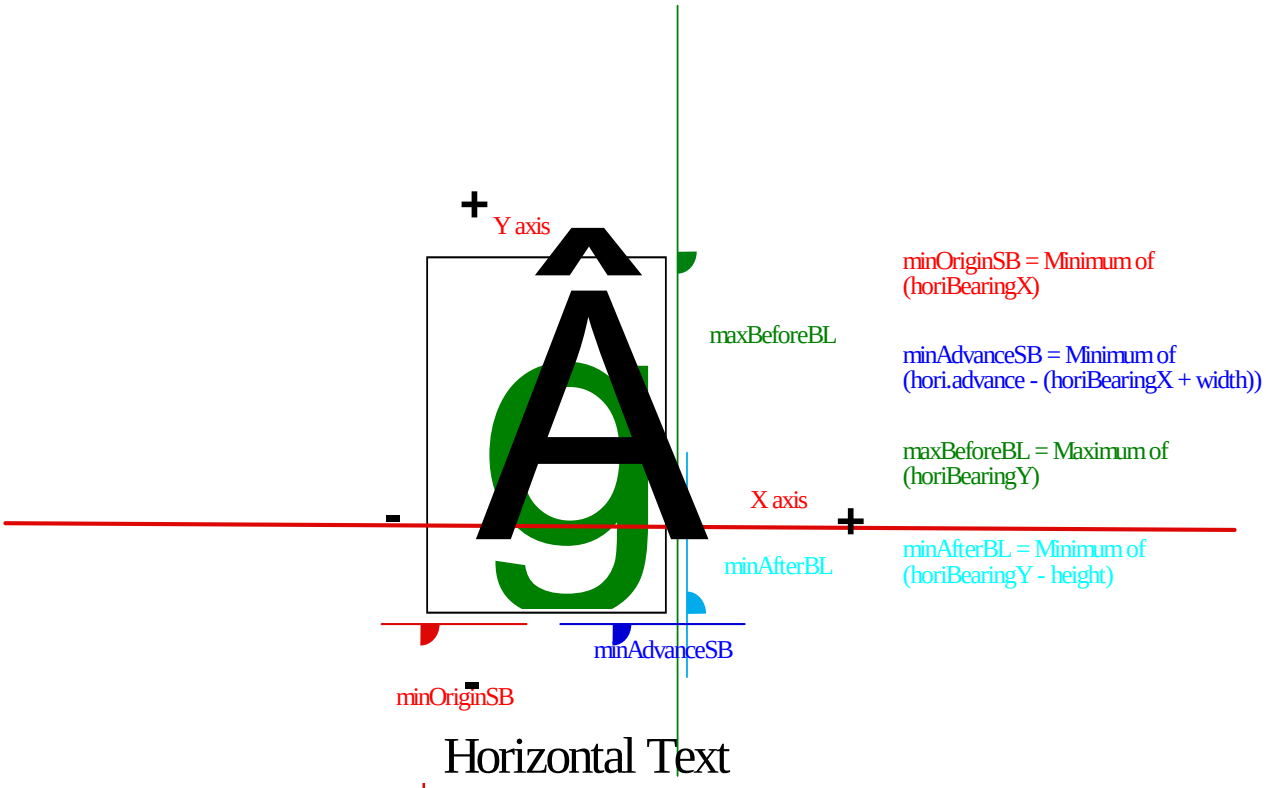
The ppemX and ppemY fields describe the size of the strike in pixels per Em. The ppem measurement is equivalent to point size on a 72 dots per inch device. Typically, ppemX will be equal to ppemY for devices with 'square pixels'. To accommodate devices with rectangular pixels, and to allow for bitmaps with other aspect ratios, ppemX and ppemY may differ.

The 'flags' byte contains two bits to indicate the direction of small glyph metrics: horizontal or vertical. The remaining bits are reserved.

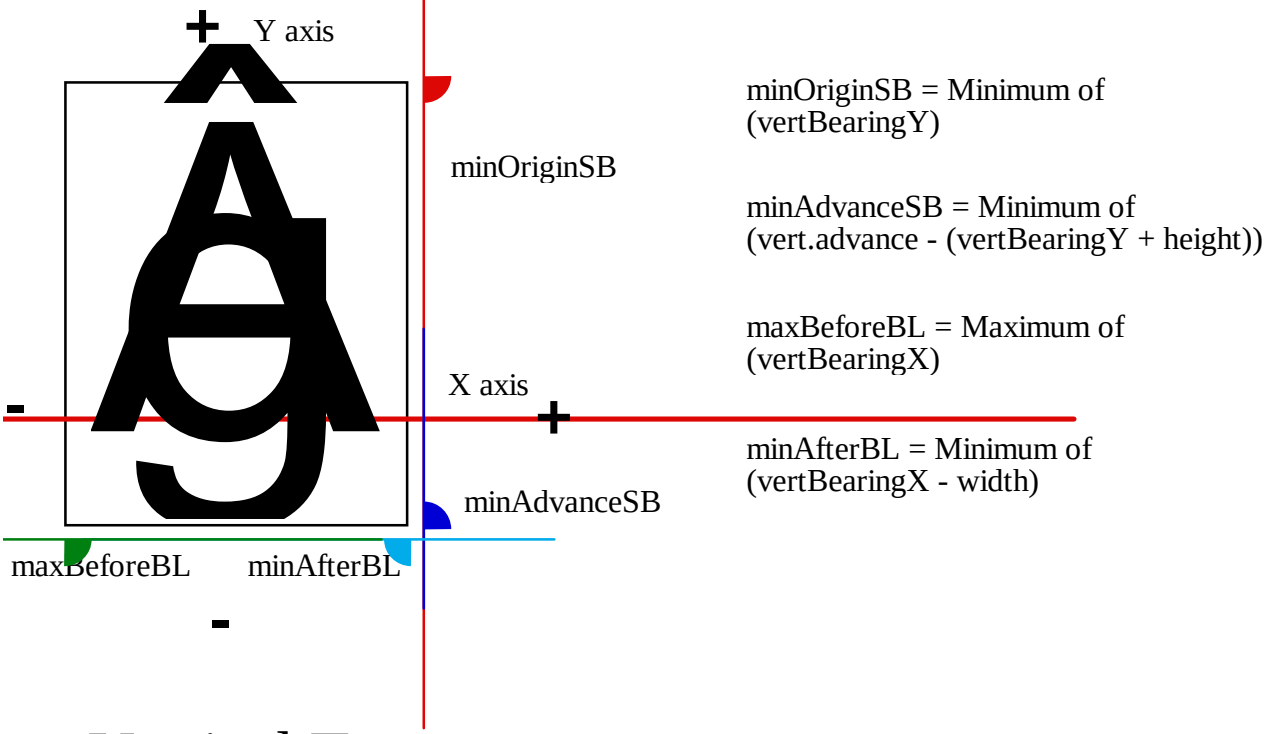
Bitmap Flags

Type	Value	Description
CHAR	0x01	Horizontal
CHAR	0x02	Vertical

The colorRef and bitDepth fields are reserved for future enhancements. For monochrome bitmaps they should have the values colorRef=0 and bitDepth=1.



Horizontal Text



Vertical Text

Associated with the image data for every glyph in a strike is a set of glyph metrics. These glyph metrics describe bounding box height and width, as well as side bearing and advance width information. The glyph metrics can be found in one of two places. For ranges of glyphs (not necessarily the whole strike) whose metrics may be different for each glyph, the glyph metrics are stored along with the glyph image data in the 'EBDT' table. Details of how this is done is described in the 'EBDT' section of this document. For ranges of glyphs whose metrics are identical for every glyph, we save significant space by storing a single copy of the glyph metrics in the indexSubTable in the 'EBLC'.

There are also two different formats for glyph metrics: big glyph metrics and small glyph metrics. Big glyph metrics define metrics information for both horizontal and vertical layouts. This is important in fonts (such as Kanji) where both types of layout may be used. Small glyph metrics define metrics information for one layout direction only. Which direction applies, horizontal or vertical, is determined by the 'flags' field in the bitmapSizeTable.

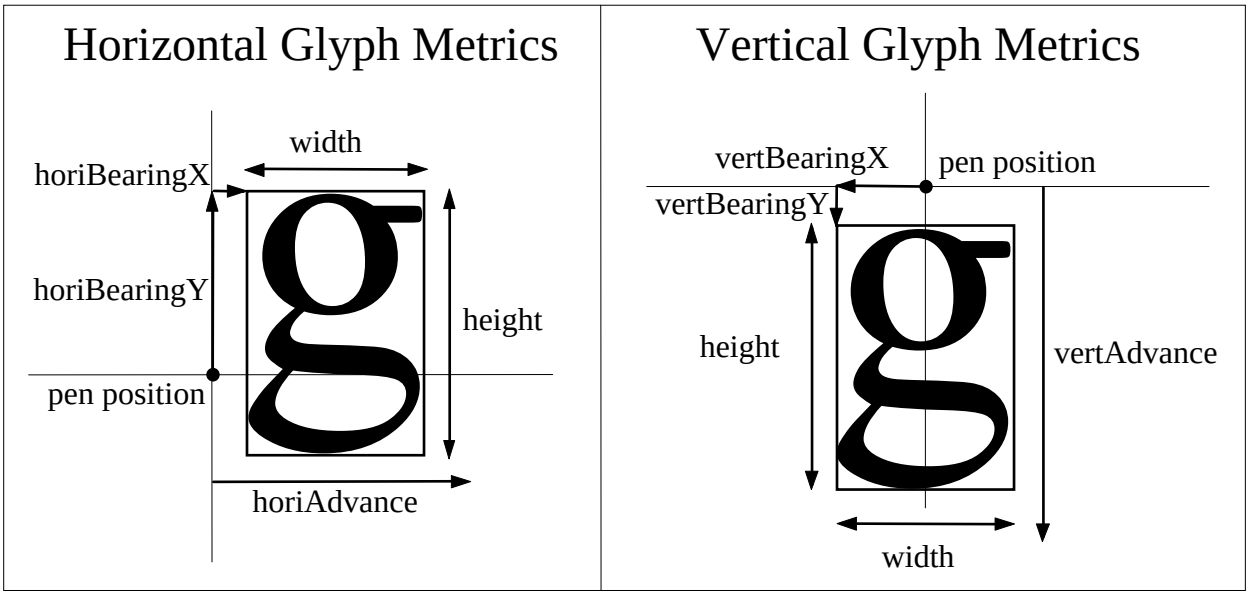
bigGlyphMetrics

Type	Name
BYTE	height
BYTE	width
CHAR	horiBearingX
CHAR	horiBearingY
BYTE	horiAdvance
CHAR	vertBearingX
CHAR	vertBearingY
BYTE	vertAdvance

smallGlyphMetrics

Type	Name
BYTE	height
BYTE	width
CHAR	BearingX
CHAR	BearingY
BYTE	Advance

The following diagram illustrates the meaning of the glyph metrics.



The bitmapSizeTable for each strike contains the offset to an array of indexSubTableArray elements. Each element describes a glyph code range and an offset to the indexSubTable for that range. This allows a strike to contain multiple glyph code ranges and to be represented in multiple index formats if desirable.

indexSubTableArray

Type	Name	Description
USHORT	firstGlyphIndex	first glyph code of this range
USHORT	lastGlyphIndex	last glyph code of this range (inclusive)
ULONG	additionalOffsetToIndexSubtable	add to indexSubTableArrayOffset to get offset from beginning of 'EBLC'

After determining the strike, the rasterizer searches this array for the range containing the given glyph code. When the range is found, the additionalOffsetToIndexSubtable is added to the indexSubTableArrayOffset to get the offset of the indexSubTable in the 'EBLC'.

The first indexSubTableArray is located after the last bitmapSizeSubTable entry. Then the indexSubTables for the strike follow. Another indexSubTableArray (if more than one strike) and its indexSubTables are next. The 'EBLC' continues with an array and indexSubTables for each strike.

We now have the offset to the indexSubTable. All indexSubTable formats begin with an indexSubHeader which identifies the indexSubTable format, the format of the 'EBDT' image data, and the offset from the beginning of the 'EBDT' table to the beginning of the image data for this range.

indexSubHeader

Type	Name	Description
USHORT	indexFormat	format of this indexSubTable
USHORT	imageFormat	format of 'EBDT' image data
ULONG	imageDataOffset	offset to image data in 'EBDT' table

There are currently five different formats used for the indexSubTable, depending upon the size and type of bitmap data in the glyph code range. Apple 'bloc' tables support only formats 1 through 3.

The choice of which indexSubTable format to use is up to the font manufacturer, but should be made with the aim of minimizing the size of the font file. Ranges of glyphs with variable metrics — that is, where glyphs may differ from each other in bounding box height, width, side bearings or advance — must use format 1, 3 or 4. Ranges of glyphs with constant metrics can save space by using format 2 or 5, which keep a single copy of the metrics information in the indexSubTable rather than a copy per glyph in the 'EBDT' table. In some monospaced fonts it makes sense to store extra white space around some of the glyphs to keep all metrics identical, thus permitting the use of format 2 or 5.

Structures for each indexSubTable format are listed below.

indexSubTable1: variable metrics glyphs with 4 byte offsets

Type	Name	Description
indexSubHeader	header	header info
ULONG	offsetArray[]	offsetArray[glyphIndex]+imageDataOffset= glyphData sizeOfArray=(lastGlyph-firstGlyph+1)+1+1 pad if needed

indexSubTable2: all glyphs have identical metrics

Type	Name	Description
indexSubHeader	header	header info
ULONG	imageSize	all the glyphs are of the same size
bigGlyphMetrics	bigMetrics	all glyphs have the same metrics; glyph data may be compressed, byte-aligned, or bit-aligned

indexSubTable3: variable metrics glyphs with 2 byte offsets

Type	Name	Description
indexSubHeader	header	header info
USHORT	offsetArray[]	offsetArray[glyphIndex]+imageDataOffset= glyphData sizeOfArray=(lastGlyph-firstGlyph+1)+1+1 pad if needed

indexSubTable4: variable metrics glyphs with sparse glyph codes

Type	Name	Description
indexSubHeader	header	header info
ULONG	numGlyphs	array length
codeOffsetPair	glyphArray[]	one per glyph; sizeOfArray=numGlyphs+1

codeOffsetPair: used by indexSubTable4

Type	Name	Description
USHORT	glyphCode	code of glyph present
USHORT	offset	location in EBDT

indexSubTable5: constant metrics glyphs with sparse glyph codes

Type	Name	Description
indexSubHeader	header	header info
ULONG	imageSize	all glyphs have the same data size
bigGlyphMetrics	bigMetrics	all glyphs have the same metrics
ULONG	numGlyphs	array length
USHORT	glyphCodeArray[]	one per glyph, sorted by glyph code; sizeOfArray=numGlyphs

The size of the ‘EBDT’ image data can be calculated from the indexSubTable information. For the constant metrics formats (2 and 5) the image data size is constant, and is given in the imageSize field. For the variable metrics formats (1, 3, and 4) image data must be stored contiguously and in glyph code order, so the image data size may be calculated by subtracting the offset for the current glyph from the offset of the next glyph. Because of this, it is necessary to store one extra element in the offsetArray pointing just past the end of the range’s image data. This will allow the correct calculation of the image data size for the last glyph in the range.

Contiguous, or nearly contiguous, ranges of glyph codes are handled best by formats 1, 2, and 3 which store an offset for every glyph code in the range. Very sparse ranges of glyph codes should use format 4 or 5 which explicitly call out the glyph codes represented in the range. A small number of missing glyphs can be efficiently represented in formats 1 or 3 by having the offset for the missing glyph be followed by the same offset for the next glyph, thus indicating a data size of zero.

The only difference between formats 1 and 3 is the size of the offsetArray elements: format 1 uses ULONG's while format 3 uses USHORT's. Therefore format 1 can cover a greater range (> 64k bytes) while format 3 saves more space in the 'EBLC' table. Since the offsetArray elements are added to the imageDataOffset base address in the indexSubHeader, a very large set of glyph bitmap data could be addressed by splitting it into multiple ranges, each less than 64k bytes in size, allowing the use of the more efficient format 3.

The 'EBLC' table specification requires double word (ULONG) alignment for all subtables. This occurs naturally for indexSubTable formats 1, 2, and 4, but may not for formats 3 and 5, since they include arrays of type USHORT. When there is an odd number of elements in these arrays it is necessary to add an extra padding element to maintain proper alignment.

EBSC - Embedded Bitmap Scaling Table

The ‘EBSC’ table provides a mechanism for describing embedded bitmaps which are created by scaling other embedded bitmaps. While this is the sort of thing that outline font technologies were invented to avoid, there are cases (small sizes of Kanji, for example) where scaling a bitmap produces a more legible font than scan-converting an outline. For this reason the ‘EBSC’ table allows a font to define a bitmap strike as a scaled version of another strike.

The ‘EBSC’ table begins with a header containing the table version and number of strikes.

ebscHeader

Type	Name	Description
FIXED	version	initially defined as 0x00020000
ULONG	numSizes	

The ebscHeader is followed immediately by the bitmapScaleTable array. The numSizes in the ebscHeader indicates the number of bitmapScaleTables in the array. Each strike is defined by one bitmapScaleTable.

bitmapScaleTable

Type	Name	Description
sbitLineMetrics	hori	line metrics
sbitLineMetrics	vert	line metrics
BYTE	ppemX	target horizontal pixels per Em
BYTE	ppemY	target vertical pixels per Em
BYTE	substitutePpemX	use bitmaps of this size
BYTE	substitutePpemY	use bitmaps of this size

The line metrics have the same meaning as those in the bitmapSizeTable, and refer to font wide metrics after scaling. The ppemX and ppemY values describe the size of the font after scaling. The substitutePpemX and substitutePpemY values describe the size of a strike that exists as an sbit in the ‘EBLC’ and ‘EBDT’, and that will be scaled up or down to generate the new strike.

Notice that scaling in the x direction is independent of scaling in the y direction, and their scaling values may differ. A square aspect-ratio strike could be scaled to a non-square aspect ratio. Glyph metrics are scaled by the same factor as the pixels per Em (in the appropriate direction), and are rounded to the nearest integer pixel.

fpgm - Font Program

This table is similar to the CVT Program, except that it is only run once, when the font is first used. It is used only for FDEFs and IDEFs. Thus the CVT Program need not contain function definitions. However, the CVT Program may redefine existing FDEFs or IDEFs.

This table is optional.

Type	Description
BYTE[*n*]	Instructions

gasp - Grid-fitting And Scan-conversion Procedure

This table contains information which describes the preferred rasterization techniques for the typeface when it is rendered on grayscale-capable devices. This table also has some use for monochrome devices, which may use the table to turn off hinting at very large or small sizes, to improve performance.

At very small sizes, the best appearance on grayscale devices can usually be achieved by rendering the glyphs in grayscale without using hints. At intermediate sizes, hinting and monochrome rendering will usually produce the best appearance. At large sizes, the combination of hinting and grayscale rendering will typically produce the best appearance.

If the ‘gasp’ table is not present in a typeface, TrueType will apply default rules to decide how to render the glyphs on grayscale devices.

The ‘gasp’ table consists of a header followed by groupings of ‘gasp’ records:

gasp Table

Type	Name	Description
USHORT	version	Version number (set to 0)
USHORT	numRanges	Number of records to follow
GASPRANGE	gaspRange[numRanges]	Sorted by ppem

Each GASPRANGE record looks like this:

Type	Name	Description
USHORT	rangeMaxPPEM	Upper limit of range, in PPEM
USHORT	rangeGaspBehavior	Flags describing desired rasterizer behavior.

There are two flags for the rangeGaspBehavior flags:

Flag	Meaning
GASP_GRIDFIT	Use gridfitting
GASP_DOGRAY	Use grayscale rendering

The set of bit flags may be extended in the future.

The four currently defined values of rangeGaspBehavior would have the following uses:

Flag	Value	Meaning
GASP_DOGRAY	0x0002	small sizes, typically ppem<9
GASP_GRIDFIT	0x0001	medium sizes, typically 9<=ppem<=16
GASP_DOGRAY| GASP_GRIDFIT	0x0003	large sizes, typically ppem>16
(neither)	0x0000	optional for very large sizes, typically ppem>2048

The records in the gaspRange[] array must be sorted in order of increasing rangeMaxPPEM value. The last record should use 0xFFFF as a sentinel value for rangeMaxPPEM and should describe the behavior desired at all sizes larger than the previous record's upper limit. If the only entry in 'gasp' is the 0xFFFF sentinel value, the behavior described will be used for *all* sizes.

Sample 'gasp' table

Field	Value		Meaning
version	0x0000		
numRanges	0x0003		
Range[0], Flag	0x0008	0x0002	ppem<=8, grayscale only
Range[1], Flag	0x0010	0x0001	9<=ppem<=16, gridfit only
Range[2], Flag	0xFFFF	0x0003	16<ppem, gridfit and grayscale

glyf - Glyph Data

This table contains information that describes the glyphs in the font. Each glyph begins with the following header:

Type	Name	Description
SHORT	numberOfContours	If the number of contours is greater than or equal to zero, this is a single glyph; if negative, this is a composite glyph.
FWORD	xMin	Minimum x for coordinate data.
FWORD	yMin	Minimum y for coordinate data.
FWORD	xMax	Maximum x for coordinate data.
FWORD	yMax	Maximum y for coordinate data.

Note that the bounding rectangle from each character is defined as the rectangle with a lower left corner of (xMin, yMin) and an upper right corner of (xMax, yMax).

Note: The scaler will perform better if the glyph coordinates have been created such that the xMin is equal to the lsb. For example, if the lsb is 123, then xMin for the glyph should be 123. If the lsb is -12 then the xMin should be -12. If the lsb is 0 then xMin is 0. If all glyphs are done like this, set bit 1 of flags field in the 'head' table.

Simple Glyph Description

This is the table information needed if numberOfContours is greater than zero, that is, a glyph is not a composite.

Type	Name	Description
USHORT	endPtsOfContours[*n*]	Array of last points of each contour; *n* is the number of contours.
USHORT	instructionLength	Total number of bytes for instructions.
BYTE	instructions[*n*]	Array of instructions for each glyph; *n* is the number of instructions.
BYTE	flags[*n*]	Array of flags for each coordinate in outline; *n* is the number of flags.
BYTE or SHORT	xCoordinates[ ]	First coordinates relative to (0,0); others are relative to previous point.
BYTE or SHORT	yCoordinates[ ]	First coordinates relative to (0,0); others are relative to previous point.

Note: In the glyf table, the position of a point is not stored in absolute terms but as a vector relative to the previous point. The delta-x and delta-y vectors represent these (often small) changes in position.

Each flag is a single byte. Their meanings are shown below.

Flags	Bit	Description
On Curve	0	If set, the point is on the curve; otherwise, it is off the curve.
x-Short Vector	1	If set, the corresponding x-coordinate is 1 byte long, not 2.
y-Short Vector	2	If set, the corresponding y-coordinate is 1 byte long, not 2.
Repeat	3	If set, the next byte specifies the number of additional times this set of flags is to be repeated. In this way, the number of flags listed can be smaller than the number of points in a character.
This x is same (Positive x-Short Vector)	4	This flag has two meanings, depending on how the x-Short Vector flag is set. If x-Short Vector is set, this bit describes the sign of the value, with 1 equalling positive and 0 negative. If the x-Short Vector bit is not set and this bit is set, then the current x-coordinate is the same as the previous x-coordinate. If the x-Short Vector bit is not set and this bit is also not set, the current x-coordinate is a signed 16-bit delta vector.
This y is same (Positive y-Short Vector)	5	This flag has two meanings, depending on how the y-Short Vector flag is set. If y-Short Vector is set, this bit describes the sign of the value, with 1 equalling positive and 0 negative. If the y-Short Vector bit is not set and this bit is set, then the current y-coordinate is the same as the previous y-coordinate. If the y-Short Vector bit is not set and this bit is also not set, the current y-coordinate is a signed 16-bit delta vector.
Reserved	6	This bit is reserved. Set it to zero.
Reserved	7	This bit is reserved. Set it to zero.

Composite Glyph Description

This is the table information needed for composite glyphs (numberOfContours is -1). A composite glyph starts with two USHORT values (“flags” and “glyphIndex,” i.e. the index of the first contour in this composite glyph); the data then varies according to “flags”). The C pseudo-code fragment below shows how the composite glyph information is stored and parsed; definitions for “flags” bits follow this fragment:

```
do {
     USHORT flags;
     USHORT glyphIndex;
     if ( flags & ARG_1_AND_2_ARE_WORDS) {
     (SHORT or FWord) argument1;
     (SHORT or FWord) argument2;
     } else {
            USHORT arg1and2; /* (arg1 << 8) | arg2 */
     }
     if ( flags & WE_HAVE_A_SCALE ) {
            F2Dot14  scale;   /* Format 2.14 */
     } else if ( flags & WE_HAVE_AN_X_AND_Y_SCALE ) {
            F2Dot14  xscale;   /* Format 2.14 */
            F2Dot14  yscale;   /* Format 2.14 */
     } else if ( flags & WE_HAVE_A_TWO_BY_TWO ) {
            F2Dot14  xscale;   /* Format 2.14 */
            F2Dot14  scale01;  /* Format 2.14 */
            F2Dot14  scale10;  /* Format 2.14 */
            F2Dot14  yscale;   /* Format 2.14 */
     }
} while ( flags & MORE_COMPONENTS )

if (flags & WE_HAVE_INSTR){

     USHORT numInstr

     BYTE instr[numInstr]
```

Argument1 and argument2 can be either x and y offsets to be added to the glyph or two point numbers. In the latter case, the first point number indicates the point that is to be matched to the new glyph. The second number indicates the new glyph’s “matched” point. Once a glyph is added, its point numbers begin directly after the last glyphs (endpoint of first glyph + 1).

When arguments 1 and 2 are an x and a y offset instead of points and the bit ROUND_XY_TO_GRID is set to 1, the values are rounded to those of the closest grid lines before they are added to the glyph. X and Y offsets are described in FUnits.

If the bit WE_HAVE_A_SCALE is set, the scale value is read in 2.14 format—the value can be between -2 to almost +2. The glyph will be scaled by this value before grid-fitting.

The bit WE_HAVE_A_TWO_BY_TWO allows for an interrelationship between the x and y coordinates. This could be used for 90-degree rotations, for example.

These are the constants for the flags field:

Flags	Bit	Description
ARG_1_AND_2_ARE_WORDS	0	If this is set, the arguments are words; otherwise, they are bytes.
ARGS_ARE_XY_VALUES	1	If this is set, the arguments are xy values; otherwise, they are points.
ROUND_XY_TO_GRID	2	For the xy values if the preceding is true.
WE_HAVE_A_SCALE	3	This indicates that there is a simple scale for the component. Otherwise, scale = 1.0.
RESERVED	4	This bit is reserved. Set it to 0.
MORE_COMPONENTS	5	Indicates at least one more glyph after this one.
WE_HAVE_AN_X_AND_Y_SCALE	6	The x direction will use a different scale from the y direction.
WE_HAVE_A_TWO_BY_TWO	7	There is a 2 by 2 transformation that will be used to scale the component.
WE_HAVE_INSTRUCTIONS	8	Following the last component are instructions for the composite character.
USE_MY_METRICS	9	If set, this forces the aw and lsb (and rsb) for the composite to be equal to those from this original glyph. This works for hinted and unhinted characters.

The purpose of USE_MY_METRICS is to force the lsb and rsb to take on a desired value. For example, an i-circumflex (Unicode 00ef) is often composed of the circumflex and a dotless-i. In order to force the composite to have the same metrics as the dotless-i, set USE_MY_METRICS for the dotless-i component of the composite. Without this bit, the rsb and lsb would be calculated from the HMTX entry for the composite (or would need to be explicitly set with TrueType instructions).

Note that the behavior of the USE_MY_METRICS operation is undefined for rotated composite components.

hdmx - Horizontal Device Metrics

The Horizontal Device Metrics table stores integer advance widths scaled to particular pixel sizes. This allows the font manager to build integer width tables without calling the scaler for each glyph. Typically this table contains only selected screen sizes. This table is sorted by pixel size. The checksum for this table applies to both subtables listed.

Note that for non-square pixel grids (for example, on an EGA), the character width (in pixels) will be used to determine which device record to use. For example, a 12 point character on an EGA (resolution of 72x96) would be 12 pixels high, and 16 pixels wide, and the hdmx device record for 16 pixel characters would be used.

If bit 4 of the flag field in the 'head' table is not set, then it is assumed that the font scales linearly; in this case an 'hdmx' table is not necessary and should not be built. If bit 4 of the flag field is set, then one or more glyphs in the font are assumed to scale nonlinearly. In this case, performance can be improved by including the 'hdmx' table with one or more important DeviceRecord's for important sizes. Please see the chapter "Recommendations for Windows Fonts" for more detail.

The table begins as follows:

Type	Description
USHORT	Table version number (starts at 0)
SHORT	Number of device records.
LONG	Size of a device record, long aligned.
DeviceRecord	Records[number of device records].

Each DeviceRecord for format 0 looks like this.

Type	Description
BYTE	Pixel size for following widths (as ppem).
BYTE	Maximum width.
BYTE	Widths[numGlyphs] (numGlyphs is from the 'maxp' table).

Each DeviceRecord is padded with 0's to make it long word aligned.

Each Width value is the width of the particular glyph, in pixels, at the pixels per em (ppem) size listed at the start of the DeviceRecord.

The ppem sizes are measured along the y axis.

head - Font Header

This table gives global information about the font. The bounding box values should be computed using *only* glyphs that have contours. Glyphs with no contours should be ignored for the purposes of these calculations.

Type	Name	Description
FIXED	Table version number	0x00010000 for version 1.0.
FIXED	fontRevision	Set by font manufacturer.
ULONG	checkSumAdjustment	To compute: set it to 0, sum the entire font as ULONG, then store 0xB1B0AFBA - sum.
ULONG	magicNumber	Set to 0x5F0F3CF5.
USHORT	flags	Bit 0 - baseline for font at y=0; Bit 1 - left sidebearing at x=0; Bit 2 - instructions may depend on point size; Bit 3 - force ppem to integer values for all internal scaler math; may use fractional ppem sizes if this bit is clear; Bit 4 - instructions may alter advance width (the advance widths might not scale linearly); Note: All other bits must be zero.
USHORT	unitsPerEm	Valid range is from 16 to 16384
longDateTime	created	International date (8-byte field).
longDateTime	modified	International date (8-byte field).
FWORD	xMin	For all glyph bounding boxes.
FWORD	yMin	For all glyph bounding boxes.
FWORD	xMax	For all glyph bounding boxes.
FWORD	yMax	For all glyph bounding boxes.
USHORT	macStyle	Bit 0 bold (if set to 1); Bit 1 italic (if set to 1) Bits 2-15 reserved (set to 0).
USHORT	lowestRecPPEM	Smallest readable size in pixels.
SHORT	fontDirectionHint	0 Fully mixed directional glyphs; 1 Only strongly left to right; 2 Like 1 but also contains neutrals[1]; -1 Only strongly right to left; -2 Like -1 but also contains neutrals.
SHORT	indexToLocFormat	0 for short offsets, 1 for long.
SHORT	glyphDataFormat	0 for current format.

[1] A neutral character has no inherent directionality; it is not a character with zero (0) width. Spaces and punctuation are examples of neutral characters. Non-neutral characters are those with inherent directionality. For example, Roman letters (left-to-right) and Arabic letters (right-to-left) have directionality. In a “normal” Roman font where spaces and punctuation are present, the font direction hints should be set to two (2).

Note that macStyle bits must agree with the ‘OS/2’ table fsSelection bits. The fsSelection bits are used over the macStyle bits in Microsoft Windows. The PANOSE values and ‘post’ table values are ignored for determining bold or italic fonts.

The Date format used in this table follows the Macintosh convention of the number of seconds since 1904 (see Apple’s *Inside Macintosh* series).

hhea - Horizontal Header

This table contains information for horizontal layout. The values in the minRightSidebearing, minLeftSideBearing and xMaxExtent should be computed using *only* glyphs that have contours. Glyphs with no contours should be ignored for the purposes of these calculations. All reserved areas must be set to 0.

Type	Name	Description
FIXED	Table version number	0x00010000 for version 1.0.
FWORD	Ascender	Typographic ascent.
FWORD	Descender	Typographic descent.
FWORD	LineGap	Typographic line gap. Negative LineGap values are treated as zero in Windows 3.1, System 6, and System 7.
UFWORD	advanceWidthMax	Maximum advance width value in 'hmtx' table.
FWORD	minLeftSideBearing	Minimum left sidebearing value in 'hmtx' table.
FWORD	minRightSideBearing	Minimum right sidebearing value; calculated as Min(aw - lsb - (xMax - xMin)).
FWORD	xMaxExtent	Max(lsb + (xMax - xMin)).
SHORT	caretSlopeRise	Used to calculate the slope of the cursor (rise/run); 1 for vertical.
SHORT	caretSlopeRun	0 for vertical.
SHORT	(reserved)	set to 0
SHORT	(reserved)	set to 0
SHORT	(reserved)	set to 0
SHORT	(reserved)	set to 0
SHORT	(reserved)	set to 0
SHORT	metricDataFormat	0 for current format.
USHORT	numberOfHMetrics	Number of hMetric entries in 'hmtx' table; may be smaller than the total number of glyphs in the font.

hmtx - Horizontal Metrics

The type longHorMetric is defined as an array where each element has two parts: the advance width, which is of type uFWord, and the left side bearing, which is of type FWord. Or, more formally:

```
typedef struct    _longHorMertric {
    uFWord advanceWidth;
    FWord  lsb;

}  longHorMetric;
```

Field	Type	Description
hMetrics	longHorMetric [numberOfHMetrics]	Paired advance width and left side bearing values for each glyph. The value numOfHMetrics comes from the 'hhea' table. If the font is monospaced, only one entry need be in the array, but that entry is required. The last entry applies to all subsequent glyphs.
leftSideBearing	FWord[ ]	Here the advanceWidth is assumed to be the same as the advanceWidth for the last entry above. The number of entries in this array is derived from numGlyphs (from 'maxp' table) minus numberOfHMetrics. This generally is used with a run of monospaced glyphs (e.g., Kanji fonts or Courier fonts). Only one run is allowed and it must be at the end. This allows a monospaced font to vary the left side bearing values for each glyph.

For any glyph, xmax and xmin are given in 'glyf' table, lsb and aw are given in 'hmtx' table. rsb is calculated as follows:

```
rsb = aw - (lsb + xmax - xmin)
```

If pp1 and pp2 are phantom points used to control lsb and rsb, their initial position in x is calculated as follows:

```
pp1 = xmin - lsb          pp2 = pp1 + aw
```

kern- Kerning

The kerning table contains the values that control the intercharacter spacing for the glyphs in a font. There is currently no system level support for kerning (other than returning the kern pairs and kern values).

Each subtable varies in format, and can contain information for vertical or horizontal text, and can contain kerning values or minimum values. Kerning values are used to adjust inter-character spacing, and minimum values are used to limit the amount of adjustment that the scaler applies by the combination of kerning and tracking. Because the adjustments are additive, the order of the subtables containing kerning values is not important. However, tables containing minimum values should usually be placed last, so that they can be used to limit the total effect of other subtables.

The kerning table in the TrueType font file has a header, which contains the format number and the number of subtables present, and the subtables themselves.

Type	Field	Description
USHORT	version	Table version number (starts at 0)
USHORT	nTables	Number of subtables in the kerning table.

Kerning subtables will share the same header format. This header is used to identify the format of the subtable and the kind of information it contains:

Type	Field	Description
USHORT	version	Kern subtable version number
USHORT	length	Length of the subtable, in bytes (including this header).
USHORT	coverage	What type of information is contained in this table.

The coverage field is divided into the following sub-fields, with sizes given in bits:

Sub-field	Bits #'s	Size	Description
horizontal	0	1	1 if table has horizontal data, 0 if vertical.
minimum	1	1	If this bit is set to 1, the table has minimum values. If set to 0, the table has kerning values.
cross-stream	2	1	If set to 1, kerning is perpendicular to the flow of the text. If the text is normally written horizontally, kerning will be done in the up and down directions. If kerning values are positive, the text will be kerned upwards; if they are negative, the text will be kerned downwards. If the text is normally written vertically, kerning will be done in the left and right directions. If kerning values are positive, the text will be kerned to the right; if they are negative, the text will be kerned to the left. The value 0x8000 in the kerning data resets the cross-stream kerning back to 0.
override	3	1	If this bit is set to 1 the value in this table should replace the value currently being accumulated.
reserved1	4-7	4	Reserved. This should be set to zero.
format	8-15	8	Format of the subtable. Only formats 0 and 2 have been defined. Formats 1 and 3 through 255 are reserved for future use.

Format 0

This is the only format that will be properly interpreted by Windows and OS/2.

This subtable is a sorted list of kerning pairs and values. The list is preceded by information which makes it possible to make an efficient binary search of the list:

Type	Field	Description
USHORT	nPairs	This gives the number of kerning pairs in the table.
USHORT	searchRange	The largest power of two less than or equal to the value of nPairs, multiplied by the size in bytes of an entry in the table.
USHORT	entrySelector	This is calculated as $\log_2$ of the largest power of two less than or equal to the value of nPairs. This value indicates how many iterations of the search loop will have to be made. (For example, in a list of eight items, there would have to be three iterations of the loop).
USHORT	rangeShift	The value of nPairs minus the largest power of two less than or equal to nPairs, and then multiplied by the size in bytes of an entry in the table.

This is followed by the list of kerning pairs and values. Each has the following format:

Type	Field	Description
USHORT	left	The glyph index for the left-hand glyph in the kerning pair.
USHORT	right	The glyph index for the right-hand glyph in the kerning pair.
FWORD	value	The kerning value for the above pair, in FUnits. If this value is greater than zero, the characters will be moved apart. If this value is less than zero, the character will be moved closer together.

The left and right halves of the kerning pair make an unsigned 32-bit number, which is then used to order the kerning pairs numerically.

A binary search is most efficiently coded if the search range is a power of two. The search range can be reduced by half by shifting instead of dividing. In general, the number of kerning pairs, nPairs, will not be a power of two. The value of the search range, searchRange, should be the largest power of two less than or equal to nPairs. The number of pairs not covered by searchRange (that is, nPairs – searchRange) is the value rangeShift.

Windows v3.1 does not make use of the 'kern' data other than to expose it to applications through the GetFontData() API.

Format 2

This subtable is a two-dimensional array of kerning values. The glyphs are mapped to classes, using a different mapping for left- and right-hand glyphs. This allows glyphs that have similar right- or left-side shapes to be handled together. Each similar right- or left-hand shape is said to be single class.

Each row in the kerning array represents one left-hand glyph class, each column represents one right-hand glyph class, and each cell contains a kerning value. Row and column 0 always represent glyphs that do not kern and contain all zeros.

The values in the right class table are stored pre-multiplied by the number of bytes in a single kerning value, and the values in the left class table are stored pre-multiplied by the number of bytes in one row. This eliminates needing to multiply the row and column values together to determine the location of the kerning value. The array can be indexed by doing the right- and left-hand class mappings, adding the class values to the address of the array, and fetching the kerning value to which the new address points.

The header for the simple array has the following format:

Type	Field	Description
USHORT	rowWidth	The width, in bytes, of a row in the table.
USHORT	leftClassTable	Offset from beginning of this subtable to left-hand class table.
USHORT	rightClassTable	Offset from beginning of this subtable to right-hand class table.
USHORT	array	Offset from beginning of this subtable to the start of the kerning array.

Each class table has the following header:

Type	Field	Description
USHORT	firstGlyph	First glyph in class range.
USHORT	nGlyphs	Number of glyph in class range.

This header is followed by nGlyphs number of class values, which are in USHORT format. Entries for glyphs that don't participate in kerning should point to the row or column at position zero.

The array itself is a left by right array of kerning values, which are FWords, where left is the number of left-hand classes and R is the number of right-hand classes. The array is stored by row.

Note that this format is the quickest to process since each lookup requires only a few index operations. The table can be quite large since it will contain the number of cells equal to the product of the number of right-hand classes and the number of left-hand classes, even though many of these classes do not kern with each other.

loca - Index to Location

The indexToLoc table stores the offsets to the locations of the glyphs in the font, relative to the beginning of the glyphData table. In order to compute the length of the last glyph element, there is an extra entry after the last valid index.

By definition, index zero points to the “missing character,” which is the character that appears if a character is not found in the font. The missing character is commonly represented by a blank box (such as) or a space. If the font does not contain an outline for the missing character, then the first and second offsets should have the same value. This also applies to any other character without an outline, such as the space character.

Most routines will look at the ‘maxp’ table to determine the number of glyphs in the font, but the value in the ‘loca’ table should agree.

There are two versions of this table, the short and the long. The version is specified in the indexToLocFormat entry in the ‘head’ table.

Short version

Type	Name	Description
USHORT	offsets[*n*]	The actual local offset divided by 2 is stored. The value of *n* is numGlyphs + 1. The value for numGlyphs is found in the ‘maxp’ table.

Long version

Type	Name	Description
ULONG	offsets[*n*]	The actual local offset is stored. The value of *n* is numGlyphs + 1. The value for numGlyphs is found in the ‘maxp’ table.

Note that the local offsets should be long-aligned, i.e., multiples of 4. Offsets which are not long-aligned may seriously degrade performance of some processors.

LTSH - Linear Threshold

There are noticeable improvements to fonts on the screen when instructions are carefully applied to the sidebearings. The gain in readability is offset by the necessity for the OS to grid fit the glyphs in order to find the actual advance width for the glyphs (since instructions may be moving the sidebearing points). TrueType already has one mechanism to side step the speed issues: the 'hdmx' table, where precomputed advance widths may be saved for selected ppem sizes. The 'LTSH' table (Linear ThreSHold) is a second, complementary method.

The LTSH table defines the point at which it is reasonable to assume linearly scaled advance widths on a glyph-by-glyph basis. This table should *not* be included unless bit 4 of the "flags" field in the 'head' table is set. The criteria for linear scaling is:

> a. (ppem size is ≥ 50) AND (difference between the rounded linear width and the rounded instructed width ≤ 2% of the rounded linear width)
>
> or b. Linear width == Instructed width

The LTSH table records the ppem for each glyph at which the scaling becomes linear again, despite instructions effecting the advance width. It is a requirement that, at and above the recorded threshold size, the glyph remain linear in its scaling (i.e., not legal to set threshold at 55 ppem if glyph becomes non-linear again at 90 ppem). The format for the table is:

Type	Name	Description
USHORT	version	Version number (starts at 0).
USHORT	numGlyphs	Number of glyphs (from "numGlyphs" in 'maxp' table).
BYTE	yPels[numGlyphs]	The vertical pel height at which the glyph can be assumed to scale linearly. On a per glyph basis.

Note that glyphs which do not have instructions on their sidebearings should have yPels = 1; i.e., always scales linearly.

maxp - Maximum Profile

This table establishes the memory requirements for this font.

Type	Name	Description
Fixed	Table version number	0x00010000 for version 1.0.
USHORT	numGlyphs	The number of glyphs in the font.
USHORT	maxPoints	Maximum points in a non-composite glyph.
USHORT	maxContours	Maximum contours in a non-composite glyph.
USHORT	maxCompositePoints	Maximum points in a composite glyph.
USHORT	maxCompositeContours	Maximum contours in a composite glyph.
USHORT	maxZones	1 if instructions do not use the twilight zone (Z0), or 2 if instructions do use Z0; should be set to 2 in most cases.
USHORT	maxTwilightPoints	Maximum points used in Z0.
USHORT	maxStorage	Number of Storage Area locations.
USHORT	maxFunctionDefs	Number of FDEFs.
USHORT	maxInstructionDefs	Number of IDEFs.
USHORT	maxStackElements	Maximum stack depth[2].
USHORT	maxSizeOfInstructions	Maximum byte count for glyph instructions.
USHORT	maxComponentElements	Maximum number of components referenced at “top level” for any composite glyph.
USHORT	maxComponentDepth	Maximum levels of recursion; 1 for simple components.

2 This includes Font and CVT Programs, as well as the instructions for each glyph.

name - Naming Table

The naming table allows multilingual strings to be associated with the TrueType font file. These strings can represent copyright notices, font names, family names, style names, and so on. To keep this table short, the font manufacturer may wish to make a limited set of entries in some small set of languages; later, the font can be "localized" and the strings translated or added. Other parts of the TrueType font file that require these strings can then refer to them simply by their index number. Clients that need a particular string can look it up by its platform ID, character encoding ID, language ID and name ID. Note that some platforms may require single byte character strings, while others may require double byte strings.

The Naming Table is organized as follows:

Type	Description
USHORT	Format selector (=0).
USHORT	Number of NameRecords that follow *n*.
USHORT	Offset to start of string storage (from start of table).
n NameRecords	The NameRecords.
(Variable)	Storage for the actual string data.

Each **NameRecord** looks like this:

Type	Description
USHORT	Platform ID.
USHORT	Platform-specific encoding ID.
USHORT	Language ID.
USHORT	Name ID.
USHORT	String length (in bytes).
USHORT	String offset from start of storage area (in bytes).

Following are the descriptions of the four kinds of ID. Note that the specific values listed here are the only ones that are predefined; new ones may be added by registry with Apple Developer Technical Support. Similar to the character encoding table, the NameRecords is sorted by platform ID, then platform-specific ID, then language ID, and then by name ID.

Platform ID

ID	Platform	Specific encoding
0	Apple Unicode	none
1	Macintosh	Script manager code
2	ISO	ISO encoding
3	*Microsoft*	Microsoft encoding

The values 240 through 255 are reserved for user-defined platforms. The DTS registry will never assign these values to a registered platform.

***Microsoft* platform-specific encoding ID's (platform ID = 3)**

Code	Description
0	Undefined character set or indexing scheme
1	UGL character set with Unicode indexing scheme (see chapter, "Character Sets.")

When building a Unicode font for Windows, the platform ID should be 3 and the encoding ID should be 1. When building a symbol font for Windows, the platform ID should be 3 and the encoding ID should be 0. When building a font that will be used on the Macintosh, the platform ID should be 1 and the encoding ID should be 0.

The PanEuropean Windows product will contain locale data for the following locales. This is also the list from which the user may choose a locale in custom setup and the mapping to Windows and DOS codepages based on that choice. The language ID (LCID in the table below) refers to a value which identifies the language in which a particular string is written.

Primary Language	Locale Name	LCID	Win CP	DOS CP
Albanian	Albania	(041c; SQI)		
Basque	Basque	(042D; EUQ)	1252	850
Byelorussian	Byelorussia	(0423, BEL)	1251	866
Bulgarian	Bulgaria	(0402, BGR)	1251	866
Catalan	Catalan	(0403; CAT)	1252	850
Croatian	Croatian	(041a, SHL)	1250	852
Czech	Czech	(0405; CSY)	1250	852
Danish	Danish	(0406; DAN)	1252	865
Dutch (2):	Dutch (Standard)	(0413; NLD)	1252	850
Dutch (2):	Belgian (Flemish)	(0813; NLB)	1252	850
English (6):	American	(0409; ENU)	1252	437
English (6):	British	(0809; ENG)	1252	850
English (6):	Australian	(0c09; ENA)	1252	850
English (6):	Canadian	(1009; ENC)	1252	850
English (6):	New Zealand	(1409; ENZ)	1252	850
English (6):	Ireland	(1809; ENI)	1252	850
Estonian	Estonia	(0425, ETI)	1257	775
Finnish	Finnish	(040b; FIN)	1252	850
French	French (Standard)	(040c; FRA)	1252	850

Primary Language	Locale Name	LCID	Win CP	DOS CP
French	Belgian	(080c; FRB)	1252	850
French	Canadian	(0c0c; FRC)	1252	850
French	Swiss	(100c; FRS)	1252	850
French	Luxembourg	(140c; FRL)	1252	850
German	German (Standard)	(0407; DEU)	1252	850
German	Swiss	(0807; DES)	1252	850
German	Austrian	(0c07; DEA)	1252	850
German	Luxembourg	(1007; DEL)	1252	850
German	Liechtenstein	(1407; DEC)	1252	850
Greek	Greek	(0408; ELL)	1253	737 or 869[3]
Hungarian	Hungarian	(040e; HUN)	1250	852
Icelandic	Icelandic	(040F; ISL)	1252	850
Italian (2):	Italian (Standard)	(0410; ITA)	1252	850
Italian (2):	Swiss	(0810; ITS)	1252	850
Latvian	Latvia	(0426, LVI)	1257	775
Lithuanian	Lithuania	(0427, LTH)	1257	775
Norwegian (2):	Norwegian (Bokmal)	(0414; NOR)	1252	850
Norwegian (2):	Norwegian (Nynorsk)	(0814; NON)	1252	850
Polish	Polish	(0415; PLK)	1250	852
Portuguese (2):	Portuguese (Brazilian)	(0416; PTB)	1252	850
Portuguese (2):	Portuguese (Standard)	(0816; PTG)	1252	850
Romanian (2):	Romania	(0418, ROM)	1250	852
Russian	Russian	(0419; RUS)	1251	866
Slovak	Slovak	(041b; SKY)	1250	852
Slovenian	Slovenia	(0424, SLV)	1250	852
Spanish (3):	Spanish (Traditional Sort)	(040a; ESP)	1252	850
Spanish (3):	Mexican	(080a; ESM)	1252	850
Spanish (3):	Spanish (Modern Sort)	(0c0a; ESN)	1252	850

[3] 737 is default, but 869 (IBM Greek) will be available at setup time through the selection of a bogus Greek locale in Custom setup.

Primary Language	Locale Name	LCID	Win CP	DOS CP
Swedish	Swedish	(041D; SVE)	1252	850
Turkish	Turkish	(041f; TRK)	1254	857
Ukrainian	Ukraine	(0422, UKR)	1251	866

Macintosh **platform-specific encoding ID's (script manager codes) (platform ID = 1)**

Code	Script	Code	Script
0	Roman	17	Malayalam
1	Japanese	18	Sinhalese
2	Chinese	19	Burmese
3	Korean	20	Khmer
4	Arabic	21	Thai
5	Hebrew	22	Laotian
6	Greek	23	Georgian
7	Russian	24	Armenian
8	RSymbol	25	Maldivian
9	Devanagari	26	Tibetan
10	Gurmukhi	27	Mongolian
11	Gujarati	28	Geez
12	Oriya	29	Slavic
13	Bengali	30	Vietnamese
14	Tamil	31	Sindhi
15	Telugu	32	Uninterp
16	Kannada		

Macintosh language ID's:

Code	Language	Code	Language
0	English	12	Arabic
1	French	13	Finnish
2	German	14	Greek
3	Italian	15	Icelandic
4	Dutch	16	Maltese
5	Swedish	17	Turkish
6	Spanish	18	Yugoslavian
7	Danish	19	Chinese
8	Portuguese	20	Urdu
9	Norwegian	21	Hindi
10	Hebrew	22	Thai
11	Japanese		

***ISO* specific encodings (platform ID = 2)**

Code	ISO encoding
0	7-bit ASCII
1	ISO 10646
2	ISO 8859-1

There are not any ISO-specific language ID's.

The following *name ID's* are defined, and they apply to all platforms. Extensions to this table will be registered with Apple DTS.

Name ID's

Code	Meaning
0	Copyright notice.
1	Font Family name
2	Font Subfamily name; for purposes of definition, this is assumed to address style (italic, oblique) and weight (light, bold, black, etc.) *only*. A font with no particular differences in weight or style (e.g. medium weight, not italic and fsSelection bit 6 set) should have the string "Regular" stored in this position.
3	Unique font identifier
4	Full font name; this should simply be a combination of strings 1 and 2. Exception: if string 2 is "Regular," then use only string 1. This is the font name that Windows will expose to users.
5	Version string. In n.nn format.
6	Postscript name for the font.
7	Trademark; this is used to save any trademark notice/information for this font. Such information should be based on legal advice. This is *distinctly* separate from the copyright.

Note that while both Apple and Microsoft support the same set of name strings, the interpretations may be somewhat different. But since name strings are stored by platform, encoding and language (placing separate strings in for both Apple and MS platforms), this should not present a problem.

The key information for this table for MS fonts relates to the use of strings 1, 2 and 4. Some examples:

Helvetica Narrow Oblique	1 = Helvetica Narrow 2 = Oblique 4 = Helvetica Narrow Oblique
Helvetica Narrow	1 = Helvetica Narrow 2 = Regular 4 = Helvetica Narrow
Helvetica Narrow Light Italic	1 = Helvetica Narrow 2 = Light Italic 4 = Helvetica Narrow Light Italic

Note that OS/2 and Windows both require that all name strings be defined in Unicode. Thus all 'name' table strings for platform ID = 3 (Microsoft) will require two bytes per character. See the chapter, "Character Sets," for a list of the current Unicode character codes supported by Microsoft. Macintosh fonts require single byte strings.

Examples of how these strings might be defined:

0 The copyright string from the font vendor.
© Copyright the Monotype Corporation plc, 1990

1 The name the user sees.
Times New Roman

2 The name of the style.
Bold

3 A unique identifier that applications can store to identify the font being used.
Monotype: Times New Roman Bold:1990

4 The complete, hopefully unique, human readable name of the font. This name is used by Windows.
Times New Roman Bold

5 Release and version information from the font vendor.
June 1, 1990; 1.00, initial release

6 The name the font will be known by on a PostScript printer.
TimesNewRoman-Bold

7 Trademark string,
Times New Roman is a registered trademark of the Monotype Corporation.

OS/2 - OS/2 and Windows Metrics

The OS/2 table consists of a set of metrics that are required by Windows and OS/2. The layout of this table is as follows:

Type	Name of Entry	Comments
USHORT	version	0x0001
SHORT	xAvgCharWidth;	
USHORT	usWeightClass;	
USHORT	usWidthClass;	
SHORT	fsType;	
SHORT	ySubscriptXSize;	
SHORT	ySubscriptYSize;	
SHORT	ySubscriptXOffset;	
SHORT	ySubscriptYOffset;	
SHORT	ySuperscriptXSize;	
SHORT	ySuperscriptYSize;	
SHORT	ySuperscriptXOffset;	
SHORT	ySuperscriptYOffset;	
SHORT	yStrikeoutSize;	
SHORT	yStrikeoutPosition;	
SHORT	sFamilyClass;	
PANOSE	panose;	
ULONG	ulUnicodeRange1	Bits 0–31
ULONG	ulUnicodeRange2	Bits 32–63
ULONG	ulUnicodeRange3	Bits 64–95
ULONG	ulUnicodeRange4	Bits 96–127
CHAR	achVendID[4];	
USHORT	fsSelection;	
USHORT	usFirstCharIndex	
USHORT	usLastCharIndex	
USHORT	sTypoAscender	
USHORT	sTypoDescender	
USHORT	sTypoLineGap	
USHORT	usWinAscent	
USHORT	usWinDescent	
ULONG	ulCodePageRange1	Bits 0-31
ULONG	ulCodePageRange2	Bits 32-63

version

Format: 2-byte unsigned short

Units: n/a

Title: OS/2 table version number.

Description: The version number for this OS/2 table.

Comments: The version number allows for identification of the precise contents and layout for the OS/2 table. The version number for this layout is one (1). The version number for the previous layout (in rev.1.5 of this spec and earlier) was zero (0). Version 0 of the OS/2 table was 78 bytes; Version 1 is 86 bytes, having added the ulCodePageRange1 and ulCodePageRange2 fields.

xAvgCharWidth

Format: 2-byte signed short

Units: Pels / em units

Title: Average weighted escapement.

Description: The Average Character Width parameter specifies the arithmetic average of the escapement (width) of all of the 26 lowercase letters a through z of the Latin alphabet and the space character. If any of the 26 lowercase letters are not present, this parameter should equal the weighted average of *all* glyphs in the font. For non-UGL (platform 3, encoding 0) fonts, use the unweighted average.

Comments: This parameter is a descriptive attribute of the font that specifies the spacing of characters for comparing one font to another for selection or substitution. For proportionally spaced fonts, this value is useful in estimating the length for lines of text. The weighting factors provided with this example are only valid for Latin lowercase letters. If other character sets, or capital letters are used, different frequency of use values should be used. One needs to be careful when comparing fonts that use different frequency of use values for font mapping. The average character width is calculated according to this formula: For the lowercase letters only, sum the individual character widths multiplied by the following weighting factors and then divide by 1000. For example:

Letter	Weight Factor
a	64
b	14
c	27
d	35
e	100
f	20
g	14
h	42
i	63
j	3
k	6
l	35
m	20
n	56
o	56
p	17
q	4
r	49
s	56
t	71
u	31
v	10
w	18
x	3
y	18
z	2
space	166

usWeightClass

Format: 2-byte unsigned short

Title: Weight class.

Description: Indicates the visual weight (degree of blackness or thickness of strokes) of the characters in the font.

Comments:

Value	Description	C Definition (from windows.h)
100	Thin	FW_THIN
200	Extra-light (Ultra-light)	FW_EXTRALIGHT
300	Light	FW_LIGHT
400	Normal (Regular)	FW_NORMAL
500	Medium	FW_MEDIUM
600	Semi-bold (Demi-bold)	FW_SEMIBOLD
700	Bold	FW_BOLD
800	Extra-Bold (Ultra-bold)	FW_EXTRABOLD
900	Black (Heavy)	FW_BLACK

usWidthClass

Format: 2-byte unsigned short

Title: Width class.

Description: Indicates a relative change from the normal aspect ratio (width to height ratio) as specified by a font designer for the glyphs in a font.

Comments:

Value	Description	C Definition	% of normal
1	Ultra-condensed	FWIDTH_ULTRA_CONDENSED	50
2	Extra-condensed	FWIDTH_EXTRA_CONDENSED	62.5
3	Condensed	FWIDTH_CONDENSED	75
4	Semi-condensed	FWIDTH_SEMI_CONDENSED	87.5
5	Medium (normal)	FWIDTH_NORMAL	100
6	Semi-expanded	FWIDTH_SEMI_EXPANDED	112.5
7	Expanded	FWIDTH_EXPANDED	125
8	Extra-expanded	FWIDTH_EXTRA_EXPANDED	150
9	Ultra-expanded	FWIDTH_ULTRA_EXPANDED	200

Although every character in a font may have a different numeric aspect ratio, each character in a font of normal width has a relative aspect ratio of one. When a new type style is created of a different width class (either by a font designer or by some automated means) the relative aspect ratio of the characters in the new font is some percentage greater or less than those same characters in the normal font -- it is this difference that this parameter specifies.

fsType

Format: 2-byte unsigned short

Title: Type flags.

Description: Indicates font embedding licensing rights for the font. Embeddable fonts may be stored in a document. When a document with embedded fonts is opened on a system that does not have the font installed (the remote system), the embedded font may be loaded for temporary (and in some cases, permanent) use on that system by an embedding-aware application. Embedding licensing rights are granted by the vendor of the font.

The TrueType Font Embedding DLL Specification and DLL release notes describe the APIs used to implement support for TrueType font embedding and loading. ***Applications that implement support for font embedding, either through use of the Font Embedding DLL or through other means, must not embed fonts which are not licensed to permit embedding. Further, applications loading embedded fonts for temporary use (see Preview & Print and Editable embedding below) must delete the fonts when the document containing the embedded font is closed.***

Bit	Bit Mask	Description
0		Reserved, must be zero.
1	0x0002	Restricted License embedding: When *only* this bit is set, this font may not be embedded, copied or modified.
2	0x0004	Preview & Print embedding: When this bit is set, the font may be embedded, and temporarily loaded on the remote system. Documents containing Preview & Print fonts must be opened "read-only;" no edits can be applied to the document.
3	0x0008	Editable embedding: When this bit is set, the font may be embedded and temporarily loaded on other systems. Documents containing Editable fonts *may* be opened for reading and writing.
4-15		Reserved, must be zero.

Comments: If multiple embedding bits are set, the *least* restrictive license granted takes precedence. For example, if bits 1 and 3 are set, bit 3 takes precedence over bit 1and the font may be embedded with Editable rights. For compatibility purposes, most vendors granting Editable embedding rights are also setting the Preview & Print bit (0x000C). This will permit an application that only supports Preview & Print embedding to detect that font embedding is allowed.

Restricted License embedding (0x0002): Fonts that have this bit set **must not be modified, embedded or exchanged in any manner** without first obtaining permission of the legal owner. *Caution:* note that for Restricted License embedding to take effect, it must be the only level of embedding selected (as noted in the previous paragraph).

Preview & Print embedding (0x0004): Fonts with this bit set indicate that they may be embedded within documents but must only be installed *temporarily* on the remote system. Any document which includes a Preview & Print embedded font must be opened "read-only;" the application must not allow the user to edit the document; it can only be viewed and/or printed.

Editable embedding (0x0008): Fonts with this bit set indicate that they may be embedded in documents, but must only be installed *temporarily* on the remote system. In contrast to Preview & Print fonts, documents containing Editable fonts may be opened "read-write;" editing is permitted, and changes may be saved.

Installable embedding (0x0000): Fonts with this setting indicate that they may be embedded and permanently installed on the remote system by an application. The user of the remote system acquires the identical rights, obligations and licenses for that font as the original purchaser of the font, and is subject to the same end-user license agreement, copyright, design patent, and/or trademark as was the original purchaser.

ySubscriptXSize

Format:	2-byte signed short
Units:	Font design units
Title:	Subscript horizontal font size.
Description:	The recommended horizontal size in font design units for subscripts for this font.
Comments:	If a font has two recommended sizes for subscripts, e.g., numerics and other, the numeric sizes should be stressed. This size field maps to the em square size of the font being used for a subscript. The horizontal font size specifies a font designer's recommended horizontal font size for subscript characters associated with this font. If a font does not include all of the required subscript characters for an application, and the application can substitute characters by scaling the character of a font or by substituting characters from another font, this parameter specifies the recommended em square for those subscript characters. For example, if the em square for a font is 2048 and ySubScriptXSize is set to 205, then the horizontal size for a simulated subscript character would be 1/10th the size of the normal character.

ySubscriptYSize

Format:	2-byte signed short
Units:	Font design units
Title:	Subscript vertical font size.
Description:	The recommended vertical size in font design units for subscripts for this font.
Comments:	If a font has two recommended sizes for subscripts, e.g. numerics and other, the numeric sizes should be stressed. This size field maps to the emHeight of the font being used for a subscript. The horizontal font size specifies a font designer's recommendation for horizontal font size of subscript characters associated with this font. If a font does not include all of the required subscript characters for an application, and the application can substitute characters by scaling the characters in a font or by substituting characters from another font, this parameter specifies the recommended horizontal EmInc for those subscript characters. For example, if the em square for a font is 2048 and ySubScriptYSize is set to 205, then the vertical size for a simulated subscript character would be 1/10th the size of the normal character.

ySubscriptXOffset

Format: 2-byte signed short

Units: Font design units

Title: Subscript x offset.

Description: The recommended horizontal offset in font design untis for subscripts for this font.

Comments: The Subscript X Offset parameter specifies a font designer's recommended horizontal offset -- from the character origin of the font to the character origin of the subscript's character -- for subscript characters associated with this font. If a font does not include all of the required subscript characters for an application, and the application can substitute characters, this parameter specifies the recommended horizontal position from the character escapement point of the last character before the first subscript character. For upright characters, this value is usually zero; however, if the characters of a font have an incline (italic characters) the reference point for subscript characters is usually adjusted to compensate for the angle of incline.

ySubscriptYOffset

Format: 2-byte signed short

Units: Font design units

Title: Subscript y offset.

Description: The recommended vertical offset in font design units from the baseline for subscripts for this font.

Comments: The Subscript Y Offset parameter specifies a font designer's recommended vertical offset from the character baseline to the character baseline for subscript characters associated with this font. Values are expressed as a positive offset below the character baseline. If a font does not include all of the required subscript for an application, this parameter specifies the recommended vertical distance below the character baseline for those subscript characters.

ySuperscriptXSize

Format: 2-byte signed short

Units: Font design units

Title: Superscript horizontal font size.

Description: The recommended horizontal size in font design units for superscripts for this font.

Comments: If a font has two recommended sizes for subscripts, e.g., numerics and other, the numeric sizes should be stressed. This size field maps to the em square size of the font being used for a subscript. The horizontal font size specifies a font designer's recommended horizontal font size for superscript characters associated with this font. If a font does not include all of the required superscript characters for an application, and the application can substitute characters by scaling the character of a font or by substituting characters from another font, this parameter specifies the recommended em square for those superscript characters.

For example, if the em square for a font is 2048 and ySuperScriptXSize is set to 205, then the horizontal size for a simulated superscript character would be 1/10th the size of the normal character.

ySuperscriptYSize

Format:	2-byte signed short
Units:	Font design units
Title:	Superscript vertical font size.
Description:	The recommended vertical size in font design units for superscripts for this font.
Comments:	If a font has two recommended sizes for subscripts, e.g., numerics and other, the numeric sizes should be stressed. This size field maps to the emHeight of the font being used for a subscript. The vertical font size specifies a font designer's recommended vertical font size for superscript characters associated with this font. If a font does not include all of the required superscript characters for an application, and the application can substitute characters by scaling the character of a font or by substituting characters from another font, this parameter specifies the recommended EmHeight for those superscript characters. For example, if the em square for a font is 2048 and ySuperScriptYSize is set to 205, then the vertical size for a simulated superscript character would be 1/10th the size of the normal character.

ySuperscriptXOffset

Format:	2-byte signed short
Units:	Font design units
Title:	Superscript x offset.
Description:	The recommended horizontal offset in font design units for superscripts for this font.
Comments:	The Superscript X Offset parameter specifies a font designer's recommended horizontal offset -- from the character origin to the superscript character's origin for the superscript characters associated with this font. If a font does not include all of the required superscript characters for an application, this parameter specifies the recommended horizontal position from the escapement point of the character before the first superscript character. For upright characters, this value is usually zero; however, if the characters of a font have an incline (italic characters) the reference point for superscript characters is usually adjusted to compensate for the angle of incline.

ySuperscriptYOffset

Format:	2-byte signed short
Units:	Font design units
Title:	Superscript y offset.
Description:	The recommended vertical offset in font design units from the baseline for superscripts for this font.
Comments:	The Superscript Y Offset parameter specifies a font designer's recommended vertical offset -- from the character baseline to the superscript character's baseline associated with this font. Values for this parameter are expressed as a positive offset above the character baseline. If a font does not include all of the required superscript characters for an application, this parameter specifies the recommended vertical distance above the character baseline for those superscript characters.

yStrikeoutSize

Format:	2-byte signed short
Units:	Font design units
Title:	Strikeout size.
Description:	Width of the strikeout stroke in font design units.
Comments:	This field should normally be the width of the em dash for the current font. If the size is one, the strikeout line will be the line represented by the strikeout position field. If the value is two, the strikeout line will be the line represented by the strikeout position and the line immediately *above* the strikeout position. For a Roman font with a 2048 em square, 102 is suggested.

yStrikeoutPosition

Format:	2-byte signed short
Units:	Font design units
Title:	Strikeout position.
Description:	The position of the strikeout stroke relative to the baseline in font design units.
Comments:	Positive values represent distances above the baseline, while negative values represent distances below the baseline. A value of zero falls directly on the baseline, while a value of one falls one pel above the baseline. The value of strikeout position should not interfere with the recognition of standard characters, and therefore should not line up with crossbars in the font. For a Roman font with a 2048 em square, 530 is suggested.

sFamilyClass

Format:	2-byte signed short
Title:	Font-family class and subclass. Also see section 3.4.
Description:	This parameter is a classification of font-family design.
Comments:	The font class and font subclass are registered values assigned by IBM to each font family. This parameter is intended for use in selecting an alternate font when the requested font is not available. The font class is the most general and the font subclass is the most specific. The high byte of this field contains the family class, while the low byte contains the family subclass.

See Appendix A for full information about this field.

Panose

Format: 10 byte array

Title: PANOSE classification number

International: Additional specifications are required for PANOSE to classify non Latin character sets.

Description: This 10 byte series of numbers are used to describe the visual characteristics of a given typeface. These characteristics are then used to associate the font with other fonts of similar appearance having different names. The variables for each digit are listed below. The specifications for each variable can be obtained in the specification *PANOSE v2.0 Numerical Evaluation* from Microsoft or Elseware Corporation.

Comments: The PANOSE definition contains ten digits each of which currently describes up to sixteen variations. Windows v3.1 uses bFamilyType, bSerifStyle and bProportion in the font mapper to determine family type. It also uses bProportion to determine if the font is monospaced.

Type	Name
BYTE	bFamilyType;
BYTE	bSerifStyle;
BYTE	bWeight;
BYTE	bProportion;
BYTE	bContrast;
BYTE	bStrokeVariation;
BYTE	bArmStyle;
BYTE	bLetterform;
BYTE	bMidline;
BYTE	bXHeight;

1. Family Kind (6 variations)
   - 0 = Any
   - 1 = No Fit
   - 2 = Text and Display
   - 3 = Script
   - 4 = Decorative
   - 5 = Pictorial
2. Serif Style (16 variations)
   - 0 = Any
   - 1 = No Fit
   - 2 = Cove
   - 3 = Obtuse Cove
   - 4 = Square Cove
   - 5 = Obtuse Square Cove
   - 6 = Square

7 = Thin
8 = Bone
9 = Exaggerated
10 = Triangle
11 = Normal Sans
12 = Obtuse Sans
13 = Perp Sans
14 = Flared
15 = Rounded

3. Weight (12 variations)
0 = Any
1 = No Fit
2 = Very Light
3 = Light
4 = Thin
5 = Book
6 = Medium
7 = Demi
8 = Bold
9 = Heavy
10 = Black
11 = Nord

4. Proportion (10 variations)
0 = Any
1 = No Fit
2 = Old Style
3 = Modern
4 = Even Width
5 = Expanded
6 = Condensed
7 = Very Expanded
8 = Very Condensed
9 = Monospaced

5. Contrast (10 variations)
    0 = Any
    1 = No Fit
    2 = None
    3 = Very Low
    4 = Low
    5 = Medium Low
    6 = Medium
    7 = Medium High
    8 = High
    9 = Very High

6. Stroke Variation (9 variations)
    0 = Any
    1 = No Fit
    2 = Gradual/Diagonal
    3 = Gradual/Transitional
    4 = Gradual/Vertical
    5 = Gradual/Horizontal
    6 = Rapid/Vertical
    7 = Rapid/Horizontal
    8 = Instant/Vertical

7. Arm Style (12 variations)
    0 = Any
    1 = No Fit
    2 = Straight Arms/Horizontal
    3 = Straight Arms/Wedge
    4 = Straight Arms/Vertical
    5 = Straight Arms/Single Serif
    6 = Straight Arms/Double Serif
    7 = Non-Straight Arms/Horizontal
    8 = Non-Straight Arms/Wedge
    9 = Non-Straight Arms/Vertical
    10 = Non-Straight Arms/Single Serif
    11 = Non-Straight Arms/Double Serif

8. Letterform (16 variations)
    - 0 = Any
    - 1 = No Fit
    - 2 = Normal/Contact
    - 3 = Normal/Weighted
    - 4 = Normal/Boxed
    - 5 = Normal/Flattened
    - 6 = Normal/Rounded
    - 7 = Normal/Off Center
    - 8 = Normal/Square
    - 9 = Oblique/Contact
    - 10 = Oblique/Weighted
    - 11 = Oblique/Boxed
    - 12 = Oblique/Flattened
    - 13 = Oblique/Rounded
    - 14 = Oblique/Off Center
    - 15 = Oblique/Square

9. Midline (14 variations)
    - 0 = Any
    - 1 = No Fit
    - 2 = Standard/Trimmed
    - 3 = Standard/Pointed
    - 4 = Standard/Serifed
    - 5 = High/Trimmed
    - 6 = High/Pointed
    - 7 = High/Serifed
    - 8 = Constant/Trimmed
    - 9 = Constant/Pointed
    - 10 = Constant/Serifed
    - 11 = Low/Trimmed
    - 12 = Low/Pointed
    - 13 = Low/Serifed

10. X-height (8 variations)
    - 0 = Any
    - 1 = No Fit
    - 2 = Constant/Small
    - 3 = Constant/Standard
    - 4 = Constant/Large
    - 5 = Ducking/Small
    - 6 = Ducking/Standard
    - 7 = Ducking/Large

ulUnicodeRange1 (Bits 0–31)
ulUnicodeRange2 (Bits 32–63)
ulUnicodeRange3 (Bits 64–95)
ulUnicodeRange4 (Bits 96–127)

Format*:* 32-bit unsigned long (4 copies) totaling 128 bits.

Title*:* Unicode Character Range

Description*:* This field is used to specify the Unicode blocks or ranges encompassed by the font file in the 'cmap' subtable for platform 3, encoding ID 1 (Microsoft platform). If the bit is set (1) then the Unicode range is considered functional. If the bit is clear (0) then the range is not considered functional. Each of the bits is treated as an independent flag and the bits can be set in any combination. The determination of "functional" is left up to the font designer, although character set selection should attempt to be functional by ranges if at all possible.

All reserved fields must be zero. Each long is in Big-Endian form. See the Basic Multilingual Plane of ISO/IEC 10646-1 or the Unicode Standard v.1.1 for the list of Unicode ranges and characters.

Bit	Description
0	Basic Latin
1	Latin-1 Supplement
2	Latin Extended-A
3	Latin Extended-B
4	IPA Extensions
5	Spacing Modifier Letters
6	Combining Diacritical Marks
7	Basic Greek
8	Greek Symbols And Coptic
9	Cyrillic
10	Armenian
11	Basic Hebrew
12	Hebrew Extended (A and B blocks combined)
13	Basic Arabic
14	Arabic Extended
15	Devanagari
16	Bengali
17	Gurmukhi
18	Gujarati
19	Oriya
20	Tamil
21	Telugu
22	Kannada
23	Malayalam
24	Thai

Table continued from previous page

Bit	Description
25	Lao
26	Basic Georgian
27	Georgian Extended
28	Hangul Jamo
29	Latin Extended Additional
30	Greek Extended
31	General Punctuation
32	Superscripts And Subscripts
33	Currency Symbols
34	Combining Diacritical Marks For Symbols
35	Letterlike Symbols
36	Number Forms
37	Arrows
38	Mathematical Operators
39	Miscellaneous Technical
40	Control Pictures
41	Optical Character Recognition
42	Enclosed Alphanumerics
43	Box Drawing
44	Block Elements
45	Geometric Shapes
46	Miscellaneous Symbols
47	Dingbats
48	CJK Symbols And Punctuation
49	Hiragana
50	Katakana
51	Bopomofo
52	Hangul Compatibility Jamo
53	CJK Miscellaneous
54	Enclosed CJK Letters And Months
55	CJK Compatibility
56	Hangul
57	Reserved for Unicode SubRanges
58	Reserved for Unicode SubRanges
59	CJK Unified Ideographs
60	Private Use Area
61	CJK Compatibility Ideographs
62	Alphabetic Presentation Forms
63	Arabic Presentation Forms-A
64	Combining Half Marks
65	CJK Compatibility Forms

Table continued from previous page

Bit	Description
66	Small Form Variants
67	Arabic Presentation Forms-B
68	Halfwidth And Fullwidth Forms
69	Specials
70–127	Reserved for Unicode SubRanges

achVendID

Format 4-byte character array

Title: Font Vendor Identification

Description: The four character identifier for the vendor of the given type face.

Comments: This is not the royalty owner of the original artwork. This is the company responsible for the marketing and distribution of the typeface that is being classified. It is reasonable to assume that there will be 6 vendors of ITC Zapf Dingbats for use on desktop platforms in the near future (if not already). It is also likely that the vendors will have other inherent benefits in their fonts (more kern pairs, unregularized data, hand hinted, etc.). This identifier will allow for the correct vendor's type to be used over another, possibly inferior, font file. The Vendor ID value is not required.

Microsoft has assigned values for some font suppliers as listed below. Uppercase vendor ID's are reserved by Microsoft. Other suppliers can choose their own mixed case or lowercase ID's, or leave the field blank.

Vendor ID	Vendor Name
AGFA	AGFA Compugraphic
ADBE	Adobe
APPL	Apple
ALTS	Altsys
B&H	Bigelow & Holmes
BERT	Berthold
BITS	Bitstream
CANO	Canon
CTDL	China Type Design Ltd.
DTC	Digital Typeface Corp.
ELSE	Elseware
EPSN	Epson
GLYF	Glyph Systems
GPI	Gamma Productions, Inc.
HP	Hewlett-Packard
HY	HanYang System
IBM	IBM
IMPR	Impress
KATF	Kingsley/ATF
LANS	Lanston Type Co., Ltd.
LEAF	Interleaf, Inc.
LETR	Letraset

Vendor ID	Vendor Name
LINO	Linotype
LTRX	Lighttracks
MACR	Macromedia
MONO	Monotype
MLGC	Micrologic Software
MS	Microsoft
NEC	NEC
PARA	ParaGraph Intl.
PRFS	Production First Software
QMSI	QMS/Imagen
SFUN	Soft Union
SWFT	Swfte International
TILD	SIA Tilde
URW	URW
ZSFT	ZSoft

fsSelection

Format: 2-byte bit field.
Title: Font selection flags.
Description: Contains information concerning the nature of the font patterns, as follows:

Bit #	macStyle bit	C definition	Description
0	bit 1	ITALIC	Font contains Italic characters, otherwise they are upright.
1		UNDERSCORE	Characters are underscored.
2		NEGATIVE	Characters have their foreground and background reversed.
3		OUTLINED	Outline (hollow) characters, otherwise they are solid.
4		STRIKEOUT	Characters are overstruck.
5	bit 0	BOLD	Characters are emboldened.
6		REGULAR	Characters are in the standard weight/style for the font.

Comments: All undefined bits must be zero.

This field contains information on the original design of the font. Bits 0 & 5 can be used to determine if the font was designed with these features or whether some type of machine simulation was performed on the font to achieve this appearance. Bits 1-4 are rarely used bits that indicate the font is primarily a decorative or special purpose font.

If bit 6 is set, then bits 0 and 5 must be clear, else the behavior is undefined. As noted above, the settings of bits 0 and 1 must be reflected in the macStyle bits in the 'head' table. While bit 6 on implies that bits 0 and 1 of macStyle are clear (along with bits 0 and 5 of fsSelection), the reverse is not true. Bits 0 and 1 of macStyle (and 0 and 5 of fsSelection) may be clear and that does not give any indication of whether or not bit 6 of fsSelection is clear (e.g., Arial Light would have all bits cleared; it is not the regular version of Arial).

usFirstCharIndex

Format: 2-byte USHORT

Description: The minimum Unicode index (character code) in this font, according to the cmap subtable for platform ID 3 and encoding ID 0 or 1. For most fonts supporting Win-ANSI or other character sets, this value would be 0x0020.

usLastCharIndex

Format: 2-byte USHORT

Description: The maximum Unicode index (character code) in this font, according to the cmap subtable for platform ID 3 and encoding ID 0 or 1. This value depends on which character sets the font supports.

sTypoAscender

Format: 2-byte SHORT

Description: The typographic ascender for this font. Remember that this is not the same as the Ascender value in the 'hhea' table, which Apple defines in a far different manner. One good source for usTypoAscender is the Ascender value from an AFM file.

The suggested useage for usTypoAscender is that it be used in conjunction with unitsPerEm to compute a typographically correct default line spacing. The goal is to free applications from Macintosh or Windows-specific metrics which are constrained by backward compatability requirements. These new metrics, when combined with the character design widths, will allow applications to lay out documents in a typographically correct and portable fashion. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table (unless Apple exposes the 'OS/2' table through a new API).

sTypoDescender

Format: 2-byte SHORT

Description: The typographic descender for this font. Remember that this is not the same as the Descender value in the 'hhea' table, which Apple defines in a far different manner. One good source for usTypoDescender is the Descender value from an AFM file.

The suggested useage for usTypoDescender is that it be used in conjunction with unitsPerEm to compute a typographically correct default line spacing. The goal is to free applications from Macintosh or Windows-specific metrics which are constrained by backward compatability requirements. These new metrics, when combined with the character design widths, will allow applications to lay out documents in a typographically correct and portable fashion. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table (unless Apple exposes the 'OS/2' table through a new API).

sTypoLineGap

Format: 2-byte SHORT

Description: The typographic line gap for this font. Remember that this is not the same as the LineGap value in the ‘hhea’ table, which Apple defines in a far different manner.

The suggested useage for usTypoLineGap is that it be used in conjunction with unitsPerEm to compute a typographically correct default line spacing. Typical values average 7-10% of units per em. The goal is to free applications from Macintosh or Windows-specific metrics which are constrained by backward compatability requirements (see chapter, “Recommendations for Windows Fonts). These new metrics, when combined with the character design widths, will allow applications to lay out documents in a typographically correct and portable fashion. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the ‘sfnt’ resource and parse it to extract this data from the “OS/2” table (unless Apple exposes the ‘OS/2’ table through a new API).

usWinAscent

Format: 2-byte USHORT

Description: The ascender metric for Windows. This, too, is distinct from Apple’s Ascender value and from the usTypoAscender values. usWinAscent is computed as the yMax for all characters in the Windows ANSI character set. usTypoAscent is used to compute the Windows font height and default line spacing. For platform 3 encoding 0 fonts, it is the same as yMax.

usWinDescent

Format: 2-byte USHORT

Description: The descender metric for Windows. This, too, is distinct from Apple’s Descender value and from the usTypoDescender values. usWinDescent is computed as the -yMin for all characters in the Windows ANSI character set. usTypoAscent is used to compute the Windows font height and default line spacing. For platform 3 encoding 0 fonts, it is the same as -yMin.

ulCodePageRange1 *Bits 0–31*
ulCodePageRange2 *Bits 32–63*

Format*:* 32-bit unsigned long (2 copies) totaling 64 bits.

Title*:* Code Page Character Range

Description*:* This field is used to specify the code pages encompassed by the font file in the 'cmap' subtable for platform 3, encoding ID 1 (Microsoft platform). If the font file is encoding ID 0, then the Symbol Character Set bit should be set. If the bit is set (1) then the code page is considered functional. If the bit is clear (0) then the code page is not considered functional. Each of the bits is treated as an independent flag and the bits can be set in any combination. The determination of "functional" is left up to the font designer, although character set selection should attempt to be functional by code pages if at all possible.

Symbol character sets have a special meaning. If the symbol bit (31) is set, and the font file contains a 'cmap' subtable for platform of 3 and encoding ID of 1, then all of the characters in the Unicode range 0xF000 - 0xF0FF (inclusive) will be used to enumerate the symbol character set. If the bit is not set, any characters present in that range will not be enumerated as a symbol character set.

All reserved fields must be zero. Each long is in Big-Endian form.

Bit	Code Page	Description
0	1252	Latin 1
1	1250	Latin 2: Eastern Europe
2	1251	Cyrillic
3	1253	Greek
4	1254	Turkish
5	1255	Hebrew
6	1256	Arabic
7	1257	Windows Baltic
8–15		Reserved for Alternate ANSI
16	874	Thai
17	932	JIS/Japan
18	936	Chinese: Simplified chars--PRC and Singapore
19	949	Korean Wansung
20	950	Chinese: Traditional chars--Taiwan and Hong Kong
21	1361	Korean Johab
22–28		Reserved for Alternate ANSI & OEM
29		Macintosh Character Set (US Roman)
30		OEM Character Set
31		Symbol Character Set
32-47		Reserved for OEM
48	869	IBM Greek
49	866	MS-DOS Russian
50	865	MS-DOS Nordic
51	864	Arabic

52	863	MS-DOS Canadian French

Bit	Code Page	Description
53	862	Hebrew
54	861	MS-DOS Icelandic
55	860	MS-DOS Portuguese
56	857	IBM Turkish
57	855	IBM Cyrillic; primarily Russian
58	852	Latin 2
59	775	MS-DOS Baltic
60	737	Greek; former 437 G
61	708	Arabic; ASMO 708
62	850	WE/Latin 1
63	437	US

PCLT - PCL 5 Table

The 'PCLT' table is an optional table that is not used directly by Microsoft Windows v3.1, but it is highly recommended that this table be present in all TrueType font files. Extra information on many of these fields can be found in the *HP PCL 5 Printer Language Technical Reference Manual* available from Hewlett-Packard Boise Printer Division.

The format for the table is:

Type	Name of Entry
FIXED	Version
ULONG	FontNumber
USHORT	Pitch
USHORT	xHeight
USHORT	Style
USHORT	TypeFamily
USHORT	CapHeight
USHORT	SymbolSet
CHAR	Typeface[16]
CHAR	CharacterComplement[8]
CHAR	FileName[6]
CHAR	StrokeWeight
CHAR	WidthType
BYTE	SerifStyle
BYTE	Reserved (pad)

Version

Table version number 1.0 is represented as 0x00010000.

FontNumber

This 32-bit number is segmented in two parts. The most significant bit indicates native versus converted format. Only font vendors should create fonts with this bit zeroed. The 7 next most significant bits are assigned by Hewlett-Packard Boise Printer Division to major font vendors. The least significant 24 bits are assigned by the vendor. Font vendors should attempt to insure that each of their fonts are marked with unique values.

Vendor codes:

A	Adobe Systems
B	Bitstream Inc.
C	Agfa Corporation
H	Bigelow & Holmes
L	Linotype Company
M	Monotype Typography Ltd.

Pitch

The width of the space in FUnits (FUnits are described by the unitsPerEm field of the 'head' table). Monospace fonts derive the width of all characters from this field.

xHeight

The height of the optical line describing the height of the lowercase x in FUnits. This might not be the same as the measured height of the lowercase x.

Style

The most significant 6 bits are reserved. The 5 next most significant bits encode structure. The next 3 most significant bits encode appearance width. The 2 least significant bits encode posture.

Structure (bits 5-9)

0	Solid (normal, black)
1	Outline (hollow)
2	Inline (incised, engraved)
3	Contour, edged (antique, distressed)
4	Solid with shadow
5	Outline with shadow
6	Inline with shadow
7	Contour, or edged, with shadow
8	Pattern filled
9	Pattern filled #1 (when more than one pattern)
10	Pattern filled #2 (when more than two patterns)
11	Pattern filled #3 (when more than three patterns)
12	Pattern filled with shadow
13	Pattern filled with shadow #1 (when more than one pattern or shadow)
14	Pattern filled with shadow #2 (when more than two patterns or shadows)
15	Pattern filled with shadow #3 (when more than three patterns or shadows)
16	Inverse
17	Inverse with border
18-31	reserved

Width (bits 2-4)

0	normal
1	condensed
2	compressed, extra condensed
3	extra compressed
4	ultra compressed
5	reserved
6	expanded, extended
7	extra expanded, extra extended

Posture (bits 0-1)

0	upright
1	oblique, italic
2	alternate italic (backslanted, cursive, swash)
3	reserved

TypeFamily

The 4 most significant bits are font vendor codes. The 12 least significant bits are typeface family codes. Both are assigned by HP Boise Division.

Vendor Codes (bits 12-15)

0	reserved
1	Agfa Corporation
2	Bitstream Inc.
3	Linotype Company
4	Monotype Typography Ltd.
5	Adobe Systems
6	font repackagers
7	vendors of unique typefaces
8-15	reserved

CapHeight

The height of the optical line describing the top of the uppercase H in FUnits. This might not be the same as the measured height of the uppercase H.

SymbolSet

The most significant 11 bits are the value of the symbol set “number” field. The value of the least significant 5 bits, when added to 64, is the ASCII value of the symbol set “ID” field. Symbol set values are assigned by HP Boise Division. Unbound fonts, or “typefaces” should have a symbol set value of 0. See the *PCL 5 Printer Language Technical Reference Manual* or the *PCL 5 Comparison Guide* for the most recent published list of codes.

Examples

	PCL	decimal
Windows 3.1 “ANSI”	19U	629
Windows 3.0 “ANSI”	9U	309
Adobe “Symbol”	19M	621
Macintosh	12J	394
PostScript ISO Latin 1	11J	362
PostScript Std. Encoding	10J	330
Code Page 1004	9J	298
DeskTop	7J	234

TypeFace

This 16-byte ASCII string appears in the “font print” of PCL printers. Care should be taken to insure that the base string for all typefaces of a family are consistent, and that the designators for bold, italic, etc. are standardized.

Example:

```
Times New
Times New       Bd
Times New       It
Times New     BdIt
Courier New
Courier New     Bd
Courier New     It
Courier New   BdIt
```

CharacterComplement

This 8-byte field identifies the symbol collections provided by the font, each bit identifies a symbol collection and is independently interpreted. Symbol set bound fonts should have this field set to all F’s (except bit 0).

Example:

DOS/PCL Complement	0xFFFFFFFF003FFFFE
Windows 3.1 “ANSI”	0xFFFFFFFF37FFFFFE
Macintosh	0xFFFFFFFF36FFFFFE
ISO 8859-1 Latin 1	0xFFFFFFFF3BFFFFFE
ISO 8859-1,2,9 Latin 1,2,5	0xFFFFFFFF0BFFFFFE

The character collections identified by each bit are as follows:

31	ASCII (supports several standard interpretations)
30	Latin 1 extensions
29	Latin 2 extensions
28	Latin 5 extensions
27	Desktop Publishing Extensions
26	Accent Extensions (East and West Europe)
25	PCL Extensions
24	Macintosh Extensions
23	PostScript Extensions
22	Code Page Extensions

The character complement field also indicates the index mechanism used with an unbound font. Bit 0 must always be cleared when the font elements are provided in Unicode order.

FileName

This 6-byte field is composed of 3 parts. The first 3 bytes are an industry standard typeface family string. The fourth byte is a treatment character, such as R, B, I. The last two characters are either zeroes for an unbound font or a two character mnemonic for a symbol set if symbol set found.

Examples:

`TNRR00`	Times New (text weight, upright)
`TNRI00`	Times New Italic
`TNRB00`	Times New Bold
`TNRJ00`	Times New Bold Italic
`COUR00`	Courier
`COUI00`	Courier Italic
`COUB00`	Courier Bold
`COUJ00`	Courier Bold Italic

Treatment Flags:

R	Text, normal, book, etc.
I	Italic, oblique, slanted, etc.
B	Bold
J	Bold Italic, Bold Oblique
D	Demibold
E	Demibold Italic, Demibold Oblique
K	Black
G	Black Italic, Black Oblique
L	Light
P	Light Italic, Light Oblique
C	Condensed
A	Condensed Italic, Condensed Oblique
F	Bold Condensed
H	Bold Condensed Italic, Bold Condensed Oblique
S	Semibold (lighter than demibold)
T	Semibold Italic, Semibold Oblique

other treatment flags are assigned over time.

StrokeWeight

This signed 1-byte field contains the PCL stroke weight value. Only values in the range -7 to 7 are valid:

-7	Ultra Thin
-6	Extra Thin
-5	Thin
-4	Extra Light
-3	Light
-2	Demilight
-1	Semilight
0	Book, text, regular, etc.
1	Semibold (Medium, when darker than Book)
2	Demibold
3	Bold
4	Extra Bold
5	Black
6	Extra Black
7	Ultra Black, or Ultra

Type designers often use interesting names for weights or combinations of weights and styles, such as Heavy, Compact, Inserat, Bold No. 2, etc. PCL stroke weights are assigned on the basis of the entire family and use of the faces. Typically, display faces don’t have a “text” weight assignment.

WidthType

This signed 1-byte field contains the PCL appearance width value. The values are not directly related to those in the appearance with field of the style word above. Only values in the range -5 to 5 are valid.

-5	Ultra Compressed
-4	Extra Compressed
-3	Compressed, or Extra Condensed
-2	Condensed
0	Normal
2	Expanded
3	Extra Expanded

SerifStyle

This signed 1-byte field contains the PCL serif style value. The most significant 2 bits of this byte specify the serif/sans or contrast/monoline characterisitics of the typeface.

Bottom 6 bit values:

0	Sans Serif Square
1	Sans Serif Round
2	Serif Line
3	Serif Triangle
4	Serif Swath
5	Serif Block
6	Serif Bracket
7	Rounded Bracket
8	Flair Serif, Modified Sans
9	Script Nonconnecting
10	Script Joining
11	Script Calligraphic
12	Script Broken Letter

Top 2 bit values:

0	reserved
1	Sans Serif/Monoline
2	Serif/Contrasting
3	reserved

Reserved

Should be set to zero.

post - PostScript

This table contains additional information needed to use TrueType fonts on PostScript printers. This includes data for the FontInfo dictionary entry and the PostScript names of all the glyphs.

The table begins as follows:

Type	Name	Description
FIXED	Format Type	0x00010000 for format 1.0, 0x00020000 for format 2.0, and so on...
FIXED	italicAngle	Italic angle in counter-clockwise degrees from the vertical. Zero for upright text, negative for text that leans to the right (forward)
FWORD	underlinePosition	Suggested values for the underline position (negative values indicate below baseline).
FWORD	underlineThickness	Suggested values for the underline thickness.
ULONG	isFixedPitch	Set to 0 if the font is proportionally spaced, non-zero if the font is not proportionally spaced (i.e. monospaced).
ULONG	minMemType42	Minimum memory usage when a TrueType font is downloaded.
ULONG	maxMemType42	Maximum memory usage when a TrueType font is downloaded.
ULONG	minMemType1	Minimum memory usage when a TrueType font is downloaded as a Type 1 font.
ULONG	maxMemType1	Maximum memory usage when a TrueType font is downloaded as a Type 1 font.

The last four entries in the table are present because PostScript drivers can do better memory management if the virtual memory (VM) requirements of a downloadable TrueType font are known before the font is downloaded. This information should be supplied if known. If it is not known, set the value to zero. The driver will still work but will be less efficient.

Maximum memory usage is minimum memory usage plus maximum runtime memory use. Maximum runtime memory use depends on the maximum band size of any bitmap potentially rasterized by the TrueType font scaler. Runtime memory usage could be calculated by rendering characters at different point sizes and comparing memory use.

How to calculate VM usage

The memory usage of a downloaded TrueType font will vary with whether it is defined as a TrueType or Type 1 font on the printer. Minimum memory usage can be calculated by calling *VMStatus*, downloading the font, and calling *VMStatus* a second time.

If the format is 1.0 or 3.0, the table ends here. The additional entries for formats 2.0 and 2.5 are shown below. Apple has defined a format 4.0 for use with QuickDraw GX, which is described in their documentation.

Format 1.0

This TrueType font file contains exactly the 258 glyphs in the standard Macintosh TrueType font file in the order specified in Appendix C, “Standard Macintosh Character Set to UGL.” As a result, the glyph names are taken from the system with no storage required by the font.

Format 2.0

This is the format required by Microsoft fonts.

Type	Description
USHORT	Number of glyphs (this is the same as numGlyphs in ‘maxp’ table).
USHORT	glyphNameIndex[numGlyphs].
CHAR	Glyph names with length bytes [variable] (a Pascal string).

This TrueType font file contains glyphs not in the standard Macintosh set or the ordering of the glyphs in the TrueType font file is non-standard (again, for the Macintosh). The glyph name array maps the glyphs in this font to name index. If the name index is between 0 and 257, treat the name index as a glyph index in the Macintosh standard order. If the name index is between 258 and 32767, then subtract 258 and use that to index into the list of Pascal strings at the end of the table. Thus a given font may map some of its glyphs to the standard glyph names, and some to its own names.

Index numbers 32768 through 65535 are reserved for future use. If you do not want to associate a PostScript name with a particular glyph, use index number 0 which points the name *.notdef.*

Format 2.5

This format provides a space saving table for fonts which contain a pure subset of, or a simple reordering of, the standard Macintosh glyph set.

Type	Description
CHAR	offset[numGlyphs]

This format is useful for font files that contain only glyphs in the standard Macintosh glyph set but which have those glyphs arranged in a non-standard order or which are missing some glyphs. The table contains one byte for each glyph in the font file. The byte is treated as a signed offset that maps the glyph index used in this font into the standard glyph index. In other words, assuming that the 'sfnt' contains the three glyphs A, B, and C which are the 37th, 38th, and 39th glyphs in the standard ordering, the 'post' table would contain the bytes +36, +36, +36.

Format 3.0

This format makes it possible to create a special font that is not burdened with a large 'post' table set of glyph names.

This format specifies that no PostScript name information is provided for the glyphs in this font file. The printing behavior of this format on PostScript printers is unspecified, except that it should not result in a fatal or unrecoverable error. Some drivers may print nothing, other drivers may attempt to print using a default naming scheme.

*Windows v3.1 makes use of the italic angle value in the 'post' table but does not actually **require** any glyph names to be stored as Pascal strings .*

prep - Control Value Program

The Control Value Program consists of a set of TrueType instructions that will be executed whenever the font or point size or transformation matrix change and before each glyph is interpreted. Any instruction is legal in the CVT Program but since no glyph is associated with it, instructions intended to move points within a particular glyph outline cannot be used in the CVT Program. The name ‘prep’ is anachronistic.

Type	Description
BYTE[ ]	Set of instructions executed whenever point size or font or transformation change

VDMX - Vertical Device Metrics

Under Windows, the usWinAscent and usWinDescent values from the 'OS/2' table will be used to determine the maximum black height for a font at any given size. Windows calls this distance the Font Height. Because TrueType instructions can lead to Font Heights that differ from the actual scaled and rounded values, basing the Font Height strictly on the yMax and yMin can result in "lost pixels." Windows will clip any pixels that extend above the yMax or below the yMin. In order to avoid grid fitting the entire font to determine the correct height, the VDMX table has been defined.

The VDMX table consists of a header followed by groupings of VDMX records:

Type	Name	Description
USHORT	version	Version number (starts at 0).
USHORT	numRecs	Number of VDMX groups present
USHORT	numRatios	Number of aspect ratio groupings
Ratios	ratRange[numRatios]	Ratio ranges (see below for more info)
USHORT	offset[numRatios]	Offset from start of this table to the VDMX group for this ratio range.
Vdmx	groups	The actual VDMX groupings (documented below)

```
struct Ratios {
       BYTE   bCharSet;     /* Character set (see below) */
       BYTE   xRatio;       /* Value to use for x-Ratio */
       BYTE   yStartRatio;  /* Starting y-Ratio value */
       BYTE   yEndRatio     /* Ending y-ratio value */
}
```

Ratios are set up as follows:

For a 1:1 aspect ratio	Ratios.xRatio = 1; Ratios.yStartRatio = 1; Ratios.yEndRatio = 1;
For 1:1 through 2:1 ratio	Ratios.xRatio = 2; Ratios.yStartRatio = 1; Ratios.yEndRatio = 2;
For 1.33:1 ratio	Ratios.xRatio = 4; Ratios.yStartRatio = 3; Ratios.yEndRatio = 3;
For *all* aspect ratios	Ratio.xRatio = 0; Ratio.yStartRatio = 0; Ratio.yEndRatio = 0;

All values set to zero signal the default grouping to use; if present, this must be the *last* Ratio group in the table. Ratios of 2:2 are the same as 1:1.

Aspect ratios are matched against the target device by normalizing the entire ratio range record based on the current X resolution and performing a range check of Y resolutions for each record after normalization. Once a match is found, the search stops. If the 0,0,0 group is encountered during the search, it is used (therefore if this group is not at the end of the ratio groupings, no group that follows it will be used). If there is not a match and there is no 0,0,0 record, then there is no VDMX data for that aspect ratio.

Note that range checks are conceptually performed as follows:

```
(deviceXRatio == Ratio.xRatio) && (deviceYRatio >=
        Ratio.yStartRatio) && (deviceYRatio <= Ratio.yEndRatio)
```

Each ratio grouping refers to a specific VDMX record group; there must be at least 1 VDMX group in the table.

The uCharSet value is used to denote cases where the VDMX group was computed based on a subset of the glyphs present in the font file. The currently defined values for character set are:

uCharSet	Description
0	No subset; the VDMX group applies to all glyphs in the font. This is used for symbol or dingbat fonts.
1	Windows ANSI subset; the VDMX group was computed using only the glyphs required to complete the Windows ANSI character set. Windows will ignore any VDMX entries that are not for the ANSI subset (i.e. uCharSet = 1)

VDMX groups immediately follow the table header. Each set of records (there need only be one set) has the following layout:

Type	Name	Description
USHORT	recs	Number of height records in this group
BYTE	startsz	Starting yPelHeight
BYTE	endsz	Ending yPelHeight
vTable	entry[recs]	The VDMX records

```
struct vTable {
       USHORT yPelHeight;   /* yPelHeight to which values apply */
       SHORT  yMax;         /* yMax (in pels) for this yPelHeight */
       SHORT  yMin;         /* yMin (in pels) for this yPelHeight */
}
```

This table must appear in sorted order (sorted by yPelHeight), but need not be continous. It should have an entry for every pel height where the yMax and yMin do not scale linearly, where linearly scaled heights are defined as:

Hinted yMax and yMin are identical to scaled/rounded yMax and yMin

It is assumed that once yPelHeight reaches 255, all heights will be linear, or at least close enough to linear that it no longer matters. Please note that while the Ratios structure can only support ppem sizes up to 255, the vTable structure can support much larger pel heights (up to 65535). The choice of SHORT and USHORT for vTable is dictated by the requirement that yMax and yMin be signed values (and 127 to -128 is too small a range) and the desire to word-align the vTable elements.

vhea - Vertical Header Table

The vertical header table (tag name: ‘vhea’) contains information needed for vertical fonts. The glyphs of vertical fonts are written either top to bottom or bottom to top. This table contains information that is general to the font as a whole. Information that pertains to specific glyphs is given in the vertical metrics table (tag name: ‘vmtx’) described separately. The formats of these tables are similar to those for horizontal metrics (hhea and hmtx).

Data in the vertical header table must be consistent with data that appears in the vertical metrics table. The advance height and top sidebearing values in the vertical metrics table must correspond with the maximum advance height and minimum bottom sidebearing values in the vertical header table.

The vertical header table format follows:

Vertical Header Table

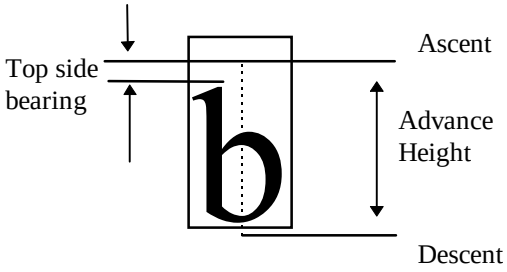
Type	Name	Description
FIXED32	version	Version number of the vertical header table (0x00010000 for the initial version).
SHORT	ascent	Distance in FUnits from the centerline to the previous line's descent.
SHORT	descent	Distance in FUnits from the centerline to the next line's ascent.
SHORT	lineGap	Reserved; set to 0
SHORT	advanceHeightMax	The maximum advance height measurement in FUnits found in the font. This value must be consistent with the entries in the vertical metrics table.
SHORT	minTopSideBearing	The minimum top sidebearing measurement found in the font, in FUnits. This value must be consistent with the entries in the vertical metrics table.
SHORT	minBottomSideBearing	The minimum bottom sidebearing measurement found in the font, in FUnits. This value must be consistent with the entries in the vertical metrics table.
SHORT	yMaxExtent	Defined as yMaxExtent=minTopSideBearing+(yMax-yMin)
SHORT	caretSlopeRise	The value of the caretSlopeRise field divided by the value of the caretSlopeRun Field determines the slope of the caret. A value of 0 for the rise and a value of 1 for the run specifies a horizontal caret. A value of 1 for the rise and a value of 0 for the run specifies a vertical caret. Intermediate values are desirable for fonts whose glyphs are oblique or italic. For a vertical font, a horizontal caret is best.
SHORT	caretSlopeRun	See the caretSlopeRise field. Value=1 for nonslanted vertical fonts.
SHORT	caretOffset	The amount by which the highlight on a slanted glyph needs to be shifted away from the glyph in order to produce the best appearance. Set value equal to 0 for nonslanted fonts.
SHORT	reserved	Set to 0.
SHORT	reserved	Set to 0.
SHORT	reserved	Set to 0.
SHORT	reserved	Set to 0.
SHORT	metricDataFormat	Set to 0.
USHORT	numOfLongVerMetrics	Number of advance heights in the vertical metrics table.

Vertical Header Table Example

Offset/ length	Value	Name	Comment
0/4	0x00010000	version	Version number of the vertical header table, in fixed-point format, is 1.0
4/2	1024	ascent	Half the em-square height.
6/2	-1024	descent	Minus half the em-square height.
8/2	0	lineGap	Typographic line gap is 0 FUnits.
10/2	2079	advanceHeightMax	The maximum advance height measurement found in the font is 2079 FUnits.
12/2	-342	minTopSideBearing	The minimum top sidebearing measurement found in the font is -342 FUnits.
14/2	-333	minBottomSideBearing	The minimum bottom sidebearing measurement found in the font is -333 FUnits.
16/2	2036	yMaxExtent	minTopSideBearing+(yMax-yMin)=2036.
18/2	0	caretSlopeRise	The caret slope rise of 0 and a caret slope run of 1 indicate a horizontal caret for a vertical font.
20/2	1	caretSlopeRun	The caret slope rise of 0 and a caret slope run of 1 indicate a horizontal caret for a vertical font.
22/2	0	caretOffset	Value set to 0 for nonslanted fonts.
24/4	0	reserved	Set to 0.
26/2	0	reserved	Set to 0.
28/2	0	reserved	Set to 0.
30/2	0	reserved	Set to 0.
32/2	0	metricDataFormat	Set to 0.
34/2	258	numOfLongVerMetrics	Number of advance heights in the vertical metrics table is 258.

vmtx - Vertical Metrics Table

The vertical metrics table (tag name: ‘vmtx’) allows you to specify the vertical spacing for each glyph in a vertical font. This table consists of either one or two arrays that contain metric information (the advance heights and top sidebearings) for the vertical layout of each of the glyphs in the font. The vertical metrics coordinate system is shown below.



TrueType vertical fonts require both a vertical header table (tag name: ‘vhea’) discussed previously and the vertical metrics table discussed below. The vertical header table contains information that is general to the font as a whole. The vertical metrics table contains information that pertains to specific glyphs. The formats of these tables are similar to those for horizontal metrics (hhea and hmtx).

Vertical Metrics Table Format

The overall structure of the vertical metrics table consists of two arrays shown below: the vMetrics array followed by an array of top side bearings.

This table does not have a header, but does require that the number of glyphs included in the two arrays equals the total number of glyphs in the font.

The number of entries in the vMetrics array is determined by the value of the numOfLongVerMetrics field of the vertical header table.

The vMetrics array contains two values for each entry. These are the advance height and the top sidebearing for each glyph included in the array.

In monospaced fonts, such as Courier or Kanji, all glyphs have the same advance height. If the font is monospaced, only one entry need be in the first array, but that one entry is required.

The format of an entry in the vertical metrics array is given below.

Type	Name	Description
USHORT	advanceHeight	The advance height of the glyph. Unsigned integer in FUnits
SHORT	topSideBearing	The top sidebearing of the glyph. Signed integer in FUnits.

The second array is optional and generally is used for a run of monospaced glyphs in the font. Only one such run is allowed per font, and it must be located at the end of the font. This array contains the top sidebearings of glyphs not represented in the first array, and all the glyphs in this array must have the same advance height as the last entry in the vMetrics array. All entries in this array are therefore monospaced.

The number of entries in this array is calculated by subtracting the value of numOfLongVerMetrics from the number of glyphs in the font. The sum of glyphs represented in the first array plus the glyphs represented in the second array therefore equals the number of glyphs in the font. The format of the top sidebearing array is given below.

Type	Name	Description
SHORT	topSideBearing[]	The top sidebearing of the glyph. Signed integer in FUnits.