INTERNATIONAL STANDARD

ISO/IEC 14496-22

Second edition
2009-08-15

Information technology — Coding of audio-visual objects —

Part 22: Open Font Format

Technologies de l'information — Codage des objets audiovisuels —

Partie 22: Format de police de caractères ouvert

Reference number
ISO/IEC 14496-22:2009(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

ISO/IEC 14496-22 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This second edition cancels and replaces the first edition (ISO/IEC 14496-22:2007) which has been technically revised.

ISO/IEC 14496 consists of the following parts, under the general title *Information technology — Coding of audio-visual objects*:

— *Part 1: Systems*

— *Part 2: Visual*

— *Part 3: Audio*

— *Part 4: Conformance testing*

— *Part 5: Reference software*

— *Part 6: Delivery Multimedia Integration Framework (DMIF)*

— *Part 7: Optimized reference software for coding of audio-visual objects*

— *Part 8: Carriage of ISO/IEC 14496 contents over IP networks*

— *Part 9: Reference hardware description*

— *Part 10: Advanced Video Coding*

— *Part 11: Scene description and application engine*

— *Part 12: ISO base media file format*

— *Part 13: Intellectual Property Management and Protection (IPMP) extensions*

— *Part 14: MP4 file format*

— *Part 15: Advanced Video Coding (AVC) file format*

— *Part 16: Animation Framework eXtension (AFX)*

— *Part 17: Streaming text format*

— *Part 18: Font compression and streaming*

— *Part 19: Synthesized texture stream*

— *Part 20: Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF)*

— *Part 21: MPEG-J Graphics Framework eXtensions (GFX)*

— *Part 22: Open Font Format*

— *Part 23: Symbolic Music Representation*

— *Part 24: Audio and systems interaction*

— *Part 25: 3D Graphics Compression Model*

Introduction

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

The ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO and IEC that he is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO and IEC. Information may be obtained from the companies listed in Annex A.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified in Annex A. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Information technology — Coding of audio-visual objects —

Part 22:
Open Font Format

1 Scope

This part of ISO/IEC 14496 specifies the Open Font Format (OFF) specification, the TrueType™ [i] and Compact Font Format (CFF) outline formats, and the TrueType hinting language. Many references to both TrueType and PostScript exist throughout this document, as Open Font Format fonts combine the two technologies.

NOTE This specification is based on the OpenType®[ii] font format specification, and is technically equivalent to that specification.

Multimedia applications require a broad range of media-related standards. In addition to the typical audio and video applications, multimedia presentations include scalable 2D graphics and text supporting all languages of the world. Faithful reproduction of scalable multimedia content requires additional components including scalable font technology. The Open Font Format is an extension of the TrueType font format, adding support for PostScript font data. OFF fonts and the operating system services which support OFF fonts provide users with a simple way to install and use fonts, whether the fonts contain TrueType outlines or CFF (PostScript) outlines.

The Open Font Format addresses the following goals:

- broader multi-platform support;
- excellent support for international character sets;
- excellent protection for font data;
- smaller file sizes to make font distribution more efficient;
- excellent support for advanced typographic control.

PostScript®[iii] data included in OFF fonts may be directly rasterized or converted to the TrueType outline format for rendering, depending on which rasterizers have been installed in the host operating system. But the user model is the same: OFF fonts just work. Users will not need to be aware of the type of outline data in OFF fonts. And font creators can use whichever outline format they feel provides the best set of features for their work, without worrying about limiting a font's usability.

OFF fonts can include the OFF Layout tables, which allow font creators to design broader international and high-end typographic fonts. The OFF Layout tables contain information on glyph substitution, glyph positioning, justification, and baseline positioning, enabling text-processing applications to improve text layout.

---

[i] TrueType is a trademark of Apple Computer Incorporated.

[ii] OPENTYPE is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

[iii] PostScript is a registered trademark of Adobe Systems Incorporated.

As with TrueType fonts, OFF fonts allow the handling of large glyph sets using Unicode encoding. Such encoding allows broad international support, as well as support for typographic glyph variants.

Additionally, OFF fonts may contain digital signatures, which enable operating systems and browsing applications to identify the source and integrity of font files, including embedded font files obtained in web documents, before using them. Also, font developers can encode embedding restrictions in OFF fonts which cannot be altered in a font signed by the developer.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 10646, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*

ISO/IEC 14496-18, *Information technology — Coding of audio-visual objects — Part 18: Font compression and streaming*

TrueType Instruction Set, <http://www.microsoft.com/typography/otspec/ttinst.htm>

Unicode 5.1, <http://www.unicode.org/versions/Unicode5.1.0/>

3 Abbreviated terms

List of abbreviated terms.

ACF	Average Character Face
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
ATM	Adobe Type Manager
BMP	[Unicode] Basic Multilingual Plane (also known as UCS-2)
BTBD	Baseline To Baseline Distance
CFF	Compact Font Format
CID	Character Identifier
CJK	Chinese Japanese Korean [characters, ideographs, fonts, etc.]
CJKV	Chinese Japanese Korean and Vietnamese
CV	Control Value
CVT	Control Value Table
DLL	Dynamically Linked Library
FDEF	Function Definition
GID	Glyph ID
ICF	Ideographic Character Face
IDEF	Instruction Definition
IETF	Internet Engineering Task Force
JIS	Japanese Industrial Standard
LTR	Left To Right

NLC	National Language Council of Japan
OFF	Open Font Format
OMPL	OFF Mirroring Pairs List
OTF	OpenType Font
PCL	Printer Control Language
PPM, PPEM	Pixels Per EM
PRC	People's Republic of China
RTL	Right To Left
TTC	TrueType Collection
TTF	TrueType Font
UCS	Universal Character Set
UTF	Unicode Transformation Format
UVS	Unicode Variation Sequence
VM	Virtual Memory
W3C	World Wide Web Consortium

4 The Open font file format

4.1 Description

An Open font file contains data, in table format, that comprises either a TrueType or a PostScript outline font. Rasterizers use combinations of data from the tables contained in the font to render the TrueType or PostScript glyph outlines. Some of this supporting data is used no matter which outline format is used; some of the supporting data is specific to either TrueType or PostScript.

4.2 Filenames

OFF fonts may have the extension .OTF or .TTF, depending on the kind of outlines in the font and the creator's desire for compatibility on systems without native OFF support.

- In all cases, fonts with only CFF data (no TrueType outlines) always have an .OTF extension.
- Fonts containing TrueType outlines may have either .OTF or .TTF, depending on the desire for backward compatibility on older systems or with previous versions of the font. TrueType Collection fonts should have a .TTC extension whether or not the fonts have OFF layout tables present.

4.3 Data types

The following data types are used in the OFF font file. All OFF fonts use big-endian (network byte order):

Data Type	Description
BYTE	8-bit unsigned integer.
CHAR	8-bit signed integer.
USHORT	16-bit unsigned integer.
SHORT	16-bit signed integer.
UINT24	24-bit unsigned 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	16-bit unsigned integer (USHORT) that describes a quantity in FUnits.
F2DOT14	16-bit signed fixed number with the low 14 bits of fraction (2.14).
LONGDATETIME	Date represented in number of seconds since 12:00 midnight, January 1, 1904. The value is represented as a signed 64-bit integer.
Tag	Array of four uint8s (length = 32 bits) used to identify a script, language system, feature, or baseline
GlyphID	Glyph index number, same as uint16(length = 16 bits)
Offset	Offset to a table, same as uint16 (length = 16 bits), NULL Offset = 0x0000

The F2DOT14 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	12288/16384
0.000061	0x0001	0	1/16384
0.0	0x0000	0	0/16384
-0.000061	0xffff	-1	16383/16384
-2.0	0x8000	-2	0/16384

4.4 Table version numbers

Most tables have version numbers, and the version number for the entire font is contained in the Table Directory. It should be noted that there are two different table version number types, each with its own numbering scheme. USHORT version numbers always start at zero (0). Fixed version numbers start at one (1.0 or 0x00010000), except where noted (EBDT, EBLC and EBSC tables).

Implementations reading tables must include code to check version numbers so that if and when the format and therefore the version number changes, older implementations will reject newer versions gracefully, if the changes are incompatible.

When a Fixed number is used as a version, the upper 16 bits comprise a major version number and the lower 16 bits a minor. Tables with non-zero minor version numbers always specify the literal value of the version number since the normal representation of Fixed numbers is not necessarily followed. For example, the version number of 'maxp' table version 0.5 is 0x00005000, and that of 'vhea' table version 1.1 is 0x00011000. If an implementation understands a major version number, then it can safely proceed reading the table. The minor version number indicates extensions to the format that are undetectable by implementations that do not support them.

The only exception to this is the Offset Table's sfnt version. This serves solely to identify whether the OFF font contains TrueType outlines (a value of 1.0) or CFF data (the tag 'OTTO'), as described in subclause 3.5, 'Open Font Structure.'

When a USHORT number is used to indicate version, it should be treated as though it were a minor version number; i.e., all format changes are compatible extensions.

4.5 Open font structure

A key characteristic of the OFF format is the TrueType sfnt "wrapper", which provides organization for a collection of tables in a general and extensible manner.

The OFF font with the Offset Table. If the font file contains only one font, the Offset Table will begin at byte 0 of the file. If the font file is a TrueType collection, the beginning point of the Offset Table for each font is indicated in the TTCHeader.

Offset Table		
Type	Name	Description
Fixed	sfnt version	0x00010000 for version 1.0 or 'OTTO'.
USHORT	numTables	Number of tables.
USHORT	searchRange	(Maximum power of 2 <= numTables) x 16.
USHORT	entrySelector	Log2(maximum power of 2 <= numTables).
USHORT	rangeShift	NumTables x 16-searchRange.

OFF fonts that contain TrueType outlines should use the value of 1.0 for the sfnt version. OFF fonts containing CFF data should use the tag 'OTTO' as the sfnt version number.

4.5.1 Table directory

The Offset Table is followed immediately by the Table Record entries. Entries in the Table Record must be sorted in ascending order by tag. Offset values in the Table Record are measured from the start of the font file.

Table Record		
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 Record 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 OFF font file. All tag names consist of four characters. Names with less than four letters are allowed if followed by the necessary trailing spaces. All tag names defined within a font (e.g., table names, feature tags, language tags) must be built from printing characters represented by ASCII values 32-126.

NOTE Tag names are case sensitive.

4.5.2 Calculating checksums

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. In fact, a font is not considered structurally proper without the correct padding. All tables must begin on four byte boundries, and any remaining space between tables is padded with zeros. The length of all tables should be recorded in the table record with their actual length (not their padded length).

To calculate the checkSum for the 'head' table which itself includes the checkSumAdjustment entry for the entire font, do the following:

1. Set the checkSumAdjustment to 0.
2. Calculate the checksum for all the tables including the 'head' table and enter that value into the table directory.
3. Calculate the checksum for the entire font.
4. Subtract that value from the hex value B1B0AFBA.
5. Store the result in checkSumAdjustment.

The checkSum for the head table which includes the checkSumAdjustment entry for the entire font is now incorrect. That is not a problem. Do not change it. An application attempting to verify that the 'head' table has not changed should calculate the checkSum for that table by not including the checkSumAdjustment value, and compare the result with the entry in the table directory.

4.6 TrueType collections

A TrueType Collection (TTC) is a means of delivering multiple OFF fonts in a single file structure. TrueType Collections are most useful when the fonts to be delivered together share many glyphs in common. By allowing multiple fonts to share glyph sets, TTCs can result in a significant saving of file space.

For example, a group of Japanese fonts may each have their own designs for the kana glyphs, but share identical designs for the kanji. With ordinary OFF font files, the only way to include the common kanji glyphs is to copy their glyph data into each font. Since the kanji represent much more data than the kana, this results in a great deal of wasteful duplication of glyph data. TTCs were defined to solve this problem.

The CFF rasterizer does not currently support TTC files.

4.6.1 The TTC file structure

The TTC File Structure

A TrueType Collection file consists of a single TTC Header table, one or more Offset Tables with Table Directories, and a number of OFF tables. The TTC Header must be located at the beginning of the TTC file.

The TTC file must contain a complete Offset Table and Table Directory for each font. A TTC file Table Directory has exactly the same format as a TTF file Table Directory. The table Offsets in all Table Directories within a TTC file are measured from the beginning of the TTC file.

Each OFF table in a TTC file is referenced through the Offset Table and Table Directory of each font which uses that table. Some of the OFF tables must appear multiple times, once for each font included in the TTC; while other tables may be shared by multiple fonts in the TTC.

As an example, consider a TTC file which combines two Japanese fonts (Font1 and Font2). The fonts have different kana designs (Kana1 and Kana2) but use the same design for kanji. The TTC file contains a single 'glyf' table which includes both designs of kana together with the kanji; both fonts' Table Directories point to this 'glyf' table. But each font's Table Directory points to a different 'cmap' table, which identifies the glyph set to use. Font1's 'cmap' table points to the Kana1 region of the 'loca' and 'glyf' tables for kana glyphs, and to the kanji region for the kanji. Font2's 'cmap' table points to the Kana2 region of the 'loca' and 'glyf' tables for kana glyphs, and to the same kanji region for the kanji.

The tables that should have a unique copy per font are those that are used by the system in identifying the font and its character mapping, including 'cmap', 'name', and 'OS/2'. The tables that should be shared by fonts in the TTC are those that define glyph and instruction data or use glyph indices to access data: 'glyf', 'loca', 'hmtx', 'hdmx', 'LTSH', 'cvt ', 'fpgm', 'prep', 'EBLC', 'EBDT', 'EBSC', 'maxp', and so on. In practice, any tables which have identical data for two or more fonts may be shared.

NOTE Tools are available to help build .TTC files. The process involves paying close attention the issue of glyph renumbering in a font and the side effects that can result, in the 'cmap' table and elsewhere. The fonts to be merged must also have compatible TrueType instructions-that is, their pre-programs, function definitions, and control values must not conflict.

TrueType Collection files use the filename suffix .TTC.

4.6.2 TTC header

There are two versions of the TTC Header: Version 1.0 has been used for TTC files without digital signatures. Version 2.0 can be used for TTC files with *or* without digital signatures -- if there's no signature, then the last three fields of the version 2.0 header are left null.

If a digital signature is used, the DSIG table for the file must be the last table in the TTC file. Signatures in a TTC file are expected to be Format 1 signatures.

The purpose of the TTC Header table is to locate the different Offset Tables within a TTC file. The TTC Header is located at the beginning of the TTC file (Offset = 0). It consists of an identification tag, a version number, a count of the number of OFF fonts in the file, and an array of Offsets to each Offset Table.

TTC Header Version 1.0

Type	Name	Description
TAG	TTCTag	TrueType Collection ID string: 'ttcf'
FIXED	Version	Version of the TTC Header (1.0), 0x00010000
ULONG	numFonts	Number of fonts in TTC
ULONG	OffsetTable[numFonts]	Array of Offsets to the OffsetTable for each font from the beginning of the file

TTC Header Version 2.0		
Type	Name	Description
TAG	TTCTag	TrueType Collection ID string: 'ttcf'
FIXED	Version	Version of the TTC Header (2.0), 0x00020000
ULONG	numFonts	Number of fonts in TTC
ULONG	OffsetTable[numFonts]	Array of Offsets to the OffsetTable for each font from the beginning of the file
ULONG	ulDsigTag	Tag indicating that a DSIG table exists, 0x44534947 ('DSIG') (null if no signature)
ULONG	ulDsigLength	The length (in bytes) of the DSIG table (null if no signature)
ULONG	ulDsigOffset	The Offset (in bytes) of the DSIG table from the beginning of the TTC file (null if no signature)

5 Open font tables

5.1 General

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

5.2 Required common tables

Whether TrueType or PostScript outlines are used in an OFF font, the following tables are required for the font to function correctly:

Tag	Name
cmap	Character to glyph mapping
head	Font header
hhea	Horizontal header
hmtx	Horizontal metrics
maxp	Maximum profile
name	Naming table
OS/2	OS/2 and Windows specific metrics
post	PostScript information

5.2.1 cmap – Character to glyph index mapping table

5.2.1.1 Table structure

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, commonly known as .notdef.

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

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

cmap Header

Type	Name	Description
USHORT	Version	Table version number (0)
USHORT	numTables	Number of encoding tables that follow

The cmap table header is followed by an array of encoding records that specify the particular encoding and the Offset to the subtable for that encoding. The number of encoding records is *numTables*. An encoding record entry looks like:

Encoding Record

Type	Name	Description
USHORT	platformID	Platform ID.
USHORT	encodingID	Platform-specific encoding ID.
ULONG	Offset	Byte Offset frombeginning of table to the subtable for this encoding

NOTE The values of platformID, encodingID and languageID are defined in the 'name' table description in subclause 4.2.6.

The platform ID and platform-specific encoding ID in the header entry (and, in the case of the Macintosh platform, the language field in the subtable itself) are used to specify a particular 'cmap' encoding. The header entries must be sorted first by platform ID, then by platform-specific encoding ID, and then by the language field in the corresponding subtable. Each platform ID, platform-specific encoding ID, and subtable language combination may appear only once in the 'cmap' table.

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 Windows Unicode BMP encodings (Platform ID = 3, Encoding ID = 1) must provide at least a Format 4 'cmap' subtable. If the font is meant to support supplementary (non-BMP) Unicode characters, it will additionally need a Format 12 subtable with a platform encoding ID 10. The contents of the Format 12 subtable need to be a superset of the contents of the Format 4 subtable. It is strongly recommended using a BMP Unicode 'cmap' for all fonts. However, some other encodings that appear in current fonts follow:

Windows Encodings		
Platform ID	Encoding ID	Description
3	0	Symbol
3	1	Unicode BMP (UCS-2)
3	2	ShiftJIS
3	3	PRC
3	4	Big5
3	5	Wansung
3	6	Johab
3	7	Reserved
3	8	Reserved
3	9	Reserved
3	10	Unicode UCS-4

Unicode Variation Sequences supported by the font may be specified in the 'cmap' table under platform ID 0 and encoding ID 5, using a format 14 cmap subtable specified in the subclause 4.2.1.3.8.

5.2.1.2 OTF Windows NT compatibility mapping

If a platform ID 4 (custom), encoding ID 0-255 (OTF Windows NT compatibility mapping) 'cmap' encoding is present in an OFF font with CFF outlines, then the OTF font driver in Windows NT will: (a) superimpose the glyphs encoded at character codes 0-255 in the encoding on the corresponding Windows character set (code page 1252) Unicode values in the Unicode encoding it reports to the system; (b) add Windows character set (CharSet 0) to the list of CharSets supported by the font; and (c) consider the value of the encoding ID to be a Windows CharSet value and add it to the list of CharSets supported by the font.

NOTE This 'cmap' subtable must use Format 0 or 6 for its subtable, and the encoding must be identical to the CFF's encoding.

5.2.1.3 cmap subtable formats

This field must be set to zero for all cmap subtables whose platform IDs are other than Macintosh (platform ID 1). For cmap subtables whose platform IDs are Macintosh, set this field to the Macintosh language ID of the cmap subtable plus one, or to zero if the cmap subtable is not language-specific. For example, a Mac OS Turkish cmap subtable must set this field to 18, since the Macintosh language ID for Turkish is 17. A Mac OS Roman cmap subtable must set this field to 0, since Mac OS Roman is not a language-specific encoding.

5.2.1.3.1 Format 0: Byte encoding table

This is the Macintosh platform standard character to glyph index mapping table which is available via the informative reference [7] in the bibliography.

Type	Name	Description
USHORT	format	Format number is set to 0.
USHORT	length	This is the length in bytes of the subtable.
USHORT	language	Please see "Note on the language field in 'cmap' subtables" in this document.
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. If this format is used to index into a larger glyph set, only the first 256 glyphs will be accessible.

5.2.1.3.2 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).

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	This is the length in bytes of the subtable.
USHORT	language	Please see "Note on the language field in 'cmap' subtables" in this document.
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.
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.

5.2.1.3.3 Format 4: Segment mapping to delta values

This is the Windows standard character to glyph index mapping table for fonts that support Unicode ranges other than the range [U+D800 - U+DFFF] (defined as Surrogates Area, in the Unicode Standard) which is used for UCS-4 characters. If a font supports this character range (i.e. in turn supports the UCS-4 characters) a subtable in this format with a platform specific encoding ID 1 is yet needed, in addition to a subtable in format 12 with a platform specific encoding ID 10. Please see details on format 12 below, for fonts that support UCS-4 characters on Windows.

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	This is the length in bytes of the subtable.
USHORT	language	Please see "Note on the language field in 'cmap' subtables" in this document.
USHORT	segCountX2	2 x segCount.
USHORT	searchRange	2 x (2**floor(log2(segCount)))
USHORT	entrySelector	log2(searchRange/2)
USHORT	rangeShift	2 x segCount - searchRange
USHORT	endCode[segCount]	End characterCode for each segment, last=0xFFFF.
USHORT	reservedPad	Set to 0.
USHORT	startCode[segCount]	Start character code for each segment.
SHORT	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	log2 (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 153-480 onto a contiguous range of glyph indices may look like this:

segCountX2:	8			
searchRange:	8			
entrySelector:	4			
rangeShift:	0			
endCode:	20	90	480	0Xffff
reservedPad:	0			
startCode:	10	30	153	0Xffff
idDelta:	-9	-18	-80	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.
```

It should be noted that the delta values could be reworked so as to reorder the segments.

5.2.1.3.4 Format 6: Trimmed table mapping

Type	Name	Description
USHORT	format	Format number is set to 6.
USHORT	length	This is the length in bytes of the subtable.
USHORT	language	Please see "Note on the language field in 'cmap' subtables" in this document.
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.

NOTE Supporting 4-byte character codes: While the four existing 'cmap' subtable formats which currently exist have served us well, the introduction of the Surrogates Area in the Unicode Standard has stressed them past the point of utility. This clause specifies three formats, format 8, 10 and 12; which directly support 4-byte character codes. A major change introduced with these three formats is a more pure 32-bit orientation. The 'cmap' table version number will continue to be 0x0000, for those fonts that make use of these formats.

5.2.1.3.5 Format 8: mixed 16-bit and 32-bit coverage

Format 8 is a bit like format 2, in that it provides for mixed-length character codes. If a font contains characters from the Unicode Surrogates Area (U+D800-U+DFFF), which are UCS-4 characters; it's likely that it will also include other, regular 16-bit Unicodes as well. We therefore need a format to map a mixture of 16-bit and 32-bit character codes, just as format 2 allows a mixture of 8-bit and 16-bit codes. A simplifying assumption is made: namely, that there are no 32-bit character codes which share the same first 16 bits as any 16-bit character code. This means that the determination as to whether a particular 16-bit value is a standalone character code or the start of a 32-bit character code can be made by looking at the 16-bit value directly, with no further information required.

Here's the format 8 subtable format:

Type	Name	Description
USHORT	format	Subtable format; set to 8.
USHORT	reserved	Reserved; set to 0
ULONG	length	Byte length of this subtable (including the header)
ULONG	language	Please see "Note on the language field in 'cmap' subtables" in this document.
BYTE	is32[8192]	Tightly packed array of bits (8K bytes total) indicating whether the particular 16-bit (index) value is the start of a 32-bit character code
ULONG	nGroups	Number of groupings which follow

Here follow the individual groups. Each group has the following format:

Type	Name	Description
ULONG	startCharCode	First character code in this group; note that if this group is for one or more 16-bit character codes (which is determined from the is32 array), this 32-bit value will have the high 16-bits set to zero
ULONG	endCharCode	Last character code in this group; same condition as listed above for the startCharCode
ULONG	startGlyphID	Glyph index corresponding to the starting character code

A few notes here. The endCharCode is used, rather than a count, because comparisons for group matching are usually done on an existing character code, and having the endCharCode be there explicitly saves the necessity of an addition per group. Groups must be sorted by increasing startCharCode. A group's endCharCode must be less than the startCharCode of the following group, if any.

To determine if a particular word (cp) is the first half of 32-bit code points, one can use an expression such as (is32[cp / 8] & (1 << (7 - (cp % 8)))). If this is non-zero, then the word is the first half of a 32-bit code point.

0 is not a special value for the high word of a 32-bit code point. A font may not have both a glyph for the code point 0x0000 and glyphs for code points with a high word of 0x0000.

The presence of the packed array of bits indicating whether a particular 16-bit value is the start of a 32-bit character code is useful even when the font contains no glyphs for a particular 16-bit start value. This is because the system software often needs to know how many bytes ahead the next character begins, even if the current character maps to the missing glyph. By including this information explicitly in this table, no "secret" knowledge needs to be encoded into the OS.

Although this format might work advantageously on some platforms for non-Unicode encodings, Windows does not support it for Unicode encoded UCS-4 characters.

5.2.1.3.6 Format 10: Trimmed array

Format 10 is a bit like format 6, in that it defines a trimmed array for a tight range of 32-bit character codes:

Type	Name	Description
USHORT	format	Subtable format; set to 10.
USHORT	reserved	Reserved; set to 0
ULONG	length	Byte length of this subtable (including the header)
ULONG	language	Please see "Note on the language field in 'cmap' subtables" in this document.
ULONG	startCharCode	First character code covered
ULONG	numChars	Number of character codes covered
USHORT	glyphs[]	Array of glyph indices for the character codes covered

5.2.1.3.7 Format 12: Segmented coverage

This is the Windows platform standard character to glyph index mapping table for fonts supporting the UCS-4 characters in the Unicode Surrogates Area (U+D800 - U+DFFF). It is a bit like format 4, in that it defines segments for sparse representation in 4-byte character space. Here's the subtable format:

Type	Name	Description
USHORT	format	Subtable format; set to 12.
USHORT	reserved	Reserved; set to 0
ULONG	length	Byte length of this subtable (including the header)
ULONG	language	Please see "Note on the language field in 'cmap' subtables" in this document.
ULONG	nGroups	Number of groupings which follow

Fonts providing Unicode encoded UCS-4 character support for Windows 2000 and later, need to have a subtable with platform ID 3, platform specific encoding ID 1 in format 4; and in addition, need to have a subtable for platform ID 3, platform specific encoding ID 10 in format 12. Please note that the content of format 12 subtable, needs to be a super set of the content in the format 4 subtable. The format 4 subtable needs to be in the cmap table to enable backward compatibility needs.

Here follow the individual groups, each of which has the following format:

Type	Name	Description
ULONG	startCharCode	First character code in this group
ULONG	endCharCode	Last character code in this group
ULONG	startGlyphID	Glyph index corresponding to the starting character code

Groups must be sorted by increasing startCharCode. A group's endCharCode must be less than the startCharCode of the following group, if any. The endCharCode is used, rather than a count, because comparisons for group matching are usually done on an existing character code, and having the endCharCode be there explicitly saves the necessity of an addition per group.

5.2.1.3.8 Format 13: Last resort font

This subtable deals with situations where the same glyph is used for hundreds or even thousands of consecutive characters, from one end of Unicode to the other. This is common when creating a "Last Resort" font. Here's the subtable format:

Type	Name	Description
USHORT	format	Subtable format; set to 13.
USHORT	reserved	Reserved; set to 0
ULONG	length	Byte length of this subtable (including the header)
ULONG	language	Please see "Note on the language field in 'cmap' subtables" in this document.
ULONG	nGroups	Number of groupings which follow

Here follow the individual groups, each of which has the following format:

Type	Name	Description
ULONG	startCharCode	First character code in this group
ULONG	endCharCode	Last character code in this group
ULONG	glyphID	Glyph index to be used for all the characters in the group's range.

5.2.1.3.9 Format 14: Unicode variation sequences

Subtable format 14 specifies the Unicode Variation Sequences (UVSes) supported by the font. A Variation Sequence, according to the Unicode Standard, comprises a base character followed by a variation selector; e.g. <U+82A6, U+E0101>.

The subtable partitions the UVSes supported by the font into two categories: “default” and “non-default” UVSes. Given a UVS, if the glyph obtained by looking up the base character of that sequence in the Unicode cmap subtable (i.e. the UCS-4 or the BMP cmap subtable) is the glyph to use for that sequence, then the sequence is a “default” UVS; otherwise it is a “non-default” UVS, and the glyph to use for that sequence is specified in the format 14 subtable itself.

The example in the end of this subclause shows how a font vendor can use format 14 for a JIS-2004–aware font.

Format 14 header		
Type	Name	Description
USHORT	format	Subtable format. Set to 14.
ULONG	length	Byte length of this subtable (including this header)
ULONG	numVarSelectorRecords	Number of variation Selector Records

This is immediately followed by ‘numVarSelectorRecords’ Variation Selector Records.

Variation Selector Record		
Type	Name	Description
UINT24	varSelector	Variation selector
ULONG	defaultUVSOffset	Offset to Default UVS Table. May be 0.
ULONG	nonDefaultUVSOffset	Offset to Non-Default UVS Table. May be 0.

The Variation Selector Records are sorted in increasing order of ‘varSelector’. No two records may have the same ‘varSelector’. All offsets in a record are relative to the beginning of the format 14 cmap subtable.

A Variation Selector Record and the data its offsets point to specify those UVSes supported by the font for which the variation selector is the ‘varSelector’ value of the record. The base characters of the UVSes are stored in the tables pointed to by the offsets. The UVSes are partitioned by whether they are default or non-default UVSes.

Glyph IDs to be used for non-default UVSes are specified in the Non-Default UVS table.

Default UVS table

A Default UVS Table is simply a range-compressed list of Unicode scalar values, representing the base characters of the default UVSes which use the 'varSelector' of the associated Variation Selector Record.

Default UVS Table header		
Type	Name	Description
ULONG	numUnicodeValueRanges	Number of ranges that follow

This is immediately followed by 'numUnicodeValueRanges' Unicode Value Ranges, each of which represents a contiguous range of Unicode values.

Unicode Value Range		
Type	Name	Description
UINT24	startUnicodeValue	First value in this range
BYTE	additionalCount	Number of additional values in this range

For example, the range U+4E4D–U+4E4F (3 values) will set 'startUnicodeValue' to 0x004E4D and 'additionalCount' to 2. A singleton range will set 'additionalCount' to 0.

('startUnicodeValue' + 'additionalCount') must not exceed 0xFFFFFF.

The Unicode Value Ranges are sorted in increasing order of 'startUnicodeValue'. The ranges must not overlap; i.e., ('startUnicodeValue' + 'additionalCount') must be less than the 'startUnicodeValue' of the following range (if any).

Non-default UVS table

A Non-Default UVS Table is a list of pairs of Unicode scalar values and glyph IDs. The Unicode values represent the base characters of all non-default UVSes which use the 'varSelector' of the associated Variation Selector Record, and the glyph IDs specify the glyph IDs to use for the UVSes.

Non-Default UVS Table header		
Type	Name	Description
ULONG	numUVSMappings	Number of UVS Mappings that follow

This is immediately followed by 'numUVSMappings' UVS Mappings.

UVS Mapping		
Type	Name	Description
UINT24	unicodeValue	Base Unicode value of the UVS
USHORT	glyphID	Glyph ID of the UVS

The UVS Mappings are sorted in increasing order of 'unicodeValue'. No two mappings in this table may have the same 'unicodeValue' values.

Example

Here is an example of how a format 14 cmap subtable may be used in a font that is aware of JIS-2004 variant glyphs. The CIDs (character IDs) in this example refer to those in the Adobe Character Collection "Adobe-Japan1", and may be assumed to be identical to the glyph IDs in the font in our example.

JIS-2004 changed the default glyph variants for some of its code points. For example:

JIS-90: U+82A6 -> CID 1142
JIS-2004: U+82A6 -> CID 7961

Both of these glyph variants are supported through the use of Unicode Variation Sequences, as the following examples from Unicode's UVS registry show:

U+82A6 U+E0100 -> CID 1142
U+82A6 U+E0101 -> CID 7961

If the font wants to support the JIS-2004 variants by default, it will:

- encode glyph ID 7961 at U+82A6 in the Unicode cmap subtable,
- specify in the UVS cmap subtable's Default UVS Table ('varSelector' will be 0x0E0101 and 'defaultUVSOffset' will point to data containing a 0x0082A6 Unicode value)
- specify -> glyph ID 1142 in the UVS cmap subtable's Non-Default UVS Table ('varSelector' will be 0x0E0100 and 'nonDefaultBaseUVOffset' will point to data containing a 'unicodeValue' 0x82A6 and 'glyphID' 1142).

If, however, the font wants to support the JIS-90 variants by default, it will:

- encode glyph ID 1142 at U+82A6 in the Unicode cmap subtable,
- specify in the UVS cmap subtable's Default UVS Table
- specify -> glyph ID 7961 in the UVS cmap subtable's Non-Default UVS Table

5.2.2 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 point 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); Bits 5-10: These bits are not defined in OFF Bit 11: Font data is 'lossless,' as a result of having been compressed and decompressed with the MicroType®[iv] Express engine, as defined in ISO/IEC 14496-18. Bit 12: Font converted (produce compatible metrics) Bit 13: Font optimized for ClearType®[v]. Note, fonts that rely on embedded bitmaps (EBDT) for rendering should not be considered optimized for ClearType, and therefore should keep this bit cleared. Bit 14: Reserved, set to 0 Bit 15: Reserved, set to 0

[iv] MicroType is a registered trademark of Monotype Imaging Inc.

[v] ClearType is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

USHORT	unitsPerEm	Valid range is from 16 to 16384. This value should be a power of 2 for fonts that have TrueType outlines.
LONGDATETIME	created	Number of seconds since 12:00 midnight, January 1, 1904. 64-bit integer
LONGDATETIME	modified	Number of seconds since 12:00 midnight, January 1, 1904. 64-bit integer
SHORT	xMin	For all glyph bounding boxes.
SHORT	yMin	For all glyph bounding boxes.
SHORT	xMax	For all glyph bounding boxes.
SHORT	yMax	For all glyph bounding boxes.
USHORT	macStyle	Bit 0: Bold (if set to 1); Bit 1: Italic (if set to 1) Bit 2: Underline (if set to 1) Bit 3: Outline (if set to 1) Bit 4: Shadow (if set to 1) Bit 5: Condensed (if set to 1) Bit 6: Extended (if set to 1) Bits 7-15: Reserved (set to 0).
USHORT	lowestRecPPEM	Smallest readable size in pixels.
SHORT	fontDirectionHint	Deprecated (Set to 2). 0: Fully mixed directional glyphs; 1: Only strongly left to right; 2: Like 1 but also contains neutrals; -1: Only strongly right to left; -2: Like -1 but also contains neutrals. [1]
SHORT	indexToLocFormat	0 for short Offsets, 1 for long.
SHORT	glyphDataFormat	0 for current format.

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).

It should be noted that the macStyle bits must agree with the 'OS/2' table fsSelection bits. The fsSelection bits are used over the macStyle bits in Windows. The PANOSE values and 'post' table values are ignored for determining bold or italic fonts.

For historical reasons, the fontRevision value contained in this table is not used by Windows to determine the version of a font. Instead, Windows evaluates the version string (id 5) in the 'name' table.

5.2.3 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. (Distance from baseline of highest ascender)
FWORD	Descender	Typographic descent. (Distance from baseline of lowest descender)
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	caretOffset	The amount by which a slanted highlight on a glyph needs to be shifted to produce the best appearance. Set to 0 for non-slanted fonts
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

NOTE The ascender, descender and linegap values in this table are Macintosh platform specific.These are not ignored by Windows platform. They are used to identify fixed pitch fonts. Also see information in the OS/2 table.

5.2.4 hmtx – Horizontal metrics

The type longHorMetric is defined as an array where each element has two parts: the advance width, which is of type USHORT, and the left side bearing, which is of type SHORT. These fields are in font design units.

```
typedef struct    _longHorMetric {
        USHORT  advanceWidth;
        SHORT                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	SHORT[ ]	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.

In CFF OFF fonts, every glyph's advanceWidth as recorded in the 'hmtx' table must be identical to its x width in the 'CFF ' table.

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

5.2.5 maxp – Maximum profile

This table establishes the memory requirements for this font. Fonts with CFF data must use Version 0.5 of this table, specifying only the numGlyphs field. Fonts with TrueType outlines must use Version 1.0 of this table, where all data is required.

Version 0.5

Type	Name	Description
Fixed	Table version number	0x00005000 for version 0.5 (It should be noted that the difference in the representation of a non-zero fractional part, in Fixed numbers.)
USHORT	numGlyphs	The number of glyphs in the font.

Version 1.0

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 CV Programs, as well as the instructions for each glyph.

5.2.6 name – Naming table

5.2.6.1 Table structure

The naming table allows multilingual strings to be associated with the OFF 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 OFF 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. It should be noted that some platforms may require single byte character strings, while others may require double byte strings.

For historical reasons, some applications which install fonts perform version control using Macintosh platform (platform ID 1) strings from the 'name' table. Because of this, it is strongly recommended that the 'name' table of all fonts include Macintosh platform strings and that the syntax of the version number (name id 5) follows the guidelines given in this document.

There are two formats for the Naming Table. Format 0 uses platform-specific numeric language identifiers. Format 1 allows for use of language-tag strings to indicate the language of Naming-Table strings.

The format 0 Naming Table is organized as follows:

Type	Name	Description
USHORT	format	Format selector (=0).
USHORT	count	Number of name records.
USHORT	stringOffset	Offset to start of string storage (from start of table).
NameRecord	nameRecord[count]	The name records where *count* is the number of records.
(Variable)		Storage for the actual string data.

Each string in the string storage is referenced by a name record. As described below, each name record is keyed using four numeric identifiers. Two of these are a platform identifier and a language identifier. When used in a format 0 Naming table, the language identifiers must be less than 0x8000 and are given platform-specific interpretations. (Exceptions to that limit may be made in the case of user-defined platforms, however.) Complete details of name records, the various identifiers and their interpretations are provided below.

The format 1 Naming Table adds additional elements, as follows:

Type	Name	Description
USHORT	format	Format selector (=1).
USHORT	count	Number of name records.
USHORT	stringOffset	Offset to start of string storage (from start of table).
NameRecord	nameRecord[count]	The name records where *count* is the number of records.
USHORT	langTagCount	Number of language-tag records.
LangTagRecord	langTagRecord[langTagCount]	The language-tag records where *langTagCount* is the number of records.
(Variable)		Storage for the actual string data.

When format 1 is used, the language IDs in name records can be less than or greater than 0x8000. If a language ID is less than 0x8000, it has a platform-specific interpretation as with a format 0 Naming table. If a language ID is equal to or greater than 0x8000, it is associated with a language-tag record that references a language-tag string. In this way, the language ID is associated with a language-tag string that specifies the language for name records using that language ID, regardless of the platform. These can be used for any platform that supports this language-tag mechanism.

A font using a format 1 Naming table may use a combination of platform-specific language IDs as well as language-tag records for a given platform and encoding.

Each *langTagRecord* is organized as follows:

Type	Name	Description
USHORT	length	Language-tag string length (in bytes).
USHORT	offset	Language-tag string offset from start of storage area (in bytes).

Language-tag strings stored in the Naming table must be encoded in UTF-16BE. The language tags must conform to IETF specification BCP 47. This provides tags such as "en", "fr-CA" and "zh-Hant" to identify languages, including dialects, written form and other variations.

The language-tag records are associated sequentially with language IDs starting with 0x8000. Each language-tag record corresponds to a language ID one greater than that for the previous language-tag record. Thus, language IDs associated with language-tag records must be within the range 0x8000 to 0x8000 + langTagCount - 1. If a name record uses a language ID that is greater than this, the identify of the language is unknown; such name records should not be used.

For example, suppose a font has two language-tag records referencing strings in the storage: the first references the string "en", and the second references the string "zh-Hant-HK" In this case, the language ID 0x8000 is used in name records to index English-language strings. The language ID 0x8001 is used in name records to index strings in Traditional Chinese as used in Hong Kong.

Name records

Each *NameRecord* looks like this:

Type	Name	Description
USHORT	platformID	Platform ID.
USHORT	encodingID	Platform-specific encoding ID.
USHORT	languageID	Language ID.
USHORT	nameID	Name ID.
USHORT	length	String length (in bytes).
USHORT	Offset	String Offset from start of storage area (in bytes).

Following are the descriptions of the four kinds of ID. The specific values listed here are predefined; new ones may be added in the future. Similar to the character encoding table, the NameRecords are sorted by platform ID, then platform-specific ID, then language ID, and then by name ID.

5.2.6.2 Platform IDs, Platform-specific encoding IDs and Language IDs

Language IDs refer to a value that identifies the language in which a particular string is written. Values less than 0x8000 are defined on a platform-specific basis. Values greater than or equal to 0x8000 are associated with language-tag records, as described above. Not all platforms have platform-specific language IDs, and not all platforms support language-tag records.

Platform ID	Platform name	Platform-specific encoding IDs	Language IDs
0	Unicode	Various	None
1	Macintosh	Script manager code	Various
2	*ISO [deprecated]*	*ISO encoding [deprecated]*	*None*
3	Windows	Windows encoding	Various
4	Custom	Custom	None

It should be noted that platform ID 2 (ISO) has been deprecated as of OpenType v1.3. It was intended to represent ISO/IEC 10646, as opposed to Unicode; both standards have identical character code assignments.

Platform ID values 240 through 255 are reserved for user-defined platforms. This specification will never assign these values to a registered platform.

Unicode platform-specific encoding IDs (platform ID = 0)

Encoding ID	Description
0	Unicode 1.0 semantics
1	Unicode 1.1 semantics
2	ISO/IEC 10646 semantics
3	Unicode 2.0 and onwards semantics, Unicode BMP only.
4	Unicode 2.0 and onwards semantics, Unicode full repertoire.
5	Unicode Variation Sequences.

A new encoding ID for the Unicode platform will be assigned if a new version of Unicode moves characters, in order to properly specify character code semantics. (Because of Unicode stability policies, such a need is not anticipated.) The distinction between Unicode platform-specific encoding IDs 1 and 2 is for historical reasons only; The Unicode Standard is in fact identical in repertoire and encoding to ISO/IEC 10646. For all practical purposes in current fonts, the distinctions provided by encoding IDs 0, 1 and 2 are not important, thus these encoding IDs are deprecated.

Unicode platform encoding ID 5 can be used for encodings in the ‘cmap’ table but not for strings in the ‘name’ table.

There are no platform-specific language IDs defined for the Unicode platform. Language ID = 0 may be used for Unicode-platform strings, but this does not indicate any particular language. Language IDs greater than or equal to 0x8000 may be used together with language-tag records, as described above.

Windows platform-specific encoding IDs (platform ID= 3)

Platform ID	Encoding ID	Description
3	0	Symbol
3	1	Unicode BMP (UCS-2)
3	2	ShiftJIS
3	3	PRC
3	4	Big5
3	5	Wansung
3	6	Johab
3	7	Reserved
3	8	Reserved
3	9	Reserved
3	10	Unicode UCS-4

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.

Windows Language IDs (platform ID = 3)

The language ID (LCID in the table below) refers to a value which identifies the language in which a particular string is written.The platform-specific Language IDs assigned by Microsoft are listed below. Any platform-specific language IDs must be less than 0x8000. Language IDs greater than or equal to 0x8000 may be used together with language-tag records.

Primary Language	Region	LCID
Afrikaans	South Africa	0436
Albanian	Albania	041C
Alsatian	France	0484
Amharic	Ethiopia	045E
Arabic	Algeria	1401
Arabic	Bahrain	3C01
Arabic	Egypt	0C01
Arabic	Iraq	0801
Arabic	Jordan	2C01

Arabic	Kuwait	3401
Arabic	Lebanon	3001
Arabic	Libya	1001
Arabic	Morocco	1801
Arabic	Oman	2001
Arabic	Qatar	4001
Arabic	Saudi Arabia	0401
Arabic	Syria	2801
Arabic	Tunisia	1C01
Arabic	U.A.E.	3801
Arabic	Yemen	2401
Armenian	Armenia	042B>
Assamese	India	044D
Azeri (Cyrillic)	Azerbaijan	082C
Azeri (Latin)	Azerbaijan	042C
Bashkir	Russia	046D
Basque	Basque	042D
Belarusian	Belarus	0423
Bengali	Bangladesh	0845
Bengali	India	0445
Bosnian (Cyrillic)	Bosnia and Herzegovina	201A
Bosnian (Latin)	Bosnia and Herzegovina	141A
Breton	France	047E
Bulgarian	Bulgaria	0402
Catalan	Catalan	0403
Chinese	Hong Kong S.A.R.	0C04
Chinese	Macao S.A.R.	1404
Chinese	People's Republic of China	0804
Chinese	Singapore	1004
Chinese	Taiwan	0404
Corsican	France	0483
Croatian	Croatia	041A
Croatian (Latin)	Bosnia and Herzegovina	101A
Czech	Czech Republic	0405
Danish	Denmark	0406

Dari	Afghanistan	048C
Divehi	Maldives	0465
Dutch	Belgium	0813
Dutch	Netherlands	0413
English	Australia	0C09
English	Belize	2809
English	Canada	1009
English	Caribbean	2409
English	India	4009
English	Ireland	1809
English	Jamaica	2009
English	Malaysia	4409
English	New Zealand	1409
English	Republic of the Philippines	3409
English	Singapore	4809
English	South Africa	1C09
English	Trinidad and Tobago	2C09
English	United Kingdom	0809
English	United States	0409
English	Zimbabwe	3009
Estonian	Estonia	0425
Faroese	Faroe Islands	0438
Filipino	Philippines	0464
Finnish	Finland	040B
French	Belgium	080C
French	Canada	0C0C
French	France	040C
French	Luxembourg	140c
French	Principality of Monoco	180C
French	Switzerland	100C
Frisian	Netherlands	0462
Galician	Galician	0456
Georgian	Georgia	0437
German	Austria	0C07
German	Germany	0407

German	Liechtenstein	1407
German	Luxembourg	1007
German	Switzerland	0807
Greek	Greece	0408
Greenlandic	Greenland	046F
Gujarati	India	0447
Hausa (Latin)	Nigeria	0468
Hebrew	Israel	040D
Hindi	India	0439
Hungarian	Hungary	040E
Icelandic	Iceland	040F
Igbo	Nigeria	0470
Indonesian	Indonesia	0421
Inuktitut	Canada	045D
Inuktitut (Latin)	Canada	085D
Irish	Ireland	083C
isiXhosa	South Africa	0434
isiZulu	South Africa	0435
Italian	Italy	0410
Italian	Switzerland	0810
Japanese	Japan	0411
Kannada	India	044B
Kazakh	Kazakhstan	043F
Khmer	Cambodia	0453
K'iche	Guatemala	0486
Kinyarwanda	Rwanda	0487
Kiswahili	Kenya	0441
Konkani	India	0457
Korean	Korea	0412
Kyrgyz	Kyrgyzstan	0440
Lao	Lao P.D.R.	0454
Latvian	Latvia	0426
Lithuanian	Lithuania	0427
Lower Sorbian	Germany	082E
Luxembourgish	Luxembourg	046E

Macedonian (FYROM)	Former Yugoslav Republic of Macedonia	042F
Malay	Brunei Darussalam	083E
Malay	Malaysia	043E
Malayalam	India	044C
Maltese	Malta	043A
Maori	New Zealand	0481
Mapudungun	Chile	047A
Marathi	India	044E
Mohawk	Mohawk	047C
Mongolian (Cyrillic)	Mongolia	0450
Mongolian (Traditional)	People's Republic of China	0850
Nepali	Nepal	0461
Norwegian (Bokmal)	Norway	0414
Norwegian (Nynorsk)	Norway	0814
Occitan	France	0482
Oriya	India	0448
Pashto	Afghanistan	0463
Polish	Poland	0415
Portuguese	Brazil	0416
Portuguese	Portugal	0816
Punjabi	India	0446
Quechua	Bolivia	046B
Quechua	Ecuador	086B
Quechua	Peru	0C6B
Romanian	Romania	0418
Romansh	Switzerland	0417
Russian	Russia	0419
Sami (Inari)	Finland	243B
Sami (Lule)	Norway	103B
Sami (Lule)	Sweden	143B
Sami (Northern)	Finland	0C3B
Sami (Northern)	Norway	043B
Sami (Northern)	Sweden	083B
Sami (Skolt)	Finland	203B
Sami (Southern)	Norway	183B

Sami (Southern)	Sweden	1C3B
Sanskrit	India	044F
Serbian (Cyrillic)	Bosnia and Herzegovina	1C1A
Serbian (Cyrillic)	Serbia	0C1A
Serbian (Latin)	Bosnia and Herzegovina	181A
Serbian (Latin)	Serbia	081A
Sesotho sa Leboa	South Africa	046C
Setswana	South Africa	0432
Sinhala	Sri Lanka	045B
Slovak	Slovakia	041B
Slovenian	Slovenia	0424
Spanish	Argentina	2C0A
Spanish	Bolivia	400A
Spanish	Chile	340A
Spanish	Colombia	240A
Spanish	Costa Rica	140A
Spanish	Dominican Republic	1C0A
Spanish	Ecuador	300A
Spanish	El Salvador	440A
Spanish	Guatemala	100A
Spanish	Honduras	480A
Spanish	Mexico	080A
Spanish	Nicaragua	4C0A
Spanish	Panama	180A
Spanish	Paraguay	3C0A
Spanish	Peru	280A
Spanish	Puerto Rico	500A
Spanish (Modern sort)	Spain	0C0A
Spanish (Traditional sort)	Spain	040A
Spanish	United States	540A
Spanish	Uruguay	380A
Spanish	Venezuela	200A
Sweden	Finland	081D
Swedish	Sweden	041D
Syriac	Syria	045A

Tajik (Cyrillic)	Tajikistan	0428
Tamazight (Latin)	Algeria	085F
Tamil	India	0449
Tatar	Russia	0444
Telugu	India	044A
Thai	Thailand	041E
Tibetan	PRC	0451
Turkish	Turkey	041F
Turkmen	Turkmenistan	0442
Uighur	PRC	0480
Ukrainian	Ukraine	0422
Upper Sorbian	Germany	042E
Urdu	Islamic Republic of Pakistan	0420
Uzbek (Cyrillic)	Uzbekistan	0843
Uzbek (Latin)	Uzbekistan	0443
Vietnamese	Vietnam	042A
Welsh	United Kingdom	0452
Wolof	Senegal	0448
Yakut	Russia	0485
Yi	PRC	0478
Yoruba	Nigeria	046A

Macintosh platform-specific encoding IDs (script manager codes), (platform ID = 1)

Code	Script	Code	Script
0	Roman	17	Malayalam
1	Japanese	18	Sinhalese
2	Chinese (Traditional)	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	Chinese (Simplified)
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	Uninterpreted
16	Kannada		

Macintosh language IDs (platform ID = 1)

The platform-specific language ID's assigned by Apple are listed below. Any platform-specific language IDs must be less than 0x8000. Language IDs greater than or equal to 0x8000 may be used together with language-tag records.

Code	Language	Code	Language
0	English	59	Pashto
1	French	60	Kurdish
2	German	61	Kashmiri
3	Italian	62	Sindhi
4	Dutch	63	Tibetan
5	Swedish	64	Nepali
6	Spanish	65	Sanskrit
7	Danish	66	Marathi
8	Portuguese	67	Bengali
9	Norwegian	68	Assamese
10	Hebrew	69	Gujarati
11	Japanese	70	Punjabi
12	Arabic	71	Oriya
13	Finnish	72	Malayalam
14	Greek	73	Kannada
15	Icelandic	74	Tamil
16	Maltese	75	Telugu
17	Turkish	76	Sinhalese
18	Croatian	77	Burmese
19	Chinese (Traditional)	78	Khmer
20	Urdu	79	Lao
21	Hindi	80	Vietnamese
22	Thai	81	Indonesian
23	Korean	82	Tagalong
24	Lithuanian	83	Malay (Roman script)

25	Polish	84	Malay (Arabic script)
26	Hungarian	85	Amharic
27	Estonian	86	Tigrinya
28	Latvian	87	Galla
29	Sami	88	Somali
30	Faroese	89	Swahili
31	Farsi/Persian	90	Kinyarwanda/Ruanda
32	Russian	91	Rundi
33	Chinese (Simplified)	92	Nyanja/Chewa
34	Flemish	93	Malagasy
35	Irish Gaelic	94	Esperanto
36	Albanian	128	Welsh
37	Romanian	129	Basque
38	Czech	130	Catalan
39	Slovak	131	Latin
40	Slovenian	132	Quenchua
41	Yiddish	133	Guarani
42	Serbian	134	Aymara
43	Macedonian	135	Tatar
44	Bulgarian	136	Uighur
45	Ukrainian	137	Dzongkha
46	Byelorussian	138	Javanese (Roman script)
47	Uzbek	139	Sundanese (Roman script)
48	Kazakh	140	Galician
49	Azerbaijani (Cyrillic script)	141	Afrikaans
50	Azerbaijani (Arabic script)	142	Breton
51	Armenian	14	Inuktitut
52	Georgian	144	Scottish Gaelic
53	Moldavian	145	Manx Gaelic
54	Kirghiz	146	Irish Gaelic (with dot above)
55	Tajiki	147	Tongan
56	Turkmen	148	Greek (polytonic)
57	Mongolian (Mongolian script)	149	Greenlandic
58	Mongolian (Cyrillic script)	150	Azerbaijani (Roman script)

ISO specific encodings (platform ID=2) [deprecated]

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

There are no ISO-specific language IDs, and language-tag records are not supported on this platform. This means that it can be used for encodings in the 'cmap' table but not for strings in the 'name' table.

Custom platform-specific encoding IDs (platform ID = 4)

ID	Custom encoding
0-255	OTF Windows NT compatibility mapping

In cases where a custom platform cmap is present for OTF Windows NT compatibility, the encoding ID must be set to the Windows charset value (in the range 0 to 255, inclusive) present in the .PFM file of the original Type 1 font. See the 'cmap' table for more details on the OTF Windows NT compatibility cmap.

There are no platform-specific language IDs defined for the Custom platform, and language-tag records are not supported on this platform. This means that it can be used for encodings in the 'cmap' table but not for strings in the 'name' table.

5.2.6.3 Name IDs

The following name IDs are pre-defined and they apply to all platforms unless indicated otherwise. Name IDs 23 to 255, inclusive, are reserved for future standard names. Name IDs 256 to 32767, inclusive, are reserved for font-specific names such as those referenced by a font's layout features.

Code	Meaning
0	Copyright notice.
1	Font Family name. Up to four fonts can share the Font Family name, forming a font style linking group (regular, italic, bold, bold italic - as defined by OS/2.fsSelection bit settings).
2	Font Subfamily name. The Font Subfamily name distiguishes the font in a group with the same Font Family name (name ID 1). This is assumed to address style (italic, oblique) and weight (light, bold, black, etc.). 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 be a combination of strings 1 and 2. Exception: if the font is "Regular" as indicated in string 2, then use only the family name contained in string 1. An exception to the above definition of Full font name is for Windows platform strings for CFF OFF fonts: in this case, the Full font name string must be identical to the PostScript FontName in the CFF Name INDEX.
5	Version string. Should begin with the syntax 'Version <number>.<number>' (upper case, lower case, or mixed, with a space between "Version" and the number). The string must contain a version number of the following form: one or more digits (0-9) of value less than 65,535, followed by a period, followed by one or more digits of value less than 65,535. Any character other than a digit will terminate the minor number. A character such as ";" is helpful to separate different pieces of version information.

The first such match in the string can be used by installation software to compare font versions. Some installers may require the string to start with "Version ", followed by a version number as above.

6 Postscript name for the font; Name ID 6 specifies a string which is used to invoke a PostScript language font that corresponds to this OFF font. If no name ID 6 is present, then there is no defined method for invoking this font on a PostScript interpreter.
OFF fonts which include a name with name ID of 6 shall include these two names with name ID 6, and characteristics as follows:

a. Platform: 1 [Macintosh]; Platform-specific encoding: 0 [Roman]; Language: 0 [English].

b. Platform: 3 [Windows]; Platform-specific encoding: 1 [Unicode]; Language: 0x409 [English (American)].

Names with name ID 6 other than the above two, if present, must be ignored.

When translated to ASCII, these two name strings must be identical; no longer than 63 characters; and restricted to the printable ASCII subset, codes 33 through 126, except for the 10 characters: '[', ']', '(', ')', '{', '}', '<', '>', '/', '%'.

In CFF OFF fonts, these two name strings, when translated to ASCII, must also be identical to the font name as stored in the CFF's Name INDEX.

The term "PostScript Name" here means a string identical to the two identical name ID 6 strings described above.

Depending on the particular font format that PostScript language font uses, the invocation method for the PostScript font differs, and the semantics of the resulting PostScript language font differ. The method used to invoke this font depends on the presence of name ID 20.

If a name ID 20 is present in this font, then the default assumption should be that the PostScript Name defined by name ID 6 should be used with the "composefont" invocation. This PostScript Name is then the name of a PostScript language CIDFont resource which corresponds to the glyphs of the OFF font. This name is valid to pass, with an appropriate PostScript language CMap reference, and an instance name, to the PostScript language "composefont" operator.

If no name ID 20 is present in this font, then the default assumption should be that the PostScript Name defined by name ID 6 should be used with the "findfont" invocation, for locating the font in the context of a PostScript interpreter. This PostScript Name is then the name of a PostScript language Font resource which corresponds to the glyphs of the OFF font. This name is valid to pass to the PostScript language "findfont" operator. This does *not* necessarily imply that the resulting font dictionary accepts an /Encoding array, such as when the font referenced is a Type 0 PostScript font.

This specification applies only to data fork OFF fonts. Macintosh resource-fork TrueType and other Macintosh sfnt-wrapped fonts supply the PostScript font name to be used with the "findfont" invocation, in order to invoke the font in a PostScript interpreter, in the FOND resource style-mapping table.

Developers may choose to ignore the default usage when appropriate. For example, PostScript printers whose version is earlier than 2015 cannot process CID font resources, and a CJK OFF/CFF-CID font can be downloaded only as a set of Type 1 PostScript fonts. Legacy CJK TrueType fonts, which do not have a name ID 20, may still be most effectively downloaded as a CID font resource. Definition of the full set of situations in which the defaults should not be followed is outside the scope of this document.

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.
8	Manufacturer Name.
9	Designer; name of the designer of the typeface.
10	Description; description of the typeface. Can contain revision information, usage recommendations, history, features, etc.
11	URL Vendor; URL of font vendor (with protocol, e.g., http://, ftp://). If a unique serial number is embedded in the URL, it can be used to register the font.
12	URL Designer; URL of typeface designer (with protocol, e.g., http://, ftp://).
13	License Description; description of how the font may be legally used, or different example scenarios for licensed use. This field should be written in plain language, not legalese.
14	License Info URL; URL where additional licensing information can be found.
15	Reserved.
16	Preferred Family; For historical reasons, font families have contained a maximum of four styles, but font designers may group more than four fonts to a single family. The Preferred Family allows font designers to include the preferred family grouping which contains more than four fonts. This ID is only present if it is different from ID 1.
17	Preferred Subfamily; Allows font designers to include the preferred subfamily grouping that is more descriptive than ID 2. This ID is only present if it is different from ID 2 and must be unique for the the Preferred Family.
18	Compatible Full (Macintosh only); On the Macintosh, the menu name is constructed using the FOND resource. This usually matches the Full Name. If you want the name of the font to appear differently than the Full Name, you can insert the Compatible Full Name in ID 18.
19	Sample text; This can be the font name, or any other text that the designer thinks is the best sample to display the font in.
20	PostScript CID findfont name; Its presence in a font means that the nameID 6 holds a PostScript font name that is meant to be used with the "composefont" invocation in order to invoke the font in a PostScript interpreter. See the definition of name ID 6. The value held in the name ID 20 string is interpreted as a PostScript font name that is meant to be used with the "findfont" invocation, in order to invoke the font in a PostScript interpreter. If the name ID 20 is present in a font, there must be one name ID 20 record for every Macintosh platform cmap subtable in that font. A particular name ID 20 record is associated with the encoding specified by the matching cmap subtable. A name ID 20 record is matched to a cmap subtable when they have the same platform and platform-specific encoding IDs, and corresponding language/version IDs. Name ID 20 records are meant to be used only with Macintosh cmap subtables. The version field for a cmap subtable is one more than the language ID value for the corresponding name ID 20 record, with the exception of the cmap subtable version field 0. This version field,

meaning "not language-specific", corresponds to the language ID value 0xFFFF, or decimal 65535, for the corresponding name ID 20 record.

When translated to ASCII, this name string must be restricted to the printable ASCII subset, codes 33 through 126, except for the 10 characters: '[', ']', '(', ')', '{', '}', '<', '>', '/', '%'.

This specification applies only to data fork OFF fonts. Macintosh resource-fork TrueType and other Macintosh sfnt-wrapped fonts supply the PostScript font name to be used with the "findfont" invocation, in order to invoke the font in a PostScript interpreter, in the FOND resource style-mapping table.

A particular Name ID 20 string always corresponds to a particular Macintosh cmap subtable. However, some host OFF/TTF fonts also contain a post table, format 4, which provides a mapping from glyph ID to encoding value, and also corresponds to a particular Macintosh cmap subtable. Unfortunately, the post table format 4 contains no provision for identifying which Macintosh cmap subtable it matches, nor for providing more than one mapping. Host fonts which contain a post table format 4, should also contain only a single Macintosh cmap subtable, and a single Name ID 20 string. In the case where there is more than one Macintosh cmap subtable and more than one Name ID 20 string, there is no definition of which one matches the post table format 4.

21 WWS family name (see OS/2 fsSelection field for details). If bit 8 of 'fsSelection' field is set, the font belongs to WWS font families that are composed of font faces that differ only in Weight, Width and Slope. Non-WWS font families may contain faces for weight, width and slope, in addition to faces for other traditional attributes such as “handwriting”, “caption”, “subheading”, “display”, “optical” etc. This ID may define the additional attributes of non-WWS font families. Examples of name ID 21: "Minion Pro Caption" and "Minion Pro Display". (Name ID 16 would be "Minion Pro" for these examples.)

22 WWS subfamily name; Should be similar to ID 21, but reflect only weight, width and slope attributes of the font. Examples of name ID 22: "Semibold Italic", "Bold Condensed". (Name ID 17 could be "Semibold Italic Caption", or "Bold Condensed Display", for example.)

NOTE All implementations support the same set of name strings but the interpretations may be somewhat different for the Macintosh and Windows platforms. But since name strings are stored by platform, encoding and language (placing separate strings for both 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. To better help understand, some examples of name usage, weight class and style flags have been created.

The following is an example of how name strings would be made for the Arial family.

Font	Name ID 1	Name ID 2	Name ID 4	Name ID 16	Name ID 17
Arial Narrow	Arial Narrow	Regular	Arial Narrow	Arial	Narrow
Arial Narrow Italic	Arial Narrow	Italic	Arial Narrow Italic	Arial	Narrow Italic
Arial Narrow Bold	Arial Narrow	Bold	Arial Narrow Bold	Arial	Narrow Bold
Arial Narrow Bold Italic	Arial Narrow	Bold Italic	Arial Narrow Bold Italic	Arial	Narrow Bold Italic
Arial	Arial	Regular	Arial	Arial	

Arial Italic	Arial	Italic	Arial Italic	Arial	Italic
Arial Bold	Arial	Bold	Arial Bold	Arial	Bold
Arial Bold Italic	Arial	Bold Italic	Arial Bold Italic	Arial	Bold Italic
Arial Black	Arial Black	Regular	Arial Black	Arial	Black
Arial Black Italic	Arial Black	Italic	Arial Black Italic	Arial	Black Italic

In addition to name strings, OS/2.usWeightClass, OS/2.usWidthClass, OS/2.fsSelection style bits, and head.macStyle bits are shown. These settings allow the fonts to fit together into a single family of varying weight and compression/expansion.

Font	OS/2 usWeight Class	OS/2 usWidthClass	OS/2 fsSelection Italic	OS/2 fsSelection Bold	OS/2 fsSelection Regular	head macStyle Bold	head macStyle Italic	head macStyle Condensed	head macStyle Extended
Arial Narrow	400	3			x			x	
Arial Narrow Italic	400	3	x				x	x	
Arial Narrow Bold	700	3		x		x		x	
Arial Narrow Bold Italic	700	3	x	x		x	x	x	
Arial	400	5			x				
Arial Italic	400	5	x				x		
Arial Bold	700	5		x		x			
Arial Bold Italic	700	5	x	x		x	x		
Arial Black	900	5							
Arial Black Italic	900	5	x				x		

All 'name' table strings for platform ID 3 (Windows platform) must be in Unicode, using the UTF-16 encoding form. The character set encding for 'name' table strings with platform ID 0 (Macintosh) is determined by the encoding ID.

Note that, for a typographic family that includes non-WWS variations, some of the member faces will differ from Regular in relation to attributes other than weight, width or slope, but there will likely also be some member faces that differ only in relation to these three attributes. IDs 21 and 22 should be used only in those fonts that differ from the Regular face in terms of an attribute other than weight, width or slope.

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*
(If this were the Windows platform string for a CFF OFF font, then the value would be TimesNewRoman-Bold, as described in the definition of name ID 4 above.)

5. Release and version information from the font vendor. *Version 1.00 June 1, 1990, 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.*

8. Manufacturer. *Monotype Corporation*

9. Designer. *Stanley Morison*

10. Description. *Designed in 1932 for the Times of London newspaper. Excellent readability and a narrow overall width, allowing more words per line than most fonts.*

11. URL of Vendor. *http://www.monotypeimaging.com*

12. URL of Designer. *http://www.monotypeimaging.com*

13. License Description. *This font may be installed on all of your machines and printers, but you may not sell or give these fonts to anyone else.*

14. License Info URL. *http://www.monotype.com/license/*

15. Reserved. Set to zero.

16. Preferred Family. No name string present, since it is the same as name ID 1 (Font Family name).

17. Preferred Subfamily. No name string present, since it is the same as name ID 2 (Font Subfamily name).

18. Compatible Full (Macintosh only). No name string present, since it is the same as name ID 4 (Full name).

19. Sample text. The quick brown fox jumps over the lazy dog.

20. PostScript CID findfont name. No name string present. Thus, the PostScript Name defined by name ID 6 should be used with the "findfont" invocation for locating the font in the context of a PostScript interpreter.

21. WWS family name: Since Times New Roman is a WWS font, this field does not need to be specified. If the font contained styles such as "caption", "display", "handwriting", etc, that would be noted here.

22. WWS subfamily name: Since Times New Roman is a WWS font, this field does not need to be specified.

The following is an example of only name IDs 6 and 20 in the CFF OFF Japanese font Kozuka Mincho Std Regular (other name IDs are also present in this font):

6. PostScript name: *KozMinStd-Regular*. Since a name ID 20 is present in the font (see below), then the PostScript name defined by name ID 6 should be used with the "composefont" invocation for locating the font in the context of a PostScript interpreter.

20. PostScript CID findfont name: *KozMinStd-Regular-83pv-RKSJ-H*, in a name record of Platform 1 [Macintosh], Platform-specific script 1 [Japanese], Language: 0xFFFF [English]. This name string is a PostScript name that should be used with the "findfont" invocation for locating the font in the context of a PostScript interpreter, and is associated with the encoding specified by the following cmap subtable, which must be present in the font: Platform: 1 [Macintosh]; Platform-specific encoding: 1 [Japanese]; Language: 0 [not language-specific].

The following is an example of family/subfamily naming for an extended, WWS-only family. Consider Adobe Caslon Pro, with six members: upright and italic versions of regular, semibold and bold weights. (Bit 8 of the fsSelection field of the OS/2 table, version 4, should be set for all six fonts, and none should include 'name' entries for IDs 21 or 22.)

Adobe Caslon Pro Regular:
Name ID 1: Adobe Caslon Pro
Name ID 2: Regular

Adobe Caslon Pro Italic:
Name ID 1: Adobe Caslon Pro
Name ID 2: Italic

Adobe Caslon Pro Semibold:
Name ID 1: Adobe Caslon Pro
Name ID 2: Bold
Name ID 16: Adobe Caslon Pro
Name ID 17: Semibold

Adobe Caslon Pro Semibold Italic:
Name ID 1: Adobe Caslon Pro
Name ID 2: Bold Italic
Name ID 16: Adobe Caslon Pro
Name ID 17: Semibold Italic

Adobe Caslon Pro Bold:
Name ID 1: Adobe Caslon Pro Bold
Name ID 2: Regular
Name ID 16: Adobe Caslon Pro
Name ID 17: Bold

Adobe Caslon Pro Bold Italic:
Name ID 1: Adobe Caslon Pro Bold
Name ID 2: Italic
Name ID 16: Adobe Caslon Pro
Name ID 17: Bold Italic

The following is an example of family/subfamily naming for an extended, non-WWS family. Consider Minion Pro Opticals, with 32 member fonts: upright and italic versions of regular, medium, semibold and bold weights in each of four optical sizes: regular, caption, display and subhead. The following show names for a sampling of the fonts in this family. (Bit 8 of the fsSelection field in the OS/2 table, version 4, should be set in those fonts that do not include 'name' entries for IDs 21 or 22, and only in those fonts.)

Minion Pro Regular:
Name ID 1: Minion Pro
Name ID 2: Regular

Minion Pro Italic:
Name ID 1: Minion Pro
Name ID 2: Italic

Minion Pro Semibold:
Name ID 1: Minion Pro SmBd
Name ID 2: Regular
Name ID 16: Minion Pro
Name ID 17: Semibold

Minion Pro Semibold Italic:
Name ID 1: Minion Pro SmBd
Name ID 2: Italic
Name ID 16: Minion Pro
Name ID 17: Semibold Italic

Minion Pro Caption:
Name ID 1: Minion Pro Capt
Name ID 2: Regular
Name ID 16: Minion Pro
Name ID 17: Caption
Name ID 21: Minion Pro Caption
Name ID 22: Regular

Minion Pro Semibold Italic Caption:
Name ID 1: Minion Pro SmBd Capt
Name ID 2: Italic
Name ID 16: Minion Pro
Name ID 17: Semibold Italic Caption
Name ID 21: Minion Pro Caption
Name ID 22: Semibold Italic

5.2.7 OS/2 – Global font information table

The OS/2 table consists of a set of metrics that are required in OFF fonts.

Type	Name of Entry	Comments
USHORT	Version	0x0000, 0x0001, 0x0002, 0x0003, 0x0004
SHORT	xAvgCharWidth	
USHORT	usWeightClass	
USHORT	usWidthClass	
USHORT	fsType	
SHORT	ySubscriptXSize	
SHORT	ySubscriptYSize	
SHORT	ySubscriptXOffset	
SHORT	ySubscriptYOffset	
SHORT	ySuperscriptXSize	

SHORT	ySuperscriptYSize	
SHORT	ySuperscriptXOffset	
SHORT	ySuperscriptYOffset	
SHORT	yStrikeoutSize	
SHORT	yStrikeoutPosition	
SHORT	sFamilyClass	
BYTE	Panose[10]	
ULONG	ulUnicodeRange1	Bits 0-31
ULONG	ulUnicodeRange2	Bits 32-63 version 0x0001 and later
ULONG	ulUnicodeRange3	Bits 64-95 version 0x0001 and later
ULONG	ulUnicodeRange4	Bits 96-127 version 0x0001 and later
CHAR	achVendID[4]	
USHORT	fsSelection	
USHORT	usFirstCharIndex	
USHORT	usLastCharIndex	
SHORT	sTypoAscender	
SHORT	sTypoDescender	
SHORT	sTypoLineGap	
USHORT	usWinAscent	
USHORT	usWinDescent	
ULONG	ulCodePageRange1	Bits 0-31 version 0x0001 and later
ULONG	ulCodePageRange2	Bits 32-63 version 0x0001 and later
SHORT	sxHeight	version 0x0002 and later
SHORT	sCapHeight	version 0x0002 and later
USHORT	usDefaultChar	version 0x0002 and later
USHORT	usBreakChar	version 0x0002 and later
USHORT	usMaxContext	version 0x0002 and later

5.2.7.1 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 four (4). See Annex C.

5.2.7.2 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 non-zero width glyphs in the font.

Comments: The value for xAvgCharWidth is calculated by obtaining the arithmetic average of the width of all non-zero width glyphs in the font. Furthermore, it is strongly recommended that implementers do not rely on this value for computing layout for lines of text. Especially, for cases where complex scripts are used. The calculation algorithm differs from one being used in previous versions of OS/2 table. For details see Annex B.

5.2.7.3 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

5.2.7.4 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: 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.

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

5.2.7.5 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 OFF Font Embedding DLL 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.

This version of the OS/2 table makes bits 0 - 3 a set of exclusive bits. In other words, at most one bit in this range may be set at a time. The purpose is to remove misunderstandings caused by previous behavior of using the least restrictive of the bits that are set.

Bit	Bit Mask	Description
	0x0000	Installable Embedding: No fsType bit is set. Thus fsType is zero. 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.
0	0x0001	Reserved, must be zero.
1	0x0002	Restricted License embedding: Fonts that have **only** this bit set **must not be modified, embedded or exchanged in any manner** without first obtaining permission of the legal owner. *Caution:* For Restricted License embedding to take effect, it must be the only level of embedding selected.
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 but must only be installed **temporarily** on other systems. In contrast to Preview & Print fonts, documents containing Editable fonts *may* be opened for reading, editing is permitted, and changes may be saved.
4-7		Reserved, must be zero.
8	0x0100	No subsetting: When this bit is set, the font may not be subsetted prior to embedding. Other embedding restrictions specified in bits 0-3 and 9 also apply.
9	0x0200	Bitmap embedding only: When this bit is set, only bitmaps contained in the font may be embedded. No outline data may be embedded. If there are no bitmaps available in the font, then the font is considered unembeddable and the embedding services will fail. Other embedding restrictions specified in bits 0-3 and 8 also apply.
10-15		Reserved, must be zero.

5.2.7.6 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.

5.2.7.7 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.

5.2.7.8 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.

5.2.7.9 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.

5.2.7.10 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.

5.2.7.11 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.

5.2.7.12 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.

5.2.7.13 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.

5.2.7.14 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.

5.2.7.15 yStrikeoutPosition

Format: 2-byte signed short

Units: Font design units

Title: Strikeout position.

Description: The position of the top 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, 460 is suggested.

5.2.7.16 sFamilyClass

Format: 2-byte signed short

Title: Font-family class and subclass.

Description: This parameter is a classification of font-family design.

Comments: The font class and font subclass are registered values per Annex B. the 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.

5.2.7.17 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 is used to describe the visual characteristics of a given typeface. If provided, these characteristics are then used to associate the font with other fonts of similar appearance having different names; the default values should be set to 'zero'. The variables for each digit are listed below.

Comments: The specification for assigning PANOSE values [14] can be found in bibliography and is maintained by Monotype Imaging Inc.

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

5.2.7.18 ulUnicodeRange

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, Unicode) and platform 3, encoding ID 10 (Microsoft platform, UCS-4). 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 ISO/IEC 10646 or the most recent version of the Unicode Standard for the list of Unicode ranges and characters.

Bit	Unicode Range	Block range
0	Basic Latin	0000-007F
1	Latin-1 Supplement	0080-00FF
2	Latin Extended-A	0100-017F
3	Latin Extended-B	0180-024F
4	IPA Extensions	0250-02AF
	Phonetic Extensions	1D00-1D7F
	Phonetic Extensions Supplement	1D80-1DBF
5	Spacing Modifier Letters	02B0-02FF
	Modifier Tone Letters	A700-A71F
6	Combining Diacritical Marks	0300-036F
	Combining Diacritical Marks Supplement	1DC0-1DFF
7	Greek and Coptic	0370-03FF
8	Coptic	2C80-2CFF
9	Cyrillic	0400-04FF
	Cyrillic Supplement	0500-052F
	Cyrillic Extended-A	2DE0-2DFF
	Cyrillic Extended-B	A640-A69F
10	Armenian	0530-058F
11	Hebrew	0590-05FF
12	Vai	A500-A63F
13	Arabic	0600-06FF

	Arabic Supplement	0750-077F
14	NKo	07C0-07FF
15	Devanagari	0900-097F
16	Bengali	0980-09FF
17	Gurmukhi	0A00-0A7F
18	Gujarati	0A80-0AFF
19	Oriya	0B00-0B7F
20	Tamil	0B80-0BFF
21	Telugu	0C00-0C7F
22	Kannada	0C80-0CFF
23	Malayalam	0D00-0D7F
24	Thai	0E00-0E7F
25	Lao	0E80-0EFF
26	Georgian	10A0-10FF
	Georgian Supplement	2D00-2D2F
27	Balinese	1B00-1B7F
28	Hangul Jamo	1100-11FF
29	Latin Extended Additional	1E00-1EFF
	Latin Extended-C	2C60-2C7F
	Latin Extended-D	A720-A7FF
30	Greek Extended	1F00-1FFF
31	General Punctuation	2000-206F
	Supplemental Punctuation	2E00-2E7F
32	Superscripts And Subscripts	2070-209F
33	Currency Symbols	20A0-20CF
34	Combining Diacritical Marks For Symbols	20D0-20FF
35	Letterlike Symbols	2100-214F
36	Number Forms	2150-218F
37	Arrows	2190-21FF
	Supplemental Arrows-A	27F0-27FF
	Supplemental Arrows-B	2900-297F
	Miscellaneous Symbols and Arrows	2B00-2BFF
38	Mathematical Operators	2200-22FF
	Supplemental Mathematical Operators	2A00-2AFF
	Miscellaneous Mathematical Symbols-A	27C0-27EF

	Miscellaneous Mathematical Symbols-B	2980-29FF
39	Miscellaneous Technical	2300-23FF
40	Control Pictures	2400-243F
41	Optical Character Recognition	2440-245F
42	Enclosed Alphanumerics	2460-24FF
43	Box Drawing	2500-257F
44	Block Elements	2580-259F
45	Geometric Shapes	25A0-25FF
46	Miscellaneous Symbols	2600-26FF
47	Dingbats	2700-27BF
48	CJK Symbols And Punctuation	3000-303F
49	Hiragana	3040-309F
50	Katakana	30A0-30FF
	Katakana Phonetic Extensions	31F0-31FF
51	Bopomofo	3100-312F
	Bopomofo Extended	31A0-31BF
52	Hangul Compatibility Jamo	3130-318F
53	Phags-pa	A840-A87F
54	Enclosed CJK Letters And Months	3200-32FF
55	CJK Compatibility	3300-33FF
56	Hangul Syllables	AC00-D7AF
57	Non-Plane 0 *	D800-DFFF
58	Phoenician	10900-1091F
59	CJK Unified Ideographs	4E00-9FFF
	CJK Radicals Supplement	2E80-2EFF
	Kangxi Radicals	2F00-2FDF
	Ideographic Description Characters	2FF0-2FFF
	CJK Unified Ideographs Extension A	3400-4DBF
	CJK Unified Ideographs Extension B	20000-2A6DF
	Kanbun	3190-319F
60	Private Use Area (plane 0)	E000-F8FF
61	CJK Strokes	31C0-31EF
	CJK Compatibility Ideographs	F900-FAFF
	CJK Compatibility Ideographs Supplement	2F800-2Fa1F
62	Alphabetic Presentation Forms	FB00-FB4F

63	Arabic Presentation Forms-A	FB50-FDFF
64	Combining Half Marks	FE20-FE2F
65	Vertical Forms	FE10-FE1F
	CJK Compatibility Forms	FE30-FE4F
66	Small Form Variants	FE50-FE6F
67	Arabic Presentation Forms-B	FE70-FEFF
68	Halfwidth And Fullwidth Forms	FF00-FFEF
69	Specials	FFF0-FFFF
70	Tibetan	0F00-0FFF
71	Syriac	0700-074F
72	Thaana	0780-07BF
73	Sinhala	0D80-0DFF
74	Myanmar	1000-109F
75	Ethiopic	1200-137F
	Ethiopic Supplement	1380-139F
	Ethiopic Extended	2D80-2DDF
76	Cherokee	13A0-13FF
77	Unified Canadian Aboriginal Syllabics	1400-167F
78	Ogham	1680-169F
79	Runic	16A0-16FF
80	Khmer	1780-17FF
	Khmer Symbols	19E0-19FF
81	Mongolian	1800-18AF
82	Braille Patterns	2800-28FF
83	Yi Syllables	A000-A48F
	Yi Radicals	A490-A4CF
84	Tagalog	1700-171F
	Hanunoo	1720-173F
	Buhid	1740-175F
	Tagbanwa	1760-177F
85	Old Italic	10300-1032F
86	Gothic	10330-1034F
87	Deseret	10400-1044F
88	Byzantine Musical Symbols	1D000-1D0FF
	Musical Symbols	1D100-1D1FF

	Ancient Greek Musical Notation	1D200-1D24F
89	Mathematical Alphanumeric Symbols	1D400-1D7FF
90	Private Use (plane 15)	FF000-FFFFD
	Private Use (plane 16)	100000-10FFFD
91	Variation Selectors	FE00-FE0F
	Variation Selectors Supplement	E0100-E01EF
92	Tags	E0000-E007F
93	Limbu	1900-194F
94	Tai Le	1950-197F
95	New Tai Lue	1980-19DF
96	Buginese	1A00-1A1F
97	Glagolitic	2C00-2C5F
98	Tifinagh	2D30-2D7F
99	Yijing Hexagram Symbols	4DC0-4DFF
100	Syloti Nagri	A800-A82F
101	Linear B Syllabary	10000-1007F
	Linear B Ideograms	10080-100FF
	Aegean Numbers	10100-1013F
102	Ancient Greek Numbers	10140-1018F
103	Ugaritic	10380-1039F
104	Old Persian	103A0-103DF
105	Shavian	10450-1047F
106	Osmanya	10480-104AF
107	Cypriot Syllabary	10800-1083F
108	Kharoshthi	10A00-10A5F
109	Tai Xuan Jing Symbols	1D300-1D35F
110	Cuneiform	12000-123FF
	Cuneiform Numbers and Punctuation	12400-1247F
111	Counting Rod Numerals	1D360-1D37F
112	Sundanese	1B80-1BBF
113	Lepcha	1C00-1C4F
114	Ol Chiki	1C50-1C7F
115	Saurashtra	A880-A8DF
116	Kayah Li	A900-A92F
117	Rejang	A930-A95F

118	Cham	AA00-AA5F
119	Ancient Symbols	10190-101CF
120	Phaistos Disc	101D0-101FF
121	Carian	102A0-102DF
	Lycian	10280-1029F
	Lydian	10920-1093F
122	Domino Tiles	1F030-1F09F
	Mahjong Tiles	1F000-1F02F
123-127	Reserved	

NOTE * Setting bit 57 implies that there is at least one codepoint beyond the Basic Multilingual Plane that is supported by this font.

5.2.7.19 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. The Vendor ID list can be accessed via the informative reference 6 in the bibliography.

5.2.7.20 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 or oblique 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.
7		USE_TYPO_METRICS	If set, it is strongly recommended to use OS/2.sTypoAscender - OS/2.sTypoDescender+ OS/2.sTypoLineGap as a value for default line spacing for this font. (OS/2 version 4 and later)
8		WWS	The font family this face belongs to is composed of faces that only differ in weight, width and slope (please see more detailed description below.) (OS/2 version 4 and later)
9		OBLIQUE	Font contains oblique characters. (OS/2 version 4 and later)

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).

Bit 7 was specified in OS/2 table v. 4. If fonts created with an earlier version of the OS/2 table are updated to the current version of the OS/2 table, then, in order to minimize potential reflow of existing documents which use the fonts, the bit would be set only for fonts for which using the OS/2.usWin* metrics for line height would yield significantly inferior results than using the OS/2.sTypo* values. New fonts, however, are not constrained by backward compatibility situations, and so are free to set this bit always.

If bit 8 is set in OS/2 table v. 4, then the font's typographic family contains faces that differ only in one or more of the attributes weight, width and slope. For example, a family with only weight and slope attributes will set this bit.

If unset in OS/2 table v. 4, then this font's typographic family contains faces that differ in attributes other than weight, width or slope. For example, a family with faces that differ only by weight, slope, and optical size will not set this bit.

This bit must be unset in OS/2 table versions less than 4. In these cases, it is not possible to determine any information about the typographic family's attributes by examining this bit.

In this context, "typographic family" is the Microsoft Unicode string for name ID 16, if present, else the Microsoft Unicode string for name ID 1; "weight" is OS/2.usWeightClass; "width" is OS/2.usWidthClass; "slope" is OS/2.fsSelection bit 0 (ITALIC) and bit 9 (OBLIQUE).

If bit 9 is set in OS/2 table v. 4, then this font is to be considered an "oblique" style by processes which make a distinction between oblique and italic styles, e.g. Cascading Style Sheets font matching. For example, a font created by algorithmically slanting an upright face will set this bit.

If unset in OS/2 table v. 4, then this font is not to be considered an "oblique" style. For example, a font that has a classic italic design will not set this bit.

This bit must be unset in OS/2 table versions less than 4. In these cases, it is not possible to

determine any information about this font's attributes by examining this bit.

This bit, unlike the ITALIC bit, is not related to style-linking for Windows GDI or Mac OS applications in a traditional four-member family of regular, italic, bold and bold italic.". It may be set or unset independently of the ITALIC bit. In most cases, if OBLIQUE is set, then ITALIC will also be set, though this is not required.

5.2.7.21 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 platform- specific encoding ID 0 or 1. For most fonts supporting Win-ANSI or other character sets, this value would be 0x0020. This field cannot represent supplementary character values (codepoints greater than 0xFFFF). Fonts that support supplementary characters should set the value in this field to 0xFFFF if the minimum index value is a supplementary character.

5.2.7.22 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. This field cannot represent supplementary character values (codepoints greater than 0xFFFF). Fonts that support supplementary characters should set the value in this field to 0xFFFF.

5.2.7.23 sTypoAscender

Format: SHORT

Description: The typographic ascender for this font. Remember that this is not the same as the Ascender value in the 'hhea' table. One good source for sTypoAscender in Latin based fonts is the Ascender value from an AFM file. For CJK fonts see below.

The suggested usage for sTypoAscender 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 compatibility 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.

For CJK (Chinese, Japanese, and Korean) fonts that are intended to be used for vertical writing (in addition to horizontal writing), the required value for sTypoAscender is that which describes the top of the of the ideographic em-box. For example, if the ideographic em-box of the font extends from coordinates 0,-120 to 1000,880 (that is, a 1000x1000 box set 120 design units below the Latin baseline), then the value of sTypoAscender must be set to 880. Failing to adhere to these requirements will result in incorrect vertical layout.

Also see the Recommendations clause 7 for more on this field.

5.2.7.24 sTypoDescender

Format: SHORT

Description: The typographic descender for this font.. One good source for sTypoDescender in Latin based fonts is the Descender value from an AFM file. For CJK fonts see below.

The suggested usage for sTypoDescender is that it be used in conjunction with unitsPerEm to compute 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.
For CJK (Chinese, Japanese, and Korean) fonts that are intended to be used for vertical writing (in addition to horizontal writing), the required value for sTypoDescender is that which describes the bottom of the ideographic em-box. For example, if the ideographic em-box of the font extends from coordinates 0,-120 to 1000,880 (that is, a 1000x1000 box set 120 design units below the Latin baseline), then the value of sTypoDescender must be set to -120. Failing to adhere to these requirements will result in incorrect vertical layout.

Also see the Recommendations clause 7 for more on this field.

5.2.7.25 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.
The suggested usage for usTypoLineGap is that it be used in conjunction with unitsPerEm to compute 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 clause 7, "Recommendations for OFF 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.

5.2.7.26 usWinAscent

Format: 2-byte USHORT

Description: The ascender metric for Windows. For platform 3 encoding 0 fonts, it is the same as yMax. Windows will clip the bitmap of any portion of a glyph that appears above this value. Some applications use this value to determine default line spacing. This is strongly discouraged. The typographic ascender, descender and line gap fields in conjunction with unitsPerEm should be used for this purpose. Developers should set this field keeping the above factors in mind.
If any clipping is unacceptable, then the value should be set to yMax.
However, if a developer desires to provide appropriate default line spacing using this field, for those applications that continue to use this field for doing so (against OFF recommendations), then the value should be set appropriately. In such a case, it may result in some glyph bitmaps being clipped.

5.2.7.27 usWinDescent

Format: 2-byte USHORT

Description: The descender metric for Windows. For platform 3 encoding 0 fonts, it is the same as -yMin. Windows will clip the bitmap of any portion of a glyph that appears below this value. Some applications use this value to determine default line spacing. This is strongly discouraged. The typographic ascender, descender and line gap fields in conjunction with unitsPerEm should be used for this purpose. Developers should set this field keeping the above factors in mind.
If any clipping is unacceptable, then the value should be set to yMin.
However, if a developer desires to provide appropriate default line spacing using this field, for those applications that continue to use this field for doing so (against OFF recommendations), then the value should be set appropriately. In such a case, it may result in some glyph bitmaps being clipped.

5.2.7.28 ulCodePageRange

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 (Windows 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	1258	Vietnamese
9-15		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
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

5.2.7.29 sxHeight

Format: SHORT

Description: This metric specifies the distance between the baseline and the approximate height of non-ascending lowercase letters measured in FUnits. This value would normally be specified by a type designer but in situations where that is not possible, for example when a legacy font is being converted, the value may be set equal to the top of the unscaled and unhinted glyph bounding box of the glyph encoded at U+0078 (LATIN SMALL LETTER X). If no glyph is encoded in this position the field should be set to 0.

This metric, if specified, can be used in font substitution: the xHeight value of one font can be scaled to approximate the apparent size of another.

5.2.7.30 sCapHeight

Format: SHORT

Description: This metric specifies the distance between the baseline and the approximate height of uppercase letters measured in FUnits. This value would normally be specified by a type designer but in situations where that is not possible, for example when a legacy font is being converted, the value may be set equal to the top of the unscaled and unhinted glyph bounding box of the glyph encoded at U+0048 (LATIN CAPITAL LETTER H). If no glyph is encoded in this position the field should be set to 0.

This metric, if specified, can be used in systems that specify type size by capital height measured in millimeters. It can also be used as an alignment metric; the top of a drop capital, for instance, can be aligned to the sCapHeight metric of the first line of text.

5.2.7.31 usDefaultChar

Format: USHORT

Description: Whenever a request is made for a character that is not in the font, Windows provides this default character. If the value of this field is zero, glyph ID 0 is to be used for the default character otherwise this is the Unicode encoding of the glyph that Windows uses as the default character. This field cannot represent supplementary character values (codepoints greater than 0xFFFF), and so applications are strongly discouraged from using this field.

5.2.7.32 usBreakChar

Format: USHORT

Description: This is the Unicode encoding of the glyph that Windows uses as the break character. The break character is used to separate words and justify text. Most fonts specify 'space' as the break character. This field cannot represent supplementary character values (codepoints greater than 0xFFFF), and so applications are strongly discouraged from using this field.

5.2.7.33 usMaxContext

Format: USHORT

Description: The maximum length of a target glyph context for any feature in this font. For example, a font which has only a pair kerning feature should set this field to 2. If the font also has a ligature feature in which the glyph sequence 'f f i' is substituted by the ligature 'ffi', then this field should be set to 3. This field could be useful to sophisticated line-breaking engines in determining how far they should look ahead to test whether something could change that effect the line breaking. For chaining contextual lookups, the length of the string (covered glyph) + (input sequence) + (lookahead sequence) should be considered.

5.2.8 Font class parameters - see informative Annex B for details.

5.2.9 post – PostScript

This table contains additional information needed to use TrueType or OFF fonts on PostScript printers. This includes data for the FontInfo dictionary entry and the PostScript names of all the glyphs. For more information about PostScript names, see the Adobe document Unicode and Glyph Names in the informative reference 3 in the bibliography.

Table Versions 1.0, 2.0, and 2.5 refer to TrueType fonts and OFF fonts with TrueType data. OFF fonts with TrueType data may also use Version 3.0. OFF fonts with CFF data use Version 3.0 only.

The table begins as follows:

Type	Name	Description
Fixed	Version	0x00010000 for version 1.0 0x00020000 for version 2.0 0x00025000 for version 2.5 *(deprecated)* 0x00030000 for version 3.0
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	This is the suggested distance of the top of the underline from the baseline (negative values indicate below baseline). The PostScript definition of this FontInfo dictionary key (the y coordinate of the center of the stroke) is not used for historical reasons. The value of the PostScript key may be calculated by subtracting half the underlineThickness from the value of this field.
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 an OFF font is downloaded.
ULONG	maxMemType42	Maximum memory usage when an OFF font is downloaded.

ULONG	minMemType1	Minimum memory usage when an OFF font is downloaded as a Type 1 font.
ULONG	maxMemType1	Maximum memory usage when an OFF 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 OFF 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 font scaler. Runtime memory usage could be calculated by rendering characters at different point sizes and comparing memory use.

If the table version is 1.0 or 3.0, the table ends here. The additional entries for versions 2.0 and 2.5 are shown below. Version 4.0 is reserved to the specification published in the informative reference [7] of the bibliography

5.2.9.1 Version 1.0

This TrueType-based font file contains exactly the 258 glyphs in the standard Macintosh TrueType font file. See the WGL4.0 Character Set in the informative reference 2 in the bibliography for a list of the Macintosh glyphs. As a result, the glyph names are taken from the system with no storage required by the font.

5.2.9.2 Version 2.0

This is the version required by TrueType-based fonts to be used on Windows.

Type	Name	Description
USHORT	numberOfGlyphs	Number of glyphs (this should be the same as numGlyphs in 'maxp' table).
USHORT	glyphNameIndex[numGlyphs].	This is not an Offset, but is the ordinal number of the glyph in 'post' string tables.
CHAR	names[numberNewGlyphs]	Glyph names with length bytes [variable] (a Pascal string).

This TrueType-based 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.*

5.2.9.3 Version 2.5

This version of the 'post' table has been deprecated.

5.2.9.4 Version 3.0

This version is used by OFF fonts with TrueType or CFF data. The version makes it possible to create a special font that is not burdened with a large 'post' table set of glyph names.

This version specifies that no PostScript name information is provided for the glyphs in this font file. The printing behavior of this version 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 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.*

5.3 TrueType outline tables

For OFF fonts based on TrueType outlines, the following tables are used:

TrueType Outlines Tables

Tag	Name
cvt	Control Value Table
fpgm	Font program
glyf	Glyph data
loca	Index to location
prep	CV Program

5.3.1 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. The length of the table must be an integral number of FWORD units.

Type	Description
FWORD[*n*]	List of *n* values referenceable by instructions. *n* is the number of FWORD items that fit in the size of the table.

5.3.2 fpgm – Font program

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

This table is optional.

Type	Description
BYTE[*n*]	Instructions. *n* is the number of BYTE items that fit in the size of the table.

5.3.3 glyf – Glyf data

5.3.3.1 Table structure

This table contains information that describes the glyphs in the font in the TrueType outline format. Information regarding the rasterizer (scaler) refers to the TrueType rasterizer.

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.
SHORT	xMin	Minimum x for coordinate data.
SHORT	yMin	Minimum y for coordinate data.
SHORT	xMax	Maximum x for coordinate data.
SHORT	yMax	Maximum y for coordinate data.

NOTE 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). 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.

5.3.3.1.1 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. See TrueType Instruction Set as listed in normative reference 3 in clause 2.
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 bit. 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. If not set, 2 bytes.
y-Short Vector	2	If set, the corresponding y-coordinate is 1 byte long. If not set, 2 bytes.
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.

5.3.3.1.2 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").

Type	Name	Description
USHORT	Flags	component flag
USHORT	glyphIndex	glyph index of component
VARIABLE	Argument1	x-Offset for component or point number; type depends on bits 0 and 1 in component flags
VARIABLE	Argument2	y-Offset for component or point number; type depends on bits 0 and 1 in component flags
Transformation Option		

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]
```

NOTE The TrueType instruction set is available via normative reference 3 in clause 2.

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.
OVERLAP_COMPOUND	10	Reserved
SCALED_COMPONENT_OFFSET	11	Reserved
UNSCALED_COMPONENT_OFFSET	12	Composite designed not to have the component Offset scaled.

The purpose of USE_MY_METRICS is to force the lsb and rsb to take on a desired value. For example, an i-circumflex (U+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 The behavior of the USE_MY_METRICS operation is undefined for rotated composite components.

5.3.4 loca – Index to location

The loca table stores the Offsets to the locations of the glyphs in the font, relative to the beginning of the glyf 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 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. If a glyph has no outlines, the offset loca[n] = loca[n+1]. In the particular case of the last glyph(s), loca[n] will be equal the length of the glyph data ('glyf') table. The offsets must be in ascending order with loca[n] <= loca[n+1].

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 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.

5.3.5 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 CV Program but since no glyph is associated with it, instructions intended to move points within a particular glyph outline cannot be used in the CV Program. The name 'prep' is anachronistic (the table used to be known as the Pre Program table).

Type	Description
BYTE[*n*]	Set of instructions executed whenever point size or font or transformation change. *n* is the number of BYTE items that fit in the size of the table.

5.4 PostScript outline tables

For OFF fonts based on PostScript outlines, the following tables are used:

Tag	Name
CFF	PostScript font program (compact font format)
VORG	Vertical Origin

It is strongly recommended that CFF OpenType fonts that are used for vertical writing include a Vertical Origin ('VORG') table.

5.4.1 CFF – PostScript font program (Compact Font Format) table

This table contains a compact representation of a PostScript Type 1, or CIDFont and is structured according to Adobe Technical Note #5176: "The Compact Font Format Specification" [5] and Adobe Technical Note #5177: "Type 2 Charstring Format" [4].

Existing TrueType fonts use a glyph index to specify and access glyphs within a font, e.g. to index the loca table and thereby access glyph data in the glyf table. This concept is retained in OFF PostScript fonts except that glyph data is accessed through the CharStrings INDEX of the CFF table.

5.4.2 VORG – Vertical origin table

This table specifies the y coordinate of the vertical origin of every glyph in the font.

This table may be optionally present only in CFF OFF fonts. If present in TrueType OFF fonts it must be ignored by font clients, just as any other unrecognized table would be. This is because this table is not needed for TrueType OFF fonts: the Vertical Metrics ('vmtx') and Glyph Data ('glyf') tables in TrueType OFF fonts provide all the information necessary to accurately calculate the y-coordinate of a glyph's vertical origin. See the "Vertical Origin and Advance Height" in the 'vmtx' table specification for more details.

The 'vmtx' and Vertical Header ('vhea') tables continue to be required for all OFF fonts that support vertical writing. Advance heights must continue to be obtained from the 'vmtx' table; that is the only place they are stored.

If a 'VORG' table is present in a CFF OFF font, a font client may choose to obtain the y coordinate of a glyph's vertical origin either:

1. directly from the 'VORG', or:

2. by first calculating the top of the glyph's bounding box from the CFF charstring data and then adding to it the glyph's top side bearing from the 'vmtx' table.

The former method offers the advantage of increased accuracy and efficiency, since bounding box results calculated from the CFF charstring as in the latter method can differ depending on the rounding decisions and data types of the bounding box algorithm. The latter method provides compatibility for font clients who are either unaware of or choose not to support the 'VORG'.

Thus, the 'VORG' doesn't add any new font metric values per se; it simply improves accuracy and efficiency for CFF OFF font clients, since the intermediate step of calculating bounding boxes from the CFF charstring is rendered unnecessary.

See clause 6 "OFF CJK Font Guidelines" for more information about constructing CJK (Chinese, Japanese, and Korean) fonts.

Vertical Origin Table Format

Type	Name	Description
USHORT	majorVersion	Major version (starting at 1). Set to 1.
USHORT	minorVersion	Minor version (starting at 0). Set to 0.
SHORT	defaultVertOriginY	The y coordinate of a glyph's vertical origin, in the font's design coordinate system, to be used if no entry is present for the glyph in the vertOriginYMetrics array.
USHORT	numVertOriginYMetrics	Number of elements in the vertOriginYMetrics array.

This is immediately followed by the vertOriginYMetrics array (if numVertOriginYMetrics is non-zero), which has numVertOriginYMetrics elements of the following format:

Type	Name	Description
USHORT	glyphIndex	Glyph index.
SHORT	vertOriginY	Y coordinate, in the font's design coordinate system, of the vertical origin of glyph with index glyphIndex.

This array must be sorted by increasing glyphIndex, and must not have more than one element with the same glyphIndex. In a size-optimized implementation, glyphs whose vertical origin's y coordinate equals defaultVertOriginY will not have an entry in this array.

If all glyphs in a font share the same defaultVertOriginY value, the length of the 'VORG' table will be 8 bytes in a size-optimized implementation, since the vertOriginYMetrics array will be absent.

Typically only the full-width Latin glyphs in an East Asian font will have entries in the vertOriginYMetrics array. Glyphs rotated for vertical writing, as used in the Vertical Alternates and Rotation ('vrt2') feature, for example, can take advantage of the default value if they are designed appropriately.

In the following example of a complete 'VORG' table for a 1000-unit-em font, every glyph in the font is specified as having a vertOriginY of 880 except for glyphs with glyph indexes 10, 12, and 13:

majorVersion =1
minorVersion =0
defaultVertOriginY =880
numVertOriginYMetrics=3
--- vertOriginYMetrics[index]=(glyphIndex,vertOriginY)
[0]=(10,889)
[1]=(12,861)
[2]=(13,849)

5.5 Bitmap glyph tables

OFF fonts may also contain bitmaps of glyphs, in addition to outlines. Hand-tuned bitmaps are especially useful in OFF fonts for representing complex glyphs at very small sizes. If a bitmap for a particular size is provided in a font, it will be used by the system instead of the outline when rendering the glyph.

NOTE ATM does not currently support hinted bitmaps in OFF fonts.)

Tag	Name
EBDT	Embedded bitmap data
EBLC	Embedded bitmap location data
EBSC	Embedded bitmap scaling data

5.5.1 EBDT – Embedded bitmap data table

5.5.1.1 Table structure

Three tables are used to embed bitmaps in OFF 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.

OFF 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 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 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

5.5.1.2 Glyph bitmap data formats

The nine different formats currently defined for glyph bitmap data are listed and described below. Different formats are better for different purposes.

In all formats, if the bitDepth is greater than 1, all of a pixel's bits are stored consecutively in memory, and all of a row's pixels are stored consecutively.

NOTE Each of these formats can contain black/white or grayscale bitmaps depending on the setting of the bitDepth field in the 'EBLC' table. For performance reasons, we recommend using a byte-aligned format for embedded bitmaps with bitDepth of 8.

5.5.1.2.1 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.

5.5.1.2.2 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.

5.5.1.2.3 Format 3: (obsolete)

5.5.1.2.4 Format 4: metrics in EBLC, compressed data

NOTE Glyph bitmap format 4 is a compressed format used by Macintosh platform in some of the East Asian fonts.

5.5.1.2.5 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.

5.5.1.2.6 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.

5.5.1.2.7 Format7: 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.

5.5.1.2.8 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.

5.5.1.2.9 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

5.5.1.2.10 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.

5.5.2 EBLC – Embedded bitmap location table

5.5.2.1 Table structure & data types

Three tables are used to embed bitmaps in OFF 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. OFF 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 begins with a header containing the table version and number of strikes. An OFF 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	the Windows rasterizer supports the following bitDepth values, as described below: 1, 2, 4, and 8.
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.

5.5.2.2 Description of table entries

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 bitDepth field is used to specify the number of levels of gray used in the embedded bitmaps. The Windows rasterizer v.1.7 or greater support the following values.

bitDepth

Value	Description
1	Black/white
2	4 levels of gray
4	16 levels of gray
8	256 levels of gray

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	Name	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.

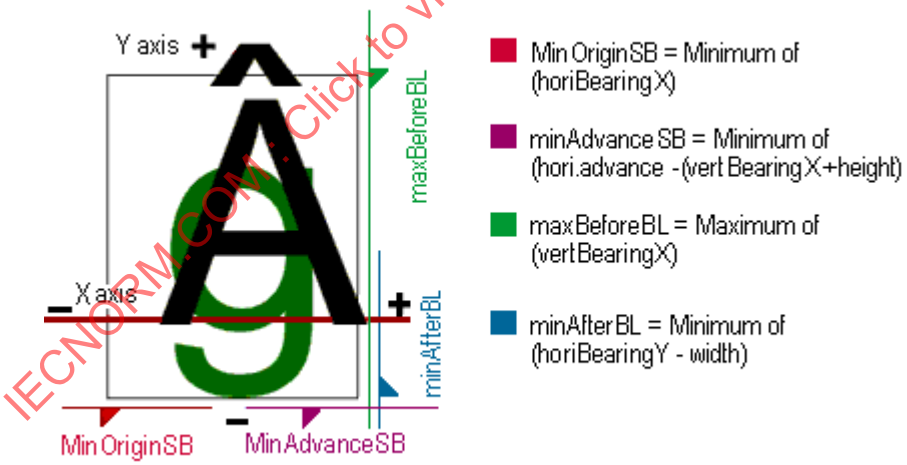

Figure 1 – Horizontal text

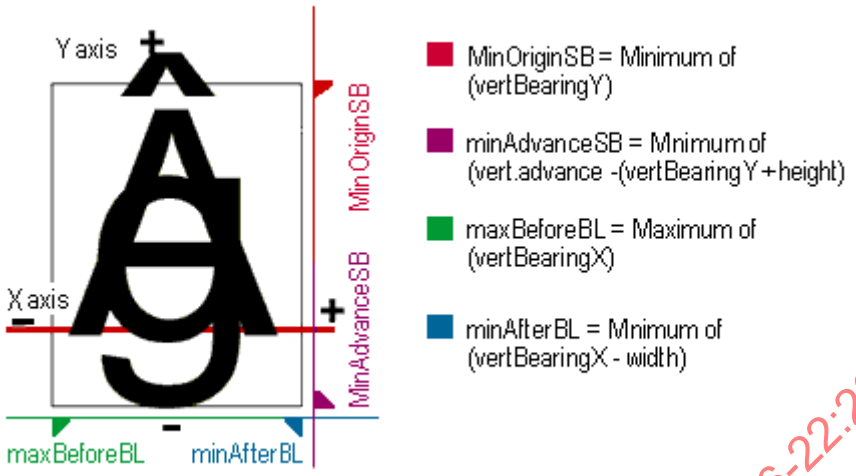

Figure 2 – Vertical text

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 'EBDT'. 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.

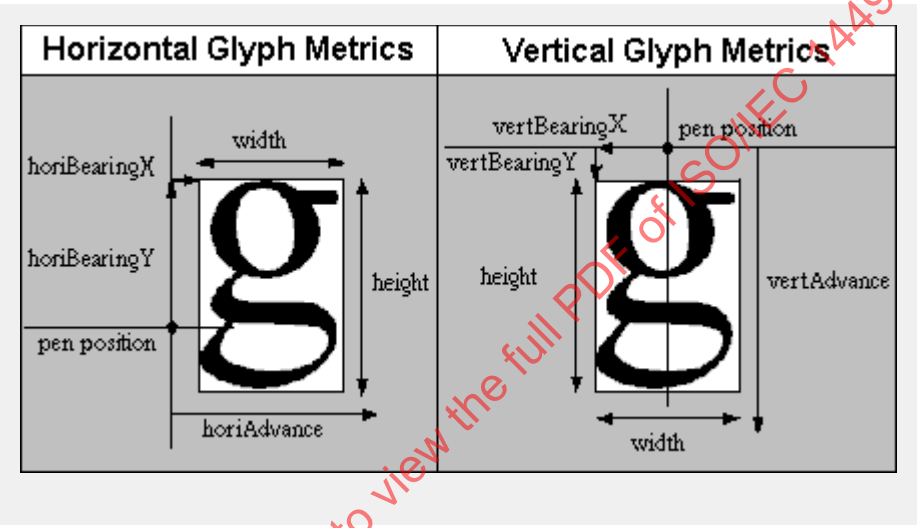

Figure 3 – Glyph metrics

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	additionalOffset ToIndexSubtable	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.

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 are an odd number of elements in these arrays it is necessary to add an extra padding element to maintain proper alignment.

5.5.2.3 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.

5.6 Optional tables

Tag	Name
DSIG	Digital signature
gasp	Grid-fitting/Scan-conversion
hdmx	Horizontal device metrics
kern	Kerning
LTSH	Linear threshold data
PCLT	PCL 5 data
VDMX	Vertical device metrics
vhea	Vertical Metrics header
vmtx	Vertical Metrics

5.6.1 DSIG – Digital signature table

The DSIG table contains the digital signature of the OFF font. Signature formats are widely documented and rely on a key pair architecture. Software developers, or publishers posting material on the Internet, create signatures using a private key. Operating systems or applications authenticate the signature using a public key.

The W3C and major software and operating system developers have specified security standards that describe signature formats, specify secure collections of web objects, and recommend authentication architecture. OFF fonts with signatures will support these standards.

OFF fonts offer many security features:

- Operating systems and browsing applications can identify the source and integrity of font files before using them,
- Font developers can specify embedding restrictions in OFF fonts, and these restrictions cannot be altered in a font signed by the developer.

The enforcement of signatures is an administrative policy, enabled by the operating system. Windows will soon require installed software components, including fonts, to be signed. Internet browsers will also give users and administrators the ability to screen out unsigned objects obtained on-line, including web pages, fonts, graphics, and software components.

Anyone can obtain identity certificates and encryption keys from a certifying agency, such as Verisign or GTE's Cybertrust, free or at a very low cost.

The DSIG table is organized as follows. The first portion of the table is the header:

DSIG Header		
Type	Name	Description
ULONG	ulVersion	Version number of the DSIG table (0x00000001)
USHORT	usNumSigs	Number of signatures in the table
USHORT	usFlag	permission flags Bit 0: cannot be resigned Bits 1-7: Reserved (Set to 0)

The version of the DSIG table is expressed as a ULONG, beginning at 0. The version of the DSIG table currently used is version 1 (0x00000001).

Permission bit 0 allows a party signing the font to prevent any other parties from also signing the font (counter-signatures). If this bit is set to zero (0) the font may have a signature applied over the existing digital signature(s). A party who wants to ensure that their signature is the last signature can set this bit.

The DSIG header information is followed by entries for each of the signatures in the table specifying format and Offset information:

Format/Offset Table		
Type	Name	Description
ULONG	ulFormat	format of the signature
ULONG	ulLength	Length of signature in bytes
ULONG	ulOffset	Offset to the signature block from the beginning of the table

This information is then followed by one or more signature blocks:

Signature Block		
Type	Name	Description
USHORT	usReserved1	Reserved for later use; 0 for now
USHORT	usReserved2	Reserved for later use; 0 for now
ULONG	cbSignature	Length (in bytes) of the PKCS#7 packet in pbSignature
BYTE []	bSignature	PKCS#7 packet

The format identifier specifies both the format of the signature object, as well as the hashing algorithm used to create and authenticate the signature. Currently only one format is defined. Format 1 supports PKCS#7 signatures with X.509 certificates and counter-signatures, as these signatures have been standardized for use by the W3C with the participation of numerous software developers.

For more information about PKCS#7 signatures see [10]

For more information about counter-signatures, see [11]

Format 1: For whole fonts, with either TrueType outlines and/or CFF data

PKCS#7 or PKCS#9. The signed content digest is created as follows:

1. If there is an existing DSIG table in the font,
    1. Remove DSIG table from font.
    2. Remove DSIG table entry from sfnt Table Directory.
    3. Adjust table Offsets as necessary.
    4. Zero out the file checksum in the head table.
    5. Add the usFlag (reserved, set at 1 for now) to the stream of bytes
2. Hash the full stream of bytes using a secure one-way hash (such as MD5) to create the content digest.
3. Create the PKCS#7 signature block using the content digest.
4. Create a new DSIG table containing the signature block.
5. Add the DSIG table to the font, adjusting table Offsets as necessary.
6. Add a DSIG table entry to the sfnt Table Directory.
7. Recalculate the checksum in the head table.

Prior to signing a font file, ensure that all the following attributes are true.

- The magic number in the head table is correct.
- Given the number of tables value in the Offset table, the other values in the Offset table are consistent.
- The tags of the tables are ordered alphabetically and there are no duplicate tags.
- The Offset of each table is a multiple of 4. (That is, tables are long word aligned.)
- The first actual table in the file comes immediately after the directory of tables.
- If the tables are sorted by Offset, then for all tables i (where index 0 means the the table with the smallest Offset), Offset[i] + Length[i] <= Offset[i+1] and Offset[i] + Length[i] >= Offset[i+1] - 3. In other words, the tables do not overlap, and there are at most 3 bytes of padding between tables.
- The pad bytes between tables are all zeros.
- The Offset of the last table in the file plus its length is not greater than the size of the file.
- The checksums of all tables are correct.
- The head table's checkSumAdjustment field is correct.

Signatures for TrueType Collections

The DSIG table for a TrueType Collection (TTC) must be the last table in the TTC file. The Offset and checksum to the table is put in the TTCHeader (version 2). Signatures of TTC files are expected to be Format 1 signatures.

The signature of a TTC file applies to the entire file, not to the individual fonts contained within the TTC. Signing the TTC file ensures that other contents are not added to the TTC.

Individual fonts included in a TrueType collection should not be individually signed as the process of making the TTC could invalidate the signature on the font.

5.6.2 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, the rasterizer may 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 or 1)
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_DOGRAY	Use grayscale rendering
GASP_GRIDFIT	Use gridfitting
GASP_SYMMETRIC_SMOOTHING	Use smoothing along multiple axes with ClearType® Only supported in version 1 of 'gasp' table
GASP_SYMMETRIC_GRIDFIT	Use gridfitting with ClearType symmetric smoothing Only supported in version 1 of 'gasp' table

The set of bit flags may be extended in the future. The first two bit flags operate independently of the following two bit flags. If font smoothing is enabled, then the first two bit flags are used. If ClearType is enabled, then the following two bit flags are used. The seven 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
GASP_SYMMETRIC_GRIDFIT	0x0004	typically always enabled
GASP_SYMMETRIC_SMOOTHING	0x0008	larger screen sizes, typically ppem>15, most commonly used with the gridfit flag.
GASP_SYMMETRIC_SMOOTHING | GASP_ SYMMETRIC_GRIDFIT	0x000C	Large screen sizes, typically ppem>15
(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

Flag	Value	Font Smoothing Meaning	ClearType with Symmetric Smoothing Meaning
version	0x0001		
numRanges	0x0004		
Range[0], Flag	0x0008 0x000a	ppem<=8, grayscale only	ppem<=8, symmetric ClearType only
Range[1], Flag	0x0010 0x0005	9<=ppem<=16, gridfit only	9<=ppem<=16, symmetric gridfit only
Range[2], Flag	0x0013 0x0007	17<=ppem<=19, gridfit and grayscale	17<=ppem<=19, symmetric gridfit
Range[3], Flag	0xFFFF 0x000F	20<=ppem, gridfit and grayscale	20<=ppem, symmetric gridfit and symmetric smoothing

5.6.3 hdmx – Horizontal device metrics

The hdmx table relates to OFF fonts with TrueType outlines. 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 For non-square pixel grids, the character width (in pixels) will be used to determine which device record to use. For example, a 12 point character on a device with a resolution of 72x96 would be 12 pixels high and 16 pixels wide. 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 clause 7 "Recommendations for OFF Fonts" for more detail.

The table begins as follows:

hdmx Header		
Type	Name	Description
USHORT	version	Table version number (0)
SHORT	numRecords	Number of device records.
LONG	sizeDeviceRecord	Size of a device record, long aligned.
DeviceRecord	records[numRecords]	Array of device records.

Each DeviceRecord for format 0 looks like this.

Device Record		
Type	Name	Description
BYTE	pixelSize	Pixel size for following widths (as ppem).
BYTE	maxWidth	Maximum width.
BYTE	widths[numGlyphs]	Array of widths (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.

5.6.4 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). OFF fonts containing CFF outlines are not supported by the 'kern' table and must use the 'GPOS' OFF Layout table.

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 OFF 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 (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 (0)
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 log2 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.

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 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.

5.6.5 LTSH – Linear threshold

The LTSH table relates to OFF fonts containing TrueType outlines. 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). The TrueType outline format already has two mechanisms to side step the speed issues: the 'hdmx' table, where precomputed advance widths may be saved for selected ppem sizes, and the 'vdmx' table, where precomputed vertical 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 nonlinear 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 Glyphs which do not have instructions on their sidebearings should have yPels = 1; i.e., always scales linearly.

5.6.6 PCLT – PCL 5 table

The 'PCLT' table is strongly discouraged for OFF fonts with TrueType outlines. 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.

Code	Vendor
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.

5.6.7 VDMX – Vertical device metrics

The VDMX table relates to OFF fonts with TrueType outlines. 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:

VDMX Header		
Type	Name	Description
USHORT	version	Version number (0 or 1).
USHORT	numRecs	Number of VDMX groups present
USHORT	numRatios	Number of aspect ratio groupings
Ratio	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)

Ratio Record		
Type	Name	Description
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 Ratios.yStartRatio Ratios.yEndRatio = 1;	= =	1; 1;
For 1:1 through 2:1 ratio	Ratios.xRatio Ratios.yStartRatio Ratios.yEndRatio = 2;	= =	2; 1;
For 1.33:1 ratio	Ratios.xRatio Ratios.yStartRatio Ratios.yEndRatio = 3;	= =	4; 3;
For *all* aspect ratios	Ratio.xRatio Ratio.yStartRatio Ratio.yEndRatio = 0;	= =	0; 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 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 bCharSet 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 semantics of bCharSet is different based on the version of the VDMX table. **It is recommended that VDMX version 1 be used.** The currently defined values for character set are:

Character Set Values - Version 0	
Value	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. ANSI_CHARSET)
Character Set Values - Version 1	
Value	Description
0	No subset; the VDMX group applies to all glyphs in the font. If adding new character sets to existing font, add this flag and the groups necessary to support it. This should only be used in conjunction with ANSI_CHARSET.
1	No subset; the VDMX group applies to all glyphs in the font. Used when creating a new font for Windows. No need to support SYMBOL_CHARSET.

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

VDMX Group		
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

vTable Record		
Type	Name	Description
USHORT	yPelHeight	yPelHeight to which values apply.
SHORT	yMax	Maximum value (in pels) for this yPelHeight.
SHORT	yMin	Minimum value (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.

5.6.8 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.

See the clause 6 "OFF CJK Font Guidelines" for more information about constructing CJK (Chinese, Japanese, and Korean) fonts.

The difference between version 1.0 and version 1.1 is the name and definition of:

- ascender becomes vertTypoAscender
- descender becomes vertTypoDescender
- lineGap becomes vertTypoLineGap

The vertical header table format follows: Vertical Header Table

Version 1.0 Type	Name	Description
Fixed	version	Version number of the vertical header table; 0x00010000 for version 1.0
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	minTop SideBearing	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	minBottom SideBearing	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	numOf LongVerMetrics	Number of advance heights in the vertical metrics table.

Version 1.1 Type	Name	Description
Fixed	version	Version number of the vertical header table; 0x00011000 for version 1.1 The representation of a non-zero fractional part, in Fixed numbers.
SHORT	vertTypoAscender	The vertical typographic ascender for this font. It is the distance in FUnits from the ideographic em-box center baseline for the vertical axis to the right of the ideographic em-box and is usually set to (head.unitsPerEm)/2. For example, a font with an em of 1000 fUnits will set this field to 500. See the baseline description of the OFF Tag Registry for a description of the ideographic em-box.
SHORT	vertTypoDescender	The vertical typographic descender for this font. It is the distance in FUnits from the ideographic em-box center baseline for the horizontal axis to the left of the ideographic em-box and is usually set to (head.unitsPerEm)/2. For example, a font with an em of 1000 fUnits will set this field to 500.
SHORT	vertTypoLineGap	The vertical typographic gap for this font. An application can determine the recommended line spacing for single spaced vertical text for an OFF font by the following expression: ideo embox width + vhea.vertTypoLineGap
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	minTop SideBearing	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	minBottom SideBearing	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	numOf LongVerMetrics	Number of advance heights in the vertical metrics table.

Vertical Header Table Example

Offset/ length	Value	Name	Comment
0/4	0x00011000	version	Version number of the vertical header table, in fixed-point format, is 1.1
4/2	1024	vertTypoAscender	Half the em-square height.
6/2	-1024	vertTypoDescender	Minus half the em-square height.
8/2	0	vertTypoLineGap	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.

5.6.9 vmtx – Vertical metric table

The vertical metrics table 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.

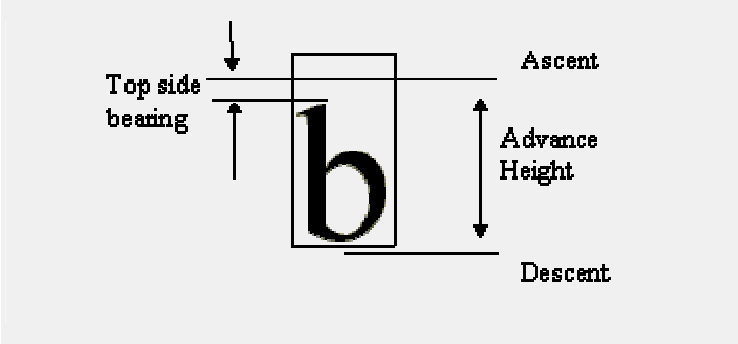

Figure 4 – Vertical Metrics

OFFvertical fonts require both a vertical header table ('vhea') 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).

See clause 6 "OFF CJK Font Guidelines" for more information about constructing CJK (Chinese, Japanese, and Korean) fonts.

Vertical Origin and Advance Height

The *y coordinate of a glyph's vertical origin* is specified as the sum of the glyph's top side bearing (recorded in the 'vmtx' table) and the top (i.e. maximum y) of the glyph's bounding box.

TrueType OFF fonts contain glyph bounding box information in the Glyph Data ('glyf') table. CFF OFF fonts do not contain glyph bounding box information, and so for these fonts the top of the glyph's bounding box must be calculated from the charstring data in the Compact Font Format ('CFF ') table.

OpenType 1.3 introduced the optional Vertical Origin ('VORG') table for CFF OFF fonts, which records the y coordinate of glyphs' vertical origins directly, thus obviating the need to calculate bounding boxes as an intermediate step. This improves accuracy and efficiency for CFF OFF clients.

The *x coordinate of a glyph's vertical origin* is not specified in the 'vmtx' table. Vertical writing implementions may make use of the baseline values in the Baseline ('BASE') table, if present, in order to align the glyphs in the x direction as appropriate to the desired vertical baseline.

The *advance height of a glyph* starts from the y coordinate of the glyph's vertical origin and advances downwards. Its endpoint is at the y coordinate of the vertical origin of the next glyph in the run, by default. Metric-adjustment OFF layout features such as Vertical Kerning ('vkrn') could modify the vertical advances in a manner similar to kerning in horizontal mode.

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. The top side bearing is measured relative to the top of the origin of glyphs, for vertical composition of ideographic glyphs.

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.

6 Advanced Open Font layout tables

6.1 Advanced Open Font layout extensions

6.1.1 Overview of advanced typographic layout extensions

The Advanced Typographic tables (OFF Layout tables) extend the functionality of fonts with either TrueType or CFF outlines. OFF Layout fonts contain additional information that extends the capabilities of the fonts to support high-quality international typography:

- OFF Layout fonts allow a rich mapping between characters and glyphs, which supports ligatures, positional forms, alternates, and other substitutions.
- OFF Layout fonts include information to support features for two-dimensional positioning and glyph attachment.
- OFF Layout fonts contain explicit script and language information, so a text-processing application can adjust its behavior accordingly.
- OFF Layout fonts have an open format that allows font developers to define their own typographical features.

This overview introduces the power and flexibility of the OFF Layout font model. The OFF Layout tables are described in more detail in clause 5 "Advanced Open Font Layout Tables".

OFF Layout Common Table Formats are documented in subclause 6.2 "OFF Layout Common Table Formats".

Registered OFF Layout Tags for scripts, languages, and baselines, are documented in subclause 6.4 "Layout Tag Registry".

OFF layout at a glance

OFF Layout addresses complex typographical issues that especially affect people using text-processing applications in multi-lingual and non-Latin environments.

OFF Layout fonts may contain alternative forms of characters and mechanisms for accessing them. For example, in Arabic, the shape of a character often varies with the character's position in a word. As shown here, the ha character will take any of four shapes, depending on whether it stands alone or whether it falls at the beginning, middle, or end of a word. OFF Layout helps a text-processing application determine which variant to substitute when composing text.

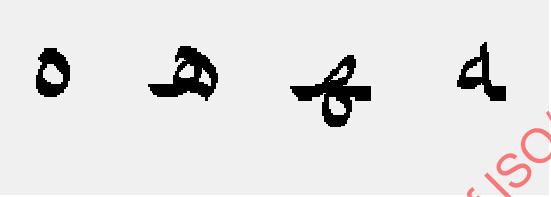

Figure 5 – Isolated, initial, medial, and final forms of the Arabic character ha.

Similarly, OFF Layout helps an application use the correct forms of characters when text is positioned vertically instead of horizontally, such as with Kanji. For example, Kanji uses alternative forms of parentheses when positioned vertically.

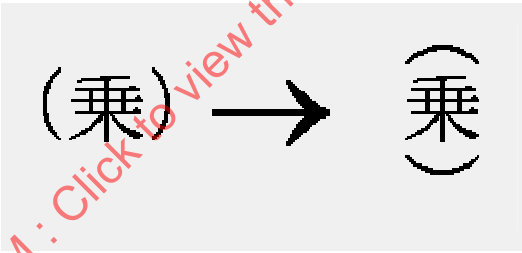

Figure 6 – Alternative forms of parentheses used when positioning Kanji vertically.

The OFF Layout font format also supports the composition and decomposition of ligatures. For example, English, French, and other languages based on Latin can substitute a single ligature, such as "fi", for its component glyphs - in this case, "f" and "i". Conversely, the individual "f" and "i" glyphs could replace the ligature, possibly to give a text-processing application more flexibility when spacing glyphs to fill a line of justified text.

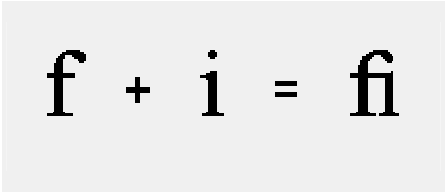

Figure 7 – Two Latin glyphs and their associated ligature.

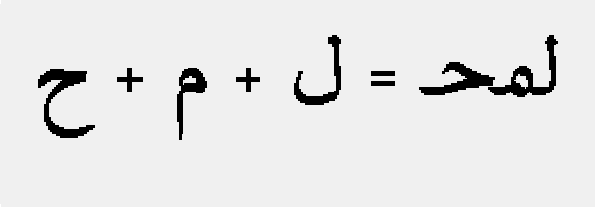

Figure 8 – Three Arabic glyphs and their associated ligature.

Glyph substitution is just one way OFF Layout extends font capabilities. Using precise X and Y coordinates for positioning glyphs, OFF Layout fonts also can identify points for attaching one glyph to another to create cursive text and glyphs that need diacritical or other special marks.

OFF Layout fonts also may contain baseline information that specifies how to position glyphs horizontally or vertically. Because baselines may vary from one script (set of characters) to another, this information is especially useful for aligning text that mixes glyphs from scripts for different languages.

Figure 9 – A line of text, baselines adjusted, mixing Latin and Arabic scripts.

6.1.2 TrueType versus OFF layout

A TrueType font is a collection of several tables that contain different types of data: glyph outlines, metrics, bitmaps, mapping information, and much more. OFF Layout fonts contain all this basic information, plus additional tables containing information for advanced typography.

Text-processing applications - referred to as "clients" of OFF Layout - can retrieve and parse the information in OFF Layout tables. So, for example, a text-processing client can choose the correct character shapes and space them properly.

As much as possible, the tables of OFF Layout define only the information that is specific to the font layout. The tables do not try to encode information that remains constant within the conventions of a particular language or the typography of a particular script. Such information that would be replicated across all fonts in a given language belongs in the text-processing application for that language, not in the fonts.

6.1.3 OFF layout terminology

The OFF Layout model is organized around glyphs, scripts, language systems, and features.

Characters versus glyphs

Users don't view or print characters: a user views or prints *glyphs*. A glyph is a representation of a character. The character "capital letter A" is represented by the glyph "A" in Times New Roman Bold and "A" in Arial Bold. A font is a collection of glyphs. To retrieve glyphs, the client uses information in the "cmap" table of the font, which maps the client's character codes to glyph indices in the table.

Glyphs can also represent combinations of characters and alternative forms of characters: glyphs and characters do not strictly correspond one-to-one. For example, a user might type two characters, which might be better represented with a single ligature glyph. Conversely, the same character might take different forms at the beginning, middle, or end of a word, so a font would need several different glyphs to represent a single character. OFF Layout fonts contain a table that provides a client with information about possible glyph substitutions.

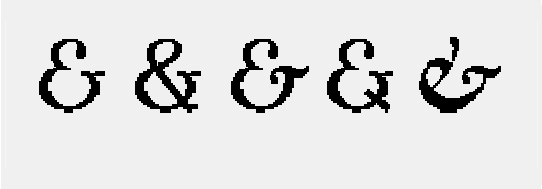

Figure 10 – Multiple glyphs for the ampersand character.

Scripts

A script is composed of a group of related characters, which may be used by one or more languages. Latin, Arabic, and Thai are examples of scripts. A font may use a single script, or it may use many scripts. Within an OFF Layout font, scripts are identified by unique 4-byte *tags*.

Figure 11 – Glyphs in the Latin, Kanji, and Arabic scripts.

Language systems

Scripts, in turn, may be divided into language systems. For example, the Latin script is used to write English, French, or German, but each language has its own special requirements for text processing. A font developer can choose to provide information that is tailored to the script, to the language system, or to both.

Language systems, unlike scripts, are not necessarily evident when a text-processing client examines the characters being used. To avoid ambiguity, the user or the operating system needs to identify the language system. Otherwise, the client will use the default language-system information provided with each script.

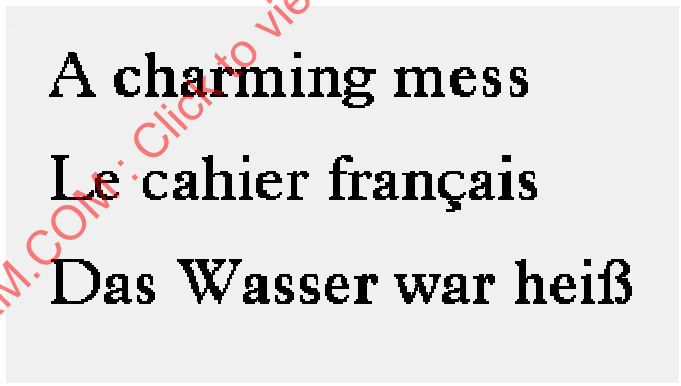

Figure 12 – Differences in the English, French, and German language system.

Features

Features define the basic functionality of the font. A font that contains tables to handle diacritical marks will have a "mark" feature. A font that supports substitution of vertical glyphs will have a "vert" feature.

The OFF Layout feature model provides great flexibility to font developers because features do not have to be predefined. Instead, font developers can work with application developers to determine useful features for fonts, add such features to OFF Layout fonts, and enable client applications to support such features.

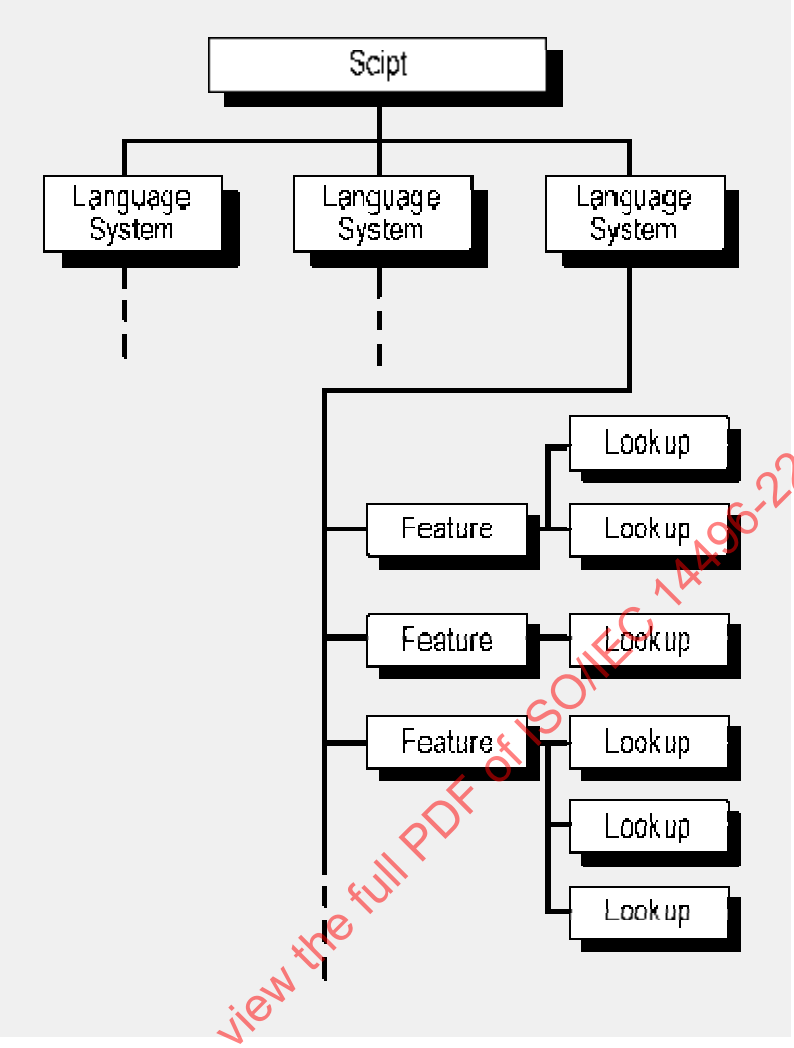

Figure 13 – The relationship of scripts, language systems, features, and lookups for substitution and positioning tables.

OFF Layout tables

OFF Layout comprises five new tables: GSUB, GPOS, BASE, JSTF, and GDEF. These tables and their formats are discussed in detail in the clauses that follow this overview.

GSUB: Contains information about glyph substitutions to handle single glyph substitution, one-to-many substitution (ligature decomposition), aesthetic alternatives, multiple glyph substitution (ligatures), and contextual glyph substitution.

GPOS: Contains information about X and Y positioning of glyphs to handle single glyph adjustment, adjustment of paired glyphs, cursive attachment, mark attachment, and contextual glyph positioning.

BASE: Contains information about baseline Offsets on a script-by-script basis.

JSTF: Contains justification information, including whitespace and Kashida adjustments.

GDEF: Contains information about all individual glyphs in the font: type (simple glyph, ligature, or combining mark), attachment points (if any), and ligature caret (if a ligature glyph).

Common Table Formats: Several common table formats are used by the OFF Layout tables.

6.1.4 Text processing with OFF layout

A text-processing client follows a standard process to convert the string of characters entered by a user into positioned glyphs. To produce text with OFF Layout fonts:

1. Using the cmap table in the font, the client converts the character codes into a string of glyph indices.
2. Using information in the GSUB table, the client modifies the resulting string, substituting positional or vertical glyphs, ligatures, or other alternatives as appropriate.
3. Using positioning information in the GPOS table and baseline Offset information in the BASE table, the client then positions the glyphs.
4. Using *design coordinates* the client determines device-independent line breaks. Design coordinates are high-resolution and device-independent.
5. Using information in the JSTF table, the client justifies the lines, if the user has specified such alignment.
6. The operating system rasterizes the line of glyphs and renders the glyphs in *device coordinates* that correspond to the resolution of the output device.

Throughout this process the text-processing client keeps track of the association between the character codes for the original text and the glyph indices of the final, rendered text. In addition, the client may save language and script information within the text stream to clearly associate character codes with typographical behavior.

Left-to-right and right-to-left text

When an OFF text layout engine applies the Unicode bidi algorithm and gets to the point where mirroring needs to be performed on runs with an even, i.e. **left-to-right** (LTR), resolved level, it does the following:

1. Glyph-level mirroring:

   Apply feature 'ltrm' to the entire LTR run to substitute mirrored forms.

2. LTR glyph alternates:

   Apply feature 'ltra' to the entire LTR run to finesse glyph selection.

For runs with an odd, i.e. **right-to-left** (RTL), resolved level, the engine does the following:

1. Character-level mirroring:

   For each character *i* in the RTL run:

   If it is mapped to character *j* by the OMPL and cmap(*j*) is non-zero:

   Use glyph cmap(*j*) at character *i*.

   Here OMPL refers to the OFF Mirroring Pairs List (see Annex D), and cmap(*j*) refers to the glyph at code point *j* in the Unicode cmap table.

   For example, suppose U+0028, LEFT PARENTHESIS, occurred in the run at resolved level 1. The glyph at that code point in the run will be replaced by cmap(U+0029), since {U+0028, U+0029} is a pair in the OMPL.

2. Glyph-level mirroring:

   The engine applies the ‘rtlm’ feature to the entire RTL run. The feature, if present, substitutes mirrored forms for characters *other* than those covered by the first elements of OMPL pairs (otherwise, it could cancel the effects of character-level mirroring).

   The data contents of the OMPL are identical to the Bidi Mirroring Glyph Property file of Unicode 5.1, and will never be revised. Thus, it will be up to the ‘rtlm’ feature to provide, if needed, mirrored forms for both (a) Unicode 5.1 code points with the “mirrored” property but no appropriate Unicode 5.1 character mirrors, as well as (b) all future “mirrored” property additions to Unicode, whether or not character mirrors exist for them.

With such a division of labor between the layout engine and the font, most fonts will not need to include an 'rtlm' feature, since the mirrored forms in their Unicode cmap subtable would be adequate.

3. RTL glyph alternates:

   The engine applies the 'rtla' feature to the entire RTL run. The feature, if present, substitutes variants appropriate for right-to-left text (other than mirrored forms).

In practice, the engine may apply features simultaneously; thus, it is up to the font vendor to ensure that the features' lookups are ordered to achieve the desired effect of the algorithms described above. The engine may optimize its implementation in various ways, e.g. by taking advantage of the fact that character- and glyph-level mirroring won't both apply on the same element in the run.

6.2 OFF layout common table formats

6.2.1 Overview

OFF Layout consists of five tables: the Glyph Substitution table (GSUB), the Glyph Positioning table (GPOS), the Baseline table (BASE), the Justification table (JSTF), and the Glyph Definition table (GDEF). These tables use some of the same data formats.

This clause explains the conventions used in all OFF Layout tables, and it describes the common table formats. Separate clauses provide complete details about the GSUB, GPOS, BASE, JSTF, and GDEF tables.

The OFF Layout tables provide typographic information for properly positioning and substituting glyphs, operations that are required for accurate typography in many language environments. OFF Layout data is organized by script, language system, typographic feature, and lookup.

Scripts are defined at the top level. A *script* is a collection of glyphs used to represent one or more languages in written form (see Figure 14). For instance, a single script-Latin is used to write English, French, German, and many other languages. In contrast, three scripts-Hiragana, Katakana, and Kanji-are used to write Japanese. With OFF Layout, multiple scripts may be supported by a single font.

Figure 14 – Glyphs in the Latin, Kanji, and Arabic scripts

A *language system* may modify the functions or appearance of glyphs in a script to represent a particular language. For example, the eszet ligature is used in the German language system, but not in French or English (see Figure 15). And the Arabic script contains different glyphs for writing the Farsi and Urdu languages. In OFF Layout, language systems are defined within scripts.

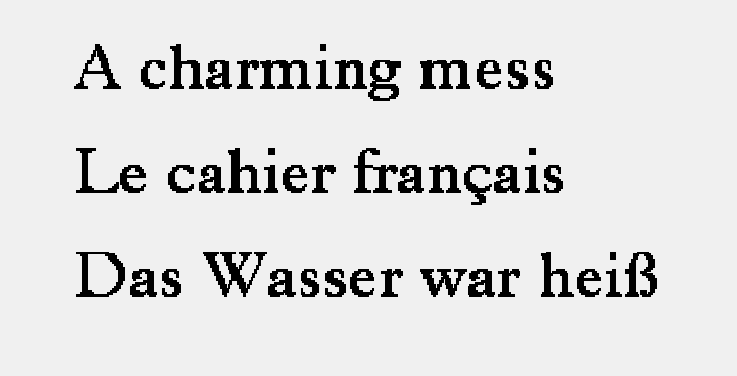

Figure15 – Differences in the English, French, and German language systems

A language system defines *features*, which are typographic rules for using glyphs to represent a language. Sample features are a "vert" feature that substitutes vertical glyphs in Japanese, a "liga" feature for using ligatures in place of separate glyphs, and a "mark" feature that positions diacritical marks with respect to base glyphs in Arabic (see Figure 16). In the absence of language-specific rules, default language system features apply to the entire script. For instance, a default language system feature for the Arabic script substitutes initial, medial, and final glyph forms based on a glyph's position in a word.

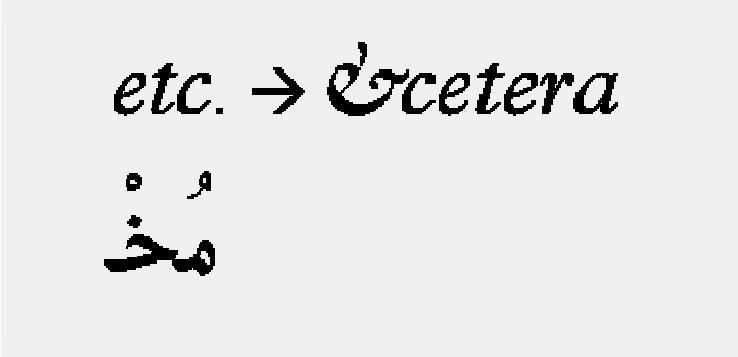

Figure 16 – A ligature glyph feature substitutes the <etc> ligature for individual glyphs, and a mark feature positions diacritical marks above an Arabic ligature glyph.

Features are implemented with lookup data that the text-processing client uses to substitute and position glyphs. *Lookups* describe the glyphs affected by an operation, the type of operation to be applied to these glyphs, and the resulting glyph output.

6.2.2 Table organization

Two OFF Layout tables, GSUB and GPOS, use the same data formats to describe the typographic functions of glyphs and the languages and scripts that they support: a ScriptList table, a FeatureList table, and a LookupList table. In GSUB, the tables define glyph substitution data. In GPOS, they define glyph positioning data. This clause describes these common table formats.

The ScriptList identifies the scripts in a font, each of which is represented by a Script table that contains script and language-system data. Language system tables reference features, which are defined in the FeatureList. Each feature table references the lookup data defined in the LookupList that describes how, when, and where to implement the feature.

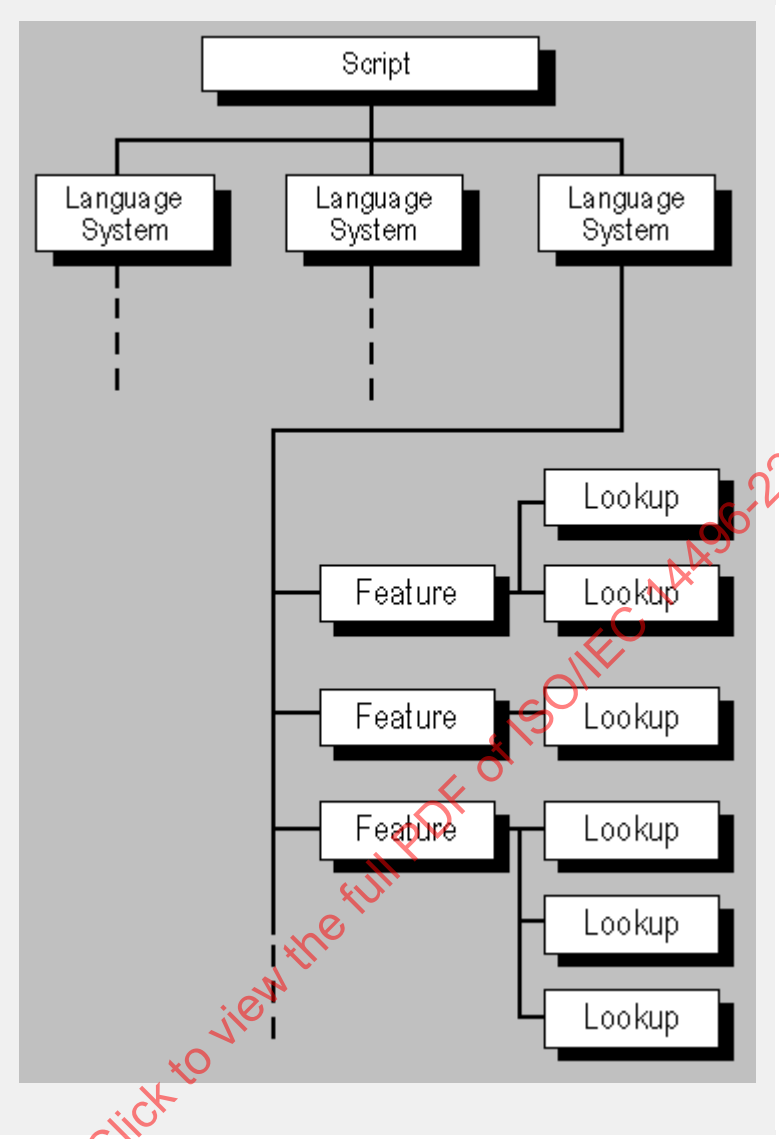

Figure 17 – The relationship of scripts, language systems, features, and lookups for substitution and positioning tables

NOTE The data in the BASE and JSTF tables also is organized by script and language system. However, the data formats differ from those in GSUB and GPOS, and they do not include a FeatureList or LookupList. The BASE and JSTF data formats are described in the BASE and JSTF clauses.

The information used to substitute and position glyphs is defined in Lookup subtables. Each subtable supplies one type of information, depending upon whether the lookup is part of a GSUB or GPOS table. For instance, a GSUB lookup might specify the glyphs to be substituted and the context in which a substitution occurs, and a GPOS lookup might specify glyph position adjustments for kerning. OFF Layout has seven types of GSUB lookups (described in the GSUB clause) and nine types of GPOS lookups (described in the GPOS clause).

Each subtable (except for an Extension LookupType subtable) includes a Coverage table that lists the "covered" glyphs that will result in a glyph substitution or positioning operation. The Coverage table formats are described in this clause.

Some substitution or positioning operations may apply to groups, or classes, of glyphs. GSUB and GPOS Lookup subtables use the Class Definition table to assign glyphs to classes. This clause includes a description of the Class Definition table formats.

Lookup subtables also may contain device tables, described in this clause, to adjust scaled contour glyph coordinates for particular output sizes and resolutions. This clauser also describes the data types used in OFF Layout. Sample tables and lists that illustrate the common data formats are supplied at the end of this clause.

6.2.3 Scripts and languages

Three tables and their associated records apply to scripts and languages: the Script List table (ScriptList) and its script record (ScriptRecord), the Script table and its language system record (LangSysRecord), and the Language System table (LangSys).

Script list table and Script record

OFF Layout fonts may contain one or more groups of glyphs used to render various scripts, which are enumerated in a ScriptList table. Both the GSUB and GPOS tables define Script List tables (ScriptList):

- The GSUB table uses the ScriptList table to access the glyph substitution features that apply to a script. For details, see the clause, The Glyph Substitution Table (GSUB).
- The GPOS table uses the ScriptList table to access the glyph positioning features that apply to a script. For details, see the clause, The Glyph Positioning Table (GPOS).

A ScriptList table consists of a count of the scripts represented by the glyphs in the font (ScriptCount) and an array of records (ScriptRecord), one for each script for which the font defines script-specific features (a script without script-specific features does not need a ScriptRecord).

If a Script table with the script tag 'DFLT' (default) is present in the ScriptList table, it must have a non-NULL DefaultLangSys and LangSysCount must be equal to 0. The 'DFLT' Script table should be used if there is not an explicit entry for the script being formatted.

The ScriptRecord array stores the records alphabetically by a ScriptTag that identifies the script. Each ScriptRecord consists of a ScriptTag and an Offset to a Script table.

Example 1 at the end of this clause shows a ScriptList table and ScriptRecords for a Japanese font that uses three scripts.

ScriptList table

Type	Name	Description
uint16	ScriptCount	Number of ScriptRecords
struct	ScriptRecord [ScriptCount]	Array of ScriptRecords -listed alphabetically by ScriptTag

ScriptRecord

Type	Name	Description
Tag	ScriptTag	4-byte ScriptTag identifier
Offset	Script	Offset to Script table-from beginning of ScriptList

Script table and Language System record

A Script table identifies each language system that defines how to use the glyphs in a script for a particular language. It also references a default language system that defines how to use the script's glyphs in the absence of language-specific knowledge.

A Script table begins with an Offset to the Default Language System table (DefaultLangSys), which defines the set of features that regulate the default behavior of the script. Next, Language System Count (LangSysCount) defines the number of language systems (excluding the DefaultLangSys) that use the script. In addition, an array of Language System Records (LangSysRecord) defines each language system (excluding the default) with an identification tag (LangSysTag) and an Offset to a Language System table (LangSys). The LangSysRecord array stores the records alphabetically by LangSysTag.

If no language-specific script behavior is defined, the LangSysCount is set to zero (0), and no LangSysRecords are allocated.

Script table

Type	Name	Description
Offset	DefaultLangSys	Offset to DefaultLangSys table-from beginning of Script table-may be NULL
uint16	LangSysCount	Number of LangSysRecords for this script-excluding the DefaultLangSys
struct	LangSysRecord [LangSysCount]	Array of LangSysRecords-listed alphabetically by LangSysTag

LangSysRecord

Type	Name	Description
Tag	LangSysTag	4-byte LangSysTag identifier
Offset	LangSys	Offset to LangSys table-from beginning of Script table

Language System table

The Language System table (LangSys) identifies language-system features used to render the glyphs in a script. (The LookupOrder Offset is reserved for future use.)

Optionally, a LangSys table may define a Required Feature Index (ReqFeatureIndex) to specify one feature as required within the context of a particular language system. For example, in the Cyrillic script, the Serbian language system always renders certain glyphs differently than the Russian language system.

Only one feature index value can be tagged as the ReqFeatureIndex. This is not a functional limitation, however, because the feature and lookup definitions in OFF Layout are structured so that one feature table can reference many glyph substitution and positioning lookups. When no required features are defined, then the ReqFeatureIndex is set to 0xFFFF.

All other features are optional. For each optional feature, a zero-based index value references a record (FeatureRecord) in the FeatureRecord array, which is stored in a Feature List table (FeatureList). The feature indices themselves (excluding the ReqFeatureIndex) are stored in arbitrary order in the FeatureIndex array. The FeatureCount specifies the total number of features listed in the FeatureIndex array.

Features are specified in full in the FeatureList table, FeatureRecord, and Feature table, which are described later in this clause. Example 2 at the end of this clause shows a Script table, LangSysRecord, and LangSys table used for contextual positioning in the Arabic script.

LangSys table

Type	Name	Description
Offset	LookupOrder	= NULL (reserved for an Offset to a reordering table)
uint16	ReqFeatureIndex	Index of a feature required for this language system- if no required features = 0xFFFF
uint16	FeatureCount	Number of FeatureIndex values for this language system-excludes the required feature
uint16	FeatureIndex[FeatureCount]	Array of indices into the FeatureList-in arbitrary order

6.2.4 Features and lookups

Features define the functionality of an OFF Layout font and they are named to convey meaning to the text-processing client. Consider a feature named "liga" to create ligatures. Because of its name, the client knows what the feature does and can decide whether to apply it. For more information, see the "Layout Tag Registry" subclause 6.4. Font developers can use these features, as well as create their own.

After choosing which features to use, the client assembles all lookups from the selected features. Multiple lookups may be needed to define the data required for different substitution and positioning actions, as well as to control the sequencing and effects of those actions.

To implement features, a client applies the lookups in the order the lookup definitions occur in the LookupList. As a result, within the GSUB or GPOS table, lookups from several different features may be interleaved during text processing. A lookup is finished when the client locates a target glyph or glyph context and performs a substitution (if specified) or a positioning (if specified).

NOTE The substitution (GSUB) lookups always occur before the positioning (GPOS) lookups. The lookup sequencing mechanism in TrueType relies on the font to determine the proper order of text-processing operations.

Lookup data is defined in one or more subtables that contain information about specific glyphs and the operations to be performed on them. Each type of lookup has one or more corresponding subtable definitions. The choice of a subtable format depends upon two factors: the precise content of the information being applied to an operation and the required storage efficiency. (For complete definitions of all lookup types and subtables, see the the GSUB and GPOS clauses of this document.)

OFF Layout features define information that is specific to the layout of the glyphs in a font. They do not encode information that is constant within the conventions of a particular language or the typography of a particular script. Information that would be replicated across all fonts in a given language belongs in the text-processing application for that language, not in the fonts.

Feature list table

The headers of the GSUB and GPOS tables contain Offsets to Feature List tables (FeatureList) that enumerate all the features in a font. Features in a particular FeatureList are not limited to any single script. A FeatureList contains the entire list of either the GSUB or GPOS features that are used to render the glyphs in all the scripts in the font.

The FeatureList table enumerates features in an array of records (FeatureRecord) and specifies the total number of features (FeatureCount). Every feature must have a FeatureRecord, which consists of a FeatureTag that identifies the feature and an Offset to a Feature table (described next). The FeatureRecord array is arranged alphabetically by FeatureTag names.

NOTE The values stored in the FeatureIndex array of a LangSys table are used to locate records in the FeatureRecord array of a FeatureList table.

FeatureList table

Type	Name	Description
uint16	FeatureCount	Number of FeatureRecords in this table
struct	FeatureRecord[FeatureCount]	Array of FeatureRecords-zero-based (first feature has FeatureIndex = 0)-listed alphabetically by FeatureTag

FeatureRecord

Type	Name	Description
Tag	FeatureTag	4-byte feature identification tag
Offset	Feature	Offset to Feature table-from beginning of FeatureList

Feature table

A Feature table defines a feature with one or more lookups. The client uses the lookups to substitute or position glyphs.

Feature tables defined within the GSUB table contain references to glyph substitution lookups, and feature tables defined within the GPOS table contain references to glyph positioning lookups. If a text-processing operation requires both glyph substitution and positioning, then both the GSUB and GPOS tables must each define a Feature table, and the tables must use the same FeatureTags.

A Feature table consists of an Offset to a Feature Parameters (FeatureParams) table (if one has been defined for this feature - see note in the following paragraph), a count of the lookups listed for the feature (LookupCount), and an arbitrarily ordered array of indices into a LookupList (LookupListIndex). The LookupList indices are references into an array of Offsets to Lookup tables.

The format of the Feature Parameters table is specific to a particular feature, and must be specified in the feature's entry in the Feature Tags subclause 6.4 of the OFF Layout Tag Registry. The length of the Feature Parameters table must be implicitly or explicitly specified in the Feature Parameters table itself. The FeatureParams field in the Feature Table records the Offset relative to the beginning of the Feature Table. If a Feature Parameters table is not needed, the FeatureParams field must be set to NULL.

To identify the features in a GSUB or GPOS table, a text-processing client reads the FeatureTag of each FeatureRecord referenced in a given LangSys table. Then the client selects the features it wants to implement and uses the LookupList to retrieve the Lookup indices of the chosen features. Next, the client arranges the indices in the LookupList order. Finally, the client applies the lookup data to substitute or position glyphs.

Example 3 at the end of this clause shows the FeatureList and Feature tables used to substitute ligatures in two languages.

Feature table

Type	Name	Description
Offset	FeatureParams	Offset to Feature Parameters table (if one has been defined for the feature), relative to the beginning of the Feature Table; = NULL if not required.
uint16	LookupCount	Number of LookupList indices for this feature
uint16	LookupListIndex [LookupCount]	Array of LookupList indices for this feature -zero-based (first lookup is LookupListIndex = 0)

Lookup list table

The headers of the GSUB and GPOS tables contain Offsets to Lookup List tables (LookupList) for glyph substitution (GSUB table) and glyph positioning (GPOS table). The LookupList table contains an array of Offsets to Lookup tables (Lookup). The font developer defines the Lookup sequence in the Lookup array to control the order in which a text-processing client applies lookup data to glyph substitution and positioning operations. LookupCount specifies the total number of Lookup table Offsets in the array.

Example 4 at the end of this clause shows three ligature lookups in the LookupList table.

LookupList table

Type	Name	Description
uint16	LookupCount	Number of lookups in this table
Offset	Lookup[LookupCount]	Array of Offsets to Lookup tables-from beginning of LookupList -zero based (first lookup is Lookup index = 0)

Lookup table

A Lookup table (Lookup) defines the specific conditions, type, and results of a substitution or positioning action that is used to implement a feature. For example, a substitution operation requires a list of target glyph indices to be replaced, a list of replacement glyph indices, and a description of the type of substitution action.

Each Lookup table may contain only one type of information (LookupType), determined by whether the lookup is part of a GSUB or GPOS table. GSUB supports eight LookupTypes, and GPOS supports nine LookupTypes (for details about LookupTypes, see the GSUB and GPOS clauses of the document).

Each LookupType is defined with one or more subtables, and each subtable definition provides a different representation format. The format is determined by the content of the information required for an operation and by required storage efficiency. When glyph information is best presented in more than one format, a single lookup may contain more than one subtable, as long as all the subtables are the same LookupType. For example, within a given lookup, a glyph index array format may best represent one set of target glyphs, whereas a glyph index range format may be better for another set of target glyphs.

During text processing, a client applies a lookup to each glyph in the string before moving to the next lookup. A lookup is finished for a glyph after the client makes the substitution/positioning operation. To move to the "next" glyph, the client will typically skip all the glyphs that participated in the lookup operation: glyphs that were substituted/positioned as well as any other glyphs that formed a context for the operation. However, in

the case of pair positioning operations (i.e., kerning), the "next" glyph in a sequence may be the second glyph of the positioned pair (see pair positioning lookup for details).

A Lookup table contains a LookupType, specified as an integer, that defines the type of information stored in the lookup. The LookupFlag specifies lookup qualifiers that assist a text-processing client in substituting or positioning glyphs. The SubTableCount specifies the total number of SubTables. The SubTable array specifies Offsets, measured from the beginning of the Lookup table, to each SubTable enumerated in the SubTable array.

Lookup table

Type	Name	Description
uint16	LookupType	Different enumerations for GSUB and GPOS
uint16	LookupFlag	Lookup qualifiers
uint16	SubTableCount	Number of SubTables for this lookup
Offset	SubTable [SubTableCount]	Array of Offsets to SubTables-from beginning of Lookup table
uint16	MarkFilteringSet	Index (base 0) into GDEF mark glyph sets structure. This field is present only if bit "UsesMarkFilteringSet" of *Lookup Flag bit enumeration* is set.

The LookupFlag uses two bytes of data:

- Each of the first four bits can be set in order to specify additional instructions for applying a lookup to a glyph string. The LookUpFlag bit enumeration table provides details about the use of these bits.
- The fifth bit indicates the presence of a MarkFilteringSet field in the *Lookup table*.
- The next three bits are reserved for future use.
- The high byte is set to specify the type of mark attachment.

LookupFlag bit enumeration

Type	Name	Description
0x0001	RightToLeft	This bit relates only to the correct processing of the cursive attachment lookup type (GPOS lookup type 3). When this bit is set, the last glyph in a given sequence to which the cursive attachment lookup is applied, will be positioned on the baseline. **Setting of this bit is not intended to be used by operating systems or applications to determine text direction.**
0x0002	IgnoreBaseGlyphs	If set, skips over base glyphs
0x0004	IgnoreLigatures	If set, skips over ligatures
0x0008	IgnoreMarks	If set, skips over all combining marks

0x0010	UseMarkFilteringSet	If set, indicates that the lookup table structure is followed by a MarkFilteringSet field. The layout engine skips over all mark glyphs not in the mark filtering set indicated.
0x00E0	Reserved	For future use (Set to zero)
0xFF00	MarkAttachmentType	If not zero, skips over all marks of attachment type different from specified.

IgnoreBaseGlyphs, IgnoreLigatures, or IgnoreMarks refer to base glyphs, ligatures and marks as defined in the Glyph Class Definition Table in the GDEF table. If any of these flags are set, a Glyph Class Definition Table must be present. If any of these bits is set, then lookups must ignore glyphs of the respective type; that is, the other glyphs must be processed just as though these glyphs were not present.

If MarkAttachmentType is non-zero, then mark attachment classes must be defined in the Mark Attachment Class Definition Table in the GDEF table. When processing glyph sequences, a lookup must ignore any mark glyphs that are not in the specified mark attachment class; only marks of the specified type are processed.

If any lookup has the UseMarkFilteringSet flag set, then the Lookup header must include the MarkFilteringSet field and a MarkGlyphSetsTable must be present in GDEF table. The lookup must ignore any mark glyphs that are not in the specified mark glyph set; only glyphs in the specified mark glyph set are processed.

If a mark filtering set is specified, this supersedes any mark attachment type indication in the lookup flag. If the IgnoreMarks bit is set, this supersedes any mark filtering set or mark attachment type indications.

For example, in Arabic text, a character string might have the pattern <base - mark - base>. That string could be converted into a ligature composed of two components, one for each base character, with the combining mark glyph over the first component. To produce this ligature, the font developer would set the IgnoreMarks bit of the ligature substitution lookup to tell the client to ignore the mark, substitute the ligature glyph first, and then position the mark glyph over the ligature in a subsequent GPOS lookup. Alternatively, a lookup which did not set the IgnoreMarks bit could be used to describe a three-component ligature glyph, composed of the first base glyph, the mark glyph, and the second base glyph.

For another example, a lookup which creates a ligature of a base glyph with a top mark could skip over all bottom marks by specifying the mark attachment type as a class that includes only top marks.

Coverage table

Each subtable (except an Extension LookupType subtable) in a lookup references a Coverage table (Coverage), which specifies all the glyphs affected by a substitution or positioning operation described in the subtable. The GSUB, GPOS, and GDEF tables rely on this notion of coverage. If a glyph does not appear in a Coverage table, the client can skip that subtable and move immediately to the next subtable.

A Coverage table identifies glyphs by glyph indices (GlyphIDs) either of two ways:

- As a list of individual glyph indices in the glyph set.
- As ranges of consecutive indices. The range format gives a number of start-glyph and end-glyph index pairs to denote the consecutive glyphs covered by the table.

In a Coverage table, a format code (CoverageFormat) specifies the format as an integer: 1 = lists, and 2 = ranges.

A Coverage table defines a unique index value (Coverage Index) for each covered glyph. This unique value specifies the position of the covered glyph in the Coverage table. The client uses the Coverage Index to look up values in the subtable for each glyph.

Coverage Format 1

Coverage Format 1 consists of a format code (CoverageFormat) and a count of covered glyphs (GlyphCount), followed by an array of glyph indices (GlyphArray). The glyph indices must be in numerical order for binary

searching of the list. When a glyph is found in the Coverage table, its position in the GlyphArray determines the Coverage Index that is returned-the first glyph has a Coverage Index = 0, and the last glyph has a Coverage Index = GlyphCount -1.

Example 5 at the end of this clause shows a Coverage table that uses Format 1 to list the GlyphIDs of all lowercase descender glyphs in a font.

Coverage Format1 table: Individual glyph indices

Type	Name	Description
uint16	CoverageFormat	Format identifier-format = 1
uint16	GlyphCount	Number of glyphs in the GlyphArray
GlyphID	GlyphArray[GlyphCount]	Array of GlyphIDs-in numerical order

Coverage Format 2

Format 2 consists of a format code (CoverageFormat) and a count of glyph index ranges (RangeCount), followed by an array of records (RangeRecords). Each RangeRecord consists of a start glyph index (Start), an end glyph index (End), and the Coverage Index associated with the range's Start glyph. Ranges must be in GlyphID order, and they must be distinct, with no overlapping.

The Coverage Indexes for the first range begin with zero (0), and the Start Coverage Indexes for each succeeding range are determined by adding the length of the preceding range (End GlyphID - Start GlyphID + 1) to the array Index. This allows for a quick calculation of the Coverage Index for any glyph in any range using the formula: Coverage Index (GlyphID) = StartCoverageIndex + GlyphID - Start GlyphID.

Example 6 at the end of this clause shows a Coverage table that uses Format 2 to identify a range of numeral glyphs in a font.

CoverageFormat2 table: Range of glyphs

Type	Name	Description
uint16	CoverageFormat	Format identifier-format = 2
uint16	RangeCount	Number of RangeRecords
struct	RangeRecord [RangeCount]	Array of glyph ranges-ordered by Start GlyphID

RangeRecord

Type	Name	Description
GlyphID	Start	First GlyphID in the range
GlyphID	End	Last GlyphID in the range
uint16	StartCoverageIndex	Coverage Index of first GlyphID in range

Class definition table

In OFF Layout, index values identify glyphs. For efficiency and ease of representation, a font developer can group glyph indices to form glyph classes. Class assignments vary in meaning from one lookup subtable to another. For example, in the GSUB and GPOS tables, classes are used to describe glyph contexts. GDEF tables also use the idea of glyph classes.

Consider a substitution action that replaces only the lowercase ascender glyphs in a glyph string. To more easily describe the appropriate context for the substitution, the font developer might divide the font's lowercase glyphs into two classes, one that contains the ascenders and one that contains the glyphs without ascenders.

A font developer can assign any glyph to any class, each identified with an integer called a class value. A Class Definition table (ClassDef) groups glyph indices by class, beginning with Class 1, then Class 2, and so on. All glyphs not assigned to a class fall into Class 0. Within a given class definition table, each glyph in the font belongs to exactly one class.

The ClassDef table can have either of two formats: one that assigns a range of consecutive glyph indices to different classes, or one that puts groups of consecutive glyph indices into the same class.

Class Definition Table Format 1

The first class definition format (ClassDefFormat1) specifies a range of consecutive glyph indices and a list of corresponding glyph class values. This table is useful for assigning each glyph to a different class because the glyph indices in each class are not grouped together.

A ClassDef Format 1 table begins with a format identifier (ClassFormat). The range of glyph indices (GlyphIDs) covered by the table is identified by two values: the GlyphID of the first glyph (StartGlyph), and the number of consecutive GlyphIDs (including the first one) that will be assigned class values (GlyphCount). The ClassValueArray lists the class value assigned to each GlyphID, starting with the class value for StartGlyph and following the same order as the GlyphIDs. Any glyph not included in the range of covered GlyphIDs automatically belongs to Class 0.

Example 7 at the end of this clause uses Format 1 to assign class values to the lowercase, x-height, ascender, and descender glyphs in a font.

ClassDefFormat1 table: Class array

Type	Name	Description
uint16	ClassFormat	Format identifier-format = 1
GlyphID	StartGlyph	First GlyphID of the ClassValueArray
uint16	GlyphCount	Size of the ClassValueArray
uint16	ClassValueArray[GlyphCount]	Array of Class Values-one per GlyphID

Class Definition Table Format 2

The second class definition format (ClassDefFormat2) defines multiple groups of glyph indices that belong to the same class. Each group consists of a discrete range of glyph indices in consecutive order (ranges cannot overlap).

The ClassDef Format 2 table contains a format identifier (ClassFormat), a count of ClassRangeRecords that define the groups and assign class values (ClassRangeCount), and an array of ClassRangeRecords ordered by the GlyphID of the first glyph in each record (ClassRangeRecord).

Each ClassRangeRecord consists of a Start glyph index, an End glyph index, and a Class value. All GlyphIDs in a range, from Start to End inclusive, constitute the class identified by the Class value. Any glyph not covered by a ClassRangeRecord is assumed to belong to Class 0.

Example 8 at the end of this clause uses Format 2 to assign class values to four types of glyphs in the Arabic script.

ClassDefFormat2 table: Class ranges

Type	Name	Description
uint16	ClassFormat	Format identifier-format = 2
uint16	ClassRangeCount	Number of ClassRangeRecords
struct	ClassRangeRecord [ClassRangeCount]	Array of ClassRangeRecords-ordered by Start GlyphID

ClassRangeRecord

Type	Name	Description
GlyphID	Start	First GlyphID in the range
GlyphID	End	Last GlyphID in the range
uint16	Class	Applied to all glyphs in the range

Device tables

Glyphs in a font are defined in design units specified by the font developer. Font scaling increases or decreases a glyph's size and rounds it to the nearest whole pixel. However, precise glyph positioning often requires adjustment of these scaled and rounded values. Hinting, applied to points in the glyph outline, is an effective solution to this problem, but it may require the font developer to redesign or re-hint glyphs.

Another solution-used by the GPOS, BASE, JSTF, and GDEF tables-is to use a Device table to specify correction values to adjust the scaled design units. A Device table applies the correction values to the range of sizes identified by StartSize and EndSize, which specify the smallest and largest pixel-per-em (ppem) sizes needing adjustment.

Because the adjustments often are very small (a pixel or two), the correction can be compressed into a 2-, 4-, or 8-bit representation per size. Two bits can represent a number in the range {-2, -1, 0, or 1}, four bits can represent a number in the range {-8 to 7}, and eight bits can represent a number in the range {-128 to 127}. The Device table identifies one of three data formats-signed 2-, 4,- or 8-bit values-for the adjustment values (DeltaFormat). A single Device table provides delta information for one coordinate at a range of sizes.

Type	Name	Description
1	2	Signed 2-bit value, 8 values per uint16
2	4	Signed 4-bit value, 4 values per uint16
3	8	Signed 8-bit value, 2 values per uint16

The 2-, 4-, or 8-bit signed values are packed into uint16's most significant bits first. For example, using a DeltaFormat of 2 (4-bit values), an array of values equal to {1, 2, 3, -1} would be represented by the DeltaValue 0x123F.

The DeltaValue array lists the number of pixels to adjust specified points on the glyph, or the entire glyph, at each ppem size in the targeted range. In the array, the first index position specifies the number of pixels to add or subtract from the coordinate at the smallest ppem size that needs correction, the second index position specifies the number of pixels to add or subtract from the coordinate at the next ppem size, and so on for each ppem size in the range.

Example 9 at the end of this clause uses a Device table to define the minimum extent value for a math script.

Device table

Type	Name	Description
uint16	StartSize	Smallest size to correct-in ppem
uint16	EndSize	Largest size to correct-in ppem
uint16	DeltaFormat	Format of DeltaValue array data: 1, 2, or 3
uint16	DeltaValue[ ]	Array of compressed data

6.2.5 Common table examples

The rest of this clause describes and illustrates examples of all the common table formats. All the examples reflect unique parameters, but the samples provide a useful reference for building tables specific to other situations.

The examples have three columns showing hex data, source, and comments.

Example 1: ScriptList table and ScriptRecords

Example 1 illustrates a ScriptList table and ScriptRecord definitions for a Japanese font with multiple scripts: Han Ideographic, Kana, and Latin. Each script has script-specific behavior.

Example 1

Hex Data	Source	Comment
	ScriptList TheScriptList	ScriptList table definintion
0003	3	ScriptCount ScriptRecord[0],in alphabetical order by ScriptTag
68616E69	"hani"	ScriptTag, Han Ideographic script
0014	HanIScriptTable	Offset to Script table ScriptRecord[1]
6B616E61	"kana"	ScriptTag, Hiragana and Katakana scripts
0018	KanaScriptTable	Offset to Script table ScriptRecord[2]
6C61746E	"latn"	ScriptTag, Latin script
001C	LatinScriptTable	Offset to Script table

Example 2: Script Table, LangSysRecord, and LangSys table

Example 2 illustrates the Script table, LangSysRecord, and LangSys table definitions for the Arabic script and the Urdu language system. The default LangSys table defines three default Arabic script features used to replace certain glyphs in words with their proper initial, medial, and final glyph forms. These contextual substitutions are invariant and occur in all language systems that use the Arabic script.

Many alternative glyphs in the Arabic script have language-specific uses. For instance, the Arabic, Farsi, and Urdu language systems use different glyphs for numerals. To maintain character-set compatibility, the Unicode Standard includes separate character codes for the Arabic and Farsi numeral glyphs. However, the standard uses the same character codes for Farsi and Urdu numerals, even though three of the Urdu glyphs (4, 6, and 7) differ from the Farsi glyphs. To access and display the proper glyphs for the Urdu numerals, users of the text-processing client must enter the character codes for the Farsi numerals. Then the text-processing client uses a required OFF Layout glyph substitution feature, defined in the Urdu LangSys table, to access the correct Urdu glyphs for the 4, 6, and 7 numerals.

NOTE The Urdu LangSys table repeats the default script features. This repetition is necessary because the Urdu language system also uses alternative glyphs in the initial, medial, and final glyph positions in words.

Example 2

Hex Data	Source	Comment
	Script ArabicScriptTable	Script table definition
000A	DefLangSys	Offset to DefaultLangSys table

0001	1	LangSysCount LangSysRecord[0], in alphabetical order by LangSysTag
55524420	"URD "	LangSysTag, Urdu language
0016	UrduLangSys	Offset to LangSys table for Urdu
	LangSys DefLangSys	default LangSys table definition
0000	NULL	LookupOrder, reserved, null
FFFF	0xFFFF	ReqFeatureIndex, no required features
0003	3	FeatureCount
0000	0	FeatureIndex[0], in arbitrary order "init" feature (initial glyph)
0001	1	FeatureIndex[1], "fina" feature (final glyph)
0002	2	FeatureIndex[2], for "medi" feature (medial glyph)
	LangSys UrduLangSys	LangSys table definition
0000	NULL	LookupOrder, reserved, null
0003	3	ReqFeatureIndex, numeral subsitution in Urdu
0003	3	FeatureCount
0000	0	FeatureIndex[0], in arbitrary order "init" feature (initial glyph)
0001	1	FeatureIndex[1], "fina" feature (final glyph)
0002	2	FeatureIndex[2], "medi" feature (medial glyph)

Example 3: FeatureList table and Feature table

Example 3 shows the FeatureList and Feature table definitions for ligatures in the Latin script. The FeatureList has three features, all optional and named "liga." One feature, also a default, implements ligatures in Latin if no language-specific feature specifies other ligatures. Two other features implement ligatures in the Turkish and German languages, respectively.

Three lookups define glyph substitutions for rendering ligatures in this font. The first lookup produces the "ffi" and "fi" ligatures; the second produces the "ffl," "fl," and "ff" ligatures; and the third produces the eszet ligature.

The ligatures that begin with an "f" are separated into two sets because Turkish has a dotless "i" glyph and so does not use "ffi" and "fi" ligatures. However, Turkish does use the "ffl," "fl," and "ff" ligatures, and the TurkishLigatures feature table lists this one lookup.

Only the German language system uses the eszet ligature, so the GermanLigatures feature table includes a lookup for rendering that ligature.

Because the Latin script can use both sets of ligatures, the DefaultLigatures feature table defines two LookupList indices: one for the "ffi" and "fi" ligatures, and one for the "ffl," "fl," and "ff" ligatures. If the text-processing client selects this feature, then the font applies both lookups.

NOTE The TurkishLigatures and DefaultLigatures feature tables both list a LookupListIndex of one (1) for the "ffl," "fl," and "ff" ligatures lookup. This is because language-specific lookups override all default language-system lookups, and a language-system feature table must explicitly list all lookups that apply to the language.

Example 3

Hex Data	Source	Comment
	FeatureList TheFeatureList	FeatureList table definition
0003	3	FeatureCount FeatureRecord[0]
6C696761	"liga"	FeatureTag
0014	TurkishLigatures	Offset to Feature table, FflFfFlLiga FeatureRecord[1]
6C696761	"liga"	FeatureTag
001A	DefaultLigatures	Offset to Feature table, FfiFiLiga, FflFfFlLiga FeatureRecord[2]
6C696761	"liga"	FeatureTag
0022	GermanLigatures	Offset to Feature table, EszetLiga
	Feature TurkishLigatures	Feature table definition
0000	NULL	FeatureParams, reserved, null
0001	1	LookupCount
0000	1	LookupListIndex[1], ffl, fl, ff ligature substitution Lookup

	Feature DefaultLigatures	Feature table definition
0000	NULL	FeatureParams - reserved, null
0002	2	LookupCount
0000	0	LookupListIndex[0], in arbitrary order, ffi, fi ligatures
0001	1	LookupListIndex[1], ffl, fl, ff ligature substitution Lookup
	Feature GermanLigatures	Feature table definition
0000	NULL	FeatureParams - reserved, null
0001	3	LookupCount
0000	0	LookupListIndex[0], in arbitrary order, ffi, fi ligatures
0001	1	LookupListIndex[1], ffl, fl, ff ligature substitution Lookup
0002	2	LookupListIndex[2], eszet ligature substitution Lookup

Example 4: LookupList table and Lookup table

A continuation of Example 3, Example 4 shows three ligature lookups in the LookupList table. The first generates the "ffi" and "fi" ligatures; the second produces the "ffl," "fl," and "ff" ligatures; and the third generates the eszet ligature. Each lookup table defines an Offset to a subtable that contains data for the ligature substitution.

Example 4

Hex Data	Source	Comment
	LookupList TheLookupList	LookupList table definition
0003	3	LookupCount
0008	FfiFiLookup	Offset to Lookup[0] table, in design order
0010	FflFlFfLookup	Offset to Lookup[1] table
0018	EszetLookup	Offset to Lookup[2] table

	Lookup FfiFiLookup	Lookup[0] table definition
0004	4	LookupType, ligature subst
000C	0x000C	LookupFlag, IgnoreLigatures, IgnoreMarks
0001	1	SubTableCount
0018	FfiFiSubtable	Offset to FfiFi ligature substitution subtable
	Lookup FflFlFfLookup	Lookup[1] table definition
0004	4	LookupType ligature subst
000C	0x000C	LookupFlag- IgnoreLigatures, IgnoreMarks
0001	1	SubTableCount
0028	FflFlFfSubtable	Offset to FflFlFf ligature substitution subtable
	Lookup EszetLookup	Lookup[2] table definition
0004	4	LookupType- ligature subst
000C	0x000C	LookupFlag- IgnoreLigatures, IgnoreMarks
0001	1	SubTableCount
0038	EszetSubtable	Offset to Eszet ligature substitution subtable

Example 5: CoverageFormat1 table (GlyphID list)

Example 5 illustrates a Coverage table that lists the GlyphIDs of all lowercase descender glyphs in a font. The table uses the list format instead of the range format because the GlyphIDs for the descender glyphs are not consecutively ordered.

Example 5

Hex Data	Source	Comment
	CoverageFormat1 DescenderCoverage	Coverage table definition
0001	1	CoverageFormat lists
0005	5	GlyphCount
0038	gGlyphID	GlyphArray[0], in GlyphID order
003B	jGlyphID	GlyphArray[1]
0041	pGlyphID	GlyphArray[2]
0042	qGlyphID	GlyphArray[3]
004A	yGlyphID	GlyphArray[4]

Example 6: CoverageFormat2 table (GlyphID ranges)

Example 6 shows a Coverage table that defines ten numeral glyphs (0 through 9). The table uses the range format instead of the list format because the GlyphIDs are ordered consecutively in the font. The StartCoverageIndex of zero (0) indicates that the first GlyphID, for the zero glyph, returns a Coverage Index of 0. The second GlyphID, for the numeral one (1) glyph, returns a Coverage Index of 1, and so on.

Example 6

Hex Data	Source	Comment
	CoverageFormat2 NumeralCoverage	Coverage table definition
0002	2	CoverageFormat, GlyphID ranges
0001	1	RangeCount RangeRecord[0]
004E	0glyphID	Start GlyphID
0057	9glyphID	End GlyphID
0000	0	StartCoverageIndex, first CoverageIndex = 0

Example 7: ClassDefFormat1 table (Class array)

The ClassDef table in Example 7 assigns class values to the lowercase glyphs in a font. The x-height glyphs are in Class 0, the ascender glyphs are in Class 1, and the descender glyphs are in Class 2. The array begins with the index for the lowercase "a" glyph.

Example 7

Hex Data	Source	Comment
	ClassDefFormat1 LowercaseClassDef	ClassDef table definition
0001	1	ClassFormat
0032	aGlyphID	StartGlyph
001A	26	GlyphCount
0000	0	aGlyph, Xheight Class 0
0001	1	bGlyph, Ascender Class 1
0000	0	cGlyph, Xheight Class 0
0001	1	dGlyph, Ascender Class 1
0000	0	eGlyph, Xheight Class 0
0001	1	fGlyph, Ascender Class 1
0002	2	gGlyph, Descender Class 2
0001	1	hGlyph, Ascender Class 1
0000	0	iGlyph, Ascender Class 1
0002	2	jGlyph, Descender Class 2
0001	1	kGlyph, Ascender Class 1
0001	1	lGlyph, Ascender Class 1
0000	0	mGlyph, Xheight Class 0
0000	0	nGlyph, Xheight Class 0
0000	0	oGlyph, Xheight Class 0

0002	2	pGlyph, Descender Class 2
0002	2	qGlyph, Descender Class 2
0000	0	rGlyph, Xheight Class 0
0000	0	sGlyph, Xheight Class 0
0001	1	tGlyph, Ascender Class 1
0000	0	uGlyph, Xheight Class 0
0000	0	vGlyph, Xheight Class 0
0000	0	wGlyph, Xheight Class 0
0000	0	xGlyph, Xheight Class 0
0002	2	yGlyph, Descender Class 2
0000	0	zGlyph, Xheight Class 0

Example 8: ClassDefFormat2 table (Class ranges)

In Example 8, the ClassDef table assigns class values to four types of glyphs in the Arabic script: medium-height base glyphs, high base glyphs, very high base glyphs, and default mark glyphs. The table lists only Class 1, Class 2, and Class 3; all glyphs not explicitly assigned a class fall into Class 0.

The table uses the range format because the GlyphIDs in each class are ordered consecutively in the font. In the ClassRange array, ClassRange definitions are ordered by the Start glyph index in each range. The indices of the high base glyphs, defined in ClassRange[0], are first in the font and have a class value of 2. ClassRange[1] defines all the very high base glyphs and assigns a class value of 3. ClassRange[2] contains all default mark glyphs; the class value is 1. Class 0 consists of all the medium-height base glyphs, which are not explicitly assigned a class value.

Example 8

Hex Data	Source	Comment
	ClassDefFormat2 GlyphHeightClassDef	Class table definition
0002	2	Class Format ranges
0003	3	ClassRangeCount ClassRange[0], ordered by StartGlyphID
0030	tahGlyphID	Start first GlyphID in the range

0031	dhahGlyphID	End Last GlyphID in the range
0002	2	Class, high base glyphs, ClassRange[1]
0040	cafGlyphID	Start, first GlyphID in the range
0041	gafGlyphID	End, Last GlyphID in the range
0003	3	Class, very high base glyphs, ClassRange[2]
00D2	fathatanDefaultGlyphID	Start, first GlyphID in the range
00D3	dammatanDefaultGlyphID	End, Last GlyphID in the range
0001	1	Class default marks

Example 9: Device table

Example 9 defines the minimum extent value for a math script, using a Device table to adjust the value according to the size of the output font. Here, the Device table defines single-pixel adjustments for font sizes from 11 ppem to 15 ppem. The DeltaFormat is 1, which signifies a packed array of signed 2-bit values, eight values per uint16.

Example 9

Hex Data	Source	Comment
	DeviceTableFormat1 MinCoordDeviceTable	Device Table definition
000B	11	StartSize, 11 ppem
000F	15	EndSize, 15 ppem
0001	1	DeltaFormat signed 2 bit value, 8 values per uint16
	1	increase 11ppem by 1 pixel
	1	increase 12ppem by 1 pixel
	1	increase 13ppem by 1 pixel
	1	increase 14ppem by 1 pixel
5540	1	increase 15ppem by 1 pixel

6.3 Advanced typographic tables

There are also several optional tables that support vertical layout as well as other advanced typographic functions:

Advanced Typographic Tables

Tag	Name
BASE	Baseline data
GDEF	Glyph definition data
GPOS	Glyph positioning data
GSUB	Glyph substitution data
JSTF	Justification data

6.3.1 BASE Baseline table

The Baseline table (BASE) provides information used to align glyphs of different scripts and sizes in a line of text, whether the glyphs are in the same font or in different fonts. To improve text layout, the Baseline table also provides minimum (min) and maximum (max) glyph extent values for each script, language system, or feature in a font.

Overview

Lines of text composed with glyphs of different scripts and point sizes need adjustment to correct interline spacing and alignment. For example, glyphs designed to be the same point size often differ in height and depth from one font to another (see Figure 18). This variation can produce interline spacing that looks too large or too small, and diacritical marks, math symbols, subscripts, and superscripts may be clipped.

Figure 18 – Incorrect alignment of glyphs from Latin and Kanji (Latin dominant)

In addition, different baselines can cause text lines to waver visually as glyphs from different scripts are placed next to one another. For example, ideographic scripts position all glyphs on a low baseline. With Latin scripts, however, the baseline is higher, and some glyphs descend below it. Finally, several Indic scripts use a high "hanging baseline" to align the tops of the glyphs.

To solve these composition problems, the BASE table recommends baseline positions and min/max extents for each script (see Figure 19). Script min/max extents can be modified for particular language systems or features.

Figure 19 – Proper alignment of glyphs from Latin and Kanji (Latin dominant)

Baseline values

The BASE table uses a model that assumes one script at one size is the "dominant run" during text processing-that is, all other baselines are defined in relation to this the dominant run.

For example, Latin glyphs and the ideographic Kanji glyphs have different baselines. If a Latin script of a particular size is specified as the dominant run, then all Latin glyphs of all sizes will be aligned on the roman baseline, and all Kanji glyphs will be aligned on the lower ideographic baseline defined for use with Latin text. As a result, all glyphs will look aligned within each line of text.

The BASE table supplies recommended baseline positions; a client can specify others. For instance, the client may want to assign baseline positions different from those in the font.

Figure 20 – Comparing Latin and Kanji baselines, with characters aligned according to the dominant run

Min/Max Extent values

The BASE table gives clients the option of using script, language system, or feature-specific extent values to improve composition (see Figure 20). For example, suppose a font contains glyphs in Latin and Arabic scripts, and the min/max extents defined for the Arabic script are larger than the Latin extents. The font also supports Urdu, a language system that includes specific variants of the Arabic glyphs, and some Urdu variants require larger min/max extents than the default Arabic extents. To accommodate the Urdu glyphs, the BASE table can define language-specific min/max extent values that will override the default Arabic extents-but only when rendering Urdu glyphs.

The BASE table also can define feature-specific min/max values that apply only when a particular feature is enabled. Suppose that the font described earlier also supports the Farsi language system, which has one feature that requires a minor alteration of the Arabic script extents to display properly. The BASE table can specify these extent values and apply them only when that feature is enabled in the Farsi language.

6.3.1.1 BASE table organization

The BASE table begins with Offsets to Axis tables that describe layout data for the horizontal and vertical layout directions of text. A font can provide layout data for both text directions or for only one text direction:

- The Horizontal Axis table (HorizAxis) defines information used to lay out text horizontally. All baseline and min/max values refer to the Y direction.
- The Vertical Axis table (VertAxis) defines information used to lay out text vertically. All baseline and min/max values refer to the X direction.

NOTE The same baseline tags can be used for both horizontal and vertical axes. For example, the 'romn' tag description used for the vertical axis would indicate the baseline of rotated Latin text.

Figure 21 shows how the BASE table is organized.

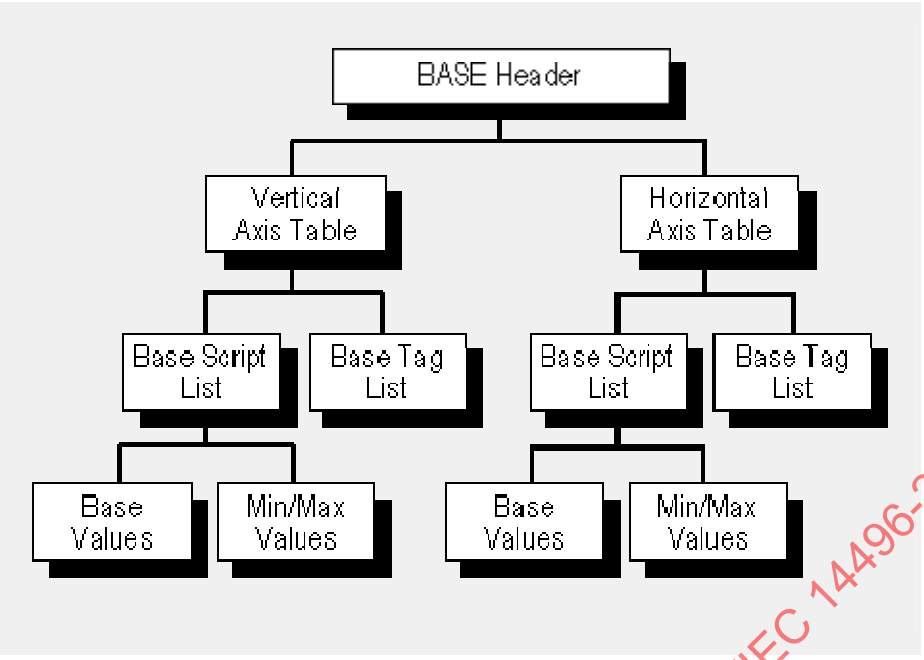

Figure 21 – High-level organization of BASE table

Text direction

The HorizAxis and VertAxis tables organize layout information by script in BaseScriptList tables. A BaseScriptList enumerates all scripts in the font that are written in a particular direction (horizontal or vertical).

For example, consider a Japanese font that contains Kanji, Kana, and Latin scripts. Because all three scripts are rendered horizontally, all three are defined in the BaseScriptList of the HorizAxis table. Kanji and Kana also are rendered vertically, so those two scripts are defined in the BaseScriptList of the VertAxis table, too.

Baseline data

Each Axis table also references a BaseTagList, which identifies all the baselines for all scripts written in the same direction (horizontal or vertical). The BaseTagList may also include baseline tags for scripts supported in other fonts.

Each script in a BaseScriptList is represented by a BaseScriptRecord. This record references a BaseScript table, which contains layout data for the script. In turn, the BaseScript table references a BaseValues table, which contains baseline information and several MinMax tables that define min/max extent values.

The BaseValues table specifies the coordinate values for all baselines in the BaseTagList. In addition, it identifies one of these baselines as the default baseline for the script. As glyphs in a script are scaled, they grow or shrink from the script's default baseline position. Each baseline can have unique coordinates. This contrasts with TrueType 1.0, which implies a single, fixed baseline for all scripts in a font. With the OFF Layout tables, each script can be aligned independently, although more than one script may use the same baseline values.

Baseline coordinates for scripts in the same font must be specified in relation to each other for correct alignment of the glyphs. Consider the font, discussed earlier, containing both Latin and Kanji glyphs. If the BaseTagList of the HorizAxis table specifies two baselines, the roman and the ideographic, then the layout data for both the Latin and Kanji scripts will specify coordinate positions for both baselines:

- The BaseValues table for the Latin script will give coordinates for both baselines and specify the roman baseline as the default.
- The BaseValues table for the Kanji script will give coordinates for both baselines and specify the ideographic baseline as the default.

Min/Max extents

The BaseScript table can define minimum and maximum extent values for each script, language system, or feature. (These values are distinct from the min/max extent values recorded for the font as a whole in the head, hhea, vhea, and OS/2 tables.) These extent values appear in three tables:

- The DefaultMinMax table defines the default min/max extents for the script.
- A MinMax table, referenced through a BaseLangSysRecord, specifies min/max extents to accommodate the glyphs in a specific language system.
- A FeatMinMaxRecord, referenced from the MinMax table, provides min/max extent values to support feature-specific glyph actions.

NOTE Language-system or feature-specific extent values may be essential to define some fonts. However, the default min/max extent values specified for each script should usually be enough to support high-quality text layout.

The actual baseline and min/max extent values used by the BASE table reside in BaseCoord tables. Three formats are defined for BaseCoord table data. All formats define single X or Y coordinate values in design units, but two formats support fine adjustments to these values based on a contour point or a Device table.

The rest of this clause describes all the tables defined within the BASE table. Sample tables and lists that illustrate typical data for a font are supplied at the end of the clause.

6.3.1.2 BASE table structure

BASE header

The BASE table begins with a header that consists of a version number for the table (Version), initially set to 1.0 (0x00010000), and Offsets to horizontal and vertical Axis tables (HorizAxis and VertAxis).

Each Axis table stores all baseline information and min/max extents for one layout direction. The HorizAxis table contains Y values for horizontal text layout; the VertAxis table contains X values for vertical text layout.

A font may supply information for both layout directions. If a font has values for only one text direction, the Axis table Offset value for the other direction will be set to NULL.

Example 1 at the end of this clause shows a sample BASE Header.

BASE Header

Type	Name	Description
fixed32	Version	Version of the BASE table-initially 0x00010000
Offset	HorizAxis	Offset to horizontal Axis table-from beginning of BASE table-may be NULL
Offset	VertAxis	Offset to vertical Axis table-from beginning of BASE table-may be NULL

Axis tables: HorizAxis and VertAxis

An Axis table is used to render scripts either horizontally or vertically. It consists of Offsets, measured from the beginning of the Axis table, to a BaseTagList and a BaseScriptList:

- The BaseScriptList enumerates all scripts rendered in the text layout direction.
- The BaseTagList enumerates all baselines used to render the scripts in the text layout direction. If no baseline data is available for a text direction, the Offset to the corresponding BaseTagList may be set to NULL.

Example 1 at the end of this clause shows an example of an Axis table.

Axis Table

Type	Name	Description
Offset	BaseTagList	Offset to BaseTagList table-from beginning of Axis table-may be NULL
Offset	BaseScriptList	Offset to BaseScriptList table-from beginning of Axis table

BaseTagList table

The BaseTagList table identifies the baselines for all scripts in the font that are rendered in the same text direction. Each baseline is identified with a 4-byte baseline tag. The Baseline Tags of the OFF Tag Registry lists currently registered baseline tags. The BaseTagList can define any number of baselines, and it may include baseline tags for scripts supported in other fonts.

Each script in the BaseScriptList table must designate one of these BaseTagList baselines as its default, which the OFF Layout Services use to align all glyphs in the script. Even though the BaseScriptList and the BaseTagList are defined independently of one another, the BaseTagList typically includes a tag for each different default baseline needed to render the scripts in the layout direction. If some scripts use the same default baseline, the BaseTagList needs to list the common baseline tag only once.

The BaseTagList table consists of an array of baseline identification tags (BaselineTag), listed alphabetically, and a count of the total number of baseline Tags in the array (BaseTagCount).

Example 1 at the end of this clause shows a sample BaseTagList table.

BaseTagList table

Type	Name	Description
uint16	BaseTagCount	Number of baseline identification tags in this text direction-may be zero (0)
Tag	BaselineTag[BaseTagCount]	Array of 4-byte baseline identification tags-must be in alphabetical order

BaseScriptList table

The BaseScriptList table identifies all scripts in the font that are rendered in the same layout direction. If a script is not listed here, then the text-processing client will render the script using the layout information specified for the entire font.

For each script listed in the BaseScriptList table, a BaseScriptRecord must be defined that identifies the script and references its layout data. BaseScriptRecords are stored in the BaseScriptRecord array, ordered alphabetically by the BaseScriptTag in each record. The BaseScriptCount specifies the total number of BaseScriptRecords in the array.

Example 1 at the end of this clause shows a sample BaseScriptList table.

BaseScriptList table

Type	Name	Description
uint16	BaseScriptCount	Number of BaseScriptRecords defined
struct	BaseScriptRecord[BaseScriptCount]	Array of BaseScriptRecords-in alphabetical order by BaseScriptTag

BaseScriptRecord

A BaseScriptRecord contains a script identification tag (BaseScriptTag), which must be identical to the ScriptTag used to define the script in the ScriptList of a GSUB or GPOS table. Each record also must include an Offset to a BaseScript table that defines the baseline and min/max extent data for the script.

Example 1 at the end of this clause shows a sample BaseScriptRecord.

BaseScriptRecord

Type	Name	Description
Tag	BaseScriptTag	4-byte script identification tag
Offset	BaseScript	Offset to BaseScript table-from beginning of BaseScriptList

BaseScript table

A BaseScript table organizes and specifies the baseline data and min/max extent data for one script. Within a BaseScript table, the BaseValues table contains baseline information, and one or more MinMax tables contain min/max extent data.

The BaseValues table identifies the default baseline for the script and lists coordinate positions for each baseline named in the corresponding BaseTagList. Each script can assign a different position to each baseline, so each script can be aligned independently in relation to any other script. (For more details, see the BaseValues table description later in this clause.)

The DefaultMinMax table defines the default min/max extent values for the script. (For details, see the MinMax table description below.) If a language system or feature defined in the font has no effect on the script's default min/max extents, the OFF Layout Services will use the default script values.

Sometimes language-specific overrides for min/max extents are needed to properly render the glyphs in a specific language system. For example, a glyph substitution required in a language system may result in a glyph whose extents exceed the script's default min/max extents. Each language system that specifies min/max extent values must define a BaseLangSysRecord. The record should identify the language system (BaseLangSysTag) and contain an Offset to a MinMax table of language-specific extent coordinates.

Feature-specific overrides for min/max extents also may be needed to accommodate the effects of glyph actions used to implement a specific feature. For example, superscript or subscript features may require changes to the default script or language system extents. Feature-specific extent values not limited to a specific language system may be specified in the DefaultMinMax table. However, extent values used for a specific language system require a BaseLangSysRecord and a MinMax table. In addition to specifying coordinate data, the MinMax table must contain Offsets to FeatMinMaxRecords that define the feature-specific min/max data.

A BaseScript table has four components:

- An Offset to a BaseValues table (BaseValues). If no baseline data is defined for the script or the corresponding BaseTagList is set to NULL, the Offset to the BaseValues table may be set to NULL.
- An Offset to the DefaultMinMax table. If no default min/max extent data is defined for the script, this Offset may be set to NULL.
- An array of BaseLangSysRecords (BaseLangSysRecord). The individual records stored in the BaseLangSysRecord array are listed alphabetically by BaseLangSysTag.
- A count of the BaseLangSysRecords included (BaseLangSysCount). If no language system or language-specific feature min/max values are defined, the BaseLangSysCount may be set to zero (0).

Example 2 at the end of this clause shows a sample BaseScript table.

BaseScript Table

Type	Name	Description
Offset	BaseValues	Offset to BaseValues table-from beginning of BaseScript table-may be NULL
Offset	DefaultMinMax	Offset to MinMax table- from beginning of BaseScript table-may be NULL
uint16	BaseLangSysCount	Number of BaseLangSysRecords defined-may be zero (0)
struct	BaseLangSysRecord [BaseLangSysCount]	Array of BaseLangSysRecords-in alphabetical order by BaseLangSysTag

BaseLangSysRecord

A BaseLangSysRecord defines min/max extents for a language system or a language-specific feature. Each record contains an identification tag for the language system (BaseLangSysTag) and an Offset to a MinMax table (MinMax) that defines extent coordinate values for the language system and references feature-specific extent data.

Example 2 at the end of this clause shows a BaseLangSysRecord.

BaseLangSysRecord

Type	Name	Description
Tag	BaseLangSysTag	4-byte language system identification tag
Offset	MinMax	Offset to MinMax table-from beginning of BaseScript table

BaseValues table

A BaseValues table lists the coordinate positions of all baselines named in the BaselineTag array of the corresponding BaseTagList and identifies a default baseline for a script.

NOTE When the Offset to the corresponding BaseTagList is NULL, a BaseValues table is not needed. However, if the Offset is not NULL, then each script must specify coordinate positions for all baselines named in the BaseTagList.

The default baseline, one per script, is the baseline used to lay out and align the glyphs in the script. The DefaultIndex in the BaseValues table identifies the default baseline with a value that equals the array index position of the corresponding tag in the BaselineTag array.

For example, the Han and Latin scripts use different baselines to align text. If a font supports both of these scripts, the BaselineTag array in the BaseTagList of the HorizAxis table will contain two tags, listed alphabetically: "ideo" in BaselineTag[0] for the Han ideographic baseline, and "romn" in BaselineTag[1] for the Latin baseline. The BaseValues table for the Latin script will specify the roman baseline as the default, so the DefaultIndex in the BaseValues table for Latin will be "1" to indicate the roman baseline tag. In the BaseValues table for the Han script, the DefaultIndex will be "0" to indicate the ideographic baseline tag.

Two or more scripts may share a default baseline. For instance, if the font described above also supports the Cyrillic script, the BaselineTag array does not need a baseline tag for Cyrillic because Cyrillic and Latin share the same baseline. The DefaultIndex defined in the BaseValues table for the Cyrillic script will specify "1" to indicate the roman baseline tag, listed in the second position in the BaselineTag array.

In addition to identifying the DefaultIndex, the BaseValues table contains an Offset to an array of BaseCoord tables (BaseCoord) that list the coordinate positions for all baselines, including the default baseline, named in the associated BaselineTag array. One BaseCoord table is defined for each baseline. The BaseCoordCount defines the total number of BaseCoord tables, which must equal the number of baseline tags listed in BaseTagCount in the BaseTagList.

Each baseline coordinate is defined as a single X or Y value in design units measured from the zero position on the relevant X or Y axis. For example, a BaseCoord table defined in the HorizAxis table will contain a Y value because horizontal baselines are positioned vertically. BaseCoord values may be negative. Each script may assign a different coordinate to each baseline.

Offsets to each BaseCoord table are stored in a BaseCoord array within the BaseValues table. The order of the stored Offsets corresponds to the order of the tags listed in the BaselineTag array of the BaseTagList. In other words, the first position in the BaseCoord array will define the Offset to the BaseCoord table for the first baseline named in the BaselineTag array, the second position will define the Offset to the BaseCoord table for the second baseline named in the BaselineTag array, and so on.

Example 3 at the end of the clause has two parts, one that shows a BaseValues table and one that shows a chart with different baseline positions defined for several scripts.

BaseValues table

Type	Name	Description
uint16	DefaultIndex	Index number of default baseline for this script-equals index position of baseline tag in BaselineArray of the BaseTagList
uint16	BaseCoordCount	Number of BaseCoord tables defined-should equal BaseTagCount in the BaseTagList
Offset	BaseCoord[BaseCoordCount]	Array of Offsets to BaseCoord-from beginning of BaseValues table-order matches BaselineTag array in the BaseTagList

The MinMax table and FeatMinMaxRecord

The MinMax table specifies extents for scripts and language systems. It also contains an array of FeatMinMaxRecords used to define feature-specific extents.

Both the MinMax table and the FeatMinMaxRecord define Offsets to two BaseCoord tables: one that defines the mimimum extent value (MinCoord), and one that defines the maximum extent value (MaxCoord). Each extent value is a single X or Y value, depending upon the text direction, and is specified in design units. Coordinate values may be negative.

Different tables define the min/max extent values for scripts, language systems, and features:

- Min/max extent values for a script are defined in the DefaultMinMax table, referenced in a BaseScript table.
- Within the DefaultMinMax table, FeatMinMaxRecords can specify extent values for features that apply to the entire script.
- Min/max extent values for a language system are defined in the MinMax table, referenced in a BaseLangSysRecord.
- FeatMinMaxRecords can be defined within the MinMax table to specify extent values for features applied within a language system.

In a FeatMinMaxRecord, the MinCoord and MaxCoord tables specify the minimum and maximum coordinate values for the feature, and a FeatureTableTag defines a 4-byte feature identification tag. The FeatureTableTag must match the tag used to identify the feature in the FeatureList of the GSUB or GPOS table.

Each feature that exceeds the default min/max values requires a FeatMinMaxRecord. All FeatMinMaxRecords are listed alphabetically by FeatureTableTag in an array (FeatMinMaxRecord) within the MinMax table. FeatMinMaxCount defines the total number of FeatMinMaxRecords.

Text-processing clients should use the following procedure to access the script, language system, and feature-specific extent data:

1. Determine script extents in relation to the text content.
2. Select language-specific extent values with respect to the language system in use.
3. Have the application or user choose feature-specific extent values.
4. If no extent values are defined for a language system or for language-specific features, use the default min/max extent values for the script.

Example 4 at the end of this clause has two parts. One shows MinMax tables and a FeatMinMaxRecord for different script, language system, and feature extents. The second part shows how to define these tables when a language system needs feature-specific extent values for an obscure feature, but otherwise the language system and script extent values match.

MinMax table

Type	Name	Description
Offset	MinCoord	Offset to BaseCoord table-defines minimum extent value-from the beginning of MinMax table-may be NULL
Offset	MaxCoord	Offset to BaseCoord table-defines maximum extent value-from the beginning of MinMax table-may be NULL
uint16	FeatMinMaxCount	Number of FeatMinMaxRecords-may be zero (0)
struct	FeatMinMaxRecord [FeatMinMaxCount]	Array of FeatMinMaxRecords-in alphabetical order, by FeatureTableTag

FeatMinMaxRecord

Type	Name	Description
Tag	FeatureTableTag	4-byte feature identification tag-must match FeatureTag in FeatureList
Offset	MinCoord	Offset to BaseCoord table-defines minimum extent value-from beginning of MinMax table-may be NULL
Offset	MaxCoord	Offset to BaseCoord table-defines maximum extent value-from beginning of MinMax table-may be NULL

BaseCoord tables

Within the BASE table, a BaseCoord table defines baseline and min/max extent values. Each BaseCoord table defines one X or Y value:

- If defined within the HorizAxis table, then the BaseCoord table contains a Y value.
- If defined within the VertAxis table, then the BaseCoord table contains an X value.

All values are defined in design units, which typically are scaled and rounded to the nearest integer when scaling the glyphs. Values may be negative.

Three formats available for BaseCoord table data define single X or Y coordinate values in design units. Two of the formats also support fine adjustments to the X or Y values based on a contour point or a Device table.

BaseCoord Format 1

The first BaseCoord format (BaseCoordFormat1) consists of a format identifier, followed by a single design unit coordinate that specifies the BaseCoord value. This format has the benefits of small size and simplicity, but the BaseCoord value cannot be hinted for fine adjustments at different sizes or device resolutions.

Example 5 at the end of the clause shows a sample of a BaseCoordFormat1 table.

BaseCoordFormat1 table: Design units only

Type	Name	Description
uint16	BaseCoordFormat	Format identifier-format = 1
int16	Coordinate	X or Y value, in design units

BaseCoord Format 2

The second BaseCoord format (BaseCoordFormat2) specifies the BaseCoord value in design units, but also supplies a glyph index and a contour point for reference. During font hinting, the contour point on the glyph outline may move. The point's final position after hinting provides the final value for rendering a given font size.

NOTE Glyph positioning operations defined in the GPOS table do not affect the point's final position.

Example 6 shows a sample of a BaseCoordFormat2 table.

BaseCoordFormat2 table: Design units plus contour point

Type	Name	Description
uint16	BaseCoordFormat	Format identifier-format = 2
int16	Coordinate	X or Y value, in design units
GlyphID	ReferenceGlyph	GlyphID of control glyph
uint16	BaseCoordPoint	Index of contour point on the ReferenceGlyph

BaseCoord Format 3

The third BaseCoord format (BaseCoordFormat3) also specifies the BaseCoord value in design units, but it uses a Device table rather than a contour point to adjust the value. This format offers the advantage of fine-tuning the BaseCoord value for any font size and device resolution. (For more information about Device tables, see the clause, Common Table Formats.)

Example 7 at the end of this clause shows a sample of a BaseCoordFormat3 table.

BaseCoordFormat3 table: Design units plus Device table

Type	Name	Description
uint16	BaseCoordFormat	Format identifier-format = 3
int16	Coordinate	X or Y value, in design units
Offset	DeviceTable	Offset to Device table for X or Y value

6.3.1.3 BASE table examples

The rest of this clause describes and illustrates examples of all the BASE tables. All the examples reflect unique parameters described below, but the samples provide a useful reference for building tables specific to other situations.

Most of the examples have three columns showing hex data, source, and comments.

Example 1: BASE header table, Axis table, BaseTagList table, BaseScriptList table, and BaseScriptRecord

Example 1 describes a sample font that contains four scripts: Cyrillic, Devanagari, Han, and Latin. All four scripts are rendered horizontally; only one script, Han, is rendered vertically. As a result, the BASE header gives Offsets to two Axis tables: HorizAxis and VertAxis. Example 1 only shows data defined in the HorizAxis table.

In the HorizAxis table, the BaseScriptList enumerates all four scripts. The BaseTagList table names three horizontal baselines for rendering these scripts: hanging, ideographic, and roman. The hanging baseline is the default for Devanagari, the ideographic baseline is the default for Han, and the roman baseline is the default for both Latin and Cyrillic.

The VertAxis table (not shown) would be defined similarly: its BaseScriptList would enumerate one script, Han, and its BaseTagList would specify the vertically centered baseline for rendering the Han script.

Example 1

Hex Data	Source	Comments
	BASEHeader TheBASEHeader	BASE table header definition
00010000	0x00010000	Version

0008	HorizontalAxisTable	Offset to HorizAxis table
010C	VerticalAxisTable	Offset to VertAxis table
	Axis HorizontalAxisTable	Axis table definition
0004	HorizBaseTagList	Offset to BaseTagList table
0012	HorizBaseScriptList	Offset to BaseScriptList table
	BaseTagList HorizBaseTagList	BaseTagList table definition
0003	3	BaseTagCount
68616E67	"hang"	BaselineTag[0], in alphabetical order
6964656F	"ideo"	BaselineTag[1]
726F6D6E	"romn"	BaselineTag[2]
	BaseScriptList HorizBaseScriptList	BaseScriptList table definition
0004	4	BaseScriptCount BaseScriptRecord[0], in alphabetical order
6379726C	"cyrl"	BaseScriptTag for Cyrillic script
001A	HorizCyrillicBaseScriptTable	Offset to BaseScript table for Cyrillic script BaseScriptRecord[1]
6465766E	"devn"	BaseScriptTag for Devanagari script
0060	HorizDevanagariBaseScriptTable	Offset to BaseScript table for Devanagari script BaseScriptRecord[2]
68616E69	"hani"	BaseScriptTag for Han script
008A	HorizHanBaseScriptTable	Offset to BaseScript table for Han script BaseScriptRecord[3]
6C61746E	"latn"	BaseScriptTag for Latin script
00B4	HorizLatinBaseScriptTable	Offset to BaseScript table for Latin script

Example 2: BaseScript table and BaseLangSysRecord

Example 2 shows the BaseScript table and BaseLangSysRecord for the Cyrillic script, one of the four scripts included in the sample font described in Example 1. The BaseScript table specifies Offsets to tables that contain the baseline and min/max extent data for Cyrillic. (The BaseScript tables for the other three scripts in the font would be defined similarly.) Again, the table specifies only the horizontal text-layout information.

The HorizCyrillicBaseValues table contains the baseline information for the script, and the HorizCyrillicDefaultMinMax table contains the default script extents. In addition, a BaseLangSysRecord defines min/max extent data for the Russian language system.

Example 2

Hex Data	Source	Comments
	BaseScript HorizCyrillicBaseScriptTable	BaseScript table definition for Cyrillic script
000C	HorizCyrillicBaseValuesTable	Offset to BaseValues table
0022	HorizCyrillicDefault MinMaxTable	Offset to DefaultMinMax table default script extents
0001	1	BaseLangSysCount, feature-specific extents BaseLangSysRecord[0] in alphabetical order
52555320	"RUS "	BaseLangSysTag, Russian language system
0030	HorizRussianMinMaxTable	Offset to MinMax table feature-specific extents

Example 3: BaseValues table

Example 3 extends the BASE table definition for the Cyrillic script described in Examples 1 and 2. It contains two parts:

- Example 3A illustrates a fully defined BaseValues table for Cyrillic. The table includes the corresponding BaseCoord table definitions.
- Example 3B shows two different sets of baseline values that can be defined for each of the four scripts in the sample font.

The examples show only horizontal text-layout data, and the font uses 2,048 design units/em.

Example 3A: BaseValues table for Cyrillic

The BaseValues table of Example 3A identifies the default baseline for Cyrillic and specifies coordinate positions for each baseline listed in the BaseTagList shown in Example 1:

- The hanging baseline is the default for the Devanagari script, and it has the highest baseline position.
- The ideographic baseline is the default for the Han script, and it has the lowest baseline position.
- The roman baseline is the default for both the Latin and Cyrillic scripts, and its position lies between the hanging and ideographic baselines.

Example 3A

Hex Data	Source	Comments
	BaseValues HorizCyrillicBaseValuesTable	BaseValues table definition for Cyrillic script
0002	2	DefaultIndex, roman baseline BaselineTag index
0003	3	BaseCoordCount, equals BaseTagCount
000A	HorizHangingBaseCoordForCyrl	Offset to BaseCoord[0] table hanging baseline coordinate, order matches order of BaselineTag array in BaseTagList
000E	HorizideographicBaseCoordForCyrl	Offset to BaseCoord[1] table ideographic baseline coordinate
0012	HorizromanBaseCoordForCyrl	Offset to BaseCoord[2] table roman baseline coordinate
	BaseCoordFormat1 HorizHangingBaseCoordForCyrl	BaseCoord table definition
0001	1	BaseCoordFormat design units only
05DC	1500	Coordinate Y value, in design units
	BaseCoordFormat1 HorizideographicBaseCoordForCyrl	BaseCoord table definition
0001	1	BaseCoordFormat design units only
FEE0	-288	Coordinate Y value, in design units
	BaseCoordFormat1 HorizromanBaseCoordinateForCyrl	BaseCoord table definition
0001	1	BaseCoordFormat, design units only
0000	0	Coordinate, Y value, in design units

Example 3B: Baseline values for four scripts

Example 3B shows two tables that contain baseline values for each of the four scripts in the sample font described in Example 1:

- The first table shows what might happen if the baseline values in all four scripts are designed consistently. Their respective BaseValues tables list identical baseline values with the roman baseline positioned at a Y value of zero (0), the ideographic baseline at 1500, and the hanging baseline at -288.
- The second table shows what might happen if the baseline values in the scripts are designed differently with the default baseline for each script at the zero (0) coordinate.

Either method of assigning baseline values can be used in the BASE table.

Example 3B: Identical baseline values

Baseline type	Han	Latin	Cyrillic	Devanagari
hanging	1500	1500	1500	1500
roman	0	0	0	0
ideographic	-288	-288	-288	-288

Example 3B: Assigned baseline values with default baselines at 0

Baseline type	Han	Latin	Cyrillic	Devanagari
hanging	1788	1500	1500	0
roman	288	0	0	-1500
ideographic	0	-288	-288	-1788

Example 4: MinMax table and FeatMinMaxRecord

Example 4 shows MinMax table and FeatMinMaxRecord definitions for the same Cyrillic script described in the previous example. It contains two parts:

- Example 4A defines tables with different script, language system, and feature extents.
- Example 4B shows these same table definitions written when the language system extents match the script extents, but an obscure feature of the language system requires feature-specific extents if that feature is implemented.

The examples show only horizontal text-layout data, and the font uses 2,048 design units/em.

Example 4A: Min/Max extents for Cyrillic script, Russian language, and Russian feature

Example 4A shows two MinMax tables and a FeatMinMaxRecord for the Cyrillic script, along with sample BaseCoord tables. Only the MinCoord extent data is included.

The DefaultMinMax table defines the default minimum and maximum extents for the Cyrillic script. Another MinMax table defines language-specific min/max extents for the Russian language system to accommodate the height and width of certain glyphs used in Russian. Also, a FeatMinMaxRecord defines min/max extents for a single feature in the Russian language system that substitutes a tall integral math symbol when required.

Example 4A

Hex Data	Source	Comments
	MinMax HorizCyrillicDefault MinMaxTable	DefaultMinMax table definition, Cyrillic script
0006	HorizCyrillic MinCoordTable	MinCoord Offset to BaseCoord table
000A	HorizCyrillic MaxCoordTable	MaxCoord Offset to BaseCoord table
0000	0	FeatMinMaxCount no default feature extents FeatMinMaxRecord[], no FeatMinMaxRecords
	BaseCoordFormat1 HorizCyrillic MinCoordTable	BaseCoord table definition, default Cyrillic Min extent coordinate
0001	1	BaseCoordFormat, design units only
FF38	-200	Coordinate Y value, in design units
	BaseCoordFormat1 HorizCyrillic MaxCoordTable	BaseCoord table definition default Cyrillic Max extent coordinate
0001	1	BaseCoordFormat, design units only
0674	1652	Coordinate Y value, in design units
	MinMax HorizRussianMinMaxTable	MinMax table definition Russian language extents
000E	HorizRussianLangSys MinCoordTable	MinCoord Offset to BaseCoord table
0012	HorizRussianLangSys MaxCoordTable	MaxCoord Offset to BaseCoord table

0001	1	FeatMinMaxCount FeatMinMaxRecord[0] in alphabetical order
696E7467	"intg"	FeatureTableTag integral math symbol Feature must be same as Tag in FeatureList
0016	HorizRussianFeature MinCoordTable	MinCoord Offset to BaseCoord table
001A	HorizRussianFeature MaxCoordTable	MaxCoord Offset to BaseCoord table
	BaseCoordFormat1 HorizRussianLangSys MinCoordTable	BaseCoord table definition Russian language min extent coordinate
0001	1	BaseCoordFormat design units only
FF08	-248	Coordinate Y value, in design units, increased Min extent beyond default Cyrillic min extent
	BaseCoordFormat1 HorizRussianLangSys MaxCoordTable	BaseCoord table definition Russian language feature Max extent coordinate
0001	1	BaseCoordFormat design units only
06A4	1700	Coordinate Y value, in design units increased max extent beyond default Cyrillic max extent
	BaseCoordFormat1 HorizRussianFeature MinCoordTable	BaseCoord table definition Russian language Min extent coordinate
0001	1	BaseCoordFormat Design Units Only
FED8	-296	Coordinate Y value, in design units, increased Min extent beyond default Cyrillic script and Russian language min extents
	BaseCoordFormat1 HorizRussianFeature MaxCoordTable	BaseCoord table definition Russian language feature Max extent coordinate
0001	1	BaseCoordFormat design units only
06D8	1752	Coordinate Y value, in design units increased Max extent beyond default Cyrillic script and Russian language max extents

Example 4B: Min/Max extents for Cyrillic script and Russian feature

A particular language system does not need to define min/max extent coordinates if its extents match the default extents defined for the script. However, an obscure or infrequently used feature within the language system may require feature-specific extent values for proper rendering.

Example 4B shows the MinMax and FeatMinMaxRecord table definitions for this situation. The example also includes a BaseScript table, but not a BaseValues tables since it is not relevant in this example. The example shows horizontal text layout extents for the Cyrillic script and feature-specific extents for one feature in the Russian language system. Much of the data is repeated from Example 4A and modified here for comparison.

The BaseScript table includes a DefaultMinMax table for the Cyrillic script and a BaseLangSysRecord that defines a BaseLangSysTag and an Offset to a MinMax table for the Russian language. The MinMax table includes a FeatMinMaxRecord and specifies a FeatMinMaxCount, but both the MinCoord and MaxCoord Offsets in the MinMax table are set to NULL since no language-specific extent values are defined for Russian. The FeatMinMaxRecord defines the min/max coordinates for the Russian feature and specifies the correct FeatureTableTag.

Example 4B

Hex Data	Source	Comments
	BaseScript HorizCyrillicBaseScriptTable	BaseScript table definition Cyrillic script
0000	NULL	Offset to BaseValues table
000C	HorizCyrillicDefault MinMaxTable	Offset to DefaultMinMax table for default script extents
0001	1	BaseLangSysCount BaseLangSysRecord[0] for Russian feature-specific-extents
52555320	"RUS "	BaseLangSysTag = Russian
001A	HorizRussian MinMaxTable	Offset to MinMax table for feature-specific extents
	MinMax HorizCyrillicDefault MinMaxTable	DefaultMinMax table definition Cyrillic script
0006	HorizCyrillic MinCoordTable	MinCoord Offset to BaseCoord table
000A	HorizCyrillic MaxCoordTable	MaxCoord Offset to BaseCoord table
0000	0	FeatMinMaxCount, no default feature extents FeatMinMaxRecord[], no FeatMinMaxRecords
	BaseCoordFormat1 HorizCyrillic	BaseCoord table definition default Cyrillic Min extent coordinate

	MinCoordTable	
0001	1	BaseCoordFormat design units only
FF38	-200	Coordinate Y value, in design units
	BaseCoordFormat1 HorizCyrillic MaxCoordTable	BaseCoord table definition default Cyrillic Min extent coordinate
0001	1	BaseCoordFormat design units only
0674	1652	Coordinate Y value, in design units
	MinMax HorizRussian MinMaxTable	MinMax table definition for Russian feature no extent differences for Russian language itself
0000	NULL	Offset to Min BaseCoord table not defined, matches default
0000	NULL	Offset to Max BaseCoord table not defined, matches default
0001	1	FeatMinMaxCount, FeatMinMaxRecord[0] in alphabetical order
696E7467	"intg"	FeatureTableTag integral math sign Feature must be same as Tag in FeatureList
000E	HorizRussianFeature MinCoordTable	MinCoord Offset to BaseCoord table
0012	HorizRussianFeature MaxCoordTable	MaxCoord Offset to BaseCoord table
	BaseCoordFormat1 HorizRussianFeature MinCoordTable	BaseCoord table definition Russian Feature Min extent coordinate
0001	1	BaseCoordFormat, design units only
FED8	-296	Coordinate Y value, in design units increased Min extent beyond default Cyrillic Min extent
	BaseCoordFormat1 HorizRussianFeature MaxCoordTable	BaseCoord table definition, Russian feature Max extent coordinate

0001	1	BaseCoordFormat design units only
06D8	1752	Coordinate Y value, in design units, increased Max extent beyond default Cyrillic Max extent

Example 5: BaseCoordFormat1 table

Example 5 illustrates BaseCoordFormat1, which specifies single coordinate values in design units only. The font uses 2,048 design units/em. The example defines the default minimum extent coordinate for a math script.

Example 5

Hex Data	Source	Comments
	BaseCoordFormat1 HorizMathMinCoordTable	Definition of BaseCoord table for Math Min coordinate
0001	1	BaseCoordFormat, design units only
FEE8	-280	Coordinate Y value, in design units

Example 6: BaseCoordFormat2 table

Example 6 illustrates the BaseCoord Format 2. Like Example 5, it specifies the minimum extent coordinate for a math script. With this format, the coordinate value depends on the final position of a specific contour point on one glyph, the integral math symbol, after hinting. Again, the value is in design units (2,048 units/em).

Example 6

Hex Data	Source	Comments
	BaseCoordFormat2 HorizMathMinCoordTable	BaseCoord table definition for Math Min coordinate
0002	2	BaseCoordFormat design units plus contour point
FEE8	-280	Coordinate Y value, in design units
0128	IntegralSignGlyphID	ReferenceGlyph math integral sign
0043	67	BaseCoordPoint glyph contour point index

Example 7: BaseCoordFormat3 table

Example 7 illustrates the BaseCoord Format 3. Like Examples 5 and 6, it specifies the minimum extent coordinate for a math script in design units (2,048 units/em). This format, however, uses a Device table to modify the coordinate value for the point size and resolution of the output font. Here, the Device table defines pixel adjustments for font sizes from 11 ppem to 15 ppem. The adjustments add one pixel at each size.

Example 7

Hex Data	Source	Comments
	BaseCoordFormat3 HorizMathMinCoordTable	BaseCoord table definition for Math Min coordinate
0003	3	BaseCoordFormat design units plus device table
	-280	Coordinate Y value, in design units
000C	HorizMathMin CoordDeviceTable	Offset to Device table
	DeviceTableFormat1 HorizMathMin CoordDeviceTable	Device table definition for MinCoord
000B	11	StartSize -11 ppem
000F	15	EndSize -15 ppem
0001	1	DeltaFormat signed 2 bit value, 8 values per uint16
	1	Increase 11ppem by 1 pixel
	1	Increase 12ppem by 1 pixel
	1	Increase 13ppem by 1 pixel
	1	Increase 14ppem by 1 pixel
5540	1	Increase 15ppem by 1 pixel

6.3.2 GDEF – The glyph definition table

The Glyph Definition (GDEF) table contains four types of information in four independent tables:

- The *GlyphClassDef* table classifies the different types of glyphs in the font.
- The *AttachmentList* table identifies all attachment points on the glyphs, which streamlines data access and bitmap caching.
- The *LigatureCaretList* table contains positioning data for ligature carets, which the text-processing client uses on screen to select and highlight the individual components of a ligature glyph.
- The *MarkAttachClassDef* table classifies mark glyphs, to help group together marks that are positioned similarly.

- The *MarkGlyphSetsTable* allows the enumeration of an arbitrary number of glyph sets that can be used as an extension of the mark attachment class definition to allow lookups to filter mark glyphs by arbitrary sets of marks.

The GSUB and GPOS tables may reference certain GDEF table information.

See, for example, the LookupFlag bit enumeration in "OFF Layout Common Table Formats" (subclause 6.2.4,"Features and Lookups").

6.3.2.1 Overview

A client may use any one or more of the five GDEF tables during text processing. This overview explains how each of the five tables are organized and used (See Figure 22). The rest of this clause describes the individual GDEF tables and the tables that they reference.

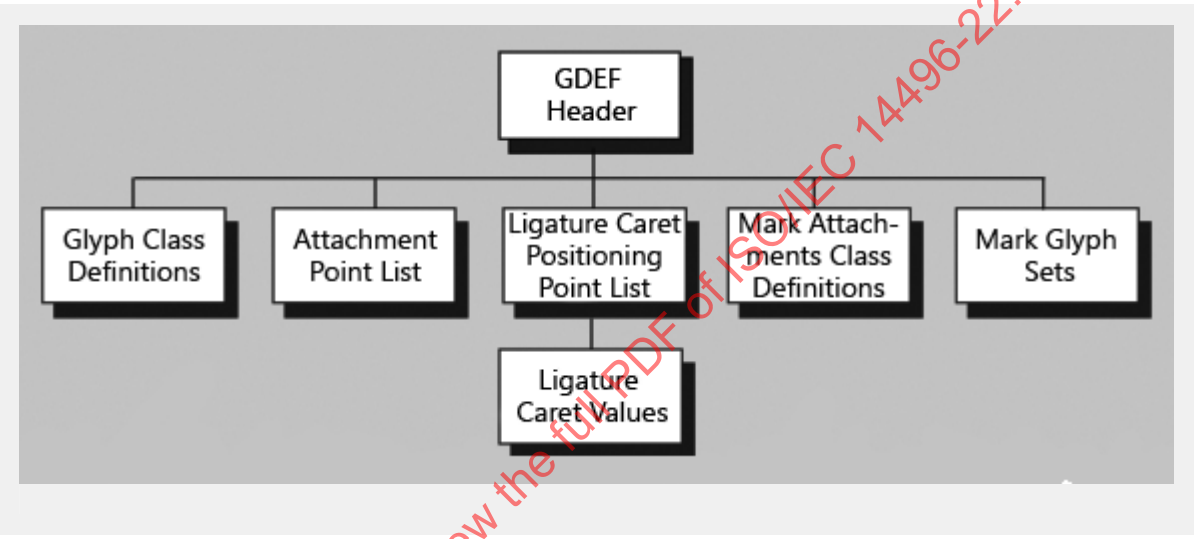

Figure 22 – High-level organization of GDEF table

6.3.2.2 GDEF table structure

The Glyph Class Definition (GlyphClassDef) table identifies four types of glyphs in a font: base glyphs, ligature glyphs, combining mark glyphs, and component glyphs (see Figure 23). GSUB and GPOS lookups define and use these glyph classes to differentiate the types of glyphs in a string. For example, GPOS uses the glyph classes to distinguish between a simple base glyph and the mark glyph that follows it.

Figure 23 – A base glyph, ligature glyph, mark glyph, and component glyphs

In addition, a client uses class definitions to apply GSUB and GPOS LookupFlag data correctly. For example, a LookupFlag may specify ignoring ligatures and marks during a glyph operation. If the font does not include a GlyphClassDef table, the client must define and maintain this information when using the GSUB and GPOS tables.

Attachment Point List table

The Attachment Point List table (AttachmentList) identifies all the attachment points defined in the GPOS table and their associated glyphs so a client can quickly access coordinates for each glyph's attachment points. As a result, the client can cache coordinates for attachment points along with glyph bitmaps and avoid

recalculating the attachment points each time it displays a glyph. Without this table, processing speed would be slower because the client would have to decode the GPOS lookups that define attachment points and compile the points in a list.

Ligature Caret List table

The Ligature Caret List table (LigatureCaretList), particularly useful in Arabic and other scripts with many ligatures, specifies coordinates for positioning carets on all ligatures in a font. The client uses this data to select and highlight ligature components in displayed text (see Figure 24).

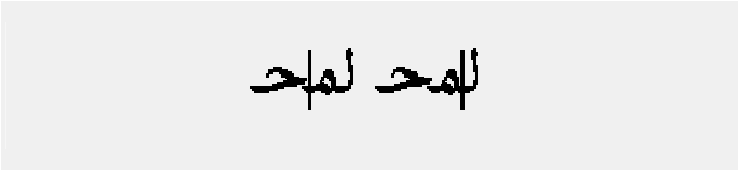

Figure 24 – Proper ligature caret postioning

Each ligature can have more than one caret position, with each position defined as an X or Y value on the baseline according to the writing direction of the script or language system. The font developer can use any of three formats to represent a caret coordinate value. One format represents values in design units only, another fine-tunes a value based on a designated contour point, and the third uses a Device table to adjust values at specific font sizes.

Without a Ligature Caret List table, the client would have to define caret positions without knowing the positions of the ligature components. The resulting highlighting or hit-testing might be ambiguous. For example, suppose a client places a caret at the midpoint position along the width of a hyphothetical "wi" ligature. Because the "w" is wider than the "i," that position would not clearly indicate which component is selected. Instead, for accurate selection, the caret should be moved to the right so that either the "w" or "i" could be clearly highlighted.

GDEF header

The GDEF table begins with a header that consists of a version number (Version), currently set to 0x00010002, an Offset to a table defining the types of glyphs in the font (GlyphClassDef), an Offset to a list defining attachment points on the glyphs(AttachList), an Offset to a ligature caret list (LigCaretList) and an Offset to a list defining types of marks that can be attached (MarkAttachClassDef). The format used for the MarkAttachClassDef is the same as that for GlyphClassDef. Please refer the 'LookupFlag bit enumeration' subclause 6.2 in the Common Table Formats for more on using lookup flags with the information in these fields.

Example 1 at the end of this clause shows a GDEF Header table.

Type	Name	Description
ULONG	Version	Version of the GDEF table-currently 0x00010002
Offset	GlyphClassDef	Offset to class definition table for glyph type-from beginning of GDEF header (may be NULL)
Offset	AttachList	Offset to list of glyphs with attachment points-from beginning of GDEF header (may be NULL)
Offset	LigCaretList	Offset to list of positioning points for ligature carets-from beginning of GDEF header (may be NULL)
Offset	MarkAttachClassDef	Offset to class definition table for mark attachment type-from beginning of GDEF header (may be NULL)
Offset	MarkGlyphSetsDef	Offset to the table of mark set definitions - from beginning of GDEF header (may be NULL)

Glyph Class Definition table

The GSUB and GPOS tables use the Glyph Class Definition table (GlyphClassDef) to identify which glyph classes to adjust with lookups.

The table uses the same format as the Class Definition table (for details, see subclause 6.2, Common Table Formats). However, the GlyphClassDef table uses class values already defined in the GlyphClassDef Enumeration list:

GlyphClassDef Enumeration List

Class	Description
1	Base glyph (single character, spacing glyph)
2	Ligature glyph (multiple character, spacing glyph)
3	Mark glyph (non-spacing combining glyph)
4	Component glyph (part of single character, spacing glyph)

The font developer does not have to classify every glyph in the font, but any glyph not assigned a class value falls into Class zero (0). For instance, class values might be useful for the Arabic glyphs in a font, but not for the Latin glyphs. Then the GlyphClassDef table will list only Arabic glyphs, and-by default-the Latin glyphs will be assigned to Class 0. Component glyphs can be put together to generate ligatures. A ligature can be generated by creating a glyph in the font that references the component glyphs, or outputting the component glyphs in the desired sequence. Component glyphs are not used in defining any GSUB or GPOS formats.

Example 2 at the end of this clause defines a GlyphClassDef table with a sample glyph for each of the assigned classes.

Attachment List table

The Attachment List table (AttachList) may be used to cache attachment point coordinates along with glyph bitmaps.

The table consists of an Offset to a Coverage table (Coverage) listing all glyphs that define attachment points in the GPOS table, a count of the glyphs with attachment points (GlyphCount), and an array of Offsets to AttachPoint tables (AttachPoint). The array lists the AttachPoint tables, one for each glyph in the Coverage table, in the same order as the Coverage Index.

AttachList table

Type	Name	Description
Offset	Coverage	Offset to Coverage table - from beginning of AttachList table
uint16	GlyphCount	Number of glyphs with attachment points
Offset	AttachPoint[GlyphCount]	Array of Offsets to AttachPoint tables-from beginning of AttachList table-in Coverage Index order

An AttachPoint table consists of a count of the attachment points on a single glyph (PointCount) and an array of contour indices of those points (PointIndex), listed in increasing numerical order.

Example 3 at the end of the clause demonstrates an AttachList table that defines attachment points for two glyphs.

AttachPoint table

Type	Name	Description
uint16	PointCount	Number of attachment points on this glyph
uint16	PointIndex[PointCount]	Array of contour point indices -in increasing numerical order

Ligature Caret List table

The Ligature Caret List table (LigCaretList) defines caret positions for all the ligatures in a font. The table consists of an Offset to a Coverage table that lists all the ligature glyphs (Coverage), a count of the defined ligatures (LigGlyphCount), and an array of Offsets to LigGlyph tables (LigGlyph). The array lists the LigGlyph tables, one for each ligature in the Coverage table, in the same order as the Coverage Index.

Example 4 at the end of this clause shows a LigCaretList table.

LigCaretList table

Type	Name	Description
Offset	Coverage	Offset to Coverage table - from beginning of LigCaretList table
uint16	LigGlyphCount	Number of ligature glyphs
Offset	LigGlyph[LigGlyphCount]	Array of Offsets to LigGlyph tables-from beginning of LigCaretList table-in Coverage Index order

Ligature Glyph table

A Ligature Glyph table (LigGlyph) contains the caret coordinates for a single ligature glyph. The number of coordinate values, each defined in a separate CaretValue table, equals the number of components in the ligature minus one (1).

The LigGlyph table consists of a count of the number of CaretValue tables defined for the ligature (CaretCount) and an array of Offsets to CaretValue tables (CaretValue).

Example 4 at the end of the clause shows a LigGlyph table.

LigGlyph table

Type	Name	Description
uint16	CaretCount	Number of CaretValues for this ligature (components - 1)
Offset	CaretValue[CaretCount]	Array of Offsets to CaretValue tables-from beginning of LigGlyph table-in increasing coordinate order

Caret Values table

A Caret Values table (CaretValues), which defines caret positions for a ligature, can be any of three possible formats. One format uses design units to define the caret position. The other two formats use a contour point

or Device table to fine-tune a caret's position at specific font sizes and device resolutions. Caret coordinates are either X or Y values, depending upon the text direction.

CaretValue Format 1

The first format (CaretValueFormat1) consists of a format identifier (CaretValueFormat), followed by a single coordinate for the caret position (Coordinate). The Coordinate is in design units.

This format has the benefits of small size and simplicity, but the Coordinate value cannot be hinted for fine adjustments at different device resolutions.

Exampel 4 at the end of this clause shows a CaretValueFormat1 table.

CaretValueFormat1 table: Design units only

Type	Name	Description
uint16	CaretValueFormat	Format identifier-format = 1
int16	Coordinate	X or Y value, in design units

CaretValue Format 2

The second format (CaretValueFormat2) specifies the caret coordinate in terms of a contour point index on a specific glyph. During font hinting, the contour point on the glyph outline may move. The point's final position after hinting provides the final value for rendering a given font size.

The table contains a format identifier (CaretValueFormat) and a contour point index (CaretValuePoint).

Example 5 at the end of this clause demonstrates a CaretValueFormat2 table.

CaretValueFormat2 table: Contour point

Type	Name	Description
uint16	CaretValueFormat	Format identifier-format = 2
uint16	CaretValuePoint	Contour point index on glyph

CaretValue Format 3

The third format (CaretValueFormat3) also specifies the value in design units, but it uses a Device table rather than a contour point to adjust the value. This format offers the advantage of fine-tuning the Coordinate value for any device resolution. (For more information about Device tables, see the clause, Common Table Formats.)

The format consists of a format identifier (CaretValueFormat), an X or Y value (Coordinate), and an Offset to a Device table (DeviceTable).

Example 6 at the end of this clause shows a CaretValueFormat3 table.

CaretValueFormat3 table: Design units plus Device table

Type	Name	Description
uint16	CaretValueFormat	Format identifier-format = 3
int16	Coordinate	X or Y value, in design units
Offset	DeviceTable	Offset to Device table for X or Y value-from beginning of CaretValue table

Mark Attachment Class Definition table

A Mark Attachment Class Definition Table defines the class to which a mark glyph may belong. This table uses the same format as the Class Definition table (for details, see subclause 6.2, Common Table Formats).

Example 7 in this document shows a MarkAttachClassDef table.

Mark Glyph Sets table

Mark glyph sets are used in GSUB and GPOS lookups to filter which marks in a string are considered or ignored. Mark glyph sets are defined in a MarkGlyphSets table, which contains offsets to individual sets each represented by a standard Coverage table.

MarkGlyphSetsTable

Type	Name	Description
uint16	MarkSetTableFormat	Format identifier = 1
uint16	MarkSetCount	Number of mark sets defined
ULONG	Coverage [MarkSetCount]	Array of offsets to mark set Coverage tables (offsets are calculated from the start of the MarkGlyphSets table)

Mark glyph sets are used for the same purpose as mark attachment classes, which is as filters for GSUB and GPOS lookups. Mark glyph sets differ from mark attachment classes, however, in that mark glyph sets may intersect as needed by the font developer. As for mark attachment classes, only one mark glyph set can be referenced in any given lookup.

Note that the array of offsets for the Coverage tables uses ULONG, not Offset.

6.3.2.3 GDEF table examples

The rest of this clause describes examples of all the GDEF table formats. All the examples reflect unique parameters described below, but the samples provide a useful reference for building tables specific to other situations.

The examples have three columns showing hex data, source, and comments.

Example 1: GDEF header

Example 1 shows a GDEF Header definition with Offsets to each of the main tables in GDEF.

Hex Data	Source	Comments
	GDEFHeader	
	TheGDEFHeader	GDEFHeader table definition
00010000	0x00010000	Version
000A	GlyphClassDefTable	Offset to GlyphClassDef table
0026	AttachListTable	Offset to AttachList table
0040	LigCaretListTable	Offset to LigCaretList table
005A	MarkAttachClassDefTable	Offset to Mark Attachment Class Definition Table

Example 2: GlyphClassDef table

The GlyphClassDef table in Example 2 specifies a glyph for the each of the glyph classes predefined in the GlyphClassDef Enumeration List.

Hex Data	Source	Comments
	ClassDefFormat2	
	GlyphClassDefTable	ClassDef table definition
0002	2	ClassFormat
0004	4	ClassRangeCount ClassRangeRecord[0]
0024	iGlyphID	Start
0024	iGlyphID	End
0001	1	Class, 1 = base glyphs ClassRangeRecord[1]
009F	ffiLigGlyphID	Start
009F	ffiLigGlyphID	End
0002	2	Class, 2 = ligature glyphs ClassRangeRecord[2]

0058	umlautAccentGlyphID	Start
0058	umlautAccentGlyphID	End
0003	3	Class, 3 = mark glyphs ClassRangeRecord[3]
018F	CurvedTailComponentGlyphID	Start
018F	CurvedTailComponentGlyphID	End
0004	4	Class, 4 = component glyphs

Example 3: AttachList table

In Example 3, the AttachList table enumerates the attachment points defined for two glyphs. The GlyphCoverage table identifies the glyphs: "a" and "e." For each covered glyph, an AttachPoint table specifies the attachment point count and point indices: one point for the "a" glyph and two for the "e" glyph.

Hex Data	Source	Comments
	AttachList AttachListTable	AttachList table definition
0012	GlyphCoverage	Offset to Coverage table
0002	2	GlyphCount
0008	aAttachPoint	AttachPoint[0]
000C	eAttachPoint	AttachPoint[1]
	AttachPoint aAttachPoint	AttachPoint table definition
0001	1	PointCount
0012	18	PointIndex[0]
	AttachPoint eAttachPoint	AttachPoint table definition
0002	2	PointCount

000E	14	PointIndex[0]
0017	23	PointIndex[1]
	CoverageFormat1 GlyphCoverage	Coverage table definition
0001	1	CoverageFormat
0002	2	GlyphCount
001C	aGlyphID	GlyphArray[0]
0020	eGlyphID	GlyphArray[1]

Example 4: LigCaretList table, LigGlyph table and CaretValueFormat1 table

Example 4 defines a list of ligature carets. The LigCoverage table lists all the ligature glyphs that define caret positions. In this example, two ligatures are covered, "ffi" and "fi." For each covered glyph, a LigGlyph table specifies the number of carets for the ligature and their coordinate values. The "fi" ligature defines one caret, positioned between the "f" and "i" components; the "ffi" ligature defines two, one positioned between the two "f" components and the other positioned between the "f" and "i." The CaretValue tables shown here use Format1, where values are specified in design units only.

Hex Data	Source	Comments
	LigCaretList LigCaretListTable	LigCaretList table definition
0008	LigCoverage	Offset to Coverage table
0002	2	LigGlyphCount
0010	fiLigGlyph	Offset to LigGlyph table[0]
0014	ffiLigGlyph	Offset to LigGlyph table[1]
	CoverageFormat1 LigCoverage	Coverage table definition
0001	1	CoverageFormat
0002	2	GlyphCount

009F	ffiLigGlyphID	GlyphArray[0]
00A5	fiLigGlyphID	GlyphArray[1]
	LigGlyph fiLigGlyph	LigGlyph table definition
0001	1	CaretCount, equals the number of components - 1
000E	CaretFI	CaretValue[0]
	LigGlyph ffiLigGlyph	LigGlyph table definition
0002	2	CaretCount, equals the number of components - 1
0006	CaretFFI1	CaretValue[0]
000E	CaretFFI2	CaretValue[1]
	CaretValueFormat1 CaretFI	CaretValue table definition
0001	1	CaretValueFormat design units only
025B	603	Coordinate X or Y value
	CaretValueFormat1 CaretFFI1	CaretValue table definition
0001	1	CaretValueFormat design units only
025B	603	Coordinate X or Y value
	CaretValueFormat1 CaretFFI2	CaretValue table definition
0001	1	CaretValueFormat design units only
04B6	1206	Coordinate X or Y value

Example 5: CaretValueFormat2 table

Example 5 shows a CaretValueFormat2 table that specifies a ligature caret coordinate in terms of a contour point index on a specific glyph. The final position of the caret depends on the location of the contour point on the glyph after hinting.

Hex Data	Source	Comments
	CaretValueFormat2	
	Caret1	CaretValue table definition
0002	2	CaretValueFormat contour point
000D	13	CaretValuePoint contour point index

Example 6: CaretValueFormat3 table

In Example 6, the CaretValueFormat3 table defines a caret position in design units, but includes a Device table to adjust the X or Y coordinate for the point size and resolution of the output font. Here, the Device table specifies pixel adjustments for font sizes from 12 ppem to 17 ppem.

Hex Data	Source	Comments
	CaretValueFormat3	
	Caret3	CaretValue table definition
0003	3	CaretValueFormat design units plus Device table
04B6	1206	Coordinate X or Y value, design units
0006	CaretDevice	Offset to Device table
	DeviceTableFormat2	
	CaretDevice	Device Table definition
000C	12	StartSize
0011	17	EndSize
0002	2	DeltaFormat
	1	increase 12ppm by 1 pixel
	1	increase 13ppm by 1 pixel

	1	increase 14ppm by 1 pixel
1111	1	increase 15ppm by 1 pixel
	2	increase 16ppm by 2 pixels
2200	2	increase 17ppm by 2 pixels

Example 7: MarkAttachClassDef table

In Example 7, the MarkAttachClassDef table specifies an attachment class for the each of the glyph ranges predefined in the GlyphClassDef Enumeration List as marks.

Hex Data	Source	Comments
	ClassDefFormat2 theMarkAttachClassDefTable	ClassDef table definition
0002	2	ClassFormat
0004	4	ClassRangeCount ClassRangeRecord[0]
0268	graveAccentGlyphID	Start
026A	circumflexAccentGlyphID	End
0001	1	Class, 1 = top marks ClassRangeRecord[1]
0270	diaeresisAccentGlyphID	Start
0272	acuteAccentGlyphID	End
0001	1	Class, 1 = top marks ClassRangeRecord[2]
028C	diaeresisBelowGlyphID	Start
028F	cedillaGlyphID	End
0002	2	Class, 2 = bottom marks ClassRangeRecord[3]
0295	circumflexBelowGlyphID	Start

0295	circumflexBelowGlyphID	End
0002	2	Class, 2 = bottom marks

6.3.3 GPOS – The glyph positioning table

The Glyph Positioning table (GPOS) provides precise control over glyph placement for sophisticated text layout and rendering in each script and language system that a font supports.

6.3.3.1 Overview

Complex glyph positioning becomes an issue in writing systems, such as Vietnamese, that use diacritical and other marks to modify the sound or meaning of characters. These writing systems require controlled placement of all marks in relation to one another for legibility and linguistic accuracy.

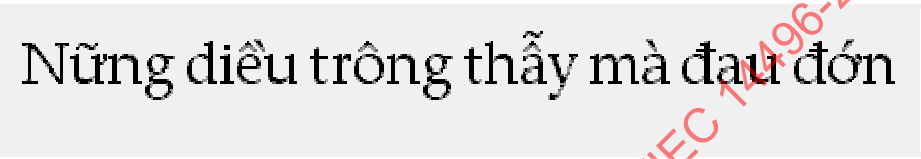

Figure 25 – Vietnamese words with marks.

Other writing systems require sophisticated glyph positioning for correct typographic composition. For instance, Urdu glyphs are calligraphic and connect to one another along a descending, diagonal text line that proceeds from right to left. To properly render Urdu, a text-processing client must modify both the horizontal (X) and vertical (Y) positions of each glyph (see Figure 26).

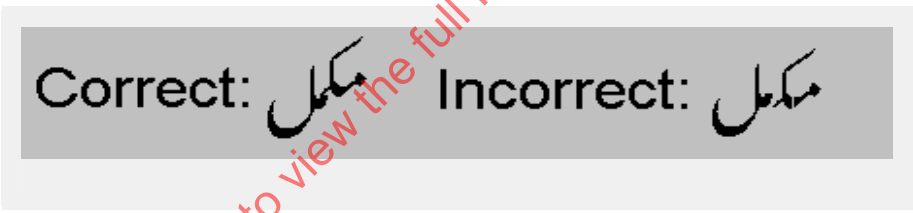

Figure 26 – Urdu layout requires glyph positioning control, as well as contextual substitution

With the GPOS table, a font developer can define a complete set of positioning adjustment features in an OFF font. GPOS data, organized by script and language system, is easy for a text-processing client to use to position glyphs.

Positioning glyphs with TrueType 1.0

Glyph positioning in TrueType uses only two values, placement and advance, to specify a glyph's position for text layout. If glyphs are positioned with respect to a virtual "pen point" that moves along a line of text, placement describes the glyph's position with respect to the current pen point, and advance describes where to move the pen point to position the next glyph (see Figure 27). For horizontal text, placement corresponds to the left side bearing, and advance corresponds to the advance width.

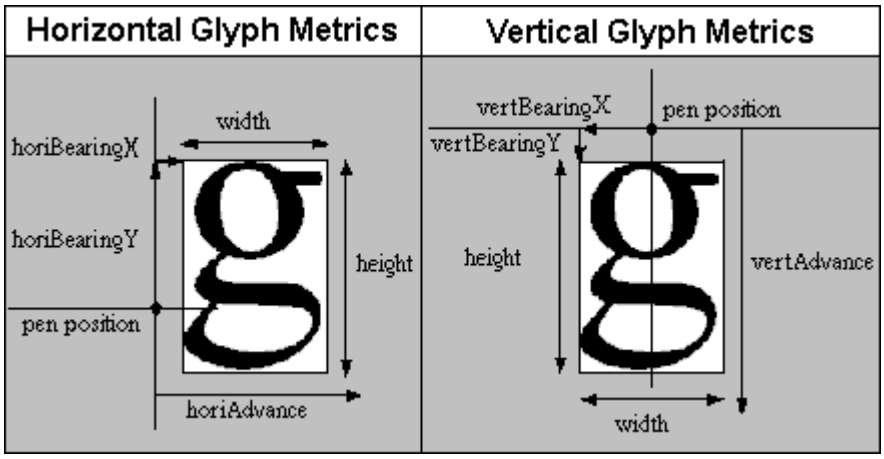

Figure 27 – Glyph positioning with TrueType

TrueType specifies placement and advance only in the X direction for horizontal layout and only in the Y direction for vertical layout. For simple Latin text layout, these two values may be adequate to position glyphs correctly. But, for texts that require more sophisticated layout, the values must cover a richer range. Placement and advance may need adjustment vertically, as well as horizontally.

The only positioning adjustment defined in TrueType is pair kerning, which modifies the horizontal spacing between two glyphs. A typical kerning table lists pairs of glyphs and specifies how much space a text-processing client should add or remove between the glyphs to properly display each pair. It does not provide specific information about how to adjust the glyphs in each pair, and cannot adjust contexts of more than two glyphs.

Positioning glyphs with OFF

OFF fonts allow excellent control and flexibility for positioning a single glyph and for positioning multiple glyphs in relation to one another. By using both X and Y values that the GPOS table defines for placement and advance and by using glyph attachment points, a client can more precisely adjust the position of a glyph.

In addition, the GPOS table can reference a Device table to define subtle, device-dependent adjustments to any placement or advance value at any font size and device resolution. For example, a Device table can specify adjustments at 51 pixels per em (ppem) that do not occur at 50 ppem.

X and Y values specified in OFF fonts for placement operations are always within the typical Cartesian coordinate system (origin at the baseline of the left side), regardless of the writing direction. Additionally, all values specified are done so in font unit measurements. This is especially convenient for font designers, since glyphs are drawn in the same coordinate system. However, it's important to note that the meaning of "advance width" changes, depending on the writing direction.

For example, in left-to-right scripts, if the first glyph has an advance width of 100, then the second glyph begins at 100,0. In right-to-left scripts, if the first glyph has an advance width of 100, then the second glyph begins at -100,0. For a top-to-bottom feature, to increase the advance height of a glyph by 100, the YAdvance = 100. For any feature, regardless of writing direction, to lower the dieresis over an 'o' by 10 units, set the YPlacement = -10.

Other GPOS features can define attachment points to combine glyphs and position them with respect to one another. A glyph might have multiple attachment points. The point used will depend on the glyph to be attached. For instance, a base glyph could have attachment points for different diacritical marks.

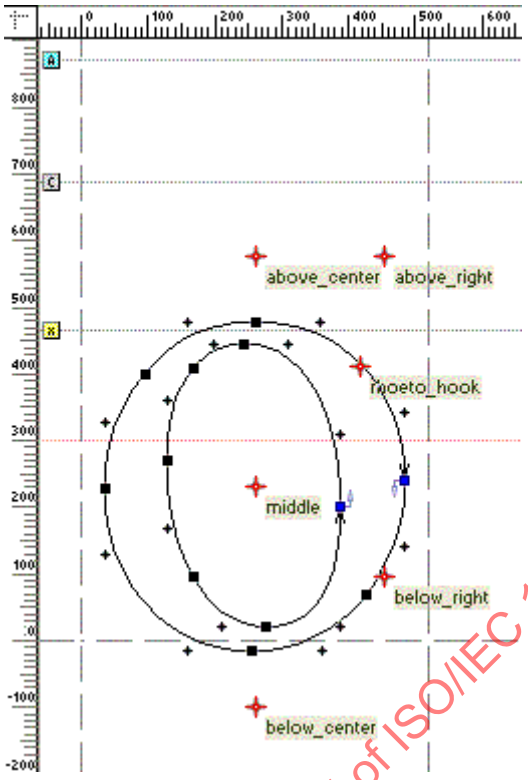

Figure 28 – Base glyph with multiple attachment points.

To reduce the size of the font file, a base glyph may use the same attachment point for all mark glyphs assigned to a particular class. For example, a base glyph could have two attachment points, one above and one below the glyph. Then all marks that attach above glyphs would be attached at the high point, and all marks that attach below glyphs would be attached at the low point. Attachment points are useful in scripts, such as Arabic, that combine numerous glyphs with vowel marks.

Attachment points also are useful for connecting cursive-style glyphs. Glyphs in cursive fonts can be designed to attach or overlap when rendered. Alternatively, the font developer can use OFF to create a cursive attachment feature and define explicit exit and entry attachment points for each glyph (see Figure 29).

Figure 29 – Entry and exit points marked on contextual Urdu glyph variations

The GPOS table supports eight types of actions for positioning and attaching glyphs:

- A *single adjustment* positions one glyph, such as a superscript or subscript.
- A *pair adjustment* positions two glyphs with respect to one another. Kerning is an example of pair adjustment.
- A *cursive attachment* describes cursive scripts and other glyphs that are connected with attachment points when rendered.
- A *MarkToBase attachment* positions combining marks with respect to base glyphs, as when positioning vowels, diacritical marks, or tone marks in Arabic, Hebrew, and Vietnamese.

- A *MarkToLigature attachment* positions combining marks with respect to ligature glyphs. Because ligatures may have multiple points for attaching marks, the font developer needs to associate each mark with one of the ligature glyph's components.
- A *MarkToMark attachment* positions one mark relative to another, as when positioning tone marks with respect to vowel diacritical marks in Vietnamese.
- *Contextual positioning* describes how to position one or more glyphs in context, within an identifiable sequence of specific glyphs, glyph classes, or varied sets of glyphs. One or more positioning operations may be performed on "input" context sequences. Figure 30 illustrates a context for positioning adjustments.
- *Chaining Contextual positioning* describes how to position one or more glyphs in a chained context, within an identifiable sequence of specific glyphs, glyph classes, or varied sets of glyphs. One or more positioning operations may be performed on "input" context sequences.

Figure 30 – Contextual positioning lowered the accent over a vowel glyph that followed an overhanging uppercase glyph

6.3.3.2 GPOS table organization and structure

The GPOS table begins with a header that defines Offsets to a ScriptList, a FeatureList, and a LookupList (see Figure 31):

- The ScriptList identifies all the scripts and language systems in the font that use glyph positioning.
- The FeatureList defines all the glyph positioning features required to render these scripts and language systems.
- The LookupList contains all the lookup data needed to implement each glyph positioning feature.

For a detailed discussion of ScriptLists, FeatureLists, and LookupLists see the OFF Common Table Formats . The following discussion summarizes how the GPOS table works.

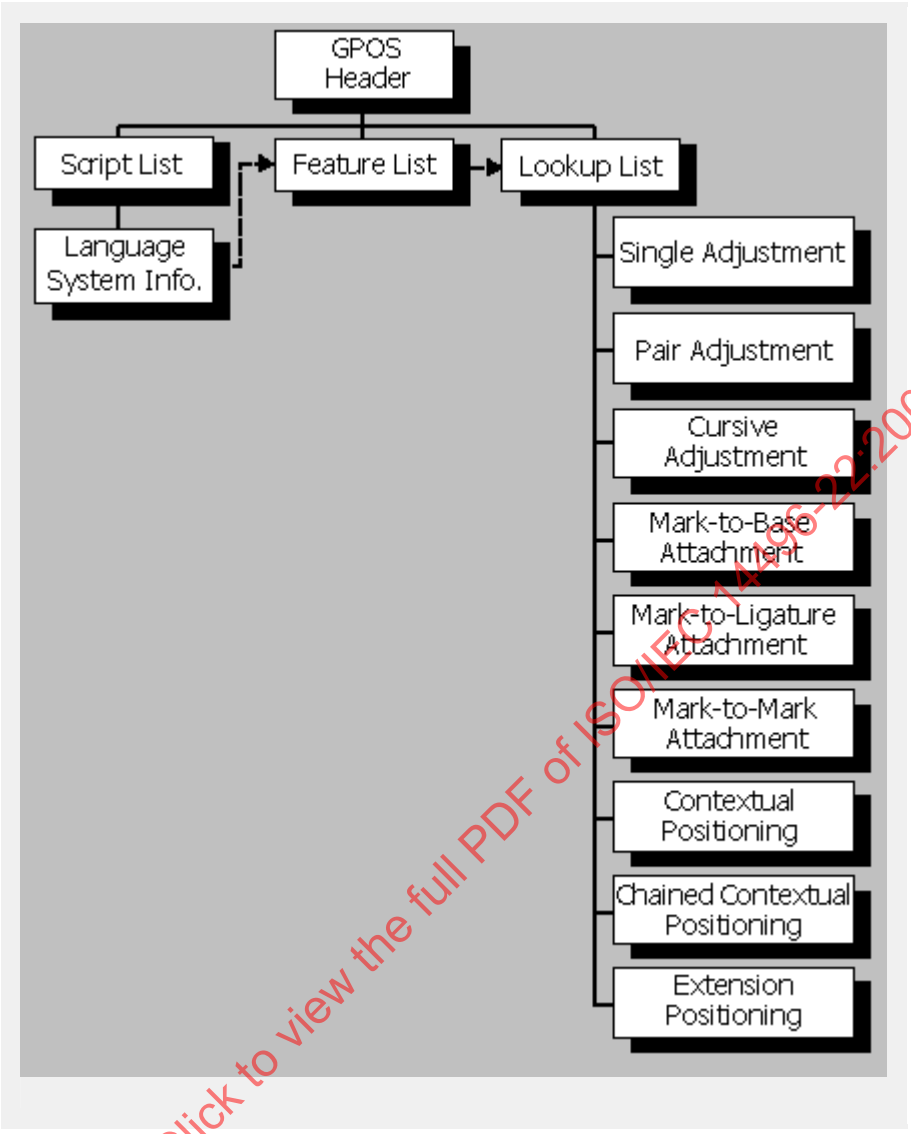

Figure 31 – High-level organization of GPOS table

The GPOS table is organized so text processing clients can easily locate the features and lookups that apply to a particular script or language system. To access GPOS information, clients should use the following procedure:

1. Locate the current script in the GPOS ScriptList table.
2. If the language system is known, search the script for the correct LangSys table; otherwise, use the script's default language system (DefaultLangSys table).
3. The LangSys table provides index numbers into the GPOS FeatureList table to access a required feature and a number of additional features.
4. Inspect the FeatureTag of each feature, and select the features to apply to an input glyph string.
5. Each feature provides an array of index numbers into the GPOS LookupList table. Lookup data is defined in one or more subtables that contain information about specific glyphs and the kinds of operations to be performed on them.
6. Assemble all lookups from the set of chosen features, and apply the lookups in the order given in the LookupList table.

A lookup uses subtables to define the specific conditions, type, and results of a positioning action used to implement a feature. All subtables in a lookup must be of the same LookupType, as listed in the LookupType Enumeration table:

LookupType Enumeration table for glyph positioning

Value	Type	Description
1	Single adjustment	Adjust position of a single glyph
2	Pair adjustment	Adjust position of a pair of glyphs
3	Cursive attachment	Attach cursive glyphs
4	MarkToBase attachment	Attach a combining mark to a base glyph
5	MarkToLigature attachment	Attach a combining mark to a ligature
6	MarkToMark attachment	Attach a combining mark to another mark
7	Context positioning	Position one or more glyphs in context
8	Chained Context positioning	Position one or more glyphs in chained context
9	Extension positioning	Extension mechanism for other positionings
10+	Reserved	For future use (must be set to zero)

Each LookupType is defined by one or more subtables, whose format depends on the type of positioning operation and the resulting storage efficiency. When glyph information is best presented in more than one format, a single lookup may define more than one subtable, as long as all the subtables are of the same LookupType. For example, within a given lookup, a glyph index array format may best represent one set of target glyphs, whereas a glyph index range format may be better for another set.

A series of positioning operations on the same glyph or string requires multiple lookups, one for each separate action. The values in the ValueRecords are accumulated in these cases. Each lookup is given a different array number in the LookupList table and is applied in the LookupList order.

During text processing, a client applies a lookup to each glyph in the string before moving to the next lookup. A lookup is finished for a glyph after the client locates the target glyph or glyph context and performs a positioning, if specified. To move to the "next" glyph, the client will typically skip all the glyphs that participated in the lookup operation: glyphs that were positioned as well as any other glyphs that formed a context for the operation.

There is just one exception: the "next" glyph in a sequence may be one of those that formed a context for the operation just performed. For example, in the case of pair positioning operations (i.e., kerning), if the position value record for the second glyph is null, that glyph is treated as the "next" glyph in the sequence.

This rest of this clause describes the GPOS header and the subtables defined for each LookupType. Several GPOS subtables share other tables: ValueRecords, Anchor tables, and MarkArrays. For easy reference, the shared tables are described at the end of this clause.

GPOS header

The GPOS table begins with a header that contains a version number (Version) initially set to 1.0 (0x00010000) and Offsets to three tables: ScriptList, FeatureList, and LookupList. For descriptions of these tables, see subclause 6.2, OFF Common Table Formats. Example 1 at the end of this clause shows a GPOS Header table definition.

GPOS Header

Value	Type	Description
Fixed	Version	Version of the GPOS table-initially = 0x00010000
Offset	ScriptList	Offset to ScriptList table-from beginning of GPOS table
Offset	FeatureList	Offset to FeatureList table-from beginning of GPOS table
Offset	LookupList	Offset to LookupList table-from beginning of GPOS table

6.3.3.3 GPOS lookup type descriptions

Lookup Type 1: Single adjustment positioning subtable

A single adjustment positioning subtable (SinglePos) is used to adjust the position of a single glyph, such as a subscript or superscript. In addition, a SinglePos subtable is commonly used to implement lookup data for contextual positioning.

A SinglePos subtable will have one of two formats: one that applies the same adjustment to a series of glyphs, or one that applies a different adjustment for each unique glyph.

Single Adjustment Positioning: Format 1

A SinglePosFormat1 subtable applies the same positioning value or values to each glyph listed in its Coverage table. For instance, when a font uses old-style numerals, this format could be applied to uniformly lower the position of all math operator glyphs.

The Format 1 subtable consists of a format identifier (PosFormat), an Offset to a Coverage table that defines the glyphs to be adjusted by the positioning values (Coverage), and the format identifier (ValueFormat) that describes the amount and kinds of data in the ValueRecord.

The ValueRecord specifies one or more positioning values to be applied to all covered glyphs (Value). For example, if all glyphs in the Coverage table require both horizontal and vertical adjustments, the ValueRecord will specify values for both XPlacement and Yplacement.

Example 2 at the end of this clause shows a SinglePosFormat1 subtable used to adjust the placement of subscript glyphs.

SinglePosFormat1 subtable: Single positioning value

Value	Type	Description
uint16	PosFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of SinglePos subtable
uint16	ValueFormat	Defines the types of data in the ValueRecord
ValueRecord	Value	Defines positioning value(s)-applied to all glyphs in the Coverage table

Single Adjustment Positioning: Format 2

A SinglePosFormat2 subtable provides an array of ValueRecords that contains one positioning value for each glyph in the Coverage table. This format is more flexible than Format 1, but it requires more space in the font file.

For example, assume that the Cyrillic script will be used in left-justified text. For all glyphs, Format 2 could define position adjustments for left side bearings to align the left edges of the paragraphs. To achieve this, the Coverage table would list every glyph in the script, and the SinglePosFormat2 subtable would define a ValueRecord for each covered glyph. Correspondingly, each ValueRecord would specify an XPlacement adjustment value for the left side bearing.

NOTE All ValueRecords defined in a SinglePos subtable must have the same ValueFormat. In this example, if XPlacement is the only value that a ValueRecord needs to optically align the glyphs, then XPlacement will be the only value specified in the ValueFormat of the subtable.

As in Format 1, the Format 2 subtable consists of a format identifier (PosFormat), an Offset to a Coverage table that defines the glyphs to be adjusted by the positioning values (Coverage), and the format identifier (ValueFormat) that describes the amount and kinds of data in the ValueRecords. In addition, the Format 2 subtable includes:

- A count of the ValueRecords (ValueCount). One ValueRecord is defined for each glyph in the Coverage table.
- An array of ValueRecords that specify positioning values (Value). Because the array follows the Coverage Index order, the first ValueRecord applies to the first glyph listed in the Coverage table, and so on.

Example 3 at the end of this clause shows how to adjust the spacing of three dash glyphs with a SinglePosFormat2 subtable.

SinglePosFormat2 subtable: Array of positioning values

Value	Type	Description
uint16	PosFormat	Format identifier-format = 2
Offset	Coverage	Offset to Coverage table-from beginning of SinglePos subtable
uint16	ValueFormat	Defines the types of data in the ValueRecord
uint16	ValueCount	Number of ValueRecords
ValueRecord	Value [ValueCount]	Array of ValueRecords-positioning values applied to glyphs

Lookup Type 2: Pair adjustment positioning subtable

A pair adjustment positioning subtable (PairPos) is used to adjust the positions of two glyphs in relation to one another-for instance, to specify kerning data for pairs of glyphs. Compared to a typical kerning table, however, a PairPos subtable offers more flexiblity and precise control over glyph positioning. The PairPos subtable can adjust each glyph in a pair independently in both the X and Y directions, and it can explicitly describe the particular type of adjustment applied to each glyph. In addition, a PairPos subtable can use Device tables to subtly adjust glyph positions at each font size and device resolution.

PairPos subtables can be either of two formats: one that identifies glyphs individually by index (Format 1), or one that identifies glyphs by class (Format 2).

Pair Positioning Adjustment: Format 1

Format 1 uses glyph indices to access positioning data for one or more specific pairs of glyphs. All pairs are specified in the order determined by the layout direction of the text.

NOTE For text written from right to left, the right-most glyph will be the first glyph in a pair; conversely, for text written from left to right, the left-most glyph will be first.

A PairPosFormat1 subtable contains a format identifier (PosFormat) and two ValueFormats:

- ValueFormat1 applies to the ValueRecord of the first glyph in each pair. ValueRecords for all first glyphs must use ValueFormat1. If ValueFormat1 is set to zero (0), the corresponding glyph has no ValueRecord and, therefore, should not be repositioned.
- ValueFormat2 applies to the ValueRecord of the second glyph in each pair. ValueRecords for all second glyphs must use ValueFormat2. If ValueFormat2 is set to null, then the second glyph of the pair is the "next" glyph for which a lookup should be performed.

A PairPos subtable also defines an Offset to a Coverage table (Coverage) that lists the indices of the first glyphs in each pair. More than one pair can have the same first glyph, but the Coverage table will list that glyph only once.

The subtable also contains an array of Offsets to PairSet tables (PairSet) and a count of the defined tables (PairSetCount). The PairSet array contains one Offset for each glyph listed in the Coverage table and uses the same order as the Coverage Index.

PairPosFormat1 subtable: Adjustments for glyph pairs

Value	Type	Description
uint16	PosFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of PairPos subtable-only the first glyph in each pair
uint16	ValueFormat1	Defines the types of data in ValueRecord1-for the first glyph in the pair -may be zero (0)
uint16	ValueFormat2	Defines the types of data in ValueRecord2-for the second glyph in the pair -may be zero (0)
uint16	PairSetCount	Number of PairSet tables
Offset	PairSetOffset [PairSetCount]	Array of Offsets to PairSet tables-from beginning of PairPos subtable-ordered by Coverage Index

A PairSet table enumerates all the glyph pairs that begin with a covered glyph. An array of PairValueRecords (PairValueRecord) contains one record for each pair and lists the records sorted by the GlyphID of the second glyph in each pair. PairValueCount specifies the number of PairValueRecords in the set.

PairSet table

Value	Type	Description
uint16	PairValueCount	Number of PairValueRecords
struct	PairValueRecord [PairValueCount]	Array of PairValueRecords-ordered by GlyphID of the second glyph

A PairValueRecord specifies the second glyph in a pair (SecondGlyph) and defines a ValueRecord for each glyph (Value1 and Value2). If ValueFormat1 is set to zero (0) in the PairPos subtable, ValueRecord1 will be empty; similarly, if ValueFormat2 is 0, Value2 will be empty.

Example 4 at the end of this clause shows a PairPosFormat1 subtable that defines two cases of pair kerning.

PairValueRecord

Value	Type	Description
GlyphID	SecondGlyph	GlyphID of second glyph in the pair-first glyph is listed in the Coverage table
ValueRecord	Value1	Positioning data for the first glyph in the pair
ValueRecord	Value2	Positioning data for the second glyph in the pair

Pair positioning adjustment: Format 2

Format 2 defines a pair as a set of two glyph classes and modifies the positions of all the glyphs in a class. For example, this format is useful in Japanese scripts that apply specific kerning operations to all glyph pairs that contain punctuation glyphs. One class would be defined as all glyphs that may be coupled with punctuation marks, and the other classes would be groups of similar punctuation glyphs.

The PairPos Format2 subtable begins with a format identifier (PosFormat) and an Offset to a Coverage table (Coverage), measured from the beginning of the PairPos subtable. The Coverage table lists the indices of the first glyphs that may appear in each glyph pair. More than one pair may begin with the same glyph, but the Coverage table lists the glyph index only once.

A PairPosFormat2 subtable also includes two ValueFormats:

- ValueFormat1 applies to the ValueRecord of the first glyph in each pair. ValueRecords for all first glyphs must use ValueFormat1. If ValueFormat1 is set to zero (0), the corresponding glyph has no ValueRecord and, therefore, should not be repositioned.
- ValueFormat2 applies to the ValueRecord of the second glyph in each pair. ValueRecords for all second glyphs must use ValueFormat2. If ValueFormat2 is set to null, then the second glyph of the pair is the "next" glyph for which a lookup should be performed.

PairPosFormat2 requires that each glyph in all pairs be assigned to a class, which is identified by an integer called a class value. (For details about classes, see subclause 6.2, OFF Common Table Formats.) Pairs are then represented in a two-dimensional array as sequences of two class values. Multiple pairs can be represented in one Format 2 subtable.

A PairPosFormat2 subtable contains Offsets to two class definition tables: one that assigns class values to all the first glyphs in all pairs (ClassDef1), and one that assigns class values to all the second glyphs in all pairs (ClassDef2). If both glyphs in a pair use the same class definition, the Offset value can be the same for ClassDef1 and ClassDef2, but they are not required to be the same. The subtable also specifies the number of glyph classes defined in ClassDef1 (Class1Count) and in ClassDef2 (Class2Count), including Class0.

For each class identified in the ClassDef1 table, a Class1Record enumerates all pairs that contain a particular class as a first component. The Class1Record array stores all Class1Records according to class value.

NOTE Class1Records are not tagged with a class value identifier. Instead, the index value of a Class1Record in the array defines the class value represented by the record. For example, the first Class1Record enumerates pairs that begin with a Class 0 glyph, the second Class1Record enumerates pairs that begin with a Class1 glyph, and so on.

PairPosFormat2 subtable: Class pair adjustment

Value	Type	Description
uint16	PosFormat	Format identifier-format = 2
Offset	Coverage	Offset to Coverage table-from beginning of PairPos subtable-for the first glyph of the pair
uint16	ValueFormat1	ValueRecord definition-for the first glyph of the pair-may be zero (0)
uint16	ValueFormat2	ValueRecord definition-for the second glyph of the pair-may be zero (0)
Offset	ClassDef1	Offset to ClassDef table-from beginning of PairPos subtable-for the first glyph of the pair
Offset	ClassDef2	Offset to ClassDef table-from beginning of PairPos subtable-for the second glyph of the pair
uint16	Class1Count	Number of classes in ClassDef1 table-includes Class0
uint16	Class2Count	Number of classes in ClassDef2 table-includes Class0
struct	Class1Record [Class1Count]	Array of Class1 records-ordered by Class1

Each Class1Record contains an array of Class2Records (Class2Record), which also are ordered by class value. One Class2Record must be declared for each class in the ClassDef2 table, including Class 0.

Class1Record

Value	Type	Description
struct	Class2Record[Class2Count]	Array of Class2 records-ordered by Class2

A Class2Record consists of two ValueRecords, one for the first glyph in a class pair (Value1) and one for the second glyph (Value2). If the PairPos subtable has a value of zero (0) for ValueFormat1 or ValueFormat2, the corresponding record (ValueRecord1 or ValueRecord2) will be empty.

Example 5 at the end of this clause demonstrates pair kerning with glyph classes in a PairPosFormat2 subtable.

Class2Record

Value	Type	Description
ValueRecord	Value1	Positioning for first glyph-empty if ValueFormat1 = 0
ValueRecord	Value2	Positioning for second glyph-empty if ValueFormat2 = 0

Lookup Type 3: Cursive attachment positioning subtable

Some cursive fonts are designed so that adjacent glyphs join when rendered with their default positioning. However, if positioning adjustments are needed to join the glyphs, a cursive attachment positioning (CursivePos) subtable can describe how to connect the glyphs by aligning two anchor points: the designated exit point of a glyph, and the designated entry point of the following glyph.

The subtable has one format: CursivePosFormat1. It begins with a format identifier (PosFormat) and an Offset to a Coverage table (Coverage), which lists all the glyphs that define cursive attachment data.

In addition, the subtable contains one EntryExitRecord for each glyph listed in the Coverage table, a count of those records (EntryExitCount), and an array of those records in the same order as the Coverage Index (EntryExitRecord).

CursivePosFormat1 subtable: Cursive attachment

Value	Type	Description
uint16	PosFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of CursivePos subtable
uint16	EntryExitCount	Number of EntryExit records
struct	EntryExitRecord[EntryExitCount]	Array of EntryExit records-in Coverage Index order

Each EntryExitRecord consists of two Offsets: one to an Anchor table that identifies the entry point on the glyph (EntryAnchor), and an Offset to an Anchor table that identifies the exit point on the glyph (ExitAnchor). (For a complete description of the Anchor table, see the end of this clause.)

To position glyphs using the CursivePosFormat1 subtable, a text-processing client aligns the ExitAnchor point of a glyph with the EntryAnchor point of the following glyph. If no corresponding anchor point exists, either the EntryAnchor or ExitAnchor Offset may be NULL.

At the end of this clause, Example 6 describes cursive glyph attachment in the Urdu language.

EntryExitRecord

Value	Type	Description
Offset	EntryAnchor	Offset to EntryAnchor table-from beginning of CursivePos subtable-may be NULL
Offset	ExitAnchor	Offset to ExitAnchor table-from beginning of CursivePos subtable-may be NULL

Lookup Type 4: MarkToBase attachment positioning subtable

The MarkToBase attachment (MarkBasePos) subtable is used to position combining mark glyphs with respect to base glyphs. For example, the Arabic, Hebrew, and Thai scripts combine vowels, diacritical marks, and tone marks with base glyphs.

In the MarkBasePos subtable, every mark glyph has an anchor point and is associated with a class of marks. Each base glyph then defines an anchor point for each class of marks it uses.

For example, assume two mark classes: all marks positioned above base glyphs (Class 0), and all marks positioned below base glyphs (Class 1). In this case, each base glyph that uses these marks would define two anchor points, one for attaching the mark glyphs listed in Class 0, and one for attaching the mark glyphs listed in Class 1.

To identify the base glyph that combines with a mark, the text-processing client must look backward in the glyph string from the mark to the preceding base glyph. To combine the mark and base glyph, the client aligns their attachment points, positioning the mark with respect to the final pen point (advance) position of the base glyph.

The MarkToBase Attachment subtable has one format: MarkBasePosFormat1. The subtable begins with a format identifier (PosFormat) and Offsets to two Coverage tables: one that lists all the mark glyphs referenced in the subtable (MarkCoverage), and one that lists all the base glyphs referenced in the subtable (BaseCoverage).

For each mark glyph in the MarkCoverage table, a record specifies its class and an Offset to the Anchor table that describes the mark's attachment point (MarkRecord). A mark class is identified by a specific integer, called a class value. ClassCount specifies the total number of distinct mark classes defined in all the MarkRecords.

The MarkBasePosFormat1 subtable also contains an Offset to a MarkArray table, which contains all the MarkRecords stored in an array (MarkRecord) by MarkCoverage Index. A MarkArray table also contains a count of the defined MarkRecords (MarkCount). (For details about MarkArrays and MarkRecords, see the end of this clause.)

The MarkBasePosFormat1 subtable also contains an Offset to a BaseArray table (BaseArray).

MarkBasePosFormat1 subtable: MarkToBase attachment point

Value	Type	Description
uint16	PosFormat	Format identifier-format = 1
Offset	MarkCoverage	Offset to MarkCoverage table-from beginning of MarkBasePos subtable
Offset	BaseCoverage	Offset to BaseCoverage table-from beginning of MarkBasePos subtable
uint16	ClassCount	Number of classes defined for marks
Offset	MarkArray	Offset to MarkArray table-from beginning of MarkBasePos subtable
Offset	BaseArray	Offset to BaseArray table-from beginning of MarkBasePos subtable

The BaseArray table consists of an array (BaseRecord) and count (BaseCount) of BaseRecords. The array stores the BaseRecords in the same order as the BaseCoverage Index. Each base glyph in the BaseCoverage table has a BaseRecord.

BaseArray table

Value	Type	Description
uint16	BaseCount	Number of BaseRecords
struct	BaseRecord[BaseCount]	Array of BaseRecords-in order of BaseCoverage Index

A BaseRecord declares one Anchor table for each mark class (including Class 0) identified in the MarkRecords of the MarkArray. Each Anchor table specifies one attachment point used to attach all the marks in a particular class to the base glyph. A BaseRecord contains an array of Offsets to Anchor tables (BaseAnchor). The zero-based array of Offsets defines the entire set of attachment points each base glyph uses to attach marks. The Offsets to Anchor tables are ordered by mark class.

NOTE Anchor tables are not tagged with class value identifiers. Instead, the index value of an Anchor table in the array defines the class value represented by the Anchor table.

Example 7 at the end of this clause defines mark positioning above and below base glyphs with a MarkBasePosFormat1 subtable.

BaseRecord

Value	Type	Description
Offset	BaseAnchor[ClassCount]	Array of Offsets (one per class) to Anchor tables-from beginning of BaseArray table-ordered by class-zero-based

Lookup Type 5: MarkToLigature attachment positioning subtable

The MarkToLigature attachment (MarkLigPos) subtable is used to position combining mark glyphs with respect to ligature base glyphs. With MarkToBase attachment, described previously, a single base glyph defines an attachment point for each class of marks. In contrast, MarkToLigature attachment describes ligature glyphs composed of several components that can each define an attachment point for each class of marks.

As a result, a ligature glyph may have multiple base attachment points for one class of marks. The specific attachment point for a mark is defined by the ligature component that the subtable associates with the mark.

The MarkLigPos subtable can be used to define multiple mark-to-ligature attachments. In the subtable, every mark glyph has an anchor point and is associated with a class of marks. Every ligature glyph specifies a two-dimensional array of data: each component in a ligature defines an array of anchor points, one for each class of marks.

For example, assume two mark classes: all marks positioned above base glyphs (Class 0), and all marks positioned below base glyphs (Class 1). In this case, each component of a base ligature glyph may define two anchor points, one for attaching the mark glyphs listed in Class 0, and one for attaching the mark glyphs listed in Class 1. Alternatively, if the language system does not allow marks on the second component, the first ligature component may define two anchor points, one for each class of marks, and the second ligature component may define no anchor points.

To position a combining mark using a MarkToLigature attachment subtable, the text-processing client must work backward from the mark to the preceding ligature glyph. To correctly access the subtables, the client must keep track of the component associated with the mark. Aligning the attachment points combines the mark and ligature.

The MarkToLigature attachment subtable has one format: MarkLigPosFormat1. The subtable begins with a format identifier (PosFormat) and Offsets to two Coverage tables that list all the mark glyphs (MarkCoverage) and Ligature glyphs (LigatureCoverage) referenced in the subtable.

For each glyph in the MarkCoverage table, a MarkRecord specifies its class and an Offset to the Anchor table that describes the mark's attachment point. A mark class is identified by a specific integer, called a class value. ClassCount records the total number of distinct mark classes defined in all MarkRecords.

The MarkBasePosFormat1 subtable contains an Offset, measured from the beginning of the subtable, to a MarkArray table, which contains all MarkRecords stored in an array (MarkRecord) by MarkCoverage Index. (For details about MarkArrays and MarkRecords, see the end of this clause.)

The MarkLigPosFormat1 subtable also contains an Offset to a LigatureArray table (LigatureArray).

MarkLigPosFormat1 subtable: MarkToLigature attachment

Value	Type	Description
uint16	PosFormat	Format identifier-format = 1
Offset	MarkCoverage	Offset to Mark Coverage table-from beginning of MarkLigPos subtable
Offset	LigatureCoverage	Offset to Ligature Coverage table-from beginning of MarkLigPos subtable
uint16	ClassCount	Number of defined mark classes
Offset	MarkArray	Offset to MarkArray table-from beginning of MarkLigPos subtable
Offset	LigatureArray	Offset to LigatureArray table-from beginning of MarkLigPos subtable

The LigatureArray table contains a count (LigatureCount) and an array of Offsets (LigatureAttach) to LigatureAttach tables. The LigatureAttach array lists the Offsets to

LigatureAttach tables, one for each ligature glyph listed in the LigatureCoverage table, in the same order as the LigatureCoverage Index.

LigatureArray table

Value	Type	Description
uint16	LigatureCount	Number of LigatureAttach table Offsets
Offset	LigatureAttach [LigatureCount]	Array of Offsets to LigatureAttach tables-from beginning of LigatureArray table-ordered by LigatureCoverage Index

Each LigatureAttach table consists of an array (ComponentRecord) and count (ComponentCount) of the component glyphs in a ligature. The array stores the ComponentRecords in the same order as the components in the ligature. The order of the records also corresponds to the writing direction of the text. For text written left to right, the first component is on the left; for text written right to left, the first component is on the right.

LigatureAttach table

Value	Type	Description
uint16	ComponentCount	Number of ComponentRecords in this ligature
struct	ComponentRecord[ComponentCount]	Array of Component records-ordered in writing direction

A ComponentRecord, one for each component in the ligature, contains an array of Offsets to the Anchor tables that define all the attachment points used to attach marks to the component (LigatureAnchor). For each mark class (including Class 0) identified in the MarkArray records, an Anchor table specifies the point used to attach all the marks in a particular class to the ligature base glyph, relative to the component.

In a ComponentRecord, the zero-based LigatureAnchor array lists Offsets to Anchor tables by mark class. If a component does not define an attachment point for a particular class of marks, then the Offset to the corresponding Anchor table will be NULL.

Example 8 at the end of this clause shows a MarkLisPosFormat1 subtable used to attach mark accents to a ligature glyph in the Arabic script.

ComponentRecord

Value	Type	Description
Offset	LigatureAnchor [ClassCount]	Array of Offsets (one per class) to Anchor tables-from beginning of LigatureAttach table-ordered by class-NULL if a component does not have an attachment for a class-zero-based array

Lookup Type 6: MarkToMark attachment positioning subtable

The MarkToMark attachment (MarkMarkPos) subtable is identical in form to the MarkToBase attachment subtable, although its function is different. MarkToMark attachment defines the position of one mark relative to another mark as when, for example, positioning tone marks with respect to vowel diacritical marks in Vietnamese.

The attaching mark is Mark1, and the base mark being attached to is Mark2. In the MarkMarkPos subtable, every Mark1 glyph has an anchor attachment point and is associated with a class of marks. Each Mark2 glyph defines an anchor point for each class of marks. For example, assume two Mark1 classes: all marks positioned to the left of Mark2 glyphs (Class 0), and all marks positioned to the right of Mark2 glyphs (Class 1). Each Mark2 glyph that uses these marks defines two anchor points: one for attaching the Mark1 glyphs listed in Class 0, and one for attaching the Mark1 glyphs listed in Class 1.

The Mark2 glyph that combines with a Mark1 glyph is the glyph preceding the Mark1 glyph in glyph string order (skipping glyphs according to LookupFlags). The subtable applies precisely when that Mark2 glyph is covered by Mark2Coverage. To combine the mark glyphs, the Mark1 glyph is moved such that the relevant attachment points coincide. The input context for MarkToBase, MarkToLigature and MarkToMark positioning tables is the mark that is being positioned. If a sequence contains several marks, a lookup may act on it several times, to position them.

The MarkToMark attachment subtable has one format: MarkMarkPosFormat1. The subtable begins with a format identifier (PosFormat) and Offsets to two Coverage tables: one that lists all the Mark1 glyphs referenced in the subtable (Mark1Coverage), and one that lists all the Mark2 glyphs referenced in the subtable (Mark2Coverage).

For each mark glyph in the Mark1Coverage table, a MarkRecord specifies its class and an Offset to the Anchor table that describes the mark's attachment point. A mark class is identified by a specific integer, called

a class value. (For details about classes, see subclause 6.2, OFF Common Table Formats.) ClassCount specifies the total number of distinct mark classes defined in all the MarkRecords.

The MarkMarkPosFormat1 subtable also contains two Offsets, measured from the beginning of the subtable, to two arrays:

- The MarkArray table contains all MarkRecords stored by Mark1Coverage Index in an array (MarkRecord). The MarkArray table also contains a count of the number of defined MarkRecords (MarkCount).
- The Mark2Array table consists of an array (Mark2Record) and count (Mark2Count) of Mark2Records.

For details about MarkArrays and MarkRecords, see the end of this clause.

MarkMarkPosFormat1 subtable: MarkToMark attachment

Value	Type	Description
uint16	PosFormat	Format identifier-format = 1
Offset	Mark1Coverage	Offset to Combining Mark Coverage table-from beginning of MarkMarkPos subtable
Offset	Mark2Coverage	Offset to Base Mark Coverage table-from beginning of MarkMarkPos subtable
uint16	ClassCount	Number of Combining Mark classes defined
Offset	Mark1Array	Offset to MarkArray table for Mark1-from beginning of MarkMarkPos subtable
Offset	Mark2Array	Offset to Mark2Array table for Mark2-from beginning of MarkMarkPos subtable

The Mark2Array, shown next, contains one Mark2Record for each Mark2 glyph listed in the Mark2Coverage table. It stores the records in the same order as the Mark2Coverage Index.

Mark2Array table

Value	Type	Description
uint16	Mark2Count	Number of Mark2 records
struct	Mark2Record [Mark2Count]	Array of Mark2 records-in Coverage order

Each Mark2Record contains an array of Offsets to Anchor tables (Mark2Anchor). The array of zero-based Offsets, measured from the beginning of the Mark2Array table, defines the entire set of Mark2 attachment points used to attach Mark1 glyphs to a specific Mark2 glyph. The Anchor tables in the Mark2Anchor array are ordered by Mark1 class value.

A Mark2Record declares one Anchor table for each mark class (including Class 0) identified in the MarkRecords of the MarkArray. Each Anchor table specifies one Mark2 attachment point used to attach all the Mark1 glyphs in a particular class to the Mark2 glyph.

Example 9 at the end of the clause shows a MarkMarkPosFormat1 subtable for attaching one mark to another in the Arabic script.

Mark2Record

Value	Type	Description
Offset	Mark2Anchor [ClassCount]	Array of Offsets (one per class) to Anchor tables-from beginning of Mark2Array table-zero-based array

Lookup Type 7: Contextual positioning subtables

A Contextual Positioning (ContextPos) subtable defines the most powerful type of glyph positioning lookup. It describes glyph positioning in context so a text-processing client can adjust the position of one or more glyphs within a certain pattern of glyphs. Each subtable describes one or more "input" glyph sequences and one or more positioning operations to be performed on that sequence.

ContextPos subtables can have one of three formats, which closely mirror the formats used for contextual glyph substitution. One format applies to specific glyph sequences (Format 1), one defines the context in terms of glyph classes (Format 2), and the third format defines the context in terms of sets of glyphs (Format 3).

All three formats of ContextPos subtables specify positioning data in a PosLookupRecord. A description of that record follows.

PosLookupRecord

All contextual positioning subtables specify the positioning data in a PosLookupRecord. Each record contains a SequenceIndex, which indicates where the positioning operation will occur in the glyph sequence. In addition, a LookupListIndex identifies the lookup to be applied at the glyph position specified by the SequenceIndex.

The order in which lookups are applied to the entire glyph sequence, called the "design order," can be significant, so PosLookupRecord data should be defined accordingly.

The contextual substitution subtables defined in Examples 10, 11, and 12 show PosLookupRecords.

PosLookupRecord

Value	Type	Description
uint16	SequenceIndex	Index to input glyph sequence-first glyph = 0
uint16	LookupListIndex	Lookup to apply to that position-zero-based

Context Positioning Subtable: Format 1

Format 1 defines the context for a glyph positioning operation as a particular sequence of glyphs. For example, a context could be <To>, <xyzabc>, <!?*#@>, or any other glyph sequence.

Within the context, Format 1 identifies particular glyph positions (not glyph indices) as the targets for specific adjustments. When a text-processing client locates a context in a string of glyphs, it makes the adjustment by applying the lookup data defined for a targeted position at that location.

For example, suppose that accent mark glyphs above lowercase x-height vowel glyphs must be lowered when an overhanging capital letter glyph precedes the vowel. When the client locates this context in the text, the subtable identifies the position of the accent mark and a lookup index. A lookup specifies a positioning action that lowers the accent mark over the vowel so that it does not collide with the overhanging capital.

ContextPosFormat1 defines the context in two places. A Coverage table specifies the first glyph in the input sequence, and a PosRule table identifies the remaining glyphs. To describe the context used in the previous example, the Coverage table lists the glyph index of the first component of the sequence (the overhanging capital), and a PosRule table defines indices for the lowercase x-height vowel glyph and the accent mark.

A single ContextPosFormat1 subtable may define more than one context glyph sequence. If different context sequences begin with the same glyph, then the Coverage table should list the glyph only once because all first glyphs in the table must be unique. For example, if three contexts each start with an "s" and two start with a "t," then the Coverage table will list one "s" and one "t."

For each context, a PosRule table lists all the glyphs, in order, that follow the first glyph. The table also contains an array of PosLookupRecords that specify the positioning lookup data for each glyph position (including the first glyph position) in the context.

All the PosRule tables defining contexts that begin with the same first glyph are grouped together and defined in a PosRuleSet table. For example, the PosRule tables that define the three contexts that begin with an "s" are grouped in one PosRuleSet table, and the PosRule tables that define the two contexts that begin with a "t" are grouped in a second PosRuleSet table. Each unique glyph listed in the Coverage table must have a PosRuleSet table that defines all the PosRule tables for a covered glyph.

To locate a context glyph sequence, the text-processing client searches the Coverage table each time it encounters a new text glyph. If the glyph is covered, the client reads the corresponding PosRuleSet table and examines each PosRule table in the set to determine whether the rest of the context defined there matches the subsequent glyphs in the text. If the context and text string match, the client finds the target glyph position, applies the lookup for that position, and completes the positioning action.

A ContextPosFormat1 subtable contains a format identifier (PosFormat), an Offset to a Coverage table (Coverage), a count of the number of PosRuleSets that are defined (PosRuleSetCount), and an array of Offsets to the PosRuleSet tables (PosRuleSet). As mentioned, one PosRuleSet table must be defined for each glyph listed in the Coverage table.

In the PosRuleSet array, the PosRuleSet tables are ordered in the Coverage Index order. The first PosRuleSet in the array applies to the first GlyphID listed in the Coverage table, the second PosRuleSet in the array applies to the second GlyphID listed in the Coverage table, and so on.

ContextPosFormat1 subtable: Simple context positioning

Value	Type	Description
uint16	PosFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of ContextPos subtable
uint16	PosRuleSetCount	Number of PosRuleSet tables
Offset	PosRuleSet [PosRuleSetCount]	Array of Offsets to PosRuleSet tables-from beginning of ContextPos subtable-ordered by Coverage Index

A PosRuleSet table consists of an array of Offsets to PosRule tables (PosRule), ordered by preference, and a count of the PosRule tables defined in the set (PosRuleCount).

PosRuleSet table: All contexts beginning with the same glyph

Value	Type	Description
uint16	PosRuleCount	Number of PosRule tables
Offset	PosRule [PosRuleCount]	Array of Offsets to PosRule tables-from beginning of PosRuleSet-ordered by preference

A PosRule table consists of a count of the glyphs to be matched in the input context sequence (GlyphCount), including the first glyph in the sequence, and an array of glyph indices that describe the context (Input). The Coverage table specifies the index of the first glyph in the context, and the Input array begins with the second glyph in the context sequence. As a result, the first index position in the array is specified with the number one (1), not zero (0). The Input array lists the indices in the order the corresponding glyphs appear in the text. For text written from right to left, the right-most glyph will be first; conversely, for text written from left to right, the left-most glyph will be first.

A PosRule table also contains a count of the positioning operations to be performed on the input glyph sequence (PosCount) and an array of PosLookupRecords (PosLookupRecord). Each record specifies a position in the input glyph sequence and a LookupList index to the positioning lookup to be applied there. The array should list records in design order, or the order the lookups should be applied to the entire glyph sequence.

Example 10 at the end of this clause demonstrates glyph kerning in context with a ContextPosFormat1 subtable.

PosRule subtable

Value	Type	Description
uint16	GlyphCount	Number of glyphs in the Input glyph sequence
uint16	PosCount	Number of PosLookupRecords
GlyphID	Input [GlyphCount - 1]	Array of input GlyphIDs-starting with the second glyph
struct	PosLookupRecord[PosCount]	Array of positioning lookups-in design order

Context positioning subtable: Format 2

Format 2, more flexible than Format 1, describes class-based context positioning. For this format, a specific integer, called a class value, must be assigned to each glyph in all context glyph sequences. Contexts are then defined as sequences of class values. This subtable may define more than one context.

To clarify the notion of class-based context rules, suppose that certain sequences of three glyphs need special kerning. The glyph sequences consist of an uppercase glyph that overhangs on the right side, a punctuation mark glyph, and then a quote glyph. In this case, the set of uppercase glyphs would constitute one glyph class (Class1), the set of punctuation mark glyphs would constitute a second glyph class (Class 2), and the set of quote mark glyphs would constitute a third glyph class (Class 3). The input context might be specified with a context rule (PosClassRule) that describes "the set of glyph strings that form a sequence of three glyph classes, one glyph from Class 1, followed by one glyph from Class 2, followed by one glyph from Class 3."

Each ContextPosFormat2 subtable contains an Offset to a class definition table (ClassDef), which defines the class values of all glyphs in the input contexts that the subtable describes. Generally, a unique ClassDef will be declared in each instance of the ContextPosFormat2 subtable that is included in a font, even though several Format 2 subtables may share ClassDef tables. Classes are exclusive sets; a glyph cannot be in more than one class at a time. The output glyphs that replace the glyphs in the context sequence do not need class values because they are specified elsewhere by GlyphID.

The ContextPosFormat2 subtable also contains a format identifier (PosFormat) and defines an Offset to a Coverage table (Coverage). For this format, the Coverage table lists indices for the complete set of glyphs (not glyph classes) that may appear as the first glyph of any class-based context. In other words, the Coverage table contains the list of glyph indices for all the glyphs in all classes that may be first in any of the context class sequences. For example, if the contexts begin with a Class 1 or Class 2 glyph, then the Coverage table will list the indices of all Class 1 and Class 2 glyphs.

A ContextPosFormat2 subtable also defines an array of Offsets to the PosClassSet tables (PosClassSet), along with a count (including Class0) of the PosClassSet tables (PosClassSetCnt). In the array, the PosClassSet tables are ordered by ascending class value (from 0 to PosClassSetCnt - 1).

A PosClassSet array contains one Offset for each glyph class, including Class 0. PosClassSets are not explicitly tagged with a class value; rather, the index value of the PosClassSet in the PosClassSet array defines the class that a PosClassSet represents.

For example, the first PosClassSet listed in the array contains all the PosClassRules that define contexts beginning with Class 0 glyphs, the second PosClassSet contains all PosClassRules that define contexts beginning with Class 1 glyphs, and so on. If no PosClassRules begin with a particular class (that is, if a PosClassSet contains no PosClassRules), then the Offset to that particular PosClassSet in the PosClassSet array will be set to NULL.

ContextPosFormat2 subtable: Class-based context glyph positioning

Value	Type	Description
uint16	PosFormat	Format identifier-format = 2
Offset	Coverage	Offset to Coverage table-from beginning of ContextPos subtable
Offset	ClassDef	Offset to ClassDef table-from beginning of ContextPos subtable
uint16	PosClassSetCnt	Number of PosClassSet tables
Offset	PosClassSet [PosClassSetCnt]	Array of Offsets to PosClassSet tables-from beginning of ContextPos subtable-ordered by class-may be NULL

All the PosClassRules that define contexts beginning with the same class are grouped together and defined in a PosClassSet table. Consequently, the PosClassSet table identifies the class of a context's first component.

A PosClassSet enumerates all the PosClassRules that begin with a particular glyph class. For instance, PosClassSet0 represents all the PosClassRules that describe contexts starting with Class 0 glyphs, and PosClassSet1 represents all the PosClassRules that define contexts starting with Class 1 glyphs.

Each PosClassSet table consists of a count of the PosClassRules defined in the PosClassSet (PosClassRuleCnt) and an array of Offsets to PosClassRule tables (PosClassRule). The PosClassRule tables are ordered by preference in the PosClassRule array of the PosClassSet.

PosClassSet table: All contexts beginning with the same class

Value	Type	Description
uint16	PosClassRuleCnt	Number of PosClassRule tables
Offset	PosClassRule[PosClassRuleCnt]	Array of Offsets to PosClassRule tables-from beginning of PosClassSet-ordered by preference

For each context, a PosClassRule table contains a count of the glyph classes in a given context (GlyphCount), including the first class in the context sequence. A class array lists the classes, beginning with the second class, that follow the first class in the context. The first class listed indicates the second position in the context sequence.

NOTE Text order depends on the writing direction of the text. For text written from right to left, the right-most glyph will be first. Conversely, for text written from left to right, the left-most glyph will be first.

The values specified in the Class array are those defined in the ClassDef table. For example, consider a context consisting of the sequence: Class 2, Class 7, Class 5, Class 0. The Class array will read: Class[0] = 7, Class[1] = 5, and Class[2] = 0. The first class in the sequence, Class 2, is defined by the index into the PosClassSet array of Offsets. The total number and sequence of glyph classes listed in the Class array must match the total number and sequence of glyph classes contained in the input context.

A PosClassRule also contains a count of the positioning operations to be performed on the context (PosCount) and an array of PosLookupRecords (PosLookupRecord) that supply the positioning data. For each position in the context that requires a positioning operation, a PosLookupRecord specifies a LookupList index and a position in the input glyph class sequence where the lookup is applied. The PosLookupRecord array lists PosLookupRecords in design order, or the order in which lookups are applied to the entire glyph sequence.

Example 11 at the end of this clause demonstrates a ContextPosFormat2 subtable that uses glyph classes to modify accent positions in glyph strings.

PosClassRule table: One class context definition

Value	Type	Description
uint16	GlyphCount	Number of glyphs to be matched
uint16	PosCount	Number of PosLookupRecords
uint16	Class [GlyphCount - 1]	Array of classes-beginning with the second class-to be matched to the input glyph sequence
struct	PosLookupRecord[PosCount]	Array of positioning lookups in design order

Context positioning subtable: Format 3

Format 3, coverage-based context positioning, defines a context rule as a sequence of coverages. Each position in the sequence may specify a different Coverage table for the set of glyphs that matches the context pattern. With Format 3, the glyph sets defined in the different Coverage tables may intersect, unlike Format 2 which specifies fixed class assignments for the lookup (they cannot be changed at each position in the context sequence) and exclusive classes (a glyph cannot be in more than one class at a time).

For example, consider an input context that contains an uppercase glyph (position 0), followed by any narrow uppercase glyph (position 1), and then another uppercase glyph (position 2). This context requires three Coverage tables, one for each position:

- In position 0, the first position, the Coverage table lists the set of all uppercase glyphs.
- In position 1, the second position, the Coverage table lists the set of all narrow uppercase glyphs, which is a subset of the glyphs listed in the Coverage table for position 0.
- In position 2, the Coverage table lists the set of all uppercase glyphs again.

NOTE Both position 0 and position 2 can use the same Coverage table.

Unlike Formats 1 and 2, this format defines only one context rule at a time. It consists of a format identifier (PosFormat), a count of the number of glyphs in the sequence to be matched (GlyphCount), and an array of Coverage Offsets that describe the input context sequence (Coverage).

NOTE The Coverage tables listed in the Coverage array must be listed in text order according to the writing direction. For text written from right to left, the right-most glyph will be first. Conversely, for text written from left to right, the left-most glyph will be first.

The subtable also contains a count of the positioning operations to be performed on the input Coverage sequence (PosCount) and an array of PosLookupRecords (PosLookupRecord) in design order, or the order in which lookups are applied to the entire glyph sequence.

Example 12 at the end of this clause changes the positions of math sign glyphs in math equations with a ContextPosFormat3 subtable.

ContextPosFormat3 subtable: Coverage-based context glyph positioning

Value	Type	Description
uint16	PosFormat	Format identifier-format = 3
uint16	GlyphCount	Number of glyphs in the input sequence
uint16	PosCount	Number of PosLookupRecords
Offset	Coverage [GlyphCount]	Array of Offsets to Coverage tables-from beginning of ContextPos subtable
struct	PosLookupRecord [PosCount]	Array of positioning lookups-in design order

LookupType 8: Chaining contextual positioning subtable

A Chaining Contextual Positioning subtable(ChainContextPos)describes glyph positioning in context with an ability to look back and/or look ahead in the sequence of glyphs. The design of the Chaining Contextual Positioning subtable is parallel to that of the Contextual Positioning subtable, including the availability of three formats.

To specify the context, the coverage table lists the first glyph in the input sequence, and the ChainPosRule subtable defines the rest. Once a covered glyph is found at position *i*, the client reads the corresponding ChainPosRuleSet table and examines each table to determine if it matches the surrounding glyphs in the text. There is a match if the string <backtrack sequence>+<input sequence>+<lookahead sequence> matches with the glyphs at position *i* - *BacktrackGlyphCount* in the text.

If there is a match, then the client finds the target glyphs for positioning and performs the operations. Please note that (just like in the ContextPosFormat1 subtable) these lookups are required to operate within the range of text from the covered glyph to the end of the input sequence. No positioning operations can be defined for the backtracking sequence or the lookahead sequence.

To clarify the ordering of glyph arrays for input, backtrack and lookahead sequences, the following illustration is provided. Input sequence match begins at *i* where the input sequence match begins. The backtrack sequence is ordered beginning at *i* - 1 and increases in Offset value as one moves away from *i*. The lookahead sequence begins after the input sequence and increases in logical order.

Logical order - a b c d e f g h i j

i

Input sequence - 0 1

Backtrack sequence - 3 2 1 0

Lookahead sequence - 0 1 2 3

Chaining context positioning Format 1: Simple chaining context glyph positioning

This Format is identical to Format 1 of Context Positioning lookup except that the PosRule table is replaced with a ChainPosRule table. (Correspondingly, the ChainPosRuleSet table differs from the PosRuleSet table only in that it lists Offsets to ChainPosRule subtables instead of PosRule tables; and the ChainContextPosFormat1 subtable lists Offsets to ChainPosRuleSet subtables instead of PosRuleSet subtables.)

ChainContextPosFormat1 subtable: Simple context positioning

Value	Type	Description
uint16	PosFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of ContextPos subtable
uint16	ChainPosRuleSetCount	Number of ChainPosRuleSet tables
Offset	ChainPosRuleSet [ChainPosRuleSetCount]	Array of Offsets to ChainPosRuleSet tables-from beginning of ContextPos subtable-ordered by Coverage Index

A ChainPosRuleSet table consists of an array of Offsets to ChainPosRule tables (ChainPosRule), ordered by preference, and a count of the ChainPosRule tables defined in the set (ChainPosRuleCount).

ChainPosRuleSet table: All contexts beginning with the same glyph

Value	Type	Description
uint16	ChainPosRuleCount	Number of ChainPosRule tables
Offset	ChainPosRule [ChainPosRuleCount]	Array of Offsets to ChainPosRule tables-from beginning of ChainPosRuleSet-ordered by preference

ChainPosRule subtable

Type	Name	Description
uint16	BacktrackGlyphCount	Total number of glyphs in the backtrack sequence (number of glyphs to be matched before the first glyph)
GlyphID	Backtrack [BacktrackGlyphCount]	Array of backtracking GlyphID's (to be matched before the input sequence)
uint16	InputGlyphCount	Total number of glyphs in the input sequence (includes the first glyph)
GlyphID	Input [InputGlyphCount - 1]	Array of input GlyphIDs (start with second glyph)
uint16	LookaheadGlyphCount	Total number of glyphs in the look ahead sequence (number of glyphs to be matched after the input sequence)

GlyphID	LookAhead [LookAheadGlyphCount]	Array of lookahead GlyphID's (to be matched after the input sequence)
uint16	PosCount	Number of PosLookupRecords
struct	PosLookupRecord [PosCount]	Array of PosLookupRecords (in design order)

Chaining context positioning Format 2: Class-based chaining context glyph positioning

This lookup Format is parallel to the Context Positioning format 2, with PosClassSet subtable changed to ChainPosClassSet subtable, and PosClassRule subtable changed to ChainPosClassRule subtable.

To chain contexts, three classes are used in the glyph ClassDef table: Backtrack ClassDef, Input ClassDef, and Lookahead ClassDef.

ChainContextPosFormat2 subtable: Chaining class-based context glyph positioning

Value	Type	Description
uint16	PosFormat	Format identifier-format = 2
Offset	Coverage	Offset to Coverage table-from beginning of ChainContextPos subtable
Offset	BacktrackClassDef	Offset to ClassDef table containing backtrack sequence context-from beginning of ChainContextPos subtable
Offset	InputClassDef	Offset to ClassDef table containing input sequence context-from beginning of ChainContextPos subtable
Offset	LookaheadClassDef	Offset to ClassDef table containing lookahead sequence context-from beginning of ChainContextPos subtable
uint16	ChainPosClassSetCnt	Number of ChainPosClassSet tables
Offset	ChainPosClassSet [ChainPosClassSetCnt]	Array of Offsets to ChainPosClassSet tables-from beginning of ChainContextPos subtable-ordered by input class-may be NULL

All the ChainPosClassRules that define contexts beginning with the same class are grouped together and defined in a ChainPosClassSet table. Consequently, the ChainPosClassSet table identifies the class of a context's first component.

ChainPosClassSet table: All contexts beginning with the same class

Value	Type	Description
uint16	ChainPosClassRuleCnt	Number of ChainPosClassRule tables
Offset	ChainPosClassRule[ChainPosClassRuleCnt]	Array of Offsets to ChainPosClassRule tables-from beginning of ChainPosClassSet-ordered by preference

ChainPosClassRule subtable

Type	Name	Description
uint16	BacktrackGlyphCount	Total number of glyphs in the backtrack sequence (number of glyphs to be matched before the first glyph)
uint16	Backtrack [BacktrackGlyphCount]	Array of backtracking classes(to be matched before the input sequence)
uint16	InputGlyphCount	Total number of classes in the input sequence (includes the first class)
uint16	Input [InputGlyphCount - 1]	Array of input classes(start with second class; to be matched with the input glyph sequence)
uint16	LookaheadGlyphCount	Total number of classes in the look ahead sequence (number of classes to be matched after the input sequence)
uint16	LookAhead [LookAheadGlyphCount]	Array of lookahead classes(to be matched after the input sequence)
uint16	PosCount	Number of PosLookupRecords
struct	PosLookupRecord [ChainPosCount]	Array of PosLookupRecords (in design order)

Chaining context positioning Format 3: Coverage-based chaining context glyph positioning

Format 3 defines a chaining context rule as a sequence of Coverage tables. Each position in the sequence may define a different Coverage table for the set of glyphs that matches the context pattern. With Format 3, the glyph sets defined in the different Coverage tables may intersect, unlike Format 2 which specifies fixed class assignments (identical for each position in the backtrack, input, or lookahead sequence) and exclusive classes (a glyph cannot be in more than one class at a time).

NOTE The order of the Coverage tables listed in the Coverage array must follow the writing direction. For text written from right to left, then the right-most glyph will be first. Conversely, for text written from left to right, the left-most glyph will be first.

The subtable also contains a count of the positioning operations to be performed on the input Coverage sequence (PosCount) and an array of PosLookupRecords (PosLookupRecord) in design order: that is, the order in which lookups should be applied to the entire glyph sequence.

ChainContextPosFormat3 subtable: Coverage-based chaining context glyph positioning

Type	Name	Description
uint16	PosFormat	Format identifier-format = 3
uint16	BacktrackGlyphCount	Number of glyphs in the backtracking sequence
Offset	Coverage[BacktrackGlyphCount]	Array of Offsets to coverage tables in backtracking sequence, in glyph sequence order

uint16	InputGlyphCount	Number of glyphs in input sequence
Offset	Coverage[InputGlyphCount]	Array of Offsets to coverage tables in input sequence, in glyph sequence order
uint16	LookaheadGlyphCount	Number of glyphs in lookahead sequence
Offset	Coverage[LookaheadGlyphCount]	Array of Offsets to coverage tables in lookahead sequence, in glyph sequence order
uint16	PosCount	Number of PosLookupRecords
struct	PosLookupRecord [PosCount]	Array of PosLookupRecords,in design order

LookupType 9: Extension positioning

This lookup provides a mechanism whereby any other lookup type's subtables are stored at a 32-bit Offset location in the 'GPOS' table. This is needed if the total size of the subtables exceeds the 16-bit limits of the various other Offsets in the 'GPOS' table. In this specification, the subtable stored at the 32-bit Offset location is termed the "extension" subtable.

ExtensionPosFormat1 subtable

Type	Name	Description
USHORT	PosFormat	Format identifier: Set to 1.
USHORT	ExtensionLookupType	Lookup type of subtable referenced by ExtensionOffset (i.e. the extension subtable).
ULONG	ExtensionOffset	Offset to the extension subtable, of lookup type ExtensionLookupType, relative to the start of the ExtensionPosFormat1 subtable.

ExtensionLookupType must be set to any lookup type other than 9. All subtables in a LookupType 9 lookup must have the same ExtensionLookupType. All Offsets in the extension subtables are set in the usual way, i.e. relative to the extension subtables themselves.

When an OFF layout engine encounters a LookupType 9 Lookup table, it shall:

- Proceed as though the Lookup table's LookupType field were set to the ExtensionLookupType of the subtables.
- Proceed as though each extension subtable referenced by ExtensionOffset replaced the LookupType 9 subtable that referenced it.

6.3.3.4 Shared tables: Value record, Anchor table and Mark array

Several lookup subtables described earlier in this clause refer to one or more of the same tables for positioning data: ValueRecord, Anchor table, and MarkArray. For easy reference, those shared tables are described here.

Example 14 at the end of the clause uses a ValueFormat table and ValueRecord to specify positioning values in GPOS.

ValueRecord

GPOS subtables use ValueRecords to describe all the variables and values used to adjust the position of a glyph or set of glyphs. A ValueRecord may define any combination of X and Y values (in design units) to add to (positive values) or subtract from (negative values) the placement and advance values provided in the font. A ValueRecord also may contain an Offset to a Device table for each of the specified values. If a ValueRecord specifies more than one value, the values should be listed in the order shown in the ValueRecord definition.

The text-processing client must be aware of the flexible and multi-dimensional nature of ValueRecords in the GPOS table. Because the GPOS table uses ValueRecords for many purposes, the sizes and contents of ValueRecords may vary from subtable to subtable.

ValueRecord (all fields are optional)

Value	Type	Description
int16	XPlacement	Horizontal adjustment for placement-in design units
int16	YPlacement	Vertical adjustment for placement-in design units
int16	XAdvance	Horizontal adjustment for advance-in design units (only used for horizontal writing)
int16	YAdvance	Vertical adjustment for advance-in design units (only used for vertical writing)
OffsetOffset	XPlaDevice	OffsetOffset to Device table for horizontal placement-measured from beginning of PosTable (may be NULL)
OffsetOffset	YPlaDevice	OffsetOffset to Device table for vertical placement-measured from beginning of PosTable (may be NULL)
OffsetOffset	XAdvDevice	OffsetOffset to Device table for horizontal advance-measured from beginning of PosTable (may be NULL)
OffsetOffset	YAdvDevice	OffsetOffset to Device table for vertical advance-measured from beginning of PosTable (may be NULL)

A data format (ValueFormat), usually declared at the beginning of each GPOS subtable, defines the types of positioning adjustment data that ValueRecords specify. Usually, the same ValueFormat applies to every ValueRecord defined in the particular GPOS subtable.

The ValueFormat determines whether the ValueRecords:

- Apply to placement, advance, or both.
- Apply to the horizontal position (X coordinate), the vertical position (Y coordinate), or both.
- May refer to one or more Device tables for any of the specified values.

Each one-bit in the ValueFormat corresponds to a field in the ValueRecord and increases the size of the ValueRecord by 2 bytes. A ValueFormat of 0x0000 corresponds to an empty ValueRecord, which indicates no positioning changes.

To identify the fields in each ValueRecord, the ValueFormat uses the bit settings shown below. To specify multiple fields with a ValueFormat, the bit settings of the relevant fields are added with a logical OR operation.

For example, to adjust the left-side bearing of a glyph, the ValueFormat will be 0x0001, and the ValueRecord will define the XPlacement value. To adjust the advance width of a different glyph, the ValueFormat will be 0x0004, and the ValueRecord will describe the XAdvance value. To adjust both the XPlacement and XAdvance of a set of glyphs, the ValueFormat will be 0x0005, and the ValueRecord will specify both values in the order they are listed in the ValueRecord definition.

ValueFormat bit enumeration (indicates which fields are present)

Mask	Name	Description
0x0001	XPlacement	Includes horizontal adjustment for placement
0x0002	YPlacement	Includes vertical adjustment for placement
0x0004	XAdvance	Includes horizontal adjustment for advance
0x0008	YAdvance	Includes vertical adjustment for advance
0x0010	XPlaDevice	Includes horizontal Device table for placement
0x0020	YPlaDevice	Includes vertical Device table for placement
0x0040	XAdvDevice	Includes horizontal Device table for advance
0x0080	YAdvDevice	Includes vertical Device table for advance
0xF000	Reserved	For future use (must be set to zero)

Anchor table

A GPOS table uses anchor points to position one glyph with respect to another. Each glyph defines an anchor point, and the text-processing client attaches the glyphs by aligning their corresponding anchor points.

To describe an anchor point, an Anchor table can use one of three formats. The first format uses design units to specify a location for the anchor point. The other two formats refine the location of the anchor point using contour points (Format 2) or Device tables (Format 3).

Anchor table: Format 1

AnchorFormat1 consists of a format identifier (AnchorFormat) and a pair of design unit coordinates (XCoordinate and YCoordinate) that specify the location of the anchor point. This format has the benefits of small size and simplicity, but the anchor point cannot be hinted to adjust its position for different device resolutions.

Example 15 at the end of this clause uses AnchorFormat1.

AnchorFormat1 table: Design units only

Value	Type	Description
uint16	AnchorFormat	Format identifier-format = 1
int16	XCoordinate	Horizontal value-in design units
int16	YCoordinate	Vertical value-in design units

Anchor table: Format 2

Like AnchorFormat1, AnchorFormat2 specifies a format identifier (AnchorFormat) and a pair of design unit coordinates for the anchor point (Xcoordinate and Ycoordinate).

For fine-tuning the location of the anchor point, AnchorFormat2 also provides an index to a glyph contour point (AnchorPoint) that is on the outline of a glyph (AnchorPoint). Hinting can be used to move the AnchorPoint. In the rendered text, the AnchorPoint will provide the final positioning data for a given ppem size.

Example 16 at the end of this clause uses AnchorFormat2.

AnchorFormat2 table: Design units plus contour point

Value	Type	Description
uint16	AnchorFormat	Format identifier-format = 2
int16	XCoordinate	Horizontal value-in design units
int16	YCoordinate	Vertical value-in design units
uint16	AnchorPoint	Index to glyph contour point

Anchor table: Format 3

Like AnchorFormat1, AnchorFormat3 specifies a format identifier (AnchorFormat) and locates an anchor point (Xcoordinate and Ycoordinate). And, like AnchorFormat 2, it permits fine adjustments to the coordinate values. However, AnchorFormat3 uses Device tables, rather than a contour point, for this adjustment.

With a Device table, a client can adjust the position of the anchor point for any font size and device resolution. AnchorFormat3 can specify Offsets to Device tables for the the X coordinate (XDeviceTable) and the Y coordinate (YDeviceTable). If only one coordinate requires adjustment, the Offset to the Device table may be set to NULL for the other coordinate.

Example 17 at the end of the clause shows an AnchorFormat3 table.

AnchorFormat3 table: Design units plus Device tables

Value	Type	Description
uint16	AnchorFormat	Format identifier-format = 3
int16	XCoordinate	Horizontal value-in design units
int16	YCoordinate	Vertical value-in design units
Offset	XDeviceTable	Offset to Device table for X coordinate- from beginning of Anchor table (may be NULL)
Offset	YDeviceTable	Offset to Device table for Y coordinate- from beginning of Anchor table (may be NULL)

Mark array

The MarkArray table defines the class and the anchor point for a mark glyph. Three GPOS subtables-MarkToBase, MarkToLigature, and MarkToMark Attachment-use the MarkArray table to specify data for attaching marks.

The MarkArray table contains a count of the number of mark records (MarkCount) and an array of those records (MarkRecord). Each mark record defines the class of the mark and an Offset to the Anchor table that contains data for the mark.

A class value can be 0 (zero), but the MarkRecord must explicitly assign that class value (this differs from the ClassDef table, in which all glyphs not assigned class values automatically belong to Class 0). The GPOS subtables that refer to MarkArray tables use the class assignments for indexing zero-based arrays that contain data for each mark class.

In Example 18 at the end of the clause, a MarkArray table and two MarkRecords define two mark classes.

MarkArray table

Value	Type	Description
uint16	MarkCount	Number of MarkRecords
struct	MarkRecord [MarkCount]	Array of MarkRecords-in Coverage order

MarkRecord

Value	Type	Description
uint16	Class	Class defined for this mark
Offset	MarkAnchor	Offset to Anchor table-from beginning of MarkArray table

6.3.3.5 GPOS subtable examples

The rest of this clause describes examples of all the GPOS subtable formats, including each of the three formats available for contextual positioning. All the examples reflect unique parameters described below, but the samples provide a useful reference for building subtables specific to other situations.

All the examples have three columns showing hex data, source, and comments.

Example 1: GPOS header table

Example 1 shows a typical GPOS Header table definition with Offsets to a ScriptList, FeatureList, and LookupList.

Example 1

Hex Data	Source	Comments
	GPOSHeader TheGPOSHeader	GPOSHeader table definition
00010000	0x00010000	Version
000A	TheScriptList	Offset to ScriptList table
001E	TheFeatureList	Offset to FeatureList table
002C	TheLookupList	Offset to LookupList table

Example 2: SinglePosFormat1 subtable

Example 2 uses the SinglePosFormat1 subtable to lower the Y placement of subscript glyphs in a font. The LowerSubscriptsSubTable defines one Coverage table, called LowerSubscriptsCoverage, which lists one range of glyph indices for the numeral/numeric subscript glyphs. The subtable's ValueFormat setting indicates that the ValueRecord specifies only the YPlacement value, lowering each subscript glyph by 80 design units.

Example 2

Hex Data	Source	Comments
	SinglePosFormat1 LowerSubscriptsSubTable	SinglePos subtable definition
0001	1	PosFormat
0008	LowerSubscriptsCoverage	Offset to Coverage table
0002	0x0002	ValueFormat, YPlacement,Value[0], move Y position down
FFB0	-80	

	CoverageFormat2 LowerSubscriptsCoverage	Coverage table definition
0002	2	CoverageFormat
0001	1	RangeCount RangeRecord[0]
01B3	ZeroSubscriptGlyphID	Start, first glyphID
01BC	NineSubscriptGlyphID	End, last glyphID
0000	0	StartCoverageIndex

Example 3: SinglePosFormat2 subtable

This example uses a SinglePosFormat2 subtable to adjust the spacing of three dash glyphs by different amounts. The em dash spacing changes by 10 units, the en dash spacing changes by 25 units, and spacing of the standard dash changes by 50 units.

The DashSpacingSubTable contains one Coverage table with three dash glyph indices, plus an array of ValueRecords, one for each covered glyph. The ValueRecords use the same ValueFormat to modify the XPlacement and XAdvance values of each glyph. The ValueFormat bit setting of 0x0005 is produced by adding the XPlacement and XAdvance bit settings.

Example 3

Hex Data	Source	Comments
	SinglePosFormat2 DashSpacingSubTable	SinglePos subtable definition
0002	2	PosFormat
0014	DashSpacingCoverage	Offset to Coverage table
0005	0x0005	ValueFormat for XPlacement and XAdvance
0003	3	ValueCount Value[0], for dash glyph
0032	50	XPlacement
0032	50	XAdvance Value[1], for en dash glyph
0019	25	XPlacement
0019	25	XAdvance Value[2], for em dash glyph

000A	10	XPlacement
000A	10	XAdvance
	CoverageFormat1 DashSpacingCoverage	Coverage table definition
0001	1	CoverageFormat
0003	3	GlyphCount
004F	DashGlyphID	GlyphArray[0]
0125	EnDashGlyphID	GlyphArray[1]
0129	EmDashGlyphID	GlyphArray[2]

Example 4: PairPosFormat1 subtable

Example 4 uses a PairPosFormat1 subtable to kern two glyph pairs - "Po" and "To" - by adjusting the XAdvance of the first glyph and the XPlacement of the second glyph. Two ValueFormats are defined, one for each glyph. The subtable contains a Coverage table that lists the index of the first glyph in each pair. It also contains an offset to a PairSet table for each covered glyph.

A PairSet table defines an array of PairValueRecords to specify all the glyph pairs that contain a covered glyph as their first component. In this example, the PPairSet table has one PairValueRecord that identifies the second glyph in the "Po" pair and two ValueRecords, one for the first glyph and one for the second. The TPairSet table also has one PairValueRecord that lists the second glyph in the "To" pair and two ValueRecords, one for each glyph.

Example 4

Hex Data	Source	Comments
	PairPosFormat1 PairKerningSubTable	PairPos subtable definition
0001	1	PosFormat
001E	PairKerningCoverage	Offset to Coverage table
0004	0x0004	ValueFormat1 XAdvance only
0001	0x0001	ValueFormat2 XPlacement only
0002	2	PairSetCount

000E	PPairSetTable	PairSet[0]
0016	TPairSetTable	PairSet[1]
	PairSetTable PPairSetTable	PairSet table definition
0001	1	PairValueCount, one pair in set PairValueRecord[0]
0059	LowercaseOGlyphID	SecondGlyph
FFE2	-30	Value 1, XAdvance adjustment for first glyph
FFEC	-20	Value 2, XPlacement adjustment for second glyph
	PairSetTable PairSetTable	PairSet table definition
0001	1	PairValueCount one pair in set PairValueRecord[0]
0059	LowercaseOGlyphID	SecondGlyph
FFD8	-40	Value1 XAdvance adjustment for first glyph
FFE7	-25	Value 2 XPlacement adjustment for second glyph
	CoverageFormat1 PairKerningCoverage	Coverage table definition
0001	1	CoverageFormat
0002	2	GlyphCount
002D	UppercasePGlyphID	GlyphArray[0]
0031	UppercaseTGlyphID	GlyphArray[1]

Example 5: PairPosFormat2 subtable

The PairPosFormat2 subtable in this example defines pairs composed of two glyph classes. Two ClassDef tables are defined, one for each glyph class. The first glyph in each pair is in a class of lowercase glyphs with diagonal shapes (v, w, y), defined Class1 in the LowercaseClassDef table. The second glyph in each pair is in a class of punctuation glyphs (comma and period), defined in Class1 in the PunctuationClassDef table. The

Coverage table only lists the indices of the glyphs in the LowercaseClassDef table since they occupy the first position in the pairs.

The subtable defines two Class1Records for the classes defined in LowecaseClassDef, including Class0. Each record, in turn, defines a Class2Record for each class defined in PunctuationClassDef, including Class0. The Class2Records specify the positioning adjustments for the glyphs.

The pairs are kerned by reducing the XAdvance of the first glyph by 50 design units. Because no positioning change applies to the second glyph, its ValueFormat2 is set to 0, to indicate that Value2 is empty for each pair.

Since no pairs begin with Class0 or Class2 glyphs, all the ValueRecords referenced in Class1Record[0] contain values of 0 or are empty. However, Class1Record[1] does define an XAdvance value in its Class2Record[1] for kerning all pairs that contain a Class1 glyph followed by a Class2 glyph.

Example 5

Hex Data	Source	Comments
	PairPosFormat2 PunctKerningSubTable	PairPos subtable definition
0002	2	PosFormat
0018	PunctKerningCoverage	Offset to Coverage table
0004	0x0004	ValueFormat1 XAdvance only
0000	0	ValueFormat2 no ValueRecord for second glyph
0022	LowercaseClassDef	Offset to ClassDef1 table for first class in pair
0032	PunctuationClassDef	Offset to ClassDef2 table for second class in pair
0002	2	Class1Count
0002	2	Class2Count Class1Record[0], no contexts begin with Class0 Class2Record[0]
0000	0	Value1- no change for first glyph, Value2 no ValueRecord for second glyph Class2Record[1]
0000	0	Value1- no change for first glyph, Value2 no ValueRecord for second glyph Class1Record[1], for contexts beginning with Class1 Class2Record[0] no contexts with Class0 as second glyph
0000	0	Value1-no change for first glyph, Value2-no ValueRecord for second glyph Class2Record[1]contexts with Class1 as second glyph
FFCE	-50	Value1- move punctuation glyph left, Value2- no ValueRecord for second glyph
	CoverageFormat1 PunctKerningCoverage	Coverage table definition

0001	1	CoverageFormat, lists
0003	3	GlyphCount
0046	LowercaseVGlyphID	GlyphArray[0]
0047	LowercaseWGlyphID	GlyphArray[1]
0049	LowercaseYGlyphID	GlyphArray[2]
	ClassDefFormat2 LowercaseClassDef	ClassDef table definition
0002	2	ClassFormat
0002	2	ClassRangeCount ClassRangeRecord[0]
0046	LowercaseVGlyphID	Start
0047	LowercaseWGlyphID	End
0001	1	Class ClassRangeRecord[1]
0049	LowercaseYGlyphID	Start
0049	LowercaseYGlyphID	End
0001	1	Class
	ClassDefFormat2 PunctuationClassDef	ClassDef table definition
0002	2	ClassFormat
0001	1	ClassRangeCount ClassRangeRecord[0]
006A	PeriodPunctGlyphID	Start
006B	CommaPunctGlyphID	End
0001	1	Class

Example 6: CursivePosFormat1 subtable

In Example 6, the Urdu language system uses a CursivePosFormat1 subtable to attach glyphs along a diagonal baseline that descends from right to left. Two glyphs are defined with attachment data and listed in

the Coverage table-the Kaf and Ha glyphs. For each glyph, the subtable contains an EntryExitRecord that defines Offsets to two Anchor tables, an entry attachment point, and an exit attachment point. Each Anchor table defines X and Y coordinate values. To render Urdu down and diagonally, the entry point's Y coordinate is above the baseline and the exit point's Y coordinate is located below the baseline.

Example 6

Hex Data	Source	Comments
	CursivePosFormat1 DiagonalWritingSubTable	CursivePos subtable definition
0001	1	PosFormat
000E	DiagonalWritingCoverage	Offset to Coverage table
0002	2	EntryExitCount EntryExitRecord[0] for Kaf glyph
0016	KafEntryAnchor	Offset to EntryAnchor table
001C	KafExitAnchor	Offset to ExitAnchor table EntryExitRecord[1] for Ha glyph
0022	HaEntryAnchor	Offset to EntryAnchor table
0028	HaExitAnchor	Offset to ExitAnchor table
	CoverageFormat1 DiagonalWritingCoverage	Coverage table definition
0001	1	CoverageFormat
0002	2	GlyphCount
0203	KafGlyphID	GlyphArray[0]
027E	HaGlyphID	GlyphArray[1]
	AnchorFormat1 KafEntryAnchor	Anchor table definition
0001	1	AnchorFormat
05DC	1500	XCoordinate

002C	44	YCoordinate
	AnchorFormat1 KafExitAnchor	Anchor table definition
0001	1	AnchorFormat
0000	0	XCoordinate
FFEC	-20	YCoordinate
	AnchorFormat1 HaEntryAnchor	Anchor table definition
0001	1	AnchorFormat
05DC	1500	XCoordinate
002C	44	YCoordinate
	AnchorFormat1 HaExitAnchor	Anchor table definition
0001	1	AnchorFormat
0000	0	XCoordinate
FFEC	-20	Ycoordinate

Example 7: MarkBasePosFormat1 subtable

The MarkBasePosFormat1 subtable in Example 7 defines one Arabic base glyph, Tah, and two Arabic mark glyphs: a fathatan mark above the base glyph, and a kasra mark below the base glyph. The BaseGlyphsCoverage table lists the base glyph, and the MarkGlyphsCoverage table lists the mark glyphs.

Each mark is also listed in the MarkArray, along with its attachment point data and a mark Class value. The MarkArray defines two mark classes: Class0 consists of marks located above base glyphs, and Class1 consists of marks located below base glyphs.

The BaseArray defines attachment data for base glyphs. In this array, one BaseRecord is defined for the Tah glyph with Offsets to two BaseAnchor tables, one for each class of marks. AboveBaseAnchor defines an attachment point for marks placed above the Tah base glyph, and BelowBaseAnchor defines an attachment point for marks placed below it.

Example 7

Hex Data	Source	Comments
	MarkBasePosFormat1 MarkBaseAttachSubTable	MarkBasePos subtable definition
0001	1	PosFormat
000C	MarkGlyphsCoverage	Offset to MarkCoverage table
0014	BaseGlyphsCoverage	Offset to BaseCoverage table
0002	2	ClassCount
001A	MarkGlyphsArray	Offset to MarkArray table
0030	BaseGlyphsArray	Offset to BaseArray table
	CoverageFormat1 MarkGlyphsCoverage	Coverage table definition
0001	1	CoverageFormat
0002	2	GlyphCount
0333	fathatanMarkGlyphID	GlyphArray[0]
033F	kasraMarkGlyphID	GlyphArray[1]
	CoverageFormat1 BaseGlyphsCoverage	Coverage table definition
0001	1	CoverageFormat
0001	1	GlyphCount
0190	tahBaseGlyphID	GlyphArray[0]
	MarkArray MarkGlyphsArray	MarkArray table definition
0002	2	MarkCount MarkRecord[0] in CoverageIndex order
0000	0	Class, for marks over base

000A	fathatanMarkAnchor	Offset to Anchor table MarkRecord[1]
0001	1	Class, for marks under
0010	kasraMarkAnchor	Offset to Anchor table
	AnchorFormat1 fathatanMarkAnchor	Anchor table definition
0001	1	AnchorFormat
015A	346	XCoordinate
FF9E	-98	YCoordinate
	AnchorFormat1 kasraMarkAnchor	Anchor table definition
0001	1	AnchorFormat
0105	261	XCoordinate
0058	88	YCoordinate
	BaseArray BaseGlyphsArray	BaseArray table definition
0001	1	BaseCount BaseRecord[0]
0006	AboveBaseAnchor	BaseAnchor[0]
000C	BelowBaseAnchor	BaseAnchor[1]
	AnchorFormat1 AboveBaseAnchor	Anchor table definition
0001	1	AnchorFormat
033E	830	XCoordinate
0640	1600	YCoordinate
	AnchorFormat1 BelowBaseAnchor	Anchor table definition

0001	1	AnchorFormat
033E	830	XCoordinate
FFAD	-83	Ycoordinate

Example 8: MarkLigPosFormat1 subtable

Example 8 uses the MarkLigPosFormat1 subtable to attach marks to a ligature glyph in the Arabic script. The hypothetical ligature is composed of three glyph components: a Lam (initial form), a meem (medial form), and a jeem (medial form). Accent marks are defined for the first two components: the sukun accent is positioned above lam, and the kasratan accent is placed below meem.

The LigGlyphsCoverage table lists the ligature glyph and the MarkGlyphsCoverage table lists the two accent marks. Each mark is also listed in the MarkArray, along with its attachment point data and a mark Class value. The MarkArray defines two mark classes: Class0 consists of marks located above base glyphs, and Class1 consists of marks located below base glyphs.

The LigGlyphsArray has an offset to one LigatureAttach table for the covered ligature glyph. This table, called LamWithMeemWithJeemLigAttach, defines a count and array of the component glyphs in the ligature. Each ComponentRecord defines offsets to two Anchor tables, one for each mark class.

In the example, the first glyph component, lam, specifies a high attachment point for positioning accents above, but does not specify a low attachment point for placing accents below. The second glyph component, meem, defines a low attachment point for placing accents below, but not above. The third component, jeem, has no attachment points since the example defines no accents for it.

Example 8

Hex Data	Source	Comments
	MarkLigPosFormat1 MarkLigAttachSubTable	MarkLigPos subtable definition
0001	1	PosFormat
000C	MarkGlyphsCoverage	Offset to MarkCoverage table
0014	LigGlyphsCoverage	Offset to LigatureCoverage table
0002	2	ClassCount
001A	MarkGlyphsArray	Offset to MarkArray table
0030	LigGlyphsArray	Offset to LigatureArray table
	CoverageFormat1 MarkGlyphsCoverage	Coverage table definition
0001	1	CoverageFormat

0002	2	GlyphCount
033C	sukunMarkGlyphID	GlyphArray[0]
033F	kasratanMarkGlyphID	GlyphArray[1]
	CoverageFormat1 LigGlyphsCoverage	Coverage table definition
0001	1	CoverageFormat
0001	1	GlyphCount
0234	LamWithMeemWithJeem LigatureGlyphID	GlyphArray[0]
	MarkArray MarkGlyphsArray	MarkArray table definition
0002	2	MarkCount MarkRecord[0] in CoverageIndex order
0000	0	Class, for marks above components
000A	sukunMarkAnchor	Offset to Anchor table MarkRecord[1]
0001	1	Class, for marks below components
0010	kasratanMarkAnchor	Offset to Anchor table
	AnchorFormat1 sukunMarkAnchor	Anchor table definition
0001	1	AnchorFormat
015A	346	XCoordinate
FF9E	-98	YCoordinate
	AnchorFormat1 kasratanMarkAnchor	Anchor table definition
0001	1	AnchorFormat
0105	261	XCoordinate

01E8	488	YCoordinate
	LigatureArray LigGlyphsArray	LigatureArray table definition
0001	1	LigatureCount
0004	LamWithMeemWithJeemLigAttach	Offset to LigatureAttach table
	LigatureAttach LamWithMeemWithJeemLigAttach	LigatureAttach table definition
0003	3	ComponentCount ComponentRecord[0] Right-to-Left text order
000E	AboveLamAnchor	Offset to LigatureAnchor table ordered by mark class value for Class0 marks (above)
0000	NULL	Offset to LigatureAnchor table no attachment points for Class1 marks ComponentRecord[1]
0000	NULL	Offset to LigatureAnchor table no attachment points for Class0 marks
0014	BelowMeemAnchor	Offset to LigatureAnchor table for Class1 marks (below) ComponentRecord[2]
0000	NULL	Offset to LigatureAnchor table no attachment points for Class0 marks
0000	NULL	Offset to LigatureAnchor table no attachment points for Class1 marks
	AnchorFormat1 AboveLamAnchor	Anchor table definition
0001	1	AnchorFormat
0271	625	XCoordinate
0708	1800	YCoordinate
	AnchorFormat1 BelowMeemAnchor	Anchor table definition
0001	1	AnchorFormat

0178	376	XCoordinate
FE90	-368	Ycoordinate

Example 9: MarkMarkPosFormat1 subtable

The MarkMarkPosFormat1 subtable in Example 9 defines two Arabic marks glyphs. The hanza mark, the base mark (Mark2), is identified in the Mark2GlyphsCoverage table. The damma mark, the attaching mark (Mark1), is defined in the Mark1GlyphsCoverage table.

Each Mark1 glyph is also listed in the Mark1Array, along with its attachment point data and a mark Class value. The Mark1GlyphsArray defines one mark class, Class0, that consists of marks located above Mark2 base glyphs. The Mark1GlyphsArray contains an Offset to a dammaMarkAnchor table to specify the coordinate of the damma mark's attachment point.

The Mark2GlyphsArray table defines a count and an array of Mark2Records, one for each covered Mark2 base glyph. Each record contains an Offset to a Mark2Anchor table for each Mark1 class. One Anchor table, AboveMark2Anchor, specifies a coordinate value for attaching the damma mark above the hanza base mark.

Example 9

Hex Data	Source	Comments
	MarkMarkPosFormat1 MarkMarkAttachSubTable	MarkBasePos subtable definition
0001	1	PosFormat
000C	Mark1GlyphsCoverage	Offset to Mark1Coverage table
0012	Mark2GlyphsCoverage	Offset to Mark2Coverage table
0001	1	ClassCount
0018	Mark1GlyphsArray	Offset to Mark1Array table
0024	Mark2GlyphsArray	Offset to Mark2Array table
	CoverageFormat1 Mark1GlyphsCoverage	Coverage table definition
0001	1	CoverageFormat
0001	1	GlyphCount
0296	dammaMarkGlyphID	GlyphArray[0]
	CoverageFormat1 Mark2GlyphsCoverage	Coverage table definition

0001	1	CoverageFormat
0001	1	GlyphCount
0289	hanzaMarkGlyphID	GlyphArray[1]
	MarkArray Mark1GlyphsArray	MarkArray table definition
0001	1	MarkCount MarkRecord[0] in CoverageIndex order
0000	0	Class for marks above base mark
0006	dammaMarkAnchor	Offset to Anchor table
	AnchorFormat1 dammaMarkAnchor	Anchor table definition
0001	1	AnchorFormat
00BD	189	XCoordinate
FF99	-103	YCoordinate
	Mark2Array Mark2GlyphsArray	Mark2Array table definition
0001	1	Mark2Count Mark2Record[0]
0004	AboveMark2Anchor	Offset to Anchor table[0]
	AnchorFormat1 AboveMark2Anchor	Anchor table definition
0001	1	AnchorFormat
00DD	221	XCoordinate
012D	301	Ycoordinate

Example 10: ContextPosFormat1 subtable and PosLookupRecord

Example 10 uses a ContextPosFormat1 subtable to adjust the spacing between three Arabic glyphs in a word. The context is the glyph sequence (from right to left): heh (initial form), thal (final form), and heh (isolated form). In the rendered word, the first two glyphs are connected, but the last glyph (the isolated form of heh), is separate. This subtable reduces the amount of space between the last glyph and the rest of the word.

The subtable contains a WordCoverage table that lists the first glyph in the word, heh (initial), and one PosRuleSet table, called WordPosRuleSet, that defines all contexts beginning with this covered glyph.

The WordPosRuleSet contains one PosRule that describes a word context of three glyphs and identifies the second and third glyphs (the first glyph is identified by the WordPosRuleSet). When a text-processing client locates this context in text, it applies a SinglePos lookup (not shown in the example) at position 2 to reduce the spacing between the glyphs.

Example 10

Hex Data	Source	Comments
	ContextPosFormat1 MoveHehInSubtable	ContextPos subtable definition
0001	1	PosFormat
0008	WordCoverage	Offset to Coverage table
0001	1	PosRuleSetCount
000E	WordPosRuleSet	Offset to PosRuleSet[0] table
	CoverageFormat1 WordCoverage	Coverage table Offset
0001	1	CoverageFormat
0001	1	GlyphCount
02A6	hehInitialGlyphID	GlyphArray[0]
	PosRuleSet WordPosRuleSet	PosRuleSet table definition
0001	1	PosRuleCount
0004	WordPosRule	Offset to PosRule[0] table

	PosRule WordPosRule	PosRule table definition
0003	3	GlyphCount
0001	1	PosCount
02DD	thalFinalGlyphID	Input[1]
02C6	hehIsolatedGlyphID	Input[0] PosLookupRecord[0]
0002	2	SequenceIndex
0001	1	LookupListIndex

Example 11: ContextPosFormat2 subtable

The ContextPosFormat2 subtable in Example 11 defines context strings for five glyph classes: Class1 consists of uppercase glyphs that overhang and create a wide open space on their right side; Class2 consists of uppercase glyphs that overhang and create a narrow space on their right side; Class3 contains lowercase x-height vowels; and Class4 contains accent glyphs placed over the lowercase vowels. The rest of the glyphs in the font fall into Class0.

The MoveAccentsSubtable defines two similar context strings. The first consists of a Class1 uppercase glyph followed by a Class3 lowercase vowel glyph with a Class4 accent glyph over the vowel. When this context is found in the text, the client lowers the accent glyph over the vowel so that it does not collide with the overhanging glyph shape. The second context consists of a Class2 uppercase glyph, followed by a Class3 lowercase vowel glyph with a Class4 accent glyph over the vowel. When this context is found in the text, the client increases the advance width of the uppercase glyph to expand the space between it and the accented vowel.

The MoveAccents subtable defines a MoveAccentsCoverage table that identifies the first glyphs in the two contexts and Offsets to five PosClassSet tables, one for each class defined in the ClassDef table. Since no contexts begin with Class0, Class3, or Class4 glyphs, the Offsets to the PosClassSet tables for these classes are NULL. PosClassSet[1] defines all contexts beginning with Class1 glyphs; it is called UCWideOverhangPosClass1Set. PosClassSet[2] defines all contexts beginning with Class2 glyphs, and it is called UCNarrowOverhangPosClass1Set.

Each PosClassSet defines one PosClassRule. The UCWideOverhangPosClass1Set uses the UCWideOverhangPosClassRule to specify the first context. The first class in this context string is identified by the PosClassSet that includes a PosClassRule, in this case Class1. The PosClassRule table lists the second and third classes in the context as Class3 and Class4. A SinglePos Lookup (not shown) lowers the accent glyph in position 3 in the context string.

The UCNarrowOverhangPosClass1Set defines the UCNarrowOverhangPosClassRule for the second context. This PosClassRule is identical to the UCWideOverhangPosClassRule, except that the first class in the context string is a Class2 lowercase glyph. A SinglePos Lookup (not shown) increases the advance width of the overhanging uppercase glyph in position 0 in the context string.

Example 11

Hex Data	Source	Comments
	ContextPosFormat2 MoveAccentsSubtable	ContextPos subtable definition
0002	2	PosFormat
0012	MoveAccentsCoverage	Offset to Coverage table
0020	MoveAccentsClassDef	Offset to ClassDef
0005	5	PosClassSetCnt
0000	NULL	PosClassSet[0], no contexts begin with Class0 glyphs
0060	UCWideOverhangPosClass1Set	PosClassSet[1] contexts beginning with Class1 glyphs
0070	UCNarrowOverhangPosClass2Set	PosClassSet[2] context beginning with Class2 glyphs
0000	NULL	PosClassSet[3], no contexts begin with Class3 glyphs
0000	NULL	PosClassSet[4], no contexts begin with Class4 glyphs
	CoverageFormat1 MoveAccentsCoverage	Coverage table definition
0001	1	CoverageFormat
0005	5	GlyphCount
0029	UppercaseFGlyphID	GlyphArray[0]
0033	UppercasePGlyphID	GlyphArray[1]
0037	UppercaseTGlyphID	GlyphArray[2]
0039	UppercaseVGlyphID	GlyphArray[3]
003A	UppercaseWGlyphID	GlyphArray[4]
	ClassDefFormat2 MoveAccentsClassDef	ClassDef table definition defines five classes = 0 (all else), 1 (T, V, W: UCUnderhang), 2 (F, P: UCOverhang), 3 (a, e, I, o, u: LCVowels), 4 (tilde, umlaut)
0002	2	ClassFormat, ranges

000A	10	ClassRangeCount ClassRangeRecord[0]
0029	UppercaseFGlyphID	Start
0029	UppercaseFGlyphID	End
0002	2	Class ClassRangeRecord[1]
0033	UppercasePGlyphID	Start
0033	UppercasePGlyphID	End
0002	2	Class ClassRangeRecord[2]
0037	UppercaseTGlyphID	Start
0037	UppercaseTGlyphID	End
0001	1	Class ClassRangeRecord[3]
0039	UppercaseVGlyphID	Start
003A	UppercaseWGlyphID	End
0001	1	Class ClassRangeRecord[4]
0042	LowercaseAGlyphID	Start
0042	LowercaseAGlyphID	End
0003	3	Class ClassRangeRecord[5]
0046	LowercaseEGlyphID	Start
0046	LowercaseEGlyphID	End
0003	3	Class ClassRangeRecord[6]
004A	LowercaseIGlyphID	Start
004A	LowercaseIGlyphID	End
0003	3	Class ClassRangeRecord[7]

0051	LowercaseOGlyphID	Start
0051	LowercaseOGlyphID	End
0003	3	Class ClassRangeRecord[8]
0056	LowercaseUGlyphID	Start
0056	LowercaseUGlyphID	End
0003	3	Class ClassRangeRecord[9]
00F5	TildeAccentGlyphID	Start
00F6	UmlautAccentGlyphID	End
0004	4	Class
	PosClassSet UCWideOverhangPosClass1Set	PosClassSet table definition
0001	1	PosClassRuleCnt
0004	UCWideOverhangPosClassRule	PosClassRule[0]
	PosClassRule UCWideOverhangPosClassRule	PosClassRule table definition
0003	3	GlyphCount
0001	1	PosCount
0003	3	Class[1], lowercase vowel
0004	4	Class[2], accent PosLookupRecord[0]
0002	2	SequenceIndex
0001	1	LookupListIndex, lower the accent
	PosClassSet UCNarrowOverhangPosClass2Set	PosClassSet table definition
0001	1	PosClassRuleCnt

0004	UCNarrowOverhangPosClassRule	PosClassRule[0]
	PosClassRule UCNarrowOverhangPosClassRule	PosClassRule table definition
0003	3	GlyphCount
0001	1	PosCount
0003	3	Class[1], lowercase vowel
0004	4	Class[2], accent PosLookupRecord[0]
0000	0	SequenceIndex
0002	2	LookupListIndex increase overhang advance width

Example 12: ContextPosFormat3 subtable

Example 12 uses a ContextPosFormat3 subtable to lower the position of math signs in math equations consisting of a lowercase descender or x-height glyph, a math sign glyph, and any lowercase glyph. Format3 is better to use for this context than the class-based Format2 because the sets of covered glyphs for positions 0 and 2 overlap.

The LowerMathSignsSubtable contains Offsets to three Coverage tables (XhtDescLCCoverage, MathSignCoverage, and LCCoverage), one for each position in the context glyph string. When the client finds the context in the text stream, it applies the PosLookupRecord data at position 1 and repositions the math sign.

Example 12

Hex Data	Source	Comments
	ContextPosFormat3 LowerMathSignsSubtable	ContextPos subtable definition
0003	3	PosFormat
0003	3	GlyphCount
0001	1	PosLookup
0010	XhtDescLCCoverage	Offset to Coverage[0] table
003C	MathSignCoverage	Offset to Coverage[1] table
0044	LCCoverage	Offset to Coverage[2] table PosLookupRecord[0]

0001	1	SequenceIndex
0001	1	LookupListIndex
	CoverageFormat1 XhtDescLCCoverage	Coverage table definition
0001	1	CoverageFormat
0014	20	GlyphCount
0033	LCaGlyphID	GlyphArray[0]
0035	LCcGlyphID	GlyphArray[1]
0037	LCeGlyphID	GlyphArray[2]
0039	LCgGlyphID	GlyphArray[3]
003B	LCiGlyphID	GlyphArray[4]
003C	LCjGlyphID	GlyphArray[5]
003F	LCmGlyphID	GlyphArray[6]
0040	LCnGlyphID	GlyphArray[7]
0041	LCoGlyphID	GlyphArray[8]
0042	LCpGlyphID	GlyphArray[9]
0043	LCqGlyphID	GlyphArray[10]
0044	LCrGlyphID	GlyphArray[11]
0045	LCsGlyphID	GlyphArray[12]
0046	LCtGlyphID	GlyphArray[13]
0047	LCuGlyphID	GlyphArray[14]
0048	LCvGlyphID	GlyphArray[15]
0049	LCwGlyphID	GlyphArray[16]
004A	LCxGlyphID	GlyphArray[17]

004B	LCyGlyphID	GlyphArray[18]
004C	LCzGlyphID	GlyphArray[19]
	CoverageFormat1 MathSignCoverage	Coverage table definition
0001	1	CoverageFormat
0002	2	GlyphCount
011E	EqualsSignGlyphID	GlyphArray[0]
012D	PlusSignGlyphID	GlyphArray[1]
	CoverageFormat2 LCCoverage	Coverage table definition
0002	2	CoverageFormat
0001	1	RangeCount RangeRecord[0]
0033	LCaGlyphID	Start
004C	LCzGlyphID	End
0000	0	StartCoverageIndex

Example 13: PosLookupRecord

The PosLookupRecord in Example 13 identifies a lookup to apply at the second glyph position in a context glyph string.

Example 13

Hex Data	Source	Comments
	PosLookupRecord PosLookupRecord[0]	PosLookupRecord definition
0001	1	SequenceIndex for second glyph position
0001	1	LookupListIndex, apply this lookup to second glyph position

Example 14: ValueFormat table and ValueRecord

Example 14 demonstrates how to specify positioning values in the GPOS table. Here, a SinglePosFormat1 subtable defines the ValueFormat and ValueRecord. The ValueFormat bit setting of 0x0099 says that the corresponding ValueRecord contains values for a glyph's XPlacement and YAdvance. Device tables specify pixel adjustments for these values at font sizes from 11 ppem to 15 ppem.

Example 14

Hex Data	Source	Comments
	SinglePosFormat1 OnesSubtable	SinglePos subtable definition
0001	1	PosFormat
000E	Cov	Offset to Coverage table
0099	0x0099	ValueFormat, for XPlacement, YAdvance, XPlaDevice, YAdvaDevice Value
0050	80	Xplacement value
00D2	210	Yadvance value
0018	XPlaDeviceTable	Offset to XPlaDevice table
0020	YAdvDeviceTable	Offset to YAdvDevice table
	CoverageFormat2 Cov	Coverage table definition
0002	2	CoverageFormat
0001	1	RangeCount RangeRecord[0]
00C8	200	Start, first glyph ID in range
00D1	209	End, last glyph ID in range
0000	0	StartCoverageIndex
	DeviceTableFormat1 XPlaDeviceTable	Device Table definition
000B	11	StartSize
000F	15	EndSize
0001	1	DeltaFormat

	1	increase 11ppem by 1 pixel
	1	increase 12ppem by 1 pixel
	1	increase 13ppem by 1 pixel
	1	increase 14ppem by 1 pixel
5540	1	increase 15ppem by 1 pixel
	DeviceTableFormat1 YAdvDeviceTable	Device Table definition
000B	11	StartSize
000F	15	EndSize
0001	1	DeltaFormat
	1	increase 11ppem by 1 pixel
	1	increase 12ppem by 1 pixel
	1	increase 13ppem by 1 pixel
	1	increase 14ppem by 1 pixel
5540	1	increase 15ppem by 1 pixel

Example 15: AnchorFormat1 table

Example 15 illustrates an Anchor table for the damma mark glyph in the Arabic script. Format1 is used to specify X and Y coordinate values in design units.

Example 15

Hex Data	Source	Comments
	AnchorFormat1 dammaMarkAnchor	Anchor table definition
0001	1	AnchorFormat
00BD	189	XCoordinate
FF99	-103	YCoordinate

Example 16: AnchorFormat2 table

Example 16 shows an AnchorFormat2 table for an attachment point placed above a base glyph. With this format, the coordinate value for the Anchor depends on the final position of a specific contour point on the base glyph after hinting. The coordinates are specified in design units.

Example 16

Hex Data	Source	Comments
	AnchorFormat2 AboveBaseAnchor	Anchor table definition
0002	2	AnchorFormat
0142	322	XCoordinate
0384	900	Ycoordinate
000D	13	AnchorPoint glyph contour point index

Example 17: AnchorFormat3 table

Example 17 shows an AnchorFormat3 table that specifies an attachment point above a base glyph. Device tables modify the X and Y coordinates of the Anchor for the point size and resolution of the output font. Here, the Device tables define pixel adjustments for font sizes from 12 ppem to 17 ppem.

Example 17

Hex Data	Source	Comments
	AnchorFormat3 AboveBaseAnchor	Anchor table definition
0003	3	AnchorFormat
0117	279	XCoordinate
0515	1301	YCoordinate
000A	XDevice	Offset to DeviceTable for X coordinate (may be NULL)
0014	YDevice	Offset to Device table for Y coordinate (may be NULL)
	DeviceTableFormat2 XDevice	Device Table definition
000C	12	StartSize

0011	17	EndSize
0002	2	DeltaFormat
	1	increase 12ppem by 1 pixel
	1	increase 13ppem by 1 pixel
	1	increase 14ppem by 1 pixel
1111	1	increase 15ppem by 1 pixel
	2	increase 16ppem by 1 pixel
2200	2	increase 17ppem by 1 pixel
	DeviceTableFormat2 YDevice	Device Table definition
000C	12	StartSize
0011	17	EndSize
0002	2	DeltaFormat
	1	increase 12ppem by 1 pixel
	1	increase 13ppem by 1 pixel
	1	increase 14ppem by 1 pixel
1111	1	increase 15ppem by 1 pixel
	2	increase 16ppem by 1 pixel
2200	2	increase 17ppem by 1 pixel

Example 18: MarkArray table and MarkRecord

Example 18 shows a MarkArray table with class and attachment point data for two accent marks, a grave and a cedilla. Two MarkRecords are defined, one for each covered mark glyph. The first MarkRecord assigns a mark class value of 0 to accents placed above base glyphs, such as the grave, and has an Offset to a graveMarkAnchor table. The second MarkRecord assigns a mark class value of 1 for all accents positioned below base glyphs, such as the cedilla, and has an Offset to a cedillaMarkAnchor table.

Example 18

Hex Data	Source	Comments
	MarkArray MarkGlyphsArray	MarkArray table definition
0002	2	MarkCount MarkRecord[0] for first mark in MarkCoverage table, grave
0000	0	Class, for marks placed above base glyphs
000A	graveMarkAnchor	Offset to Anchor table MarkRecord[1] for second mark in MarkCoverage table = cedilla
0001	1	Class, for marks placed below base glyphs
0010	cedillaMarkAnchor	Offset to Anchor table

6.3.4 GSUB – The glyph substitution table

The Glyph Substitution table (GSUB) contains information for substituting glyphs to render the scripts and language systems supported in a font. Many language systems require glyph substitutes. For example, in the Arabic script, the glyph shape that depicts a particular character varies according to its position in a word or text string (see Figure 32). In other language systems, glyph substitutes are aesthetic options for the user, such as the use of ligature glyphs in the English language (see Figure 33).

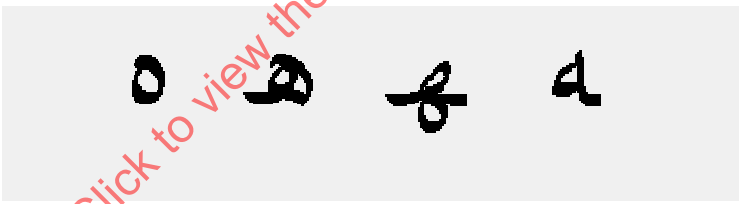

Figure 32 – Isolated, initial, medial, and final forms of the Arabic character HAH

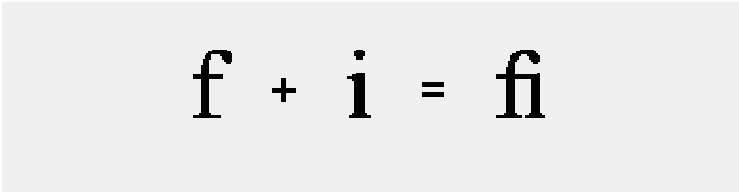

Figure 33 – Two Latin glyphs and their associated ligature

6.3.4.1 GSUB – Table overview

Many fonts use limited character encoding standards that map glyphs to characters one-to-one, assigning a glyph to each character code value in a font. Multiple character codes cannot be mapped to a single glyph, as needed for ligature glyphs, and multiple glyphs cannot be mapped to a single character code, as needed to decompose a ligature into its component glyphs.

To supply glyph substitutes, font developers must assign different character codes to the glyphs, or they must create additional fonts or character sets. To access these glyphs, users must bear the burden of switching between character codes, character sets, or fonts.

Substituting glyphs with OFF

The OFF GSUB table fully supports glyph substitution. To access glyph substitutes, GSUB maps from the glyph index or indices defined in a cmap table to the glyph index or indices of the glyph substitutes. For example, if a font has three alternative forms of an ampersand glyph, the cmap table associates the ampersand's character code with only one of these glyphs. In GSUB, the indices of the other ampersand glyphs are then referenced by this one index.

The text-processing client uses the GSUB data to manage glyph substitution actions. GSUB identifies the glyphs that are input to and output from each glyph substitution action, specifies how and where the client uses glyph substitutes, and regulates the order of glyph substitution operations. Any number of substitutions can be defined for each script or language system represented in a font.

The GSUB table supports six types of glyph substitutions that are widely used in international typography:

- A *single substitution* replaces a single glyph with another single glyph. This is used to render positional glyph variants in Arabic and vertical text in the Far East (see Figure 34).

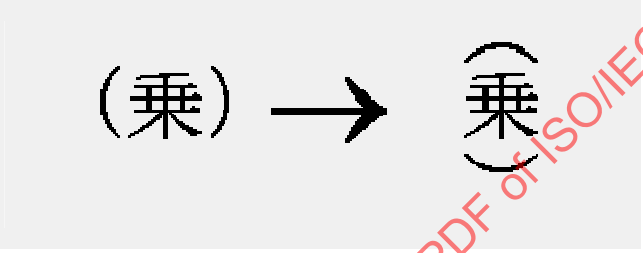

Figure 34 – Alternative forms of parentheses used when positioning Kanji vertically

- A *multiple substitution* replaces a single glyph with more than one glyph. This is used to specify actions such as ligature decomposition (see Figure 35).

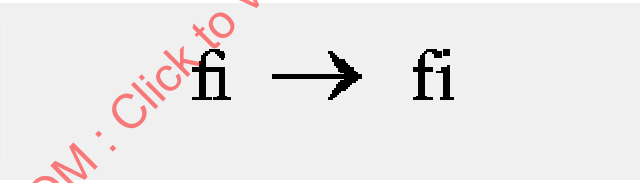

Figure 35 – Decomposing a Latin ligature glyph into its individual glyph components

- An *alternate substitution* identifies functionally equivalent but different looking forms of a glyph. These glyphs are often referred to as aesthetic alternatives. For example, a font might have five different glyphs for the ampersand symbol, but one would have a default glyph index in the cmap table. The client could use the default glyph or substitute any of the four alternatives (see Figure 36).

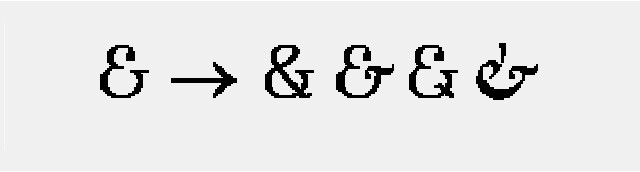

Figure 36 – Alternative ampersand glyphs in a font

- A *ligature substitution* replaces several glyph indices with a single glyph index, as when an Arabic ligature glyph replaces a string of separate glyphs (see Figure 37). When a string of glyphs can be replaced with a single ligature glyph, the first glyph is substituted with the ligature. The remaining

glyphs in the string are deleted, this includes those glyphs that are skipped as a result of lookup flags.

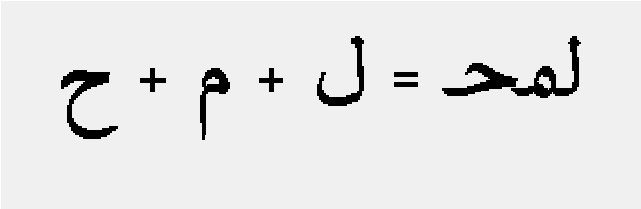

Figure 37 – Three Arabic glyphs and their associated ligature glyph

- *Contextual substitution,* the most powerful type, describes glyph substitutions in context that is, a substitution of one or more glyphs within a certain pattern of glyphs. Each substitution describes one or more input glyph sequences and one or more substitutions to be performed on that sequence. Contextual substitutions can be applied to specific glyph sequences, glyph classes, or sets of glyphs.

- *Chaining contextual substitution* extends the capabilities of contextual substitution. With this, one or more substitutions can be performed on one or more glyphs within a pattern of glyphs (input sequence), by chaining the input sequence to a 'backtrack' and/or 'lookahead' sequence. Each such substitution can be applied in three formats to handle glyphs, glyph classes or glyph sets in the input sequence. Each of these formats can describe one or more of the backtrack, input and lookahead sequences.

- *Reverse Chaining contextual single substitution,* allows one glyph to be substituted with another by chaining input glyph to a 'backtrack' and/or 'lookahead' sequence. The difference between this and other lookup types is that processing of input glyph sequence goes from end to start.

6.3.4.2 GSUB – Table organization and structure

Table organization

The GSUB table begins with a header that defines Offsets to a ScriptList, a FeatureList, and a LookupList (see Figure 38):

- The ScriptList identifies all the scripts and language systems in the font that use glyph substitutes.

- The FeatureList defines all the glyph substitution features required to render these scripts and language systems.

- The LookupList contains all the lookup data needed to implement each glyph substitution feature.

For a detailed discussion of ScriptLists, FeatureLists, and LookupLists, see clause 6.2 OFF Common Table Formats.

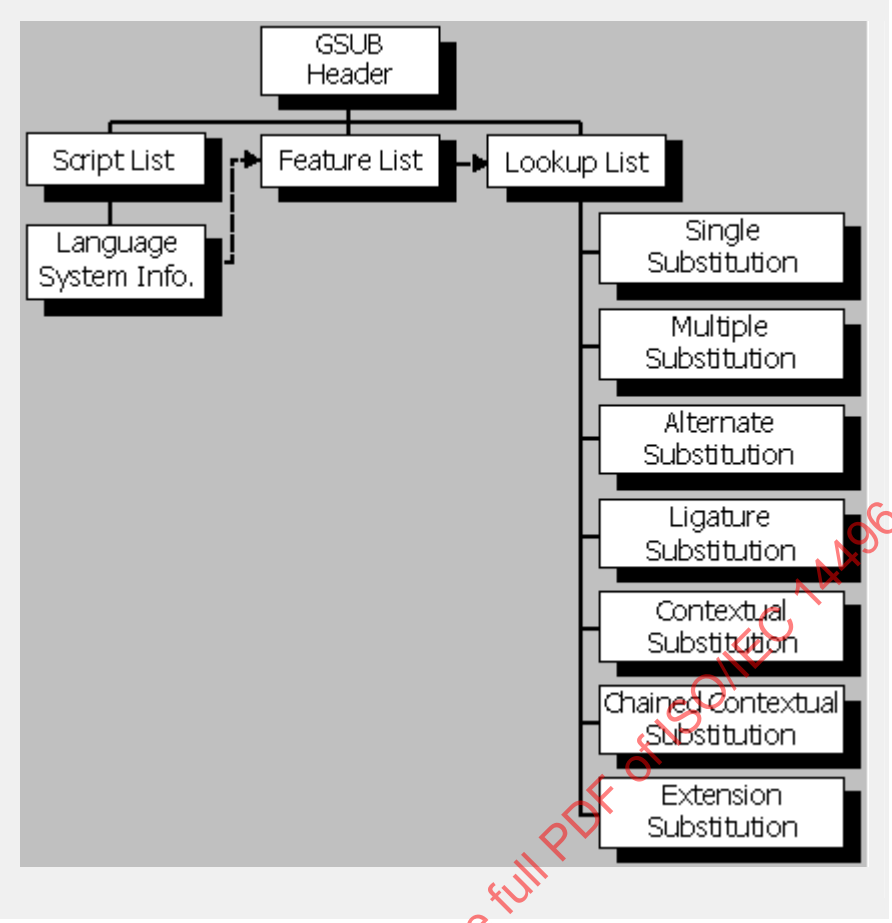

Figure 38 – High-level organization of GSUB table

This organization helps text-processing clients to easily locate the features and lookups that apply to a particular script or language system. To access GSUB information, clients should use the following procedure:

1. Locate the current script in the GSUB ScriptList table.
2. If the language system is known, search the script for the correct LangSys table; otherwise, use the script's default language system (DefaultLangSys table).
3. The LangSys table provides index numbers into the GSUB FeatureList table to access a required feature and a number of additional features.
4. Inspect the FeatureTag of each feature, and select the features to apply to an input glyph string. Each feature provides an array of index numbers into the GSUB LookupList table.
5. Assemble all lookups from the set of chosen features, and apply the lookups in the order given in the LookupList table.

Lookup data is defined in one or more subtables that define the specific conditions, type, and results of a substitution action used to implement a feature. All subtables in a lookup must be of the same LookupType, as listed in the LookupType Enumeration table:

LookupType Enumeration table for glyph substitution

Value	Type	Description
1	Single	Replace one glyph with one glyph
2	Multiple	Replace one glyph with more than one glyph
3	Alternate	Replace one glyph with one of many glyphs
4	Ligature	Replace multiple glyphs with one glyph
5	Context	Replace one or more glyphs in context
6	Chaining Context	Replace one or more glyphs in chained context
7	Extension Substitution	Extension mechanism for other substitutions (i.e. this excludes the Extension type substitution itself)
8	Reverse chaining context single	Applied in reverse order, replace single glyph in chaining context
9+	Reserved	For future use (must be set to zero)

Each LookupType subtable has one or more formats. The "best" format depends on the type of substitution and the resulting storage efficiency. When glyph information is best presented in more than one format, a single lookup may define more than one subtable, as long as all the subtables are for the same LookupType. For example, within a given lookup, a glyph index array format may best represent one set of target glyphs, whereas a glyph index range format may be better for another set.

A series of substitution operations on the same glyph or string requires multiple lookups, one for each separate action. Each lookup is given a different array number in the LookupList table and is applied in the LookupList order.

During text processing, a client applies a lookup to each glyph in the string before moving to the next lookup. A lookup is finished for a glyph after the client locates the target glyph or glyph context and performs a substitution, if specified. To move to the "next" glyph, the client will typically skip all the glyphs that participated in the lookup operation: glyphs that were substituted as well as any other glyphs that formed a context for the operation.

In the case of chained contextual lookups, glyphs comprising backtrack and lookahead sequences may participate in more than one context.

The rest of this clause describes the GSUB header and the subtables defined for each GSUB LookupType. Examples at the end of this page illustrate each of the eight LookupTypes, including the three formats available for contextual substitutions.

GSUB header

The GSUB table begins with a header that contains a version number for the table (Version) and Offsets to three tables: ScriptList, FeatureList, and LookupList. For descriptions of each of these tables, see clause 6.2, OFF Common Table Formats. Example 1 at the end of this clause shows a GSUB Header table definition.

GSUB Header

Type	Name	Description
Fixed	Version	Version of the GSUB table-initially set to 0x00010000
Offset	ScriptList	Offset to ScriptList table-from beginning of GSUB table
Offset	FeatureList	Offset to FeatureList table-from beginning of GSUB table
Offset	LookupList	Offset to LookupList table-from beginning of GSUB table

6.3.4.3 GSUB – Lookup type descriptions

LookupType 1: Single substitution subtable

Single substitution (SingleSubst) subtables tell a client to replace a single glyph with another glyph. The subtables can be either of two formats. Both formats require two distinct sets of glyph indices: one that defines input glyphs (specified in the Coverage table), and one that defines the output glyphs. Format 1 requires less space than Format 2, but it is less flexible.

Single substitution Format 1

Format 1 calculates the indices of the output glyphs, which are not explicitly defined in the subtable. To calculate an output glyph index, Format 1 adds a constant delta value to the input glyph index. For the substitutions to occur properly, the glyph indices in the input and output ranges must be in the same order. This format does not use the Coverage Index that is returned from the Coverage table.

The SingleSubstFormat1 subtable begins with a format identifier (SubstFormat) of 1. An Offset references a Coverage table that specifies the indices of the input glyphs. DeltaGlyphID is the constant value added to each input glyph index to calculate the index of the corresponding output glyph.

Example 2 at the end of this clause uses Format 1 to replace standard numerals with lining numerals.

SingleSubstFormat1 subtable: Calculated output glyph indices

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of Substitution table
int16	DeltaGlyphID	Add to original GlyphID to get substitute GlyphID

Single substitution Format 2

Format 2 is more flexible than Format 1, but requires more space. It provides an array of output glyph indices (Substitute) explicitly matched to the input glyph indices specified in the Coverage table.

The SingleSubstFormat2 subtable specifies a format identifier (SubstFormat), an Offset to a Coverage table that defines the input glyph indices, a count of output glyph indices in the Substitute array (GlyphCount), and a list of the output glyph indices in the Substitute array (Substitute).

The Substitute array must contain the same number of glyph indices as the Coverage table. To locate the corresponding output glyph index in the Substitute array, this format uses the Coverage Index returned from the Coverage table.

Example 3 at the end of this clause uses Format 2 to substitute vertically oriented glyphs for horizontally oriented glyphs.

SingleSubstFormat2 subtable: Specified output glyph indices

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 2
Offset	Coverage	Offset to Coverage table-from beginning of Substitution table
uint16	GlyphCount	Number of GlyphIDs in the Substitute array
GlyphID	Substitute [GlyphCount]	Array of substitute GlyphIDs-ordered by Coverage Index

LookupType 2: Multiple substitution subtable

A Multiple Substitution (MultipleSubst) subtable replaces a single glyph with more than one glyph, as when multiple glyphs replace a single ligature. The subtable has a single format: MultipleSubstFormat1. The subtable specifies a format identifier (SubstFormat), an Offset to a Coverage table that defines the input glyph indices, a count of Offsets in the Sequence array (SequenceCount), and an array of Offsets to Sequence tables that define the output glyph indices (Sequence). The Sequence table Offsets are ordered by the Coverage Index of the input glyphs.

For each input glyph listed in the Coverage table, a Sequence table defines the output glyphs. Each Sequence table contains a count of the glyphs in the output glyph sequence (GlyphCount) and an array of output glyph indices (Substitute).

NOTE The order of the output glyph indices depends on the writing direction of the text. For text written left to right, the left-most glyph will be first glyph in the sequence. Conversely, for text written right to left, the right-most glyph will be first.

The use of multiple substitution for deletion of an input glyph is prohibited. GlyphCount should always be greater than 0.

Example 4 at the end of this clause shows how to replace a single ligature with three glyphs.

MultipleSubstFormat1 subtable: Multiple output glyphs

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of Substitution table
uint16	SequenceCount	Number of Sequence table Offsets in the Sequence array
Offset	Sequence [SequenceCount]	Array of Offsets to Sequence tables-from beginning of Substitution table-ordered by Coverage Index

Sequence table

Type	Name	Description
uint16	GlyphCount	Number of GlyphIDs in the Substitute array. This should always be greater than 0.
GlyphID	Substitute [GlyphCount]	String of GlyphIDs to substitute

LookupType 3: Alternate substitution subtable

An Alternate Substitution (AlternateSubst) subtable identifies any number of aesthetic alternatives from which a user can choose a glyph variant to replace the input glyph. For example, if a font contains four variants of the ampersand symbol, the cmap table will specify the index of one of the four glyphs as the default glyph index, and an AlternateSubst subtable will list the indices of the other three glyphs as alternatives. A text-processing client would then have the option of replacing the default glyph with any of the three alternatives.

The subtable has one format: AlternateSubstFormat1. The subtable contains a format identifier (SubstFormat), an Offset to a Coverage table containing the indices of glyphs with alternative forms (Coverage), a count of Offsets to AlternateSet tables (AlternateSetCount), and an array of Offsets to AlternateSet tables (AlternateSet).

For each glyph, an AlternateSet subtable contains a count of the alternative glyphs (GlyphCount) and an array of their glyph indices (Alternate). Because all the glyphs are functionally equivalent, they can be in any order in the array.

Example 5 at the end of this clause shows how to replace the default ampersand glyph with alternative glyphs.

AlternateSubstFormat1 subtable: Alternative output glyphs

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of Substitution table
uint16	AlternateSetCount	Number of AlternateSet tables
Offset	AlternateSet [AlternateSetCount]	Array of Offsets to AlternateSet tables-from beginning of Substitution table-ordered by Coverage Index

AlternateSet table

Type	Name	Description
uint16	GlyphCount	Number of GlyphIDs in the Alternate array
GlyphID	Alternate[GlyphCount]	Array of alternate GlyphIDs-in arbitrary order

LookupType 4: Ligature substitution subtable

A Ligature Substitution (LigatureSubst) subtable identifies ligature substitutions where a single glyph replaces multiple glyphs. One LigatureSubst subtable can specify any number of ligature substitutions.

The subtable uses a single format: LigatureSubstFormat1. It contains a format identifier (SubstFormat), a Coverage table Offset (Coverage), a count of the ligature sets defined in this table (LigSetCount), and an array of Offsets to LigatureSet tables (LigatureSet). The Coverage table specifies only the index of the first glyph component of each ligature set.

LigatureSubstFormat1 subtable:
All ligature substitutions in a script

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of Substitution table
uint16	LigSetCount	Number of LigatureSet tables
Offset	LigatureSet [LigSetCount]	Array of Offsets to LigatureSet tables-from beginning of Substitution table-ordered by Coverage Index

A LigatureSet table, one for each covered glyph, specifies all the ligature strings that begin with the covered glyph. For example, if the Coverage table lists the glyph index for a lowercase "f," then a LigatureSet table will define the "ffl," "fl," "ffi," "fi," and "ff" ligatures. If the Coverage table also lists the glyph index for a lowercase "e," then a different LigatureSet table will define the "etc" ligature.

A LigatureSet table consists of a count of the ligatures that begin with the covered glyph (LigatureCount) and an array of Offsets to Ligature tables, which define the glyphs in each ligature (Ligature). The order in the Ligature Offset array defines the preference for using the ligatures. For example, if the "ffl" ligature is preferable to the "ff" ligature, then the Ligature array would list the Offset to the "ffl" Ligature table before the Offset to the "ff" Ligature table.

LigatureSet table: All ligatures beginning with the same glyph

Type	Name	Description
uint16	LigatureCount	Number of Ligature tables
Offset	Ligature [LigatureCount]	Array of Offsets to Ligature tables-from beginning of LigatureSet table-ordered by preference

For each ligature in the set, a Ligature table specifies the GlyphID of the output ligature glyph (LigGlyph); a count of the total number of component glyphs in the ligature, including the first component (CompCount); and an array of GlyphIDs for the components (Component). The array starts with the second component glyph (array index = 1) in the ligature because the first component glyph is specified in the Coverage table.

NOTE The Component array lists GlyphIDs according to the writing direction of the text. For text written right to left, the right-most glyph will be first. Conversely, for text written left to right, the left-most glyph will be first.

Example 6 at the end of this clause shows how to replace a string of glyphs with a single ligature.

Ligature table: Glyph components for one ligature

Type	Name	Description
GlyphID	LigGlyph	GlyphID of ligature to substitute
uint16	CompCount	Number of components in the ligature
GlyphID	Component [CompCount - 1]	Array of component GlyphIDs-start with the second component-ordered in writing direction

LookupType 5: Contextual substitution subtable

A Contextual Substitution (ContextSubst) subtable defines the most powerful type of glyph substitution lookup: it describes glyph substitutions in context that replace one or more glyphs within a certain pattern of glyphs.

ContextSubst subtables can be any of three formats that define a context in terms of a specific sequence of glyphs, glyph classes, or glyph sets. Each format can describe one or more input glyph sequences and one or more substitutions for each sequence.

All three formats of ContextSubst subtables specify substitution data in a SubstLookupRecord. A description of that record follows.

SubstLookupRecord

Type	Name	Description
uint16	SequenceIndex	Index into current glyph sequence-first glyph = 0
uint16	LookupListIndex	Lookup to apply to that position-zero-based

The SequenceIndex in a SubstLookupRecord must take into consideration the order in which lookups are applied to the entire glyph sequence. Because multiple substitutions may occur per context, the SequenceIndex and LookupListIndex refer to the glyph sequence after the text-processing client has applied any previous lookups. In other words, the SequenceIndex identifies the location for the substitution at the time that the lookup is to be applied. For example, consider an input glyph sequence of four glyphs. The first glyph does not have a substitute, but the middle two glyphs will be replaced with a ligature, and a single glyph will replace the fourth glyph:

- The first glyph is in position 0. No lookups will be applied at position 0, so no SubstLookupRecord is defined.
- The SubstLookupRecord defined for the ligature substitution specifies the SequenceIndex as position 1, which is the position of the first-glyph component in the ligature string. After the ligature replaces

the glyphs in positions 1 and 2, however, the input glyph sequence consists of only three glyphs, not the original four.

- To replace the last glyph in the sequence, the SubstLookupRecord defines the SequenceIndex as position 2 instead of position 3. This position reflects the effect of the ligature substitution applied before this single substitution.

NOTE This example assumes that the LookupList specifies the ligature substitution lookup before the single substitution lookup.

Context substitution Format 1

Format 1 defines the context for a glyph substitution as a particular sequence of glyphs. For example, a context could be <xyz>, <holiday>, <!?*#@>, or any other glyph sequence.

Within a context sequence, Format 1 identifies particular glyph positions (not glyph indices) as the targets for specific substitutions. When a text-processing client locates a context in a string of glyphs, it finds the lookup data for a targeted position and makes a substitution by applying the lookup data at that location.

For example, if a client is to replace the glyph string <abc> with its reverse glyph string <cba>, the input context is defined as the glyph sequence, <abc>, and the lookups defined for the context are (1) "a" to "c" and (2) "c" to "a". When a client encounters the context <abc>, the lookups are performed in the order stored. First, "c" is substituted for "a" resulting in <cbc>. Second, "a" is substituted for the "c" that has not yet been touched, resulting in <cba>.

To specify a context, a Coverage table lists the first glyph in the sequence, and a SubRule table identifies the remaining glyphs. To describe the <abc> context used in the previous example, the Coverage table lists the glyph index of the first component of the sequence-the "a" glyph. A SubRule table defines indices for the "b" and "c" glyphs.

A single ContextSubstFormat1 subtable may define more than one context glyph sequence. If different context sequences begin with the same glyph, then the Coverage table should list the glyph only once because all glyphs in the table must be unique. For example, if three contexts each start with an "s" and two start with a "t," then the Coverage table will list one "s" and one "t."

For each context, a SubRule table lists all the glyphs that follow the first glyph. The table also contains an array of SubstLookupRecords that specify the substitution lookup data for each glyph position (including the first glyph position) in the context.

All of the SubRule tables defining contexts that begin with the same first glyph are grouped together and defined in a SubRuleSet table. For example, the SubRule tables that define the three contexts that begin with an "s" are grouped in one SubRuleSet table, and the SubRule tables that define the two contexts that begin with a "t" are grouped in a second SubRuleSet table. Each glyph listed in the Coverage table must have a SubRuleSet table defining all the SubRule tables that apply to a covered glyph.

To locate a context glyph sequence, the text-processing client searches the Coverage table each time it encounters a new text glyph. If the glyph is covered, the client reads the corresponding SubRuleSet table and examines each SubRule table in the set to determine whether the rest of the context matches the subsequent glyphs in the text. If the context and text string match, the client finds the target glyph positions, applies the lookups for those positions, and completes the substitutions.

A ContextSubstFormat1 subtable contains a format identifier (SubstFormat), an Offset to a Coverage table (Coverage), a count of defined SubRuleSets (SubRuleSetCount), and an array of Offsets to the SubRuleSet tables (SubRuleSet). As mentioned, one SubRuleSet table must be defined for each glyph listed in the Coverage table.

In the SubRuleSet array, the SubRuleSet table Offsets are ordered in the Coverage Index order. The first SubRuleSet in the array applies to the first GlyphID listed in the Coverage table, the second SubRuleSet in the array applies to the second GlyphID listed in the Coverage table, and so on.

ContextSubstFormat1 subtable: Simple context glyph substitution

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of Substitution table
uint16	SubRuleSetCount	Number of SubRuleSet tables-must equal GlyphCount in Coverage table
Offset	SubRuleSet [SubRuleSetCount]	Array of Offsets to SubRuleSet tables-from beginning of Substitution table-ordered by Coverage Index

A SubRuleSet table consists of an array of Offsets to SubRule tables (SubRule), ordered by preference, and a count of the SubRule tables defined in the set (SubRuleCount). The order in the SubRule array can be critical. Consider two contexts, <abc> and <abcd>. If <abc> is first in the SubRule array, all instances of <abc> in the text-including all instances of <abcd>-will be changed. If <abcd> comes first in the array, however, only <abcd> sequences will be changed, without affecting any instances of <abc>.

SubRuleSet table: All contexts beginning with the same glyph

Type	Name	Description
uint16	SubRuleCount	Number of SubRule tables
Offset	SubRule [SubRuleCount]	Array of Offsets to SubRule tables-from beginning of SubRuleSet table-ordered by preference

A SubRule table consists of a count of the glyphs to be matched in the input context sequence (GlyphCount), including the first glyph in the sequence, and an array of glyph indices that describe the context (Input). The Coverage table specifies the index of the first glyph in the context, and the Input array begins with the second glyph (array index = 1) in the context sequence.

NOTE The Input array lists the indices in the order the corresponding glyphs appear in the text. For text written from right to left, the right-most glyph will be first; conversely, for text written from left to right, the left-most glyph will be first.

A SubRule table also contains a count of the substitutions to be performed on the input glyph sequence (SubstCount) and an array of SubstitutionLookupRecords (SubstLookupRecord). Each record specifies a position in the input glyph sequence and a LookupListIndex to the substitution lookup that is applied at that position. The array should list records in design order, or the order the lookups should be applied to the entire glyph sequence.

SubRule table: One simple context definition

Type	Name	Description
uint16	GlyphCount	Total number of glyphs in input glyph sequence-includes the first glyph
uint16	SubstCount	Number of SubstLookupRecords
GlyphID	Input [GlyphCount - 1]	Array of input GlyphIDs-start with second glyph
struct	SubstLookupRecord [SubstCount]	Array of SubstLookupRecords-in design order

Example 7 at the end of the clause shows how to use the ContextSubstFormat1 subtable to replace a sequence of three glyphs with a sequence preferred for the French language system.

Context substitution Format 2

Format 2, a more flexible format than Format 1, describes class-based context substitution. For this format, a specific integer, called a class value, must be assigned to each glyph component in all context glyph sequences. Contexts are then defined as sequences of glyph class values. More than one context may be defined at a time.

For example, suppose that a swash capital glyph should replace each uppercase letter glyph that is preceded by a space glyph and followed by a lowercase letter glyph (a glyph sequence of space - uppercase - lowercase). The set of uppercase glyphs would constitute one glyph class (Class 1), the set of lowercase glyphs would constitute a second class (Class 2), and the space glyph would constitute a third class (Class 3). The input context might be specified with a context rule (called a SubClassRule) that describes "the set of glyph strings that form a sequence of three glyph classes, one glyph from Class 3, followed by one glyph from Class 1, followed by one glyph from Class 2."

Each ContextSubstFormat2 subtable contains an Offset to a class definition table (ClassDef), which defines the glyph class values of all input contexts. Generally, a unique ClassDef table will be declared in each instance of the ContextSubstFormat2 table that is included in a font, even though several Format 2 tables could share ClassDef tables. Class assignments are fixed (the same for each position in the context), and classes are exclusive (a glyph cannot be in more than one class at a time). The output glyphs that replace the glyphs in the context sequences do not need class values because they are specified elsewhere by GlyphID.

The ContextSubstFormat2 subtable also contains a format identifier (SubstFormat) and defines an Offset to a Coverage table (Coverage). For this format, the Coverage table lists indices for the complete set of unique glyphs (not glyph classes) that may appear as the first glyph of any class-based context. In other words, the Coverage table contains the list of glyph indices for all the glyphs in all classes that may be first in any of the context class sequences. For example, if the contexts begin with a Class 1 or Class 2 glyph, then the Coverage table will list the indices of all Class 1 and Class 2 glyphs.

A ContextSubstFormat2 subtable also defines an array of Offsets to the SubClassSet tables (SubClassSet) and a count of the SubClassSet tables (SubClassSetCnt). The array contains one Offset for each class (including Class 0) in the ClassDef table. In the array, the class value defines an Offset's index position, and the SubClassSet Offsets are ordered by ascending class value (from 0 to SubClassSetCnt - 1).

For example, the first SubClassSet listed in the array contains all contexts beginning with Class 0 glyphs, the second SubClassSet contains all contexts beginning with Class 1 glyphs, and so on. If no contexts begin with a particular class (that is, if a SubClassSet contains no SubClassRule tables), then the Offset to that particular SubClassSet in the SubClassSet array will be set to NULL.

ContextSubstFormat2 subtable: Class-based context glyph substitution

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 2
Offset	Coverage	Offset to Coverage table-from beginning of Substitution table
Offset	ClassDef	Offset to glyph ClassDef table-from beginning of Substitution table
uint16	SubClassSetCnt	Number of SubClassSet tables
Offset	SubClassSet [SubClassSetCnt]	Array of Offsets to SubClassSet tables-from beginning of Substitution table-ordered by class-may be NULL

Each context is defined in a SubClassRule table, and all SubClassRules that specify contexts beginning with the same class value are grouped in a SubClassSet table. Consequently, the SubClassSet containing a context identifies a context's first class component.

Each SubClassSet table consists of a count of the SubClassRule tables defined in the SubClassSet (SubClassRuleCnt) and an array of Offsets to SubClassRule tables (SubClassRule). The SubClassRule tables are ordered by preference in the SubClassRule array of the SubClassSet.

SubClassSet subtable

Type	Name	Description
uint16	SubClassRuleCnt	Number of SubClassRule tables
Offset	SubClassRule [SubClassRuleCount]	Array of Offsets to SubClassRule tables-from beginning of SubClassSet-ordered by preference

For each context, a SubClassRule table contains a count of the glyph classes in the context sequence (GlyphCount), including the first class. A Class array lists the classes, beginning with the second class (array index = 1), that follow the first class in the context.

NOTE Text order depends on the writing direction of the text. For text written from right to left, the right-most class will be first. Conversely, for text written from left to right, the left-most class will be first.

The values specified in the Class array are the values defined in the ClassDef table. For example, a context consisting of the sequence "Class 2, Class 7, Class 5, Class 0" will produce a Class array of 7,5,0. The first class in the sequence, Class 2, is identified in the ContextSubstFormat2 table by the SubClassSet array index of the corresponding SubClassSet.

A SubClassRule also contains a count of the substitutions to be performed on the context (SubstCount) and an array of SubstLookupRecords (SubstLookupRecord) that supply the substitution data. For each position in the context that requires a substitution, a SubstLookupRecord specifies a LookupList index and a position in the input glyph sequence where the lookup is applied. The SubstLookupRecord array lists SubstLookupRecords in design order-that is, the order in which lookups should be applied to the entire glyph sequence.

SubClassRule table: Context definition for one class

Type	Name	Description
uint16	GlyphCount	Total number of classes specified for the context in the rule-includes the first class
uint16	SubstCount	Number of SubstLookupRecords
uint16	Class [GlyphCount - 1]	Array of classes-beginning with the second class-to be matched to the input glyph class sequence
struct	SubstLookupRecord [SubstCount]	Array of Substitution lookups-in design order

Example 8 at the end of this clause uses Format 2 to substitute Arabic mark glyphs for base glyphs of different heights.

Context substitution Format 3

Format 3, coverage-based context substitution, defines a context rule as a sequence of coverage tables. Each position in the sequence may define a different Coverage table for the set of glyphs that matches the context pattern. With Format 3, the glyph sets defined in the different Coverage tables may intersect, unlike Format 2 which specifies fixed class assignments (identical for each position in the context sequence) and exclusive classes (a glyph cannot be in more than one class at a time).

For example, consider an input context that contains a lowercase glyph (position 0), followed by an uppercase glyph (position 1), either a lowercase or numeral glyph (position 2), and then either a lowercase or uppercase vowel (position 3). This context requires four Coverage tables, one for each position:

- In position 0, the Coverage table lists the set of lowercase glyphs.
- In position 1, the Coverage table lists the set of uppercase glyphs.
- In position 2, the Coverage table lists the set of lowercase and numeral glyphs, a superset of the glyphs defined in the Coverage table for position 0.
- In position 3, the Coverage table lists the set of lowercase and uppercase vowels, a subset of the glyphs defined in the Coverage tables for both positions 0 and 1.

Unlike Formats 1 and 2, this format defines only one context rule at a time. It consists of a format identifier (SubstFormat), a count of the glyphs in the sequence to be matched (GlyphCount), and an array of Coverage Offsets that describe the input context sequence (Coverage).

NOTE The order of the Coverage tables listed in the Coverage array must follow the writing direction. For text written from right to left, then the right-most glyph will be first. Conversely, for text written from left to right, the left-most glyph will be first.

The subtable also contains a count of the substitutions to be performed on the input Coverage sequence (SubstCount) and an array of SubstLookupRecords (SubstLookupRecord) in design order-that is, the order in which lookups should be applied to the entire glyph sequence.

Example 9 at the end of this clause substitutes swash glyphs for two out of three glyphs in a sequence.

ContextSubstFormat3 subtable: Coverage-based context glyph substitution

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 3
uint16	GlyphCount	Number of glyphs in the input glyph sequence
uint16	SubstCount	Number of SubstLookupRecords
Offset	Coverage[GlyphCount]	Array of Offsets to Coverage table-from beginning of Substitution table-in glyph sequence order
struct	SubstLookupRecord [SubstCount]	Array of SubstLookupRecords-in design order

LookupType 6: Chaining contextual substitution subtable

A Chaining Contextual Substitution subtable (ChainContextSubst) describes glyph substitutions in context with an ability to look back and/or look ahead in the sequence of glyphs. The design of the Chaining Contextual

Substitution subtable is parallel to that of the Contextual Substitution subtable, including the availability of three formats for handling sequences of glyphs, glyph classes, or glyph sets. Each format can describe one or more backtrack, input, and lookahead sequences and one or more substitutions for each sequence.

Chaining context substitution Format 1: Simple chaining context glyph substitution

Format 1 defines the context for a glyph substitution as a particular sequence of glyphs. For example, a context could be <xyz>, <holiday>, <!?*#@>, or any other glyph sequence.

Within a context sequence, Format 1 identifies particular glyph positions (not glyph indices) as the targets for specific substitutions. When a text-processing client locates a context in a string of glyphs, it finds the lookup data for a targeted position and makes a substitution by applying the lookup data at that location.

To specify the context, the coverage table lists the first glyph in the input sequence, and the ChainSubRule subtable defines the rest. Once a covered glyph is found at position *i*, the client reads the corresponding ChainSubRuleSet table and examines each table to determine if it matches the surrounding glyphs in the glyph string. In the simplest of cases, there is a match if the string <backtrack sequence>+<input sequence>+<lookahead sequence> matches with the glyphs at position *i* - *BacktrackGlyphCount* in the text. LookupFlag values affect backtrack/lookahead sequences.

To clarify the ordering of glyph arrays for input, backtrack and lookahead sequences, the following illustration is provided. Input sequence match begins at *i* where the input sequence match begins. The backtrack sequence is ordered beginning at *i* - 1 and increases in Offset value as one moves away from *i*. The lookahead sequence begins after the input sequence and increases in logical order.

Logical order -	a	b	c	d	e	f	g	h	i	j
					i					
Input sequence -					0	1				
Backtrack sequence -	3	2	1	0						
Lookahead sequence -							0	1	2	3

If there is a match, then the client finds the target glyph positions for substitutions and completes the substitutions. Please note that (just like in the ContextSubstFormat1 subtable) these lookups are required to operate within the range of text from the covered glyph to the end of the input sequence. No substitutions can be defined for the backtracking sequence or the lookahead sequence.

Once the substitutions are complete, the client should move to the glyph position *immediately following the matched input sequence* and resume the lookup process from there.

A single ChainContextSubstFormat1 subtable may define more than one context glyph sequence. If different context sequences begin with the same glyph, then the Coverage table should list the glyph only once because all glyphs in the table must be unique. For example, if three contexts each start with an "s" and two start with a "t," then the Coverage table will list one "s" and one "t."

All of the ChainSubRule tables defining contexts that begin with the same first glyph are grouped together and defined in a ChainSubRuleSet table. For example, the ChainSubRule tables that define the three contexts that begin with an "s" are grouped in one ChainSubRuleSet table, and the ChainSubRule tables that define the two contexts that begin with a "t" are grouped in a second ChainSubRuleSet table. Each glyph listed in the Coverage table must have a ChainSubRuleSet table defining all the ChainSubRule tables that apply to a covered glyph.

A ChainContextSubstFormat1 subtable contains a format identifier (SubstFormat), an Offset to a Coverage table (Coverage), a count of defined ChainSubRuleSets (ChainSubRuleSetCount), and an array of Offsets to

the ChainSubRuleSet tables (ChainSubRuleSet). As mentioned, one ChainSubRuleSet table must be defined for each glyph listed in the Coverage table.

In the ChainSubRuleSet array, the ChainSubRuleSet table Offsets are ordered in the Coverage Index order. The first ChainSubRuleSet in the array applies to the first GlyphID listed in the Coverage table, the second ChainSubRuleSet in the array applies to the second GlyphID listed in the Coverage table, and so on.

ChainContextSubstFormat1 subtable: Simple context glyph substitution

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table-from beginning of Substitution table
uint16	ChainSubRuleSetCount	Number of ChainSubRuleSet tables-must equal GlyphCount in Coverage table
Offset	ChainSubRuleSet [ChainSubRuleSetCount]	Array of Offsets to ChainSubRuleSet tables-from beginning of Substitution table-ordered by Coverage Index

A ChainSubRuleSet table consists of an array of Offsets to ChainSubRule tables (ChainSubRule), ordered by preference, and a count of the ChainSubRule tables defined in the set (ChainSubRuleCount).

The order in the ChainSubRule array can be critical. Consider two contexts, <abc> and <abcd>. If <abc> is first in the ChainSubRule array, all instances of <abc> in the text-including all instances of <abcd>-will be changed. If <abcd> comes first in the array, however, only <abcd> sequences will be changed, without affecting any instances of <abc>.

ChainSubRuleSet table: All contexts beginning with the same glyph

Type	Name	Description
uint16	ChainSubRuleCount	Number of ChainSubRule tables
Offset	ChainSubRule [ChainSubRuleCount]	Array of Offsets to ChainSubRule tables-from beginning of ChainSubRuleSet table-ordered by preference

A ChainSubRule table consists of a count of the glyphs to be matched in the backtrack, input, and lookahead context sequences, including the first glyph in each sequence, and an array of glyph indices that describe each portion of the contexts. The Coverage table specifies the index of the first glyph in each context, and each array begins with the second glyph (array index = 1) in the context sequence.

NOTE All arrays list the indices in the order the corresponding glyphs appear in the text. For text written from right to left, the right-most glyph will be first; conversely, for text written from left to right, the left-most glyph will be first.

A ChainSubRule table also contains a count of the substitutions to be performed on the input glyph sequence (SubstCount) and an array of SubstitutionLookupRecords (SubstLookupRecord). Each record specifies a position in the input glyph sequence and a LookupListIndex to the substitution lookup that is applied at that position. The array should list records in design order, or the order the lookups should be applied to the entire glyph sequence.

ChainSubRule subtable

Type	Name	Description
uint16	BacktrackGlyphCount	Total number of glyphs in the backtrack sequence (number of glyphs to be matched before the first glyph)
GlyphID	Backtrack [BacktrackGlyphCount]	Array of backtracking GlyphID's (to be matched before the input sequence)
uint16	InputGlyphCount	Total number of glyphs in the input sequence (includes the first glyph)
GlyphID	Input [InputGlyphCount - 1]	Array of input GlyphIDs (start with second glyph)
uint16	LookaheadGlyphCount	Total number of glyphs in the look ahead sequence (number of glyphs to be matched after the input sequence)
GlyphID	LookAhead [LookAheadGlyphCount]	Array of lookahead GlyphID's (to be matched after the input sequence)
uint16	SubstCount	Number of SubstLookupRecords
struct	SubstLookupRecord [SubstCount]	Array of SubstLookupRecords (in design order)

Chaining context substitution Format 2: Class-based chaining context glyph substitution

Format 2 describes class-based chaining context substitution. For this format, a specific integer, called a class value, must be assigned to each glyph component in all context glyph sequences. Contexts are then defined as sequences of glyph class values. More than one context may be defined at a time.

To chain contexts, three classes are used in the glyph ClassDef table: Backtrack ClassDef, Input ClassDef, and Lookahead ClassDef.

The ChainContextSubstFormat2 subtable also contains a format identifier (SubstFormat) and defines an Offset to a Coverage table (Coverage). For this format, the Coverage table lists indices for the complete set of unique glyphs (not glyph classes) that may appear as the first glyph of any class-based context. In other words, the Coverage table contains the list of glyph indices for all the glyphs in all classes that may be first in any of the context class sequences. For example, if the contexts begin with a Class 1 or Class 2 glyph, then the Coverage table will list the indices of all Class 1 and Class 2 glyphs.

A ChainContextSubstFormat2 subtable also defines an array of Offsets to the ChainSubClassSet tables (ChainSubClassSet) and a count of the ChainSubClassSet tables (ChainSubClassSetCnt). The array contains one Offset for each class (including Class 0) in the ClassDef table. In the array, the class value defines an Offset's index position, and the ChainSubClassSet Offsets are ordered by ascending class value (from 0 to ChainSubClassSetCnt - 1).

If no contexts begin with a particular class (that is, if a ChainSubClassSet contains no ChainSubClassRule tables), then the Offset to that particular ChainSubClassSet in the ChainSubClassSet array will be set to NULL.

ChainContextSubstFormat2 subtable: Class-based chaining context glyph substitution

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 2
Offset	Coverage	Offset to Coverage table-from beginning of Substitution table
Offset	BacktrackClassDef	Offset to glyph ClassDef table containing backtrack sequence data-from beginning of Substitution table
Offset	InputClassDef	Offset to glyph ClassDef table containing input sequence data-from beginning of Substitution table
Offset	LookaheadClassDef	Offset to glyph ClassDef table containing lookahead sequence data-from beginning of Substitution table
uint16	ChainSubClassSetCnt	Number of ChainSubClassSet tables
Offset	ChainSubClassSet [ChainSubClassSetCnt]	Array of Offsets to ChainSubClassSet tables-from beginning of Substitution table-ordered by input class-may be NULL

Each context is defined in a ChainSubClassRule table, and all ChainSubClassRules that specify contexts beginning with the same class value are grouped in a ChainSubClassSet table. Consequently, the ChainSubClassSet containing a context identifies a context's first class component.

Each ChainSubClassSet table consists of a count of the ChainSubClassRule tables defined in the ChainSubClassSet (ChainSubClassRuleCnt) and an array of Offsets to ChainSubClassRule tables (ChainSubClassRule). The ChainSubClassRule tables are ordered by preference in the ChainSubClassRule array of the ChainSubClassSet.

ChainSubClassSet subtable

Type	Name	Description
uint16	ChainSubClassRuleCnt	Number of ChainSubClassRule tables
Offset	ChainSubClassRule [ChainSubClassRuleCount]	Array of Offsets to ChainSubClassRule tables-from beginning of ChainSubClassSet-ordered by preference

For each context, a ChainSubClassRule table contains a count of the glyph classes in the context sequence (GlyphCount), including the first class. A Class array lists the classes, beginning with the second class (array index = 1), that follow the first class in the context.

NOTE Text order depends on the writing direction of the text. For text written from right to left, the right-most class will be first. Conversely, for text written from left to right, the left-most class will be first.

The values specified in the Class array are the values defined in the ClassDef table. The first class in the sequence, Class 2, is identified in the ChainContextSubstFormat2 table by the ChainSubClassSet array index of the corresponding ChainSubClassSet.

A ChainSubClassRule also contains a count of the substitutions to be performed on the context (SubstCount) and an array of SubstLookupRecords (SubstLookupRecord) that supply the substitution data. For each position in the context that requires a substitution, a SubstLookupRecord specifies a LookupList index and a position in the input glyph sequence where the lookup is applied. The SubstLookupRecord array lists SubstLookupRecords in design order-that is, the order in which lookups should be applied to the entire glyph sequence.

ChainSubClassRule table: Chaining context definition for one class

Type	Name	Description
uint16	BacktrackGlyphCount	Total number of glyphs in the backtrack sequence (number of glyphs to be matched before the first glyph)
uint16	Backtrack [BacktrackGlyphCount]	Array of backtracking classes(to be matched before the input sequence)
uint16	InputGlyphCount	Total number of classes in the input sequence (includes the first class)
uint16	Input [InputGlyphCount - 1]	Array of input classes(start with second class; to be matched with the input glyph sequence)
uint16	LookaheadGlyphCount	Total number of classes in the look ahead sequence (number of classes to be matched after the input sequence)
uint16	LookAhead [LookAheadGlyphCount]	Array of lookahead classes(to be matched after the input sequence)
uint16	SubstCount	Number of SubstLookupRecords
struct	SubstLookupRecord [SubstCount]	Array of SubstLookupRecords (in design order)

Chaining context substitution Format 3: Coverage-based chaining context glyph substitution

Format 3 defines a chaining context rule as a sequence of Coverage tables. Each position in the sequence may define a different Coverage table for the set of glyphs that matches the context pattern. With Format 3, the glyph sets defined in the different Coverage tables may intersect, unlike Format 2 which specifies fixed class assignments (identical for each position in the backtrack, input, or lookahead sequence) and exclusive classes (a glyph cannot be in more than one class at a time).

NOTE The order of the Coverage tables listed in the Coverage array must follow the writing direction. For text written from right to left, then the right-most glyph will be first. Conversely, for text written from left to right, the left-most glyph will be first.

The subtable also contains a count of the substitutions to be performed on the input Coverage sequence (SubstCount) and an array of SubstLookupRecords (SubstLookupRecord) in design order: that is, the order in which lookups should be applied to the entire glyph sequence. (SubstLookupRecords are described next.)

ChainContextSubstFormat3 subtable: Coverage-based chaining context glyph substitution

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 3
uint16	BacktrackGlyphCount	Number of glyphs in the backtracking sequence
Offset	Coverage[BacktrackGlyphCount]	Array of Offsets to coverage tables in backtracking sequence, in glyph sequence order

uint16	InputGlyphCount	Number of glyphs in input sequence
Offset	Coverage[InputGlyphCount]	Array of Offsets to coverage tables in input sequence, in glyph sequence order
uint16	LookaheadGlyphCount	Number of glyphs in lookahead sequence
Offset	Coverage[LookaheadGlyphCount]	Array of Offsets to coverage tables in lookahead sequence, in glyph sequence order
uint16	SubstCount	Number of SubstLookupRecords
struct	SubstLookupRecord [SubstCount]	Array of SubstLookupRecords, in design order

LookupType 7: Extension substitution

This lookup provides a mechanism whereby any other lookup type's subtables are stored at a 32-bit Offset location in the 'GSUB' table. This is needed if the total size of the subtables exceeds the 16-bit limits of the various other Offsets in the 'GSUB' table. In this specification, the subtable stored at the 32-bit Offset location is termed the "extension" subtable.

ExtensionSubstFormat1 subtable

Type	Name	Description
USHORT	SubstFormat	Format identifier: Set to 1.
USHORT	ExtensionLookupType	Lookup type of subtable referenced by ExtensionOffset (i.e. the extension subtable).
ULONG	ExtensionOffset	Offset to the extension subtable, of lookup type ExtensionLookupType, relative to the start of the ExtensionSubstFormat1 subtable.

ExtensionLookupType must be set to any lookup type other than 7. All subtables in a LookupType 7 lookup must have the same ExtensionLookupType. All Offsets in the extension subtables are set in the usual way, i.e. relative to the extension subtables themselves.

When an OFF layout engine encounters a LookupType 7 Lookup table, it shall:

- Proceed as though the Lookup table's LookupType field were set to the ExtensionLookupType of the subtables.
- Proceed as though each extension subtable referenced by ExtensionOffset replaced the LookupType 7 subtable that referenced it.

Substitution lookup record

All contextual substitution subtables specify the substitution data in a Substitution Lookup Record (SubstLookupRecord). Each record contains a SequenceIndex, which indicates the position where the substitution will occur in the glyph sequence. In addition, a LookupListIndex identifies the lookup to be applied at the glyph position specified by the SequenceIndex.

The contextual substitution subtables defined in Examples 7, 8, and 9 at the end of this clause show SubstLookupRecords.

LookupType 8: Reverse chaining contextual single substitution subtable

Reverse Chaining Contextual Single Substitution subtable (ReverseChainSingleSubst) describes single glyph substitutions in context with an ability to look back and/or look ahead in the sequence of glyphs. The major difference between this and other lookup types is that processing of input glyph sequence goes from end to start. Comparing to Chaining Contextual Sustitution this format is restricted to only coverage based subtable format, input sequence could contain only single glyph and only single substitution allowed on this glyph. This substitution rule is integrated into subtable format.

This lookup type is designed specifically for the Arabic script writing styles, like nastaliq, where the shape of the glyph is determined by the following glyph, beginning at the last glyph of the "joor", or set of connected glyphs. An example of this lookup type is defined in Example 10 at the end of this clause.

Reverse chaining contextual single substitution Format 1: Coverage based reverse chaining contextual single glyph substitution.

Format 1 defines a chaining context rule as a sequence of Coverage tables. Each position in the sequence may define a different Coverage table for the set of glyphs that matches the context pattern. With Format 1, the glyph sets defined in the different Coverage tables may intersect.

NOTE Despite reverse order processing, the order of the Coverage tables listed in the Coverage array must be in logical order (follow the writing direction). The backtrack sequence is as illustrated in the LookupType 6: Chaining Contextual Substitution subtable. The input sequence is one glyph located at *i* in the logical string. The backtrack begins at *i* - 1 and increases in Offset value as one moves toward the logical beginning of the string. The lookahead sequence begins at *i* + 1 and increases in Offset value as one moves toward the logical end of the string. In the reverse chaining process *i* began at the logical end of the string and moves to the beginning.

The subtable contains Coverage table for input glyph and Coverage table arrays for lookahead and backtrack sequences, also count of output glyph indices in the Substitute array (GlyphCount), and a list of the output glyph indices (Substitute array). The Substitute array must contain the same number of glyph indices as the Coverage table. To locate the corresponding output glyph index in the Substitute array, this format uses the Coverage Index returned from the Coverage table.

ReverseChainSingleSubstFormat1 subtable: Coverage-based Reverse Chaining Contextual Single Glyph substitution.

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table - from beginning of Substitution table
uint16	BacktrackGlyphCount	Number of glyphs in the backtracking sequence
Offset	Coverage[BacktrackGlyphCount]	Array of Offsets to coverage tables in backtracking sequence, in glyph sequence order
uint16	LookaheadGlyphCount	Number of glyphs in lookahead sequence
Offset	Coverage[LookaheadGlyphCount]	Array of Offsets to coverage tables in lookahead sequence, in glyph sequence order
uint16	GlyphCount	Number of GlyphIDs in the Substitute array
GlyphID	Substitute[GlyphCount]	Array of substitute GlyphIDs-ordered by Coverage Index

6.3.4.4 GSUB – subtable examples

The rest of this clause describes and illustrates examples of all the GSUB subtables, including each of the three formats available for contextual substitutions. All the examples reflect unique parameters described below, but the samples provide a useful reference for building subtables specific to other situations.

All the examples have three columns showing hex data, source, and comments.

Example 1: GSUB header table

Example 1 shows a typical GSUB Header table definition.

Example 1

Hex Data	Source	Comments
	GSUBHeader TheGSUBHeader	GSUBHeader table definition
00010000	0x00010000	Version
000A	TheScriptList	Offset to ScriptList table
001E	TheFeatureList	Offset to FeatureList table
002C	TheLookupList	Offset to LookupList table

Example 2: SingleSubstFormat1 subtable

Example 2 illustrates the SingleSubstFormat1 subtable , which uses ranges to replace single input glyphs with their corresponding output glyphs. The indices of the output glyphs are calculated by adding a constant delta value to the indices of the input glyphs. In this example, the Coverage table has a format identifier of 1 to indicate the range format, which is used because the input glyph indices are in consecutive order in the font. The Coverage table specifies one range that contains a StartGlyphID for the "0" (zero) glyph and an EndGlyphID for the "9" glyph.

Example 2

Hex Data	Source	Comments
	SingleSubstFormat1 LiningNumeralSubtable	SingleSubst subtable definition
0001	1	SubstFormat, ranges
0006	LiningNumeralCoverage	Offset to Coverage table for input glyphs
00C0	192	DeltaGlyphID = 192, add to each input glyph index to produce output glyph index

	CoverageFormat2 LiningNumeralCoverage	Coverage table definition
0002	2	CoverageFormat, ranges
	1	RangeCount RangeRecord[0]
004E	78	Start GlyphID for numeral zero glyph
0058	87	End GlyphID for numeral nine glyph
0000	0	StartCoverageIndex first CoverageIndex = 0

Example 3: SingleSubstFormat2 subtable

Example 3 uses the SingleSubstFormat2 subtable for lists to substitute punctuation glyphs in Japanese text that is written vertically. Horizontally oriented parentheses and square brackets (the input glyphs) are replaced with vertically oriented parentheses and square brackets (the output glyphs).

The Coverage table, Format 1, identifies each input glyph index. The number of input glyph indices listed in the Coverage table matches the number of output glyph indices listed in the subtable. For correct substitution, the order of the glyph indices in the Coverage table (input glyphs) must match the order in the Substitute array (output glyphs).

Example 3

Hex Data	Source	Comments
	SingleSubstFormat2 VerticalPunctuationSubtable	SingleSubst subtable definition
0002	2	SubstFormat lists
000E	VerticalPunctuationCoverage	Offset to Coverage table
0004	4	GlyphCount, equals GlyphCount in Coverage table
0131	VerticalOpenBracketGlyph	Substitute[0], ordered by Coverage Index
0135	VerticalClosedBracketGlyph	Substitute[1]
013E	VerticalOpenParenthesisGlyph	Substitute[2]
0143	VerticalClosedParenthesisGlyph	Substitute[3]
	CoverageFormat1 VerticalPunctuationCoverage	Coverage table definition

0001	1	CoverageFormat lists
0004	4	GlyphCount
003C	HorizontalOpenBracketGlyph	GlyphArray[0], ordered by GlyphID
0040	HorizontalClosedBracketGlyph	GlyphArray[1]
004B	HorizontalOpenParenthesisGlyph	GlyphArray[2]
004F	HorizontalClosedParenthesisGlyph	GlyphArray[3]

Example 4: MultipleSubstFormat1 subtable

Example 4 uses a MultipleSubstFormat1 subtable to replace a single "ffi" ligature with three individual glyphs that form the string <ffi>. The subtable defines a format identifier of 1, an Offset to a Coverage table that specifies the glyph index of the "ffi" ligature (the input glyph), an Offset to a Sequence table that specifies the sequence of glyph indices for the <ffi> string in its substitute array (the output glyph sequence), and a count of Sequence table Offsets.

Example 4

Hex Data	Source	Comments
	MultipleSubstFormat1 FfiDecompSubtable	MultipleSubst subtable definition
0001	1	SubstFormat
0008	FfiDecompCoverage	Offset to Coverage table
0001	1	SequenceCount, equals GlyphCount in Coverage table
000E	FfiDecompSequence	Offset to Sequence[0] table
	CoverageFormat1 FfiDecompCoverage	Coverage table definition
0001	1	CoverageFormat lists
0001	1	GlyphCount
00F1	ffiGlyphID	ligature glyph
	Sequence FfiDecompSequence	Sequence table definition

0003	3	GlyphCount
001A	fGlyphID	first glyph in sequence order
001A	fGlyphID	second glyph
001D	iGlyphID	third glyph

Example 5: AlternateSubstFormat 1 subtable

Example 5 uses the AlternateSubstFormat1 subtable to replace the default ampersand glyph (input glyph) with one of two alternative ampersand glyphs (output glyph).

In this case, the Coverage table specifies the index of a single glyph, the default ampersand, because it is the only glyph covered by this lookup. The AlternateSet table for this covered glyph identifies the alternative glyphs: AltAmpersand1GlyphID and AltAmpersand2GlyphID.

In Example 5, the index position of the AlternateSet table Offset in the AlternateSet array is zero (0), which correlates with the index position (also zero) of the default ampersand glyph in the Coverage table.

Example 5

Hex Data	Source	Comments
	AlternateSubstFormat1 AltAmpersandSubtable	AlternateSubstFormat1 subtable definition
0001	1	SubstFormat
0008	AltAmpersandCoverage	Offset to Coverage table
0001	1	AlternateSetCnt, equals GlyphCount in Coverage table
000E	AltAmpersandSet	Offset to AlternateSet[0] table
	CoverageFormat1 AltAmpersandCoverage	Coverage table definition
0001	1	CoverageFormat
0001	1	GlyphCount
003A	DefaultAmpersandGlyphID	GlyphArray[0]
	AlternateSet AltAmpersandSet	AlternateSet table definition

0002	2	GlyphCount
00C9	AltAmpersand1GlyphID	Offset to Alternate[0], in arbitrary order
00CA	AltAmpersand2GlyphID	Offset to Alternate[1]

Example 6: LigatureSubstFormat1 subtable

Example 6 shows a LigatureSubstFormat1 subtable that defines data to replace a string of glyphs with a single ligature glyph. Because a LigatureSubstFormat1 subtable can specify glyph substitutions for more than one ligature, this subtable defines three ligatures: "etc," "ffi," and "fi."

The sample subtable contains a format identifier (4) and an Offset to a Coverage table. The Coverage table, which lists an index for each first glyph in the ligatures, lists indices for the "e" and "f" glyphs. The Coverage table range format is used here because the "e" and "f" glyph indices are numbered consecutively.

In the LigatureSubst subtable, LigSetCount specifies two LigatureSet tables, one for each covered glyph, and the LigatureSet array stores Offsets to them. In this array, the "e" LigatureSet precedes the "f" LigatureSet, matching the order of the corresponding first-glyph components in the Coverage table.

Each LigatureSet table identifies all ligatures that begin with a covered glyph. The sample LigatureSet table defined for the "e" glyph contains only one ligature, "etc." A LigatureSet table defined for the "f" glyph contains two ligatures, "ffi" and "fi."

The sample FLigaturesSet table has Offsets to two Ligature tables, one for "ffi" and one for "fi." The Ligature array lists the "ffi" Ligature table first to indicate that the "ffi" ligature is preferred to the "fi" ligature.

Example 6

Hex Data	Source	Comments
	LigatureSubstFormat1 LigaturesSubtable	LigatureSubstFormat1 subtable definition
0001	1	SubstFormat
000A	LigaturesCoverage	Offset to Coverage table
0002	2	LigSetCount
0014	ELigaturesSet	Offset to LigatureSet[0] table in Coverage Index order
0020	FLigaturesSet	Offset to LigatureSet[1] table
	CoverageFormat2 LigaturesCoverage	Coverage table definition
0002	2	CoverageFormat, ranges
0001	1	RangeCount

		RangeRecord[0]
0019	eGlyphID	Start, first GlyphID
001A	fGlyphID	End, last GlyphID in range
0000	0	StartCoverageIndex, coverage index of start glyphID
	LigatureSet ELigaturesSet	LigatureSet table definition all ligatures that start with e
0001	1	LigatureCount
0004	etcLigature	Offset to Ligature[0] table
	Ligature etcLigature	Ligature table definition
015B	etcGlyphID	LigGlyph, output GlyphID
0003	3	CompCount number of components
0028	tGlyphID	Component[1], second component in ligature
0017	cGlyphID	Component[2], third component in ligature
	LigatureSet FLigaturesSet	LigatureSet table definition all ligatures start with f
0002	2	LigatureCount
0006	ffiLigature	Offset to Ligature[0] table, listed first because ffi ligature is preferred to fi ligature
000E	fiLigature	Offset to Ligature[1] table
	Ligature ffiLigature	Ligature table definition
00F1	ffiGlyphID	LigGlyph, output GlyphID

0003	3	CompCount
001A	fGlyphID	Component[1], second component in ligature
001D	iGlyphID	Component[2], third component in ligature
	Ligature fiLigature	Ligature table definition
00F0	fiGlyphID	LigGlyph, output GlyphID
0002	2	CompCount
001D	iGlyphID	Component[1] second component in ligature

Example 7: ContextSubstFormat1 subtable and SubstLookupRecord

Example 7 uses a ContextSubstFormat1 subtable for glyph sequences to replace a string of three glyphs with another string. For the French language system, the subtable defines a contextual substitution that replaces the input sequence, space-dash-space, with the output sequence, thin space-dash-thin space.

The contextual substitution, called Dash Lookup in this example, contains one ContextSubstFormat1 subtable called the DashSubtable. The subtable specifies two contexts: a SpaceGlyph followed by a DashGlyph, and a DashGlyph followed by a SpaceGlyph. In each sequence, a single substitution replaces the SpaceGlyph with a ThinSpaceGlyph.

The Coverage table, labeled DashCoverage, lists two GlyphIDs for the first glyphs in the SpaceGlyph and DashGlyph sequences. One SubRuleSet table is defined for each covered glyph.

SpaceAndDashSubRuleSet lists all the contexts that begin with a SpaceGlyph. It contains an Offset to one SubRule table (SpaceAndDashSubRule), which specifies two glyphs in the context sequence, the second of which is a DashGlyph. The SubRule table contains an Offset to a SubstLookupRecord that lists the position in the sequence where the glyph substitution should occur (position 0) and the index of the SpaceToThinSpaceLookup applied there to replace the SpaceGlyph with a ThinSpaceGlyph. DashAndSpaceSubRuleSet lists all the contexts that begin with a DashGlyph. An Offset to a SubRule table (DashAndSpaceSubRule) specifies two glyphs in the context sequence, and the second one is a SpaceGlyph. The SubRule table contains an Offset to a SubstLookupRecord, which lists the position in the sequence where the glyph substitution should occur, and an index to the same lookup used in the SpaceAndDashSubRule. The lookup replaces the SpaceGlyph with a ThinSpaceGlyph.

Example 7

Hex Data	Source	Comments
	ContextSubstFormat1 DashSubtable	ContextSubstFormat1 subtable definition for Lookup[0], DashLookup
0001	1	SubstFormat
000A	DashCoverage	Offset to Coverage table

0002	2	SubRuleSetCount
0012	SpaceAndDashSubRuleSet	Offset to SubRuleSet[0], ordered by Coverage Index
0020	DashAndSpaceSubRuleSet	Offset to SubRuleSet[1]
	CoverageFormat1 DashCoverage	Coverage table definition
0001	1	CoverageFormat lists
0002	2	GlyphCount
0028	SpaceGlyph	GlyphArray[0], in numeric order
005D	DashGlyph	GlyphArray[1], dash GlyphID
	SubRuleSet SpaceAndDashSubRuleSet	SubRuleSet[0] table definition
0001	1	SubRuleCount
0004	SpaceAndDashSubRule	Offset to SubRule[0], ordered by preference
	SubRule SpaceAndDashSubRule	SubRule[0] table definition
0002	2	GlyphCount number in input sequence
0001	1	SubstCount
005D	DashGlyph	Input[1], starting with second glyph SpaceGlyph in Coverage table is first glyph SubstLookupRecord[0]
0000	0	SequenceIndex substitution at first glyph position (0)
0001	1	LookupListIndex index for SpaceToThinSpaceLookup in LookupList
	SubRuleSet DashAndSpaceSubRuleSet	SubRuleSet[0] table definition
0001	1	SubRuleCount

0004	DashAndSpaceSubRule	Offset to SubRule[0], ordered by preference
	SubRule DashAndSpaceSubRule	SubRule[0] table definition
0002	2	GlyphCount number in the input glyph sequence
0001	1	SubstCount
0028	SpaceGlyph	Input[1], starting with second glyph SubstLookupRecord definition
0001	1	SequenceIndex substitution at second glyph position(1)
0001	1	LookupListIndex for SpaceToThinSpaceLookup

Example 8: ContextSubstFormat2 subtable

Example 8 uses a ContextSubstFormat2 subtable with glyph classes to replace default mark glyphs with their alternative forms. Glyph alternatives are selected depending upon the height of the base glyph that they combine with-that is, the mark glyph used above a high base glyph differs from the mark glyph above a very high base glyph.

In the example, SetMarksHighSubtable contains a Class table that defines four glyph classes: medium-height glyphs (Class 0), all default mark glyphs (Class 1), high glyphs (Class 2), and very high glyphs (Class 3). The subtable also contains a Coverage table that lists each base glyph that functions as a first component in a context, ordered by glyph index.

Two SubClassSets are defined, one for substituting high marks and one for very high marks. No SubClassSets are specified for Class 0 and Class 1 glyphs because no contexts begin with glyphs from these classes. The SubClassSet array lists SubClassSets in numerical order, so SubClassSet 2 precedes SubClassSet 3.

Within each SubClassSet, a SubClassRule is defined. In SetMarksHighSubClassSet2, the SubClassRule table specifies two glyphs in the context, the first glyph in Class 2 (a high glyph) and the second in Class 1 (a mark glyph). The SubstLookupRecord specifies applying SubstituteHighMarkLookup at the second position in the sequence-that is, a high mark glyph will replace the default mark glyph.

In SetMarksVeryHighSubClassSet3, the SubClassRule specifies two glyphs in the context, the first in Class 3 (a very high glyph) and the second in Class 1 (a mark glyph). The SubstLookupRecord specifies applying SubstituteVeryHighMarkLookup at the second position in the sequence-that is, a very high mark glyph will replace the default mark glyph.

Example 8

Hex Data	Source	Comments
	ContextSubstFormat2 SetMarksHighSubtable	ContextSubstFormat2 subtable definition
0002	2	SubstFormat

0010	SetMarksHighCoverage	Offset to Coverage table
001C	SetMarksHighClassDef	Offset to Class Def table
0004	4	SubClassSetCnt
0000	NULL	Offset to SubClassSet[0] table, no contexts that begin with Class 0 glyphs are defined
0000	NULL	Offset to SubClassSet[1] table no contexts that begin with Class 1 glyphs are defined
0032	SetMarksHighSubClassSet2	Offset to SubClassSet[2] table for contexts that begin with Class 2 glyphs (high base glyphs)
0040	SetMarksVeryHighSubClassSet3	Offset to SubClassSet[3] table for contexts that begin with Class 3 glyphs (very high base glyphs)
	CoverageFormat1	
	SetMarksHighCoverage	Coverage table definition
0001	1	CoverageFormat, lists
0004	4	GlyphCount
0030	tahGlyphID	GlyphArray[0], high base glyph
0031	dhahGlyphID	GlyphArray[1], high base glyph
0040	cafGlyphID	GlyphArray[2], very high base glyph
0041	gafGlyphID	GlyphArray[3], very high base glyph
	ClassDefFormat2	
	SetMarksHighClassDef	Class table definition
0002	2	Class Format, ranges
0003	3	ClassRangeCount ClassRange[0] ordered by StartGlyphID for Class 2, high base glyphs
0030	tahGlyphID	Start, first Glyph ID in range
0031	dhahGlyphID	End, last Glyph ID in range
0002	2	Class ClassRange[1] for Class 3, very high base glyphs
0040	cafGlyphID	Start, first Glyph ID in the range

0041	gafGlyphID	End, last Glyph ID in the range
0003	3	Class ClassRange[2] for Class 1, mark gyphs
00D2	fathatanDefaultGlyphID	Start, first Glyph ID in range default fathatan mark
00D3	dammatanDefaultGlyphID	End, last Glyph ID in the range default dammatan mark
0001	1	Class
	SubClassSet SetMarksHighSubClassSet2	SubClassSet[2] table definition all contexts that begin with Class 2 glyphs
0001	1	SubClassRuleCnt
0004	SetMarksHighSubClassRule2	Offset to SubClassRule[0] table ordered by preference
	SubClassRule SetMarksHighSubClassRule2	SubClassRule[0] table definition, Class 2 glyph (high base) glyph followed by a Class 1 glyph (mark)
0002	2	GlyphCount
0001	1	SubstCount
0001	1	Offset to Class[1], beginning with the second Class in the context sequence (mark = Class 1) begin SubstLookupRecord array in design order SubstLookupRecord[0]
0001	1	SequenceIndex, apply substitution to position 2, a mark
0001	1	LookupListIndex
	SubClassSet SetMarksVeryHighSubClassSet3	SubClassSet[3] table definition all contexts that begin with Class 3 glyphs
0001	1	SubClassRuleCnt
0004	SetMarksVeryHighSubClassRule3	Offset to SubClassRule[0] table ordered by preference
	SubClassRule SetMarksVeryHighSubClassRule3	SubClassRule[0] table definition Class 3 glyph (very high base glyph) followed by a Class 1 glyph (mark)
0002	2	GlyphCount

0001	1	SubstCount
0001	1	Offset to Class[1], beginning with the second Class in the context sequence = marks, Class 1 begin SubstLookupRecord array in design order SubstLookupRecord[0]
0001	1	SequenceIndex, apply substitution to position 2, second glyph class (mark)
0002	2	LookupListIndex

Example 9: ContextualSubstFormat3 subtable

Example 9 uses the ContextSubstFormat3 subtable with Coverage tables to describe a context sequence of three lowercase glyphs in the pattern: any ascender or descender glyph in position 0 (zero), any x-height glyph in position 1, and any descender glyph in position 2. The overlapping sets of covered glyphs for positions 0 and 2 make Format 3 better for this context than the class-based Format 2.

In positions 0 and 2, swash versions of the glyphs replace the default glyphs. The contextual-substitution lookup is SwashLookup (LookupList index = 0), and its subtable is SwashSubtable. The SwashSubtable defines three Coverage tables: AscenderDescenderCoverage, XheightCoverage, and DescenderCoverage-one for each glyph position in the context sequence, respectively.

The SwashSubtable also defines two SubstLookupRecords: one that applies to position 0, and one for position 2. (No substitutions are applied to position 1.) The record for position 0 uses a single substitution lookup called AscDescSwashLookup to replace the current ascender or descender glyph with a swash ascender or descender glyph. The record for position 2 uses a single substitution lookup called DescSwashLookup to replace the current descender glyph with a swash descender glyph.

Example 9

Hex Data	Source	Comments
	ContextSubstFormat3 SwashSubtable	ContextSubstFormat3 subtable definition
0003	3	SubstFormat
0003	3	GlyphCount in input glyph sequence
0002	2	SubstCount
0030	AscenderDescenderCoverage	Offset to Coverage[0] table in context sequence order
004C	XheightCoverage	Offset to Coverage[1] table
006E	DescenderCoverage	Offset to Coverage[2] table SubstLookupRecord[0] in glyph position order
0000	0	SequenceIndex
0001	1	LookupListIndex, single substitution to output ascender or descender swash SubstLookupRecord[1]

0002	2	SequenceIndex
0002	2	LookupListIndex single substitution to output descender swash
	CoverageFormat1 AscenderDescenderCoverage	Coverage table definition
0001	1	CoverageFormat, lists
000C	12	GlyphCount
0033	bGlyphID	GlyphArray[0] in GlyphID order
0035	dGlyphID	GlyphArray[1]
0037	fGlyphID	GlyphArray[2]
0038	gGlyphID	GlyphArray[3]
0039	hGlyphID	GlyphArray[4]
003B	jGlyphID	GlyphArray[5]
003C	kGlyphID	GlyphArray[6]
003D	lGlyphID	GlyphArray[7]
0041	pGlyphID	GlyphArray[8]
0042	qGlyphID	GlyphArray[9]
0045	tGlyphID	GlyphArray[10]
004A	yGlyphID	GlyphArray[11]
	CoverageFormat1 XheightCoverage	Coverage table definition
0001	1	CoverageFormat, lists
000F	15	GlyphCount
0032	aGlyphID	GlyphArray[0] in GlyphID order
0034	cGlyphID	GlyphArray[1]
0036	eGlyphID	GlyphArray[2]

003A	iGlyphID	GlyphArray[3]
003E	mGlyphID	GlyphArray[4]
003F	nGlyphID	GlyphArray[5]
0040	oGlyphID	GlyphArray[6]
0043	rGlyphID	GlyphArray[7]
0044	sGlyphID	GlyphArray[8]
0045	tGlyphID	GlyphArray[9]
0046	uGlyphID	GlyphArray[10]
0047	vGlyphID	GlyphArray[11]
0048	wGlyphID	GlyphArray[12]
0049	xGlyphID	GlyphArray[13]
004B	zGlyphID	GlyphArray[14]
	CoverageFormat1 DescenderCoverage	Coverage table definition
0001	1	CoverageFormat, lists
0005	5	GlyphCount
0038	gGlyphID	GlyphArray[0] in GlyphID order
003B	jGlyphID	GlyphArray[1]
0041	pGlyphID	GlyphArray[2]
0042	qGlyphID	GlyphArray[3]
004A	yGlyphID	GlyphArray[4]

Example 10: ReverseChainSingleSubstFormat1 subtable and SubstLookupRecord

Example 10 uses a ReverseChainSingleSubstFormat1 subtable for glyph sequences to glyph with the correct form that has a thick connection to the left (thick exit). This allow the glyph to correctly connect to the letter form to the left of it.

The ThickExitCoverage table is the listing of glyphs to be matched for substitution.

The LookaheadCoverage table, labeled ThickEntryCoverage, lists four GlyphIDs for the glyph following a substitution coverage glyph. This lookahead coverage attempts to match the context that will cause the substitution to take place.

The Substitute table maps the glyphs to replace those in the ThickConnectCoverage table.

Example 10

Hex Data	Source	Comments
	ReverseChainSingleSubstFormat1 ThickConnect	ReverseChainSingleSubstFormat1 subtable definition
0001	1	SubstFormat
0068	ThickExitCoverage	Offset to Coverage table
0000	0	BacktrackGlyphCount
0000	null - not used	Offset to BacktrackCoverage[0]
0001	1	LookaheadGlyphCount
0026	ThickEntryCoverage	Offset to LookaheadCoverage[0]
000C	12	GlyphCount
00A7	BEm2	Substitute[0], ordered by Coverage Index
00B9	BEi3	Substitute[1]
00C5	JIMm3	Substitute[2]
00D4	JIMi2	Substitute[3]
00EA	SINm2	Substitute[4]
00F2	SINi2	Substitute[5]
00FD	SADm2	Substitute[6]
010D	SADi2	Substitute[7]
011B	TOEm3	Substitute[8]
012B	TOEi3	Substitute[9]
013B	AINm2	Substitute[10]

0141	AINi2	Substitute[11]
	CoverageFormat1 ThickEntryCoverage	Coverage table definition
0001	1	CoverageFormat, lists
001F	31	GlyphCount
00A5	ALEFf1	GlyphArray[0], in GlyphID order
00A9	BEm4	GlyphArray[1]
00AA	BEm5	GlyphArray[2]
00E2	DALf1	GlyphArray[3]
0167	KAFf1	GlyphArray[4]
0168	KAFfs1	GlyphArray[5]
0169	KAFm1	GlyphArray[6]
016D	KAFm5	GlyphArray[7]
016E	KAFm6	GlyphArray[8]
0170	KAFm8	GlyphArray[9]
0183	GAFf1	GlyphArray[10]
0184	GAFfs1	GlyphArray[11]
0185	GAFm1	GlyphArray[12]
0189	GAFm5	GlyphArray[13]
018A	GAFm6	GlyphArray[14]
018C	GAFm8	GlyphArray[15]
019F	LAMf1	GlyphArray[16]
01A0	LAMm1	GlyphArray[17]
01A1	LAMm2	GlyphArray[18]

01A2	LAMm3	GlyphArray[19]
01A3	LAMm4	GlyphArray[20]
01A4	LAMm5	GlyphArray[21]
01A5	LAMm6	GlyphArray[22]
01A6	LAMm7	GlyphArray[23]
01A7	LAMm8	GlyphArray[24]
01A8	LAMm9	GlyphArray[25]
01A9	LAMm10	GlyphArray[26]
01AA	LAMm11	GlyphArray[27]
01AB	LAMm12	GlyphArray[28]
01AC	LAMm13	GlyphArray[29]
01EC	HAYf2	GlyphArray[30]

	CoverageFormat1 ThickExitCoverage	Coverage table definition
0001	1	CoverageFormat, lists
000C	12	GlyphCount
00A6	BEm1	GlyphArray[0], ordered by GlyphID
00B7	BEi1	GlyphArray[1]
00C3	JIMm1	GlyphArray[2]
00D2	JIMi1	GlyphArray[3]
00E9	SINm1	GlyphArray[4]
00F1	SINi1	GlyphArray[5]
00FC	SADm1	GlyphArray[6]
010C	SADi1	GlyphArray[7]

0119	TOEm1	GlyphArray[8]
0129	TOEi1	GlyphArray[9]
013A	AINm1	GlyphArray[10]
0140	AINi1	GlyphArray[11]

6.3.5 JSTF – The justification table

6.3.5.1 JSTF table overview

The Justification table (JSTF) provides font developers with additional control over glyph substitution and positioning in justified text. Text-processing clients now have more options to expand or shrink word and glyph spacing so text fills the specified line length.

When justifying text, the text-processing client distributes the characters in each line to completely fill the specified line length. Whether removing space to fit more characters in the line or adding more space to spread the characters, justification can produce large gaps between words, cramped or extended glyph spacing, uneven line break patterns, and other jarring visual effects. For example:

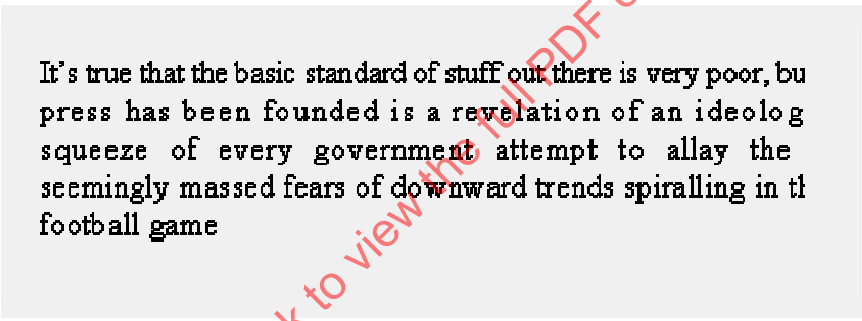

Figure 39 – Poorly justified text

To offset these effects, text-processing clients have used justification algorithms that redistribute the space with a series of glyph spacing adjustments that progress from least to most obvious. Typically, the client will begin by expanding or compressing the space between words. If these changes aren't enough or look distracting, the client might hyphenate the word at the end of the line or adjust the space between glyphs in one or more lines.

To disguise spacing inconsistencies so they won't disrupt the flow of text for a reader, the font developer can use the JSTF table to enable or disable individual glyph substitution and positioning actions that apply to specific scripts, language systems, and glyphs in the font.

For instance, a ligature glyph can replace multiple glyphs, shortening the line of text with an unobtrusive, localized adjustment (see Figure 40). Font-specific positioning changes can be applied to particular glyphs in a text line that combines two or more fonts. Other options include repositioning the individual glyphs in the line, expanding the space between specific pairs of glyphs, and decreasing the spacing within particular glyph sequences.

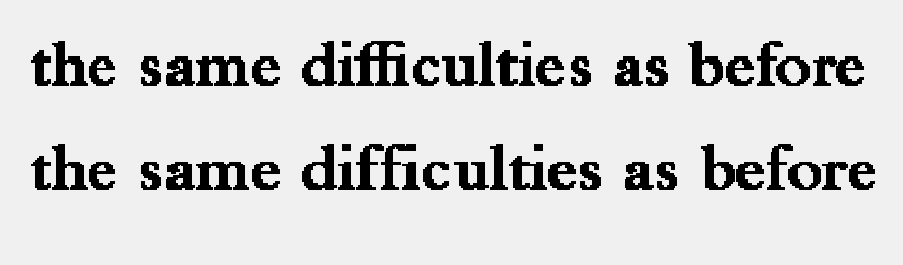

Figure 40 – JSTF shortens the top line of this example by using the "ffi" ligature

The font designer or developer defines JSTF data as prioritized suggestions. Each suggestion lists the particular actions that the client can use to adjust the line of text. Justification actions may apply to both vertical and horizonal text.

6.3.5.2 Table organization and structure

The JSTF table organizes data by script and language system, as do the GSUB and GPOS tables. The JSTF table begins with a header that lists scripts in an array of JstfScriptRecords (see Figure 41). Each record contains a ScriptTag and an Offset to a JstfScript table that contains script and language-specific data:

- A default justification language system table (DefJstfLangSys) defines script-specific data that applies to the entire script in the absence of any language-specific information.
- A justification language system table (JstfLangSys) stores the justification data for each language system.

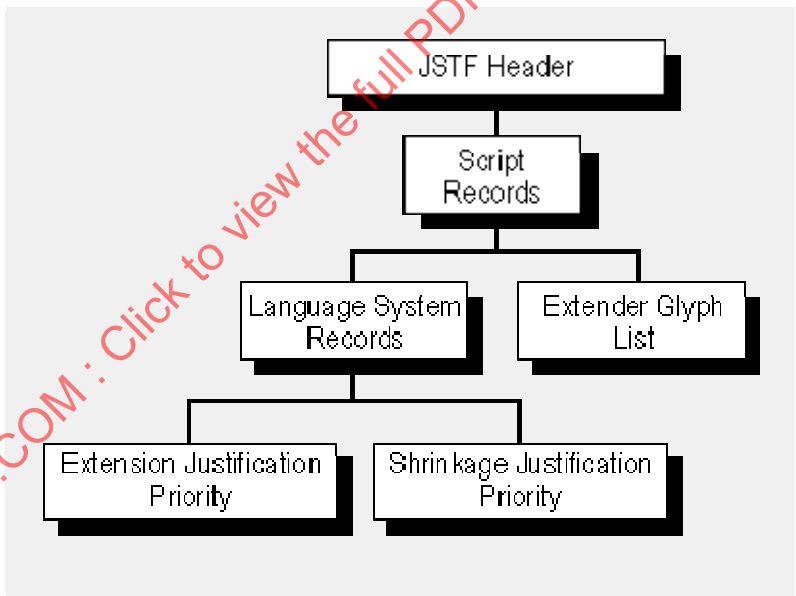

Figure 41 – High-level organization of JSTF table

A JstfLangSys table contains a list of justification suggestions. Each suggestion consists of a list of GSUB or GPOS LookupList indices to lookups that may be enabled or disabled to add or remove space in the line of text. In addition, each suggestion can include a set of dedicated justification lookups with maximum adjustment values to extend or shrink the amount of space.

The font developer prioritizes suggestions based on how they affect the appearance and function of the text line, and the client applies the suggestions in that order. Low-numbered (high-priority) suggestions correspond to "least bad" options.

Each script also may supply a list of extender glyphs, such as kashidas in Arabic. A client may use the extender glyphs in addition to the justification suggestions.

A client begins justifying a line of text only after implementing all selected GSUB and GPOS features for the string. Starting with the lowest-numbered suggestion, the client enables or disables the lookups specified in the JSTF table, reassembles the lookups in the LookupList order, and applies them to each glyph in the string one after another. If the line still is not the correct length, the client processes the next suggestion in ascending order of priority. This continues until the line length meets the justification requirements.

NOTE If any JSTF suggestion at any priority level modifies a GSUB or GPOS lookup that was previously applied to the glyph string, then the text processing client must apply the JSTF suggestion to an unmodified version of the glyph string.

The rest of this clause describes the tables and records used by the JSTF table for scripts and language systems:

- Script information includes the JstfScript table (plus its associated JstfLangSysRecords) and the ExtenderGlyph table.
- Language system information includes the JstfLangSys table, JstfPriority table (and its associated JstfDataRecord), the JstfModList table, and the JstfMax table.

JSTF header

The JSTF table begins with a header that contains a version number for the table (Version), a count of the number of scripts used in the font (JstfScriptCount), and an array of records (JstfScriptRecord). Each record contains a script tag (JstfScriptTag) and an Offset to a JstfScript table (JstfScript).

NOTE The JstfScriptTags must correspond with the ScriptTags listed in the GSUB and GPOS tables.

Example 1 at the end of this clause shows a JSTF Header table and JstfScriptRecord.

JSTF header

Type	Name	Description
fixed32	Version	Version of the JSTF table-initially set to 0x00010000
uint16	JstfScriptCount	Number of JstfScriptRecords in this table
struct	JstfScriptRecord[JstfScriptCount]	Array of JstfScriptRecords-in alphabetical order, by JstfScriptTag

JstfScriptRecord

Type	Name	Description
Tag	JstfScriptTag	4-byte JstfScript identification
Offset	JstfScript	Offset to JstfScript table-from beginning of JSTF Header

Justification script table

A Justification Script (JstfScript) table describes the justification information for a single script. It consists of an Offset to a table that defines extender glyphs (ExtenderGlyph), an Offset to a default justification table for the script (DefJstfLangSys), and a count of the language systems that define justification data (JstfLangSysCount).

If a script uses the same justification information for all language systems, the font developer defines only the DefJstfLangSys table and sets the JstfLangSysCount to zero (0). However, if any language system has unique justification suggestions, JstfLangSysCount will be a positive value, and the JstfScript table must include an array of records (JstfLangSysRecord), one for each language system. Each JstfLangSysRecord contains a language system tag (JstfLangSysTag) and an Offset to a justification language system table (JstfLangSys). In the JstfLangSysRecord array, records are ordered alphabetically by JstfLangSysTag.

NOTE No JstfLangSysRecord is defined for the default script data; the data is stored in the DefJstfLangSys table instead.

Example 2 at the end of the clause shows a JstfScript table for the Arabic script and a JstfLangSysRecord for the Farsi language system.

JstfScript table

Type	Name	Description
Offset	ExtenderGlyph	Offset to ExtenderGlyph table-from beginning of JstfScript table-may be NULL
Offset	DefJstfLangSys	Offset to Default JstfLangSys table-from beginning of JstfScript table-may be NULL
uint16	JstfLangSysCount	Number of JstfLangSysRecords in this table-may be zero (0)
struct	JstfLangSysRecord [JstfLangSysCount]	Array of JstfLangSysRecords-in alphabetical order, by JstfLangSysTag

JstfLangSysRecord

Type	Name	Description
Tag	JstfLangSysTag	4-byte JstfLangSys identifier
Offset	JstfLangSys	Offset to JstfLangSys table-from beginning of JstfScript table

Extender glyph table

The Extender Glyph table (ExtenderGlyph) lists indices of glyphs, such as Arabic kashidas, that a client may insert to extend the length of the line for justification. The table consists of a count of the extender glyphs for the script (GlyphCount) and an array of extender glyph indices (ExtenderGlyph), arranged in increasing numerical order.

Example 2 at the end of this clause shows an ExtenderGlyph table for Arabic kashida glyphs.

ExtenderGlyph table

Type	Name	Description
uint16	GlyphCount	Number of Extender Glyphs in this script
GlyphID	ExtenderGlyph[GlyphCount]	GlyphIDs-in increasing numerical order

Justification Language System table

The Justification Language System (JstfLangSys) table contains an array of justification suggestions, ordered by priority. A text-processing client doing justification should begin with the suggestion that has a zero (0) priority, and then-as necessary-apply suggestions of increasing priority until the text is justified.

The font developer defines the number and the meaning of the priority levels. Each priority level stands alone; its suggestions are not added to the previous levels. The JstfLangSys table consists of a count of the number of priority levels (JstfPriorityCnt) and an array of Offsets to Justification Priority tables (JstfPriority), stored in priority order. Example 2 at the end of the clause shows how to define a JstfLangSys table.

JstfLangSys table

Type	Name	Description
uint16	JstfPriorityCnt	Number of JstfPriority tables
Offset	JstfPriority[JstfPriorityCnt]	Array of Offsets to JstfPriority tables-from beginning of JstfLangSys table-in priority order

Justification Priority table

A Justification Priority (JstfPriority) table defines justification suggestions for a single priority level. Each priority level specifies whether to enable or disable GSUB and GPOS lookups or apply text justification lookups to shrink and extend lines of text.

JstfPriority has Offsets to four tables with line shrinkage data: two are JstfGSUBModList tables for enabling and disabling glyph substitution lookups, and two are JstfGPOSModList tables for enabling and disabling glyph positioning lookups. Offsets to JstfGSUBModList and JstfGPOSModList tables also are defined for line extension.

Example 3 at the end of this clause demonstrates two JstfPriority tables for two justification suggestions.

JstfPriority table

Type	Name	Description
Offset	ShrinkageEnableGSUB	Offset to Shrinkage Enable JstfGSUBModList table-from beginning of JstfPriority table-may be NULL
Offset	ShrinkageDisableGSUB	Offset to Shrinkage Disable JstfGSUBModList table-from beginning of JstfPriority table-may be NULL
Offset	ShrinkageEnableGPOS	Offset to Shrinkage Enable JstfGPOSModList table-from beginning of JstfPriority table-may be NULL
Offset	ShrinkageDisableGPOS	Offset to Shrinkage Disable JstfGPOSModList table-from beginning of JstfPriority table-may be NULL
Offset	ShrinkageJstfMax	Offset to Shrinkage JstfMax table-from beginning of JstfPriority table -may be NULL
Offset	ExtensionEnableGSUB	Offset to Extension Enable JstfGSUBModList table-may be NULL

Offset	ExtensionDisableGSUB	Offset to Extension Disable JstfGSUBModList table-from beginning of JstfPriority table-may be NULL
Offset	ExtensionEnableGPOS	Offset to Extension Enable JstfGSUBModList table-may be NULL
Offset	ExtensionDisableGPOS	Offset to Extension Disable JstfGSUBModList table-from beginning of JstfPriority table-may be NULL
Offset	ExtensionJstfMax	Offset to Extension JstfMax table-from beginning of JstfPriority table -may be NULL

Justification Modification List tables

The Justification Modification List tables (JstfGSUBModList and JstfGPOSModList) contain lists of indices into the lookup lists of either the GSUB or GPOS tables. The client can enable or disable the lookups to justify text. For example, to increase line length, the client might disable a GSUB ligature substitution.

Each JstfModList table consists of a count of Lookups (LookupCount) and an array of lookup indices (LookupIndex).

To justify a line of text, a text-processing client enables or disables the specified lookups in a JstfModList table, reassembles the lookups in the LookupList order, and applies them to each glyph in the string one after another.

NOTE If any JSTF suggestion at any priority level modifies a GSUB or GPOS lookup previously applied to the glyph string, then the text-processing client must apply the JSTF suggestion to an unmodified version of the glyph string.

Example 3 at the end of this clause shows JstfGSUBModList and JstfGPOSModList tables with data for shrinking and extending text line lengths.

JstfGSUBModList table

Type	Name	Description
uint16	LookupCount	Number of lookups for this modification
uint16	GSUBLookupIndex[LookupCount]	Array of LookupIndex identifiers in GSUB-in increasing numerical order

JstfGPOSModList table

Type	Name	Description
uint16	LookupCount	Number of lookups for this modification
uint16	GPOSLookupIndex[LookupCount]	Array of LookupIndex identifiers in GPOS-in increasing numerical order

Justification Maximum table

A Justification Maximum table (JstfMax) consists of an array of Offsets to justification lookups (Lookup) and a count of the defined lookups (Lookup). JstfMax lookups typically are located after the JstfMax table in the font definition.

JstfMax tables have the same format as lookup tables and subtables in the GPOS table, but the JstfMax lookups reside in the JSTF table and contain justification data only. The lookup data might specify a single adjustment value for positioning all glyphs in the script, or it might specify more elaborate adjustments, such as different values for different glyphs or special values for specific pairs of glyphs.

NOTE All GPOS lookup types except contextual positioning lookups may be defined in a JstfMax table.

JstfMax lookup values are defined in GPOS ValueRecords and may be specified for any advance or placement position, whether horizontal or vertical. These values define the maximum shrinkage or extension allowed per glyph. To justify text, a text-processing client may choose to adjust a glyph's positioning by any amount from zero (0) to the specified maximum.

Example 4 at the end of this clause shows a JstfMax table. It defines a justification lookup to change the size of the word space glyph to extend line lengths.

JstfMax table

Type	Name	Description
uint16	LookupCount	Number of lookup Indices for this modification
Offset	Lookup[LookupCount]	Array of Offsets to GPOS-type lookup tables-from beginning of JstfMax table-in design order

6.3.5.3 JSTF table examples

The rest of this clause describes examples of all the JSTF table formats. All the examples reflect unique parameters described below, but the samples provide a useful reference for building tables specific to other situations.

The examples have three columns showing hex data, source, and comments.

Example 1: JSTF header table and JstfScriptRecord

Example 1 demonstrates how a script is defined in the JSTF Header with a JstfScriptRecord that identifies the script and references its JstfScript table.

Example 1

Hex Data	Source	Comments
	JSTFHeader TheJSTFHeader	JSTFHeader table definition
00010000	0x00010000	Version
0001	1	JstfScriptCount JstfScriptRecord[0]

74686169	"thai"	JstfScriptTag
000C	ThaiScript	Offset to JstfScript table

Example 2: JstfScript table, ExtenderGlyph table, JstfLangSysRecord, and JstfLangSys table

Example 2 shows a JstfScript table for the Arabic script and the tables it references. The DefJstfLangSys table defines justification data to apply to the script in the absence of language-specific information. In the example, the table lists two justification suggestions in priority order.

JstfScript also supplies language-specific justification data for the Farsi language. The JstfLangSysRecord identifies the language and references its JstfLangSys table. The FarsiJstfLangSys lists one suggestion for justifying Farsi text.

The ExtenderGlyph table in JstfScript lists the indices of all the extender glyphs used in the script.

Example 2

Hex Data	Source	Comments
	JstfScript ArabicScript	JstfScript table definition
000C	ArabicExtenders	ExtenderGlyph
0012	ArabicDefJstfLangSys	Offset to DefJstfLangSys table
0001	1	JstfLangSysCount JstfLangSysRecord[0]
50455220	"FAR "	JstfLangSysTag
0018	FarsiJstfLangSys	JstfLangSys
	ExtenderGlyph ArabicExtenders	ExtenderGlyph table definition
0002	2	GlyphCount
01D3	TatweelGlyphID	ExtenderGlyph[0]
01D4	LongTatweelGlyphID	ExtenderGlyph[1]
	JstfLangSys ArabicDefJstfLangSys	JstfLangSys table definition
0002	2	JstfPriorityCnt

000A	ArabicScriptJstfPriority1	Offset to JstfPriority[0] table
001E	ArabicScriptJstfPriority2	Offset to JstfPriority[1] table
	JstfLangSys FarsiJstfLangSys	JstfLangSys table definition
0001	1	JstfPriorityCnt
002C	FarsiLangJstfPriority1	Offset to JstfPriority[0] table

Example 3: JstfPriority table, JstfGSUBModList table, and JstfGPOSModList table

Example 3 shows the JstfPriority and JstfModList table definitions for two justification suggestions defined in priority order. The first suggestion uses ligature substitution to shrink the lengths of text lines, and it extends line lengths by replacing ligatures with their individual glyph components. Other lookup actions are not recommended at this priority level and are set to NULL. The associated JstfModList tables enable and disable three substitution lookups.

The second suggestion enables glyph kerning to reduce line lenths and disables glyph kerning to extend line lengths. Each action uses three lookups. This suggestion also includes a JstfMax table to extend line lengths, called WordSpaceExpandMax, which is described in Example 4.

Example 3

Hex Data	Source	Comments
	JstfPriority USEnglishFirstJstfPriority	JstfPriority table definition
0028	EnableGSUBLookupsToShrink	Offset to ShrinkageEnableGSUB JstfGSUBModList table
0000	NULL	Offset to ShrinkageDisableGSUB JstfGSUBModList table
0000	NULL	Offset to ShrinkageEnableGPOS JstfGPOSModList table
0000	NULL	Offset to ShrinkageDisableGPOS JstfGPOSModList table
0000	NULL	Offset to Shrinkage JstfMax table
0000	NULL	Offset to ExtensionEnableGSUB, JstfGSUBModList table
0038	DisableGSUBLookupsToExtend	Offset to ExtensionDisableGSUB JstfGSUBModList table
0000	NULL	Offset to ExtensionEnableGPOS JstfGPOSModList table
0000	NULL	Offset to ExtensionDisableGPOS JstfGPOSModList table

0000	NULL	Offset to Extension JstfMax table
	JstfPriority USEnglishSecondJstfPriority	JstfPriority table definition
0000	NULL	Offset to ShrinkageEnableGSUB JstfGSUBModList table
0000	NULL	Offset to ShrinkageDisableGSUB JstfGSUBModList table
0000	NULL	Offset to ShrinkageEnableGPOS JstfGPOSModList table
001C	DisableGPOSLookupsToShrink	Offset to ShrinkageDisableGPOS JstfGPOSModList table
0000	NULL	Offset to Shrinkage JstfMax table
0000	NULL	Offset to ExtensionEnableGSUB JstfGSUBModList table
0000	NULL	Offset to ExtensionDisableGSUB JstfGSUBModList table
002C	EnableGPOSLookupsToExtend	Offset to ExtensionEnableGPOS JstfGPOSModList table
0000	NULL	Offset to ExtensionDisableGPOS JstfGPOSModList table
0000	NULL	Offset to Extension JstfMax table
	JstfGSUBModList EnableGSUBLookupsToShrink	JstfGSUBModList table definition, enable three ligature substitution lookups
0003	3	LookupCount
002E	46	LookupIndex[0]
0035	53	LookupIndex[1]
0063	99	LookupIndex[2]
	JstfGPOSModList DisableGPOSLookupsToShrink	JstfGPOSModList table definition, disable three tight kerning lookups
0003	3	LookupCount
006C	108	LookupIndex[0]
006E	110	LookupIndex[1]

0070	112	LookupIndex[2]
	JstfGSUBModList DisableGSUBLookupsToExtend	JstfGSUBModList table definition, disable three ligature substitution lookups
0003	3	LookupCount
002E	46	LookupIndex[0]
0035	53	LookupIndex[1]
0063	99	LookupIndex[2]
	JstfGPOSModList EnableGPOSLookupsToExtend	JstfGPOSModList table definition enable three tight kerning lookups
0003	3	LookupCount
006C	108	LookupIndex[0]
006E	110	LookupIndex[1]
0070	112	LookupIndex[2]

Example 4: JstfMax table

The JstfMax table in Example 4 defines a lookup to expand the advance width of the word space glyph and extend line lengths. The lookup definition is identical to the SinglePos lookup type in the GPOS table although it is enabled only when justifying text. The ValueRecord in the WordSpaceExpand lookup subtable specifies an XAdvance adjustment of 360 units, which is the maximum value the font developer recommends for acceptable text rendering. The text-processing client may implement the lookup using any value between zero and the maximum.

Example 4

Hex Data	Source	Comments
	JstfMax WordSpaceExpandMax	JstfMax table definition
0001	1	LookupCount
0004	WordSpaceExpandLookup	Offset to Jstf Lookup[0] table
	Lookup WordSpaceExpandLookup	Jstf Lookup table definition

0001	1	LookupType, SinglePos Lookup
0000	0x0000	LookupFlag
0001	1	SubTableCount
0008	WordSpaceExpandSubtable	Offset to Subtable[0], SinglePos subtable
	SinglePosFormat1 WordSpaceExpandSubtable	SinglePos subtable definition
0001	1	PosFormat
0008	WordSpaceCoverage	Offset to Coverage table
0004	0x0004	ValueFormat, XAdvance only
0168	360	Value XAdvance value in Jstf, this is a max value, expand word space from zero to this amount
	CoverageFormat1 WordSpaceCoverage	Coverage table definition
0001	1	CoverageFormat
0001	1	GlyphCount
0022	WordSpaceGlyphID	GlyphArray[0]

6.4 Layout tag registry

OFF Layout tags are 4-byte character strings that identify the scripts, language systems, features and baselines in a OFF Layout font. The registry establishes conventions for naming and using these tags. Registered tags have a specific meaning and convey precise information to developers and text-processing clients of OFF Layout. Font developers are encouraged to use registered tags to assure compatibility and ease of use across fonts, applications, and operating systems. Additional tags can be added to the tag registry when necessary.

6.4.1 Scripts tags

Script tags correspond to the contiguous character code ranges in Unicode.

All tags are 4-byte character strings composed of a limited set of ASCII characters in the 0x20-0x7E range. A script tag can consist of four or fewer lowercase letters. If a script tag consists less than four lowercase letters, the letters are followed by the requisite number of spaces (0x20), each consisting of a single byte.

Script	Script Tag
Arabic	arab
Armenian	armn
Balinese	bali
Bengali	beng
Bengali v.2	bng2
Bopomofo	bopo
Braille	brai
Buginese	bugi
Buhid	buhd
Byzantine Music	byzm
Canadian Syllabics	cans
Carian	cari
Cham	cham
Cherokee	cher
CJK Ideographic	hani
Coptic	copt
Cypriot Syllabary	cprt
Cyrillic	cyrl
Default	DFLT
Deseret	dsrt
Devanagari	deva
Devanagari v.2	dev2

Ethiopic	ethi
Georgian	geor
Glagolitic	glag
Gothic	goth
Greek	grek
Gujarati	gujr
Gujarati v.2	gjr2
Gurmukhi	guru
Gurmukhi v.2	gur2
Hangul	hang
Hangul Jamo	jamo
Hanunoo	hano
Hebrew	hebr
Hiragana	kana
Javanese	java
Kannada	knda
Kannada v.2	knd2
Katakana	kana
Kayah Li	kali
Kharosthi	khar
Khmer	khmr
Lao	lao
Latin	latn

Lepcha	lepc
Limbu	limb
Linear B	linb
Lycian	lyci
Lydian	lydi
Malayalam	mlym
Malayalam v.2	mlm2
Mathematical Alphanumeric Symbols	math
Mongolian	mong
Musical Symbols	musc
Myanmar	mymr
New Tai Lue	talu
N'Ko	nko
Ogham	ogam
Ol Chiki	olck
Old Italic	ital
Old Persian Cuneiform	xpeo
Oriya	orya
Oriya v.2	ory2
Osmanya	osma
Phags-pa	phag
Phoenician	phnx
Rejang	rjng

Runic	runr
Saurashtra	saur
Shavian	shaw
Sinhala	sinh
Sumero-Akkadian Cuneiform	xsux
Sundanese	sund
Syloti Nagri	sylo
Syriac	syrc
Tagalog	tglg
Tagbanwa	tagb
Tai Le	tale
Tamil	taml
Tamil v.2	tml2
Telugu	telu
Telugu v.2	tel2
Thaana	thaa
Thai	thai
Tibetan	tibt
Tifinagh	tfng
Ugaritic Cuneiform	ugar
Vai	vai
Yi	yi

When the ScriptList table is searched for a script, and no entry is found, and there is an entry for the 'DFLT' script, then this entry must be used. Furthermore, the Script table for the 'DFLT' script must have a non-NULL DefaultLangSys and a LangSysCount equal to 0; in other words, there is only a default language for the default script.

6.4.2 Language tags

Language system tags identify the language systems supported in an OFF Layout font data. Windows platform uses the standard language system tag names. What is meant by a "language system" in this context is a set of typographic conventions for how text in a given script should be presented. Such conventions may be associated with particular languages, with particular genres of usage, with different publications, and other such factors. For example, particular glyph variants for certain characters may be required for particular languages, or for phonetic transcription or mathematical notation.

In principle, a given set of conventions may be shared across multiple scenarios. For instance, two different languages (perhaps unrelated) may happen to follow the same conventions. Language system tags can be registered on a perceived-need basis, however; as a result, there is no guarantee that each tag represents a distinct and unique set of conventions. Tags can, however, be registered with the intent of representing conventions that apply to multiple languages. In such cases, the documented description for the tag should reflect that intent.

It should also be noted that there may be more than one set of typographic conventions that apply to a given language.

Therefore, in several respects, language system tags do not correspond in a one-to-one manner with languages. Even so, many registered tags are intended to represent typographic conventions for a particular language. For cases in which a correlation exists between a tag and one or more languages, the language identities are documented here by reference to ISO 639-2 and ISO 639-3.

If information is available to an application declaring the language of text content, then the application may make use of that to select a default language system tag to be applied when displaying that text. It is preferable, however, to give users control over the choice of language system tag to be used. (Depending on the application scenario, such control may be given to content authors, to content readers, or to both.)

> NOTE: ISO 639-2 provides identifiers for individual languages as well as for certain collections of languages. ISO 639-3 provides identifiers for a far more comprehensive set of individual languages, though not for collections. Entities in ISO 639 that are referenced here may include any of the individual languages covered in ISO 639-2 or ISO 639-3, or to any of the collections covered in ISO 639-2.

All tags are 4-byte character strings composed of a limited set of ASCII characters in the 0x20-0x7E range. If a language system tag consists of three or less lowercase letters, the letters are followed by the requisite number of spaces (0x20), each consisting of a single byte.

Language System	Language System Tag	Corresponding ISO 639 ID (if applicable)
Abaza	ABA	abq
Abkhazian	ABK	abk
Adyghe	ADY	ady
Afrikaans	AFK	afr
Afar	AFR	aar
Agaw	AGW	ahg
Alsatian	ALS	gsw

Altai	ALT	atv, alt
Amharic	AMH	amh
Phonetic transcription – Americanist conventions	APPH	
Arabic	ARA	ara
Aari	ARI	aiw
Arakanese	ARK	mhv, rmz, rki
Assamese	ASM	asm
Athapaskan	ATH	
Avar	AVR	ava
Awadhi	AWA	awa
Aymara	AYM	aym
Azeri	AZE	aze
Badaga	BAD	bfq
Baghelkhandi	BAG	bfy
Balkar	BAL	krc
Baule	BAU	bci
Berber	BBR	
Bench	BCH	bcq
Bible Cree	BCR	
Belarussian	BEL	bel
Bemba	BEM	bem
Bengali	BEN	ben
Bulgarian	BGR	bul
Bhili	BHI	bhi
Bhojpuri	BHO	bho
Bikol	BIK	bik
Bilen	BIL	byn
Blackfoot	BKF	bla
Balochi	BLI	bal
Balante	BLN	bjt, ble

Balti	BLT	bft
Bambara	BMB	bam
Bamileke	BML	
Bosnian	BOS	bos
Breton	BRE	bre
Brahui	BRH	brh
Braj Bhasha	BRI	bra
Burmese	BRM	mya
Bashkir	BSH	bak
Beti	BTI	btb
Catalan	CAT	cat
Cebuano	CEB	ceb
Chechen	CHE	che
Chaha Gurage	CHG	sgw
Chattisgarhi	CHH	hne
Chichewa	CHI	nya
Chukchi	CHK	ckt
Chipewyan	CHP	chp
Cherokee	CHR	chr
Chuvash	CHU	chv
Comorian	CMR	swb, wlc, wni, zdj
Coptic	COP	cop
Cree	CRE	cre
Carrier	CRR	crx, caf
Crimean Tatar	CRT	crh
Church Slavonic	CSL	chu
Czech	CSY	ces
Danish	DAN	dan
Dargwa	DAR	dar
Woods Cree	DCR	cwd

German	DEU	deu
Dogri	DGR	doi
Dhivehi	DHV (deprecated)	dv
Dhivehi	DIV	div
Djerma	DJR	dje
Dangme	DNG	ada
Dinka	DNK	din
Dari	DRI	prs
Dungan	DUN	dng
Dzongkha	DZN	dzo
Ebira	EBI	igb
Eastern Cree	ECR	crj, crl
Edo	EDO	bin
Efik	EFI	efi
Greek	ELL	ell
English	ENG	eng
Erzya	ERZ	myv
Spanish	ESP	spa
Estonian	ETI	est
Basque	EUQ	eus
Evenki	EVK	evn
Even	EVN	eve
Ewe	EWE	ewe
French Antillean	FAN	acf
Farsi	FAR	fas
Finnish	FIN	fin
Fijian	FJI	fiji
Flemish	FLE	
Forest Nenets	FNE	enf
Fon	FON	fon

Faroese	FOS	fao
French	FRA	fra
Frisian	FRI	fry
Friulian	FRL	fur
Futa	FTA	fuf
Fulani	FUL	ful
Ga	GAD	gaa
Gaelic	GAE	gla
Gagauz	GAG	gag
Galician	GAL	glg
Garshuni	GAR	
Garhwali	GAW	gbm
Ge'ez	GEZ	gez
Gilyak	GIL	niv
Gumuz	GMZ	guk
Gondi	GON	gon
Greenlandic	GRN	kal
Garo	GRO	grt
Guarani	GUA	grn
Gujarati	GUJ	guj
Haitian	HAI	hat
Halam	HAL	flm
Harauti	HAR	hoj
Hausa	HAU	hau
Hawaiin	HAW	haw
Hammer-Banna	HBN	amf
Hiligaynon	HIL	hil
Hindi	HIN	hin
High Mari	HMA	mrj
Hindko	HND	hno

Ho	HO	hoc
Harari	HRI	har
Croatian	HRV	hrv
Hungarian	HUN	hun
Armenian	HYE	hye
Igbo	IBO	ibo
Ijo	IJO	ijc
Ilokano	ILO	ilo
Indonesian	IND	ind
Ingush	ING	inh
Inuktitut	INU	iku
Phonetic transcription – IPA conventions	IPPH	
Irish	IRI	gle
Irish Traditional	IRT	gle
Icelandic	ISL	isl
Inari Sami	ISM	smn
Italian	ITA	ita
Hebrew	IWR	heb
Javanese	JAV	jav
Yiddish	JII	yid
Japanese	JAN	jpn
Judezmo	JUD	lad
Jula	JUL	dyu
Kabardian	KAB	kbd
Kachchi	KAC	kfr
Kalenjin	KAL	kln
Kannada	KAN	kan
Karachay	KAR	krc
Georgian	KAT	kat
Kazakh	KAZ	kaz

Kebena	KEB	ktb
Khutsuri Georgian	KGE	kat
Khakass	KHA	kjh
Khanty-Kazim	KHK	kca
Khmer	KHM	khm
Khanty-Shurishkar	KHS	kca
Khanty-Vakhi	KHV	kca
Khowar	KHW	khw
Kikuyu	KIK	kik
Kirghiz	KIR	kir
Kisii	KIS	kqs, kss
Kokni	KKN	kex
Kalmyk	KLM	xal
Kamba	KMB	kam
Kumaoni	KMN	kfy
Komo	KMO	kmw
Komso	KMS	kxc
Kanuri	KNR	kau
Kodagu	KOD	kfa
Korean Old Hangul	KOH	okm
Konkani	KOK	kok
Kikongo	KON	ktu
Komi-Permyak	KOP	koi
Korean	KOR	kor
Komi-Zyrian	KOZ	kpv
Kpelle	KPL	kpe
Krio	KRI	kri
Karakalpak	KRK	kaa
Karelian	KRL	krl
Karaim	KRM	kdr

Karen	KRN	kar
Koorete	KRT	kqy
Kashmiri	KSH	kas
Khasi	KSI	kha
Kildin Sami	KSM	sjd
Kui	KUI	kxu
Kulvi	KUL	kfx
Kumyk	KUM	kum
Kurdish	KUR	kur
Kurukh	KUU	kru
Kuy	KUY	kdt
Koryak	KYK	kpy
Ladin	LAD	lld
Lahuli	LAH	bfu
Lak	LAK	lbe
Lambani	LAM	lmn
Lao	LAO	lao
Latin	LAT	lat
Laz	LAZ	lzz
L-Cree	LCR	crm
Ladakhi	LDK	lbj
Lezgi	LEZ	lez
Lingala	LIN	lin
Low Mari	LMA	mhr
Limbu	LMB	lif
Lomwe	LMW	ngl
Lower Sorbian	LSB	dsb
Lule Sami	LSM	smj
Lithuanian	LTH	lit
Luxembourgish	LTZ	ltz

Luba	LUB	lua, lub
Luganda	LUG	lug
Luhya	LUH	luy
Luo	LUO	luo
Latvian	LVI	lav
Majang	MAJ	mpe
Makua	MAK	vmw
Malayalam Traditional	MAL	mal
Mansi	MAN	mns
Mapudungun	MAP	arn
Marathi	MAR	mar
Marwari	MAW	mwr
Mbundu	MBN	kmb
Manchu	MCH	mnc
Moose Cree	MCR	crm
Mende	MDE	men
Me'en	MEN	mym
Mizo	MIZ	lus
Macedonian	MKD	mkd
Male	MLE	mdy
Malagasy	MLG	mlg
Malinke	MLN	mlq
Malayalam Reformed	MLR	mal
Malay	MLY	msa
Mandinka	MND	mnk
Mongolian	MNG	mon
Manipuri	MNI	mni
Maninka	MNK	man
Manx Gaelic	MNX	glv
Mohawk	MOH	mho

Moksha	MOK	mdf
Moldavian	MOL	mol
Mon	MON	mnw
Moroccan	MOR	
Maori	MRI	mri
Maithili	MTH	mai
Maltese	MTS	mlt
Mundari	MUN	unr
Naga-Assamese	NAG	nag
Nanai	NAN	gld
Naskapi	NAS	nsk
N-Cree	NCR	csw
Ndebele	NDB	nde, ndl
Ndonga	NDG	ndo
Nepali	NEP	nep
Newari	NEW	new
Nagari	NGR	
Norway House Cree	NHC	csw
Nisi	NIS	dap
Niuean	NIU	niu
Nkole	NKL	nyn
N'Ko	NKO	ngo
Dutch	NLD	nld
Nogai	NOG	nog
Norwegian	NOR	nob
Northern Sami	NSM	sme
Northern Tai	NTA	nod
Esperanto	NTO	epo
Nynorsk	NYN	nno
Occitan	OCI	oci

Oji-Cree	OCR	ojs
Ojibway	OJB	oji
Oriya	ORI	ori
Oromo	ORO	orm
Ossetian	OSS	oss
Palestinian Aramaic	PAA	sam
Pali	PAL	pli
Punjabi	PAN	pan
Palpa	PAP	plp
Pashto	PAS	pus
Polytonic Greek	PGR	ell
Filipino	PIL	fil
Palaung	PLG	pce, rbb, pll
Polish	PLK	pol
Provencal	PRO	pro
Portuguese	PTG	por
Chin	QIN	bgr, cnh, cnw, czt, sez, tcp, csy, ctd, flm, pck, tcz, zom, cmr, dao, hlt, cka, cnk, mrh, mwg, cbl, cnb, csh
Rajasthani	RAJ	raj
R-Cree	RCR	atj
Russian Buriat	RBU	bxr
Riang	RIA	ria
Rhaeto-Romanic	RMS	roh
Romanian	ROM	ron
Romany	ROY	rom
Rusyn	RSY	rue
Ruanda	RUA	kin
Russian	RUS	rus
Sadri	SAD	sck
Sanskrit	SAN	san

Santali	SAT	sat
Sayisi	SAY	chp
Sekota	SEK	xan
Selkup	SEL	sel
Sango	SGO	sag
Shan	SHN	shn
Sibe	SIB	sjo
Sidamo	SID	sid
Silte Gurage	SIG	xst
Skolt Sami	SKS	sms
Slovak	SKY	slk
Slavey	SLA	scs
Slovenian	SLV	slv
Somali	SML	som
Samoan	SMO	smo
Sena	SNA	she
Sindhi	SND	snd
Sinhalese	SNH	sin
Soninke	SNK	snk
Sodo Gurage	SOG	gru
Sotho	SOT	nso, sot
Albanian	SQI	gsw
Serbian	SRB	srp
Saraiki	SRK	skr
Serer	SRR	srr
South Slavey	SSL	xsl
Southern Sami	SSM	sma
Suri	SUR	suq
Svan	SVA	sva
Swedish	SVE	swe

Swadaya Aramaic	SWA	aii
Swahili	SWK	swa
Swazi	SWZ	ssw
Sutu	SXT	ngo
Syriac	SYR	syr
Tabasaran	TAB	tab
Tajiki	TAJ	tgk
Tamil	TAM	tam
Tatar	TAT	tat
TH-Cree	TCR	cwd
Telugu	TEL	tel
Tongan	TGN	ton
Tigre	TGR	tig
Tigrinya	TGY	tir
Thai	THA	tha
Tahitian	THT	tah
Tibetan	TIB	bod
Turkmen	TKM	tuk
Temne	TMN	tem
Tswana	TNA	tsn
Tundra Nenets	TNE	enh
Tonga	TNG	toi
Todo	TOD	xal
Turkish	TRK	tur
Tsonga	TSG	tso
Turoyo Aramaic	TUA	tru
Tulu	TUL	tcy
Tuvin	TUV	tyv
Twi	TWI	aka
Udmurt	UDM	udm

Ukrainian	UKR	ukr
Urdu	URD	urd
Upper Sorbian	USB	hsb
Uyghur	UYG	uig
Uzbek	UZB	uzb
Venda	VEN	ven
Vietnamese	VIT	vie
Wa	WA	wbm
Wagdi	WAG	wbr
West-Cree	WCR	crk
Welsh	WEL	cym
Wolof	WLF	wol
Tai Lue	XBD	khb
Xhosa	XHS	xho
Yakut	YAK	sah
Yoruba	YBA	yor
Y-Cree	YCR	
Yi Classic	YIC	
Yi Modern	YIM	iii
Chinese Hong Kong	ZHH	zho
Chinese Phonetic	ZHP	zho
Chinese Simplified	ZHS	zho
Chinese Traditional	ZHT	zho
Zande	ZND	zne
Zulu	ZUL	zul

6.4.3 Feature tags

Features provide information about how to use the glyphs in a font to render a script or language. For example, an Arabic font might have a feature for substituting initial glyph forms, and a Kanji font might have a feature for positioning glyphs vertically. All OFF Layout features define data for glyph substitution, glyph positioning, or both.

Each OFF Layout feature has a feature tag that identifies its typographic function and effects. By examining a feature's tag, a text-processing client can determine what a feature does and decide whether to implement it.

All tags are 4-byte character strings composed of a limited set of ASCII characters in the 0x20-0x7E range. Windows platform-registered feature tags use four lowercase letters. For instance, the "mark" feature manages the placement of diacritical marks, and the "swsh" feature renders swash glyphs.

A feature definition may not provide all the information required to properly implement glyph substitution or positioning actions. In many cases, a text-processing client may need to supply additional data. For example, the function of the "init" feature is to provide initial glyph forms. Nothing in the feature's lookup tables indicates when or where to apply this feature during text processing. To correctly use the "init" feature in Arabic text where initial glyph forms appear at the beginning of words, text-processing clients must be able to identify the first glyph position in each word before making the glyph substitution. In all cases, the text-processing client is responsible for applying, combining, and arbitrating among features and rendering the result.

The tag space defined by tags consisting of four uppercase letters (A-Z) with no punctuation, spaces, or numbers, is reserved as a vendor space. Font vendors may use such tags to identify private features. For example, the feature tag "PKRN" might designate a private feature that may be used to kern punctuation marks.

NOTE There is no guarantee the compatibility or usability of private features, and it cannot be ensured that two font vendors will not choose the same tag for a private feature.

This Tag Registry describes all the OFF Layout features. Lookup information is provided for reference purposes only; the set of lookups used to implement a feature will vary across system platforms, applications, fonts, and font developers.

6.4.3.1 Feature tag list

Registered features

The features listed below are sorted in alphabetical order by tag name.

Feature Tag	Friendly Name
'aalt'	Access All Alternates
'abvf'	Above-base Forms
'abvm'	Above-base Mark Positioning
'abvs'	Above-base Substitutions
'afrc'	Alternative Fractions
'akhn'	Akhands
'blwf'	Below-base Forms
'blwm'	Below-base Mark Positioning
'blws'	Below-base Substitutions
'calt'	Contextual Alternates
'case'	Case-Sensitive Forms
'ccmp'	Glyph Composition / Decomposition
'cfar'	Conjunct Form After Ro
'cjct'	Conjunct Forms
'clig'	Contextual Ligatures

'cpct'	Centered CJK Punctuation
'cpsp'	Capital Spacing
'cswh'	Contextual Swash
'curs'	Cursive Positioning
'cv01-cv99'	Character Variants
'c2pc'	Petite Capitals From Capitals
'c2sc'	Small Capitals From Capitals
'dist'	Distances
'dlig'	Discretionary Ligatures
'dnom'	Denominators
'expt'	Expert Forms
'falt'	Final Glyph on Line Alternates
'fin2'	Terminal Forms #2
'fin3'	Terminal Forms #3
'fina'	Terminal Forms
'frac'	Fractions
'fwid'	Full Widths
'half'	Half Forms
'haln'	Halant Forms
'halt'	Alternate Half Widths
'hist'	Historical Forms
'hkna'	Horizontal Kana Alternates
'hlig'	Historical Ligatures
'hngl'	Hangul
'hojo'	Hojo Kanji Forms (JIS X 0212-1990 Kanji Forms)
'hwid'	Half Widths
'init'	Initial Forms
'isol'	Isolated Forms
'ital'	Italics
'jalt'	Justification Alternates
'jp78'	JIS78 Forms
'jp83'	JIS83 Forms
'jp90'	JIS90 Forms

'jp04'	JIS2004 Forms
'kern'	Kerning
'lfbd'	Left Bounds
'liga'	Standard Ligatures
'ljmo'	Leading Jamo Forms
'lnum'	Lining Figures
'locl'	Localized Forms
'ltra'	Left-to-right glyph alternates
'ltrm'	Left-to-right mirrored forms
'mark'	Mark Positioning
'med2'	Medial Forms #2
'medi'	Medial Forms
'mgrk'	Mathematical Greek
'mkmk'	Mark to Mark Positioning
'mset'	Mark Positioning via Substitution
'nalt'	Alternate Annotation Forms
'nlck'	NLC Kanji Forms
'nukt'	Nukta Forms
'numr'	Numerators
'onum'	Oldstyle Figures
'opbd'	Optical Bounds
'ordn'	Ordinals
'ornm'	Ornaments
'palt'	Proportional Alternate Widths
'pcap'	Petite Capitals
'pkna'	Proportional Kana
'pnum'	Proportional Figures
'pref'	Pre-Base Forms
'pres'	Pre-base Substitutions
'pstf'	Post-base Forms
'psts'	Post-base Substitutions
'pwid'	Proportional Widths

'qwid'	Quarter Widths
'rand'	Randomize
'rkrf'	Rakar Forms
'rlig'	Required Ligatures
'rphf'	Reph Forms
'rtbd'	Right Bounds
'rtla'	Right-to-left alternates
'rtlm'	Right-to-left mirrored forms
'ruby'	Ruby Notation Forms
'salt'	Stylistic Alternates
'sinf'	Scientific Inferiors
'size'	Optical size
'smcp'	Small Capitals
'smpl'	Simplified Forms
'ss01'	Stylistic Set 1
'ss02'	Stylistic Set 2
'ss03'	Stylistic Set 3
'ss04'	Stylistic Set 4
'ss05'	Stylistic Set 5
'ss06'	Stylistic Set 6
'ss07'	Stylistic Set 7
'ss08'	Stylistic Set 8
'ss09'	Stylistic Set 9
'ss10'	Stylistic Set 10
'ss11'	Stylistic Set 11
'ss12'	Stylistic Set 12
'ss13'	Stylistic Set 13
'ss14'	Stylistic Set 14
'ss15'	Stylistic Set 15
'ss16'	Stylistic Set 16
'ss17'	Stylistic Set 17
'ss18'	Stylistic Set 18
'ss19'	Stylistic Set 19

'ss20'	Stylistic Set 20
'subs'	Subscript
'sups'	Superscript
'swsh'	Swash
'titl'	Titling
'tjmo'	Trailing Jamo Forms
'tnam'	Traditional Name Forms
'tnum'	Tabular Figures
'trad'	Traditional Forms
'twid'	Third Widths
'unic'	Unicase
'valt'	Alternate Vertical Metrics
'vatu'	Vattu Variants
'vert'	Vertical Writing
'vhal'	Alternate Vertical Half Metrics
'vjmo'	Vowel Jamo Forms
'vkna'	Vertical Kana Alternates
'vkrn'	Vertical Kerning
'vpal'	Proportional Alternate Vertical Metrics
'vrt2'	Vertical Alternates and Rotation
'zero'	Slashed Zero

6.4.3.2 Feature descriptions and implementations

Tag: 'aalt'

Friendly name: Access All Alternates

Function: This feature makes all variations of a selected character accessible. This serves several purposes: An application may not support the feature by which the desired glyph would normally be accessed; the user may need a glyph outside the context supported by the normal substitution, or the user may not know what feature produces the desired glyph. Since many-to-one substitutions are not covered, ligatures would not appear in this table unless they were variant forms of another ligature.

Example: A user inputs the P in Poetica, and is presented with a choice of the four standard capital forms, the eight swash capital forms, the initial capital form and the small capital form.

Recommended implementation: The aalt table groups glyphs into semantic units. These units include the glyph which represents the default form for the underlying Unicode value stored by the application. While many of these substitutions are one-to-one (GSUB lookup type 1), others require a selection from a set (GSUB lookup type 3). The manufacturer may choose to build two tables (one for each lookup type) or only one which uses lookup type 3 for all substitutions. As in any one-from-many substitution, alternates present in

more than one face should be ordered consistently across a family, so that those alternates can work correctly when switching between family members. This feature should be ordered first in the font, to take precedence over other features.

Application interface: The application determines the GID for the default form of a given character (Unicode value with no features applied). It then checks to see whether the GID is found in the aalt coverage table. If so, the application passes this value to the feature table and gets back the GIDs in the associated group.

UI suggestion: While most one-from-many substitution features can be applied globally with reasonable results, aalt is not designed to support this use. The application should indicate to the user which glyphs in the user's document have alternative forms (i.e which are in the coverage table for aalt). When the user selects one of those glyphs and applies the aalt feature, an application could display the forms sequentially in context, or present a palette showing all the forms at once, or give the user a choice between these approaches. The application may assume that the first glyph in a set is the preferred form, so the font developer should order them accordingly. When only one alternate exists, this feature could toggle directly between the alternate and default forms.

Script/language sensitivity: None.

Feature interaction: This feature may be used in combination with other features.

Tag: 'abvf'

Friendly name: Above-base Forms

Function: Substitutes the above-base form of a vowel.

Example: In complex scripts like Khmer, the vowel OE must be split into a pre-base form and an above-base form. The above-base form of OE would be substituted to form the correct piece of the letter that is displayed above the base consonant.

Recommended implementation: This feature substitutes the GID for OE with the above part of the glyph (GSUB lookup type 1).

Application interface: In a sequence where a split vowel with an above form is used, the application must insert the pre-base glyph into the correct location and then apply the above-base form feature. The application gets back the GID for the correct form for the piece that is placed above the base glyph. The application may also choose to position this glyph if required, after this feature is called.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required in Khmer script.

Feature interaction: This feature overrides the results of all other features.

Tag: 'abvm'

Friendly name: Above-base Mark Positioning

Function: Positions marks above base glyphs.

Example: In complex scripts like Devanagari (Indic), the Anuswar needs to be positioned above the base glyph. This base glyph can be a base consonant or conjunct. The base glyph and the presence/absence of other marks above the base glyph decides the location of the Anuswar, so that they do not overlap each other.

Recommended implementation: The **abvm** table provides positioning information (x,y) to enable mark positioning (GPOS lookup type 4, 5).

Application interface: The application must define the GIDs of the base glyphs above which marks need to be positioned, and the marks themselves. If these are located in the coverage table, the application passes the

sequence to the **abvm** table and gets the positioning values (x,y) or positioning adjustments for the mark in return.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required in Indic scripts.

Feature interaction: Can be used to position default marks; or those that have been selected from a number of alternates based on contextual requirement using a feature like abvs.

Tag: 'abvs'

Friendly name: Above-base Substitutions

Function: Substitutes a ligature for a base glyph and mark that's above it.

Example: In complex scripts like Kannada (Indic), the vowel sign for the vowel I which a mark is positioned above base consonants. This mark combines with the consonant Ga to form a ligature.

Recommended implementation: Lookups for this feature map each sequence of consonant and vowel sign to the corresponding ligature in the font (GSUB lookup type 4).

Application interface: The application must define the GIDs of the base glyphs and the mark that combines with it to form a ligature. The application passes the sequence to the **abvs** table. If these are located in the coverage table, it gets the GID for the ligature in return.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required in Indic scripts.

Feature interaction: None.

Tag: 'afrc'

Friendly name: Alternative Fractions

Function: Replaces figures separated by a slash with an alternative form.

Example: The user enters 3/4 in a recipe and get the threequarters nut fraction.

Recommended implementation: The afrc table maps sets of figures separated by slash (U+002F) or fraction (U+2044) characters to corresponding fraction glyphs in the font (GSUB lookup type 4).

Application interface: The application must define the full sequence of GIDs to be replaced. When the full sequence is found in the frac coverage table, the application passes the sequence to the afrc table and gets a new GID in return.

UI suggestion: This feature should be off by default.

Script/language sensitivity: None.

Feature interaction: This feature overrides the results of all other features.

Tag: 'akhn'

Friendly name: Akhand

Function: Preferentially substitutes a sequence of characters with a ligature. This substitution is done irrespective of any characters that may precede or follow the sequence.

Example: In Devanagari script, the form Kssa is considered an Akhand character (meaning unbreakable), and the sequence Ka, Halant, Ssa should always produce the ligature Kssa, irrespective of characters that precede/follow the above given sequence.

Recommended implementation: This feature maps the sequences for generating Akhands defined in the given script, to the ligature they form (GSUB lookup type 4).

Application interface: The application passes the full sequence of GIDs. If these are located in the coverage table of the Akhand table, the application gets back the GID for the akhand ligature in return.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Required in most Indic scripts.

Feature interaction: This feature is used in conjunction with certain other features to derive required forms of Indic scripts. The application is expected to process this feature and certain other features in an appropriate order to obtain the correct set of basic forms: nukt, akhn, rphf, rkrf, pref, blwf, half, pstf, cjct. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: 'blwf'

Friendly name: Below-base Forms

Function: Substitutes the below-base form of a consonant in conjuncts.

Example: In complex scripts like Oriya (Indic), the consonant Va has a below-base form that is used to generate conjuncts. Given a sequence Gha, Virama (Halant), Va; the below-base form of Va would be substituted to form the conjunct GhVa.

Recommended Implementation: This feature substitutes the GID sequence of virama (halant) followed by a consonant; by the GID of the below base form of the consonant (GSUB lookup type 4).

Application interface: In a conjunct formation sequence, if a consonant is identified as having a below base form, the application gets back the GID for this. The application may also choose to position this glyph if required, after this feature is called.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Required in a number of Indic scripts.

Feature interaction: This feature is used in conjunction with certain other features to derive required forms of Indic and Indic-related scripts. For Indic scripts, the application is expected to process this feature and certain other features in an appropriate order to obtain the correct set of basic forms: nukt, akhn, rphf, rkrf, pref, blwf, half, pstf, cjct. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: 'blwm'

Friendly name: Below-base Mark Positioning

Function: Positions marks below base glyphs.

Example: In complex scripts like Gujarati (Indic), the vowel sign U needs to be positioned below base consonant/conjuncts that form the base glyph. This position can vary depending on the base glyph, as well as the presence/absence of other marks below the base glyph.

Recommended implementation: The **blwm** table provides positioning information (x,y) to enable mark positioning (GPOS lookup type 4, 5).

Application interface: The application must define the GIDs of the base glyphs below which marks need to be positioned, and the marks themselves. If these are located in the coverage table, the application passes the sequence to the **blwm** table and gets the positioning values (x,y) or positioning adjustments for the mark in return.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required in Indic scripts.

Feature interaction: Can be used to position default marks; or those that have been selected from a number of alternates based on contextual requirement using a feature like blws.

Tag: "blws"

Friendly name: Below-base Substitutions

Function: Produces ligatures that comprise of base glyph and below-base forms.

Example: In the Malayalam script (Indic), the conjunct Kla, requires a ligature which is formed using the base glyph Ka and the below-base form of consonant La. This feature can also be used to substitute ligatures formed using base glyphs and below base matras in Indic scripts.

Recommended implementation: The **blws** table maps the identified conjunct forming sequences; or consonant vowel sign sequences; to their ligatures (GSUB lookup type 4).

Application interface: For GIDs found in the **blws** coverage table, the application passes the sequence of GIDs to the table, and gets back the GID for the ligature.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required in Indic scripts.

Feature interaction: This feature overrides the results of all other features.

Tag: 'calt'

Friendly name: Contextual Alternates

Function: In specified situations, replaces default glyphs with alternate forms which provide better joining behavior. Used in script typefaces which are designed to have some or all of their glyphs join.

Example: In Caflisch Script, o is replaced by o.alt2 when followed by an ascending letterform.

Recommended implementation: The calt table specifies the context in which each substitution occurs, and maps one or more default glyphs to replacement glyphs (GSUB lookup type 6).

Application interface: The application passes sequences of GIDs to the feature table, and gets back new GIDs. Full sequences must be passed.

UI suggestion: This feature should be active by default.

Script/language sensitivity: Not applicable to ideographic scripts.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'case'

Friendly name: Case-Sensitive Forms

Function: Shifts various punctuation marks up to a position that works better with all-capital sequences or sets of lining figures; also changes oldstyle figures to lining figures. By default, glyphs in a text face are designed to work with lowercase characters. Some characters should be shifted vertically to fit the higher visual center of all-capital or lining text. Also, lining figures are the same height (or close to it) as capitals, and fit much better with all-capital text.

Example: The user selects a block of text and applies this feature. The dashes, bracketing characters, guillemet quotes and the like shift up to match the capitals, and oldstyle figures change to lining figures.

Recommended implementation: The font may implement this change by substituting different glyphs (GSUB lookup type 1) or by repositioning the original glyphs (GPOS lookup type 1).

Application interface: The application queries whether specific GIDs are found in the coverage table for the case feature. If so, it passes these IDs to the table and gets back either new GIDs or positional adjustments (XPlacement and YPlacement).

UI suggestion: It would be good to apply this feature (or turn it off) by default when the user changes case on a sequence of more than one character. Applications could also detect words consisting only of capitals, and apply this feature based on user preference settings.

Script/language sensitivity: Applies only to European scripts; particularly prominent in Spanish-language setting.

Feature interaction: This feature overrides the results of other features affecting the figures (e.g. onum and tnum).

Tag: "ccmp"

Friendly name: Glyph Composition/Decomposition

Function: To minimize the number of glyph alternates, it is sometimes desired to decompose a character into two glyphs. Additionally, it may be preferable to compose two characters into a single glyph for better glyph processing. This feature permits such composition/decompostion. The feature should be processed as the first feature processed, and should be processed only when it is called.

Example: In Syriac, the character 0x0732 is a combining mark that has a dot above AND a dot below the base character. To avoid multiple glyph variants to fit all base glyphs, the character is decomposed into two glyphs...a dot above and a dot below. These two glyphs can then be correctly placed using GPOS. In Arabic it might be preferred to combine the shadda with fatha (0x0651, 0x064E) into a ligature before processing shapes. This allows the font vendor to do special handling of the mark combination when doing further processing without requiring larger contextual rules.

Recommended implementation: The **ccmp** table maps the character sequence to its corresponding ligature (GSUB lookup type 4) or string of glyphs (GSUB lookup type 2). When using GSUB lookup type 4, sequences that are made up of larger number of glyphs must be placed before those that require fewer glyphs.

Application interface: For GIDs found in the **ccmp** coverage table, the application passes the sequence of GIDs to the table, and gets back the GID for the ligature, or GIDs for the multiple substitution.

UI suggestion: This feature should be on by default.

Script/language sensitivity: None.

Feature interaction: This feature needs to be implemented prior to any other feature.

Tag: 'cfar'

Friendly name: Conjunct Form After Ro

Function: Substitutes alternate below-base or post-base forms in Khmer script when occurring after conjoined Ro ("Coeng Ra").

In Khmer script, the conjoined form of Ro re-orders to the left of the base consonant. It wraps under the base consonant, however, and so can interact typographically with below-base or post-base conjoined consonant and vowel forms. After the application has re-ordered the glyph for the conjoined Ro, it is no longer in the immediate context of glyphs for below-base or post-base forms. The application can detect this and apply this feature over the range for the below-base and post-base conjoining forms, triggering lookups to substitute alternate below-base or past-base forms as may be needed.

Example: In the Khmer script, Coeng Ro is denoted by a pre-base conjoining form, and Coeng Yo is denoted by a post-base conjoining form, but in both cases part of the form wraps under the base. The consonant cluster TRYo is denoted with an alternate form of Coeng Ya that descends lower so that it does not collide below the base with the Coeng Ro.

Recommended implementation: The cfar table maps below-base or post-base conjoining form into an alternate form (GSUB lookup type 1).

Application interface: For substitutions defined in the cfar table, the application passes the GID to the table and gets back the GID for an alternate form. The application is expected to apply this feature if a syllable contains a Coeng Ra followed by other conjoining consonants or vowels.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Required in Khmer scripts.

Feature interaction: This feature is used in conjunction with certain other features to derive required forms of Khmer script. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: 'cjct'

Friendly name: Conjunct Forms

Function: Produces conjunct forms of consonants in Indic scripts. This is similar to the Akhands feature, but is applied at a different sequential point in the process of shaping an Indic syllable.

Indic scripts are associated with conjoining-consonant behaviors, such as the use of "half" forms. Some consonants may not have half forms and not exhibit conjoining behavior when combined with certain consonants, yet may conjoin as ligature forms with other consonants. Whether a given pair of consonants conjoins may impact other shaping behaviors for a syllable, such as where a re-ordering vowel mark or reph is placed. The Conjunct Forms feature can be used at a point in the shaping process immediately before final re-ordering such that the application can determine whether a re-ordering vowel or reph is placed in relation to the consonants.

More generally, the Akhands feature and Conjunct Forms feature can be used at two points in the shaping of an Indic syllable, together with other features such as Half Forms and Below Forms applied in between, providing the font developer with flexibility in how the shapes for Indic syllables are derived from the default glyphs for the character sequence.

Example: In Hindi (Devanagari script), the consonant cluster DGa is denoted with a conjunct ligature form.

Recommended implementation: The cjct table maps the sequence of a consonant (the nominal form) followed by a virama (halant) followed by a second consonant (the nominal form or a half form) to the corresponding conjunct form (GSUB lookup type 4).

Application interface: For substitution sequences defined in the cjct table, the application passes the sequence of GIDs to the table, and gets back the GID for the conjunct form.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Required in Indic scripts that show similarity to Devanagari.

Feature interaction: This feature is used in conjunction with certain other features to derive required forms of Indic scripts. The application is expected to process this feature and certain other features in an appropriate order to obtain the correct set of basic forms: nukt, akhn, rphf, rkrf, pref, blwf, half, pstf, cjct. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: 'clig'

Friendly name: Contextual Ligatures

Function: Replaces a sequence of glyphs with a single glyph which is preferred for typographic purposes. Unlike other ligature features, clig specifies the context in which the ligature is recommended. This capability is important in some script designs and for swash ligatures.

Example: The glyph for ft replaces the sequence f t in Bickham Script, except when preceded by an ascending letter.

Recommended implementation: The clig table maps sequences of glyphs to corresponding ligatures in a chained context (GSUB lookup type 8). Ligatures with more components must be stored ahead of those with fewer components in order to be found. The set of contextual ligatures will vary by design and script.

Application interface: For sets of GIDs found in the clig coverage table, the application passes the sequence of GIDs to the table and gets back a single new GID. Full sequences must be passed.

NOTE This may include a change of character code. Besides the original character code, the application should store the code for the new character.

UI suggestion: This feature should be active by default.

Script/language sensitivity: Applies to virtually all scripts.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also dlig.

Tag: 'cpct'

Friendly name: Centered CJK Punctuation

Function: Centers specific punctuation marks for those fonts that do not include centered and non-centered forms.

Example: The user may invoke this feature in a Chinese font to get centered punctuation in case it is desired. Examples include U+3001 and U+3002, including their vertical variants, specifically U+FE11 and U+FE12, respectively.

Recommended implementation: The font specifies X- and Y-axis adjustments for a small number of full-width glyphs (GPOS lookup type 1).

Application interface: For GIDs found in the cpct coverage table, the application passes the GIDs to the table and gets back positional adjustments (XPlacement, XAdvance, YPlacement and YAdvance).

UI suggestion: This feature would be off by default.

Script/language sensitivity: Used primarily in Chinese fonts.

Feature interaction: This feature is mutually exclusive with all other glyph-width features (e.g. tnum, fwid, hwid, halt, palt, twid), which should be turned off when it's applied.

Tag: 'cpsp'

Friendly name: Capital Spacing

Function: Globally adjusts inter-glyph spacing for all-capital text. Most typefaces contain capitals and lowercase characters, and the capitals are positioned to work with the lowercase. When capitals are used for words, they need more space between them for legibility and esthetics. This feature would not apply to monospaced designs. Of course the user may want to override this behavior in order to do more pronounced letterspacing for esthetic reasons.

Example: The user sets a title in all caps, and the Capital Spacing feature opens the spacing.

Recommended implementation: The cpsp table stores alternate advance widths for the capital letters covered, generally increasing them by a uniform percentage (GPOS lookup type 1).

Application interface: For GIDs found in the cpsp coverage table, the application passes a sequence of GIDs to the cpsp table and gets back a set of XPlacement and XAdvance adjustments. The application may rely on the user to apply this feature (e.g., by selecting text for a change to all-caps) or apply its own heuristics for recognizing words consisting of capitals.

UI suggestion: This feature should be on by default. Applications may want to allow the user to respecify the percentage to fit individual tastes and functions.

Script/language sensitivity: Should not be used in connecting scripts (e.g. most Arabic).

Feature interaction: May be used in addition to any other feature.

NOTE This feature is additive with other GPOS features like kern.

Tag: 'cswh'

Friendly name: Contextual Swash

Function: This feature replaces default character glyphs with corresponding swash glyphs in a specified context. There may be more than one swash alternate for a given character.

Example: The user sets the word "HOLIDAY" in Poetica with this feature active, and is presented with a choice of three alternate forms appropriate for an initial H and one alternate appropriate for a medial L.

Recommended implementation: The cswh table maps GIDs for default forms to those for one or more corresponding swash forms in a chained context, which may require a selection from a set (GSUB lookup type 8). If several styles of swash are present across the font, the set of forms for each character should be ordered consistently.

Application interface: For GIDs found in the cswh coverage table, the application passes the GIDs to the swsh table and gets back one or more new GIDs. If more than one GID is returned, the application must provide a means for the user to select the one desired.

UI suggestion: This feature should be inactive by default. When more than one GID is returned, an application could display the forms sequentially in context, or present a palette showing all the forms at once, or give the user a choice between these approaches. The application may assume that the first glyph in a set is the preferred form, so the font developer should order them accordingly.

Script/language sensitivity: Does not apply to ideographic scripts.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also swsh and init.

Tag: 'curs'

Friendly name: Cursive Positioning

Function: In cursive scripts like Arabic, this feature cursively positions adjacent glyphs.

Example: In Arabic, the Meem followed by a Reh are cursively positioned by overlapping the exit point of the Meem on the entry point of the Reh.

Recommended implementation: The **curs** table provides entry and exit points (x,y) for glyphs to be cursively positioned (GPOS lookup type 3).

Application interface: For GIDs located in the coverage table, the application gets back positioning point locations for the preceding and following glyphs.

UI suggestion: This feature could be made active or inactive by default, at the user's preference.

Script/language sensitivity: Can be used in any cursive script.

Feature interaction: None.

Tag: 'cv01' - 'cv99'

Friendly name: Character Variant 1 – Character Variant 99

Registered by: Microsoft

Function: A font may have stylistic-variant glyphs for one or more characters where the variations for one character are not systematically related to those for other characters. Or, a variation may exist for a character and its casing pair (or related pre-composed characters), but not be applicable to other unrelated characters.

In some usage scenarios, it may be necessary to provide the application with control over glyph variations for different Unicode characters individually

The function of these features is similar to the function of the Stylistic Alternates feature ('salt') and the Stylistic Set features (see 'ss01' – 'ss20'). Whereas the Stylistic Set features assume recurring stylistic variations that apply to a broad set of Unicode characters, these features are intended for scenarios in which particular characters have variations not applicable to a broad set of characters. The Stylistic Alternates feature provides access to glyph variants, but does not allow an application to control these on a character-by-character basis; the Character Variant features provide the greater granularity of control.

The function of these features is also related to that of the Localized Forms ('locl') feature, in that particular variations for a character may be preferred for particular languages. In practice, though, it may not be feasible to associate particular glyph variants with particular language systems for all the relevant languages; for example, the requirements of particular languages may not be known when a font is being developed.

The distinction between these features and the Stylistic set features is most easily understood in terms of variations applying to a single character versus variations applying across a range of characters. In practice, if a variation applies to a character in a bicameral script, then the casing-pair character may have the same variation. Also, Unicode includes pre-composed characters for certain base + mark combinations, hence a single abstract character may be incorporated into a number of Unicode characters. Therefore, a variation for a particular abstract character may be applicable to several related Unicode characters. The Character Variant features can be used for sets of related characters in these cases. The key distinction between such use and the intended use for Stylistic Set features is that a Character Variant feature should apply only to one character or a set of characters closely related in this way, while Stylistic Set features are intended for broader sets of characters.

Recommended implementation: A cvXX table maps the GID for the default form of a character to the GIDs for stylistic alternatives of that character. Each cvXX feature uses alternate (GSUB lookup type 3) substitutions. (If there is only one variant for a character, a single-substitution lookup, type 1, can also be used.)

The FeatureParams field of the Feature Table of these GSUB features may be set to 0, or to an offset to a Feature Parameters table. The Feature Parameters table for this feature is structured as follows:

Type	Name	Description
USHORT	format	Format number is set to 0.
USHORT	featUiLabelNameId	The 'name' table name ID that specifies a string (or strings, for multiple languages) for a user-interface label for this feature. (May be NULL.)
USHORT	featUiTooltipTextNameId	The 'name' table name ID that specifies a string (or strings, for multiple languages) that an application can use for tooltip text for this feature. (May be NULL.)
USHORT	sampleTextNameId	The 'name' table name ID that specifies sample text that illustrates the effect of this feature. (May be NULL.)
USHORT	numNamedParameters	Number of named parameters. (May be zero.)
USHORT	firstParamUiLabelNameId	The first 'name' table name ID used to specify strings for user-interface labels for the feature parameters. (Must be zero if numParameters is zero.)
USHORT	charCount	The count of characters for which this feature provides glyph variants. (May be zero.)
UINT24	character[charCount]	The Unicode Scalar Value of the characters for which this feature provides glyph variants.

The name ID provided by featUiLabelNameId is intended to provide a user-interface string for the feature; for example, "Capital-eng variants". If set to NULL, no 'name' table string is used for the feature name.

The name ID provided by featUiTooltipTextNameId is intended to provide a user-interface string that provides a brief description of the feature that applications can use in popup "tooltip" help windows (e.g. "Select glyph variants for capital eng."). If set to NULL, no 'name' table string is used for the feature "tooltip" help text.

The name ID provided by sampleTextNameId is intended to provide a string that can be used in a user-interface to illustrate the effect of the feature. If multiple characters are affected by the feature or if the feature affects a combining mark, it may not be evident to an application what string to use to present an illustrative sample; a 'name' table string can be provided for that purpose.

If numNamedParameters is non-zero, then firstParamUiLabelNameId and numNamedParameters specify a sequence of consecutive name IDs in the name table. These are used to provide user-interface strings for individual variants. The range of name IDs start at firstParamUiLabelNameId and end at firstParamUiLabelNameId + numNamedParameters – 1. Each of these name IDs corresponds to a feature parameter value used to select a particular GID from the array of GIDs returned by a type 3 substitution lookup; the relation between parameter values and name IDs is: name ID = parameter + firstParamUiLabelNameId - 1. The value of numNamedParameters should not exceed the number of alternate glyphs in lookups associated with the feature; note, however, that the number of GIDs in the returned array for a GSUB type 3 lookup should not be assumed to be equal to numNamedParameters: numNamedParameters should not be more than the number of GIDs in the array, but it may be less. If numNamedParameters is zero, then no 'name' table strings are associated with feature parameters.

The values of featUiLabelNameId, featUiTooltipTextNameId and firstParamUiLabelNameId are expected to be in the font-specific name ID range (256–32767), though that is not a requirement in this Feature Parameters specification. The value of firstParamUiLabelNameId + numNamedParameters – 1 should not exceed 32767.

The user-interface label for the feature, for "tooltip" help text, or for feature parameters can be provided in multiple languages. English strings for each should be included as a fallback. A sample-text string likely would not need to be localized, though different sample-text strings for different UI languages can be used. If only one sample-text string is provided, applications may use it with any UI language.

The charCount field and character array are used to identify the Unicode characters for which this feature provides glyph variants. Applications can use this information in presenting user interface or for other purposes. Content of the character list is at the discretion of the font developer — the list may be exhaustive, representative, or empty — and does not affect the operation of the feature. If a font developer chooses not to include such information, charCount can be set to zero, in which case no character array can be included.

It is left to the discretion of application developers to determine whether or how to use the data provided in the feature parameters table or associated strings in the 'name' table.

Note: Since the strings provided using this feature parameter table will be used in application user interface, length is an important consideration. Strings should be as short as possible. It is recommended that the length of the feature or feature-parameter names be 25 characters or less, and that the length of "tooltip" help text be 250 characters or less.

Application interface: The application is responsible for counting and enumerating the number of features in the font with tag names of the format 'cv01' to 'cv99', and for presenting the user with an appropriate selection mechanism. The application is also responsible for interpreting any feature parameter tables (if the application developer wishes to use that data) and presenting referenced strings in user interface. For GIDs found in the cvXX coverage table, the application passes the GIDs to the cvXX table and gets back one or more new GIDs; the application selects one of the returned GIDs for display. The application may use an index parameter as an index into the array of returned GIDs.

UI suggestion: This feature should be off by default. An application can display glyph variants for a given character as a glyph palette in the user interface. If a Feature Parameters table is provided, the feature UI label or the feature and parameter UI labels (if provided) can be presented in the application user interface; or the sample-text string (if provided) can be presented in the application user interface.

Script/language sensitivity: None.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. Note that after a cvXX feature has been applied, the user may wish to apply other typographic features, e.g. 'smcp'; font developers are responsible for ordering substitution lookups to obtain desired user experience. If it is to be used in conjunction with a complex script that requires obligatory substitution of ligatures or contextual forms, this feature should be applied before features for obligatory script behaviors.

Tag:'c2pc'

Friendly name: Petite Capitals From Capitals

Function: This feature turns capital characters into petite capitals. It is generally used for words which would otherwise be set in all caps, such as acronyms, but which are desired in petite-cap form to avoid disrupting the flow of text. See the pcap feature description for notes on the relationship of caps, smallcaps and petite caps.

Example: The user types UNICEF or NASA, applies c2pc and gets petite cap text.

Recommended implementation: The c2pc table maps capital glyphs to the corresponding petite cap forms (GSUB lookup type 1).

Application interface: For GIDs found in the c2pc coverage table, the application passes GIDs to the c2pc table, and gets back new GIDs.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies only to scripts with both upper- and lowercase forms (e.g. Latin, Cyrillic, Greek).

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. Also see pcap.

Tag: 'c2sc'

Friendly name: Small Capitals From Capitals

Function: This feature turns capital characters into small capitals. It is generally used for words which would otherwise be set in all caps, such as acronyms, but which are desired in small-cap form to avoid disrupting the flow of text.

Example: The user types UNICEF or SCUBA, applies c2sc and gets small cap text.

Recommended implementation: The c2sc table maps capital glyphs to the corresponding small-cap forms (GSUB lookup type 1).

Application interface: For GIDs found in the c2sc coverage table, the application passes GIDs to the c2sc table, and gets back new GIDs.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies only to bicameral scripts (i.e. those with case differences), such as Latin, Greek, Cyrillic, and Armenian.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. Also see smcp.

Tag: "dist"

Friendly name: Distances

Function: Provides a means to control distance between glyphs.

Example: In the Devanagari (Indic) script, the distance between the vowel sign U and a consonant can be adjusted using this.

Recommended implementation: The **dist** table provides distances by which a glyph needs to move towards or away from another glyph (GPOS lookup type 2).

Application interface: For GIDs found in the **dist** coverage table, the application passes their GID to the table and gets back the distance that needs to be maintained between them.

UI suggestion: This feature could be made active or inactive by default, at the user's preference.

Script/language sensitivity: Required in Indic scripts.

Feature interaction: None.

Tag: 'dlig'

Friendly name: Discretionary Ligatures

Function: Replaces a sequence of glyphs with a single glyph which is preferred for typographic purposes. This feature covers those ligatures which may be used for special effect, at the user's preference.

Example: The glyph for ct replaces the sequence of glyphs c t, or U+322E (Kanji ligature for "Friday") replaces the sequence U+91D1 U+66DC U+65E5.

Recommended implementation: The dlig table maps sequences of glyphs to corresponding ligatures (GSUB lookup type 4). Ligatures with more components must be stored ahead of those with fewer components in order to be found. The set of discretionary ligatures will vary by design and script.

Application interface: For sets of GIDs found in the dlig coverage table, the application passes the sequence of GIDs to the table and gets back a single new GID. Full sequences must be passed. This may include a change of character code. Besides the original character code, the application should store the code for the new character.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies to virtually all scripts.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also clig.

Tag: 'dnom'

Friendly name: Denominators

Function: Replaces selected figures which follow a slash with denominator figures.

Example: In the string 11/17 selected by the user, the application turns the 17 into denominators when the user applies the fraction feature (frac).

Recommended implementation: The dnom table maps sets of figures and related characters to corresponding numerator glyphs in the font (GSUB lookup type 1).

Application interface: For GIDs found in the dnom coverage table, the application passes a GID to the table and gets back a new GID.

UI suggestion: This feature should normally be called by an application when the user applies the frac feature.

Script/language sensitivity: None.

Feature interaction: This feature supports frac. It may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'expt'

Friendly name: Expert Forms

Function: Like the JIS78 Forms described above, this feature replaces standard forms in Japanese fonts with corresponding forms preferred by typographers. Although most of the JIS78 substitutions are included, the expert substitution goes on to handle many more characters.

Example: The user would invoke this feature to replace kanji character U+5516 with U+555E.

Recommended implementation: The expt table maps many default (JIS90) GIDs to corresponding alternates (GSUB lookup type 1).

Application interface: For GIDs found in the expt coverage table, the application passes the GIDs to the table and gets back one new GID for each.

NOTE This is a change of character code. Besides the original character code, the application should store the code for the new character.

UI suggestion: Applications may choose to have this feature active or inactive by default, depending on their target markets.

Script/language sensitivity: Applies only to Japanese.

Feature interaction: This feature is mutually exclusive with all other features, which should be turned off when it's applied, except the palt, vpal, vert and vrt2 features, which may be used in addition.

Tag: "falt"

Friendly name: Final Glyph on Line Alternates

Function: Replaces line final glyphs with alternate forms specifically designed for this purpose (they would have less or more advance width as need may be), to help justification of text.

Example: In the Arabic script, providing alternate forms for line final glyphs would result in better justification. eg. replacing a long tailed Yeh-with-tail with one that has a slightly longer/shorter tail.

Recommended implementation: The **falt** table maps line final glyphs (in isolated or final forms) to their corresponding alternate forms (GSUB lookup type 3).

Application interface: For GIDs found in the **falt** coverage table, the application passes a GID to the table and gets back a new GID.

UI suggestion: This feature could be made active or inactive by default, at the user's preference.

Script/language sensitivity: Can be used in any cursive script.

Feature interaction: Would need to be applied last, only after all other features have been applied to the run.

Tag: "fin2"

Friendly name: Terminal Form #2

Function: Replaces the Alaph glyph at the end of Syriac words with its appropriate form, when the preceding base character cannot be joined to, and that preceding base character is not a Dalath, Rish, or dotless Dalath-Rish.

Example: When an Alaph is preceded by a He, the Alaph would be replaced by an appropriate form. This feature is used only for the Syriac script alaph character.

Recommended implementation: The **fin2** table maps default alphabetic forms to corresponding final forms (GSUB lookup type 5).

Application interface: The application is responsible for noting word boundaries. For GIDs in the middle of words and found in the fin2 coverage table, the application passes a GID to the feature and gets back a new GID.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Used only with the Syriac script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also init and fina.

Tag: "fin3"

Friendly name: Terminal Form #3

Function: Replaces Alaph glyphs at the end of Syriac words when the preceding base character is a Dalath, Rish, or dotless Dalath-Rish.

Example: When an Alaph is preceded by a Dalath, the Alaph would be replaced by an appropriate form. This feature is used only for the Syriac script alaph character.

Recommended implementation: The **fin3** table maps default alphabetic forms to corresponding final forms (GSUB lookup type 5).

Application interface: The application is responsible for noting word boundaries. For GIDs in the middle of words and found in the fin3 coverage table, the application passes a GID to the feature and gets back a new GID.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Used only with the Syriac script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also init and fina.

Tag: 'fina'

Friendly name: Terminal Forms

Function: Replaces glyphs at the ends of words with alternate forms designed for this use. This is common in Latin connecting scripts, and required in various non-Latins like Arabic.

Example: In the typeface Poetica, the default e in the word 'type' is replaced with the e.end form.

Recommended implementation: The fina table maps default alphabetic forms to corresponding ending forms (GSUB lookup type 1).

Application interface: The application is responsible for noting word boundaries. For GIDs at the ends of words and found in the fina coverage table, the application passes a GID to the feature and gets back a new GID.

UI suggestion: This feature should be active by default.

Script/language sensitivity: Can be used in any alphabetic script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also init and medi.

Tag: 'frac'

Friendly name: Fractions

Function: Replaces figures separated by a slash with 'common' (diagonal) fractions.

Example: The user enters 3/4 in a recipe and gets the threequarters fraction.

Recommended implementation: The frac table maps sets of figures separated by slash or fraction characters to corresponding fraction glyphs in the font. These may be precomposed fractions (GSUB lookup type 4) or arbitrary fractions (GSUB lookup type 1).

Application interface: The application must define the full sequence of GIDs to be replaced, based on user input (i.e. user selection determines the string's delimitation). When the full sequence is found in the frac coverage table, the application passes the sequence to the frac table and gets a new GID in return. When the frac table does not contain an exact match, the application performs two steps. First, it uses the numr feature (see below) to replace figures (as used in the numr coverage table) preceding the slash with numerators, and to replace the typographic slash character (U+002F) with the fraction slash character (U+2044). Second, it uses the dnom feature (see below) to replace all remaining figures (as listed in the dnom coverage table) with denominators.

UI suggestion: This feature should be off by default.

Script/language sensitivity: None.

Feature interaction: This feature may require the application to call the numr and dnom features. It may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'fwid'

Friendly name: Full Widths

Function: Replaces glyphs set on other widths with glyphs set on full (usually em) widths. In a CJKV font, this may include "lower ASCII" Latin characters and various symbols. In a European font, this feature replaces proportionally-spaced glyphs with monospaced glyphs, which are generally set on widths of 0.6 em.

Example: The user may invoke this feature in a Japanese font to get full monospaced Latin glyphs instead of the corresponding proportionally-spaced versions.

Recommended implementation: The font may contain alternate glyphs designed to be set on full widths (GSUB lookup type 1), or it may specify alternate (full-width) metrics for the proportional glyphs (GPOS lookup type 1).

Application interface: For GIDs found in the fwid coverage table, the application passes the GIDs to the table and gets back either new GIDs or positional adjustments (XPlacement and XAdvance).

UI suggestion: This feature would normally be off by default.

Script/language sensitivity: Applies to any script which can use monospaced forms.

Feature interaction: This feature is mutually exclusive with all other glyph-width features (e.g. tnum, halt, hwid, palt, pwid, qwid and twid), which should be turned off when it's applied. It deactivates the kern feature..

Tag: "half"

Friendly name: Half Forms

Function: Produces the half forms of consonants in Indic scripts.

Example: In Hindi (Devanagari script), the conjunct KKa, obtained by doubling the Ka, is denoted with a half form of Ka followed by the full form.

Recommended implementation: The **half** table maps the sequence of a consonant followed by a virama (halant) to its half form (GSUB lookup type 4).

Application interface: For substitution sequences defined in the **half** table [consonant followed by the virama (halant)], the application passes the sequence of GIDs to the table, and gets back the GID for the half form.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Required in Indic scripts that show similarity to Devanagari.

Feature interaction: This feature is used in conjunction with certain other features to derive required forms of Indic scripts. The application is expected to process this feature and certain other features in an appropriate order to obtain the correct set of basic forms: nukt, akhn, rphf, rkrf, pref, blwf, half, pstf, cjct. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: "haln"

Friendly name: Halant Forms

Function: Produces the halant forms of consonants in Indic scripts.

Example: In Sanskrit (Devanagari script), syllable final consonants are frequently required in their halant form.

Recommended implementation: The **haln** table maps the sequence of a consonant followed by a virama (halant) to its halant form (GSUB lookup type 4).

Application interface: For substitutions defined in the **halant** table, the application passes the sequence of GIDs to the feature (essentially the consonant and virama), and gets back the GID for the halant form.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required in Indic scripts.

Feature interaction: This feature overrides the results of all other features.

Tag: 'halt'

Friendly name: Alternate Half Widths

Function: Respaces glyphs designed to be set on full-em widths, fitting them onto half-em widths. This differs from hwid in that it does not substitute new glyphs.

Example: The user may invoke this feature in a CJKV font to get better fit for punctuation or symbol glyphs without disrupting the monospaced alignment.

Recommended implementation: The font specifies alternate metrics for the full-width glyphs (GPOS lookup type 1).

Application interface: For GIDs found in the halt coverage table, the application passes the GIDs to the table and gets back positional adjustments (XPlacement, XAdvance, YPlacement and YAdvance).

UI suggestion: This feature would be off by default.

Script/language sensitivity: Used only in CJKV fonts.

Feature interaction: This feature is mutually exclusive with all other glyph-width features (e.g. tnum, fwid, hwid, palt, twid), which should be turned off when it's applied. It deactivates the kern feature. See also vhal.

Tag: 'hist'

Friendly name: Historical Forms

Function: Some letterforms were in common use in the past, but appear anachronistic today. The best-known example is the long form of s; others would include the old Fraktur k. Some fonts include the historical forms as alternates, so they can be used for a 'period' effect. This feature replaces the default (current) forms with the historical alternates. While some ligatures are also used for historical effect, this feature deals only with single characters.

Example: The user applies this feature in Adobe Jenson to get the archaic forms of M, Q and Z.

Recommended implementation: The hist table maps default forms to corresponding historical forms (GSUB lookup type 1).

Application interface: For GIDs found in the hist coverage table, the application passes the GIDs to the hist table and gets back new GIDs.

UI suggestion: This feature should be off by default.

Script/language sensitivity: None.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'hkna'

Friendly name: Horizontal Kana Alternates

Function: Replaces standard kana with forms that have been specially designed for only horizontal writing. This is a typographic optimization for improved fit and more even color. Also see vkna.

Example: Standard full-width kana (hiragana and katakana) are replaced by forms that are designed for horizontal use.

Recommended implementation: The font includes a set of specially-designed glyphs, listed in the hkna coverage table. The hkna feature maps the standard full-width forms to the corresponding special horizontal forms (GSUB lookup type 1).

Application interface: For GIDs found in the hkna coverage table, the application passes GIDs to the feature, and gets back new GIDs.

*UI suggestion:*This feature would be off by default.

Script/language sensitivity: Applies only to fonts that support kana (hiragana and katakana).

Feature interaction: This feature may be used with the kern feature. Since it is for horizontal use, features applying to vertical behaviors (e.g. vkna, vert, vrt2 or vkrn) do not apply.

Tag: 'hlig'

Friendly name: Historical Ligatures

Function: Some ligatures were in common use in the past, but appear anachronistic today. Some fonts include the historical forms as alternates, so they can be used for a 'period' effect. This feature replaces the default (current) forms with the historical alternates.

Example: The user applies this feature using Palatino Linotype, and historic ligatures are formed for all long s forms, including: long s+t, long s+b, long s+h, long s+k, and several others.

Recommended implementation: The hlig table maps default ligatures and character combinations to corresponding historical ligatures (GSUB lookup type 1).

Application interface: For GIDs found in the hlig coverage table, the application passes the GIDs to the hlig table and gets back new GIDs.

UI suggestion: This feature should be off by default.

Script/language sensitivity: None.

Feature interaction: This feature overrides the results of all other features.

Tag: 'hngl'

Friendly name: Hangul

Function: Replaces hanja (Chinese-style) Korean characters with the corresponding hangul (syllabic) characters. This effectively reverses the standard input methods, in which hangul are entered and replaced by hanja. Many of these substitutions are one-to-one (GSUB lookup type 1), but hanja substitution often requires the user to choose from several possible hangul characters (GSUB lookup type 3).

Example: The user may call this feature to get U+AC00 from U+4F3D.

Recommended implementation: This table associates each hanja character in the font with one or more hangul characters. The manufacturer may choose to build two tables (one for each lookup type) or only one which uses lookup type 3 for all substitutions. As in any one-from-many substitution, alternates should be ordered consistently across a family, so that those alternates can work correctly when switching between family members.

Application interface: For GIDs found in the hngl coverage table, the application passes the GIDs to the table and gets back one or more new GIDs. If more than one GID is returned, the application must provide a means for the user to select the one desired.

NOTE This is a change of semantic value. Besides the original character codes (when entered as hanja), the application should store the code for the new character.

UI suggestion: This feature should be inactive by default. The application may note the user's choice when selecting from multiple hangul, and offer it as a default the next time the source hanja character is encountered. In the absence of such prior information, the application may assume that the first hangul in a set is the preferred form, so the font developer should order them accordingly.

Script/language sensitivity: Korean only.

Feature interaction: This feature is mutually exclusive with all other features, which should be turned off when it's applied, except the palt, vert and vrt2 may be used in addition.

Tag: 'hojo'

Friendly name: Hojo Kanji Forms (JIS X 0212-1990 Kanji Forms)

Registered by: Adobe

Function: The JIS X 0212-1990 (aka, "Hojo Kanji") and JIS X 0213:2004 character sets overlap significantly. In some cases their prototypical glyphs differ. When building fonts that support both JIS X 0212-1990 and JIS X 0213:2004 (such as those supporting the Adobe-Japan 1-6 character collection), it is recommended that JIS X 0213:2004 forms be preferred as the encoded form. The 'hojo' feature is used to access the JIS X 0212-1990 glyphs for the cases when the JIS X 0213:2004 form is encoded.

Example: The glyph  is replaced by the glyph  .

Recommended implementation: One-for-one substitution of JIS X 0213:2004 glyphs by the corresponding JIS X 0212-1990 glyph.

Application interface: For GIDs found in the hojo coverage table, the application passes the GIDs to the table and gets back one new GID for each.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Used only with Kanji script.

Feature interaction: This feature is exclusive with jp78, jp83, jp90, nlck and similar features. It can be combined with the palt, vpal, vert and vrt2 features.

Tag: 'hwid'

Friendly name: Half Widths

Function: Replaces glyphs on proportional widths, or fixed widths other than half an em, with glyphs on half-em (en) widths. Many CJKV fonts have glyphs which are set on multiple widths; this feature selects the half-em version. There are various contexts in which this is the preferred behavior, including compatibility with older desktop documents.

Example: The user may replace a proportional Latin glyph with the same character set on a half-em width.

Recommended implementation: The font may contain alternate glyphs designed to be set on half-em widths (GSUB lookup type 1), or it may specify alternate metrics for the original glyphs (GPOS lookup type 1) which adjust their spacing to fit in half-em widths.

Application interface: For GIDs found in the hwid coverage table, the application passes the GIDs to the table and gets back either new GIDs or positional adjustments (XPlacement and XAdvance).

UI suggestion: This feature would normally be off by default.

Script/language sensitivity: Generally used only in CJKV fonts.

Feature interaction: This feature is mutually exclusive with all other glyph-width features (e.g. tnum, fwid, halt, qwid and twid), which should be turned off when it's applied. It deactivates the kern feature.

Tag: 'init'

Friendly name: Initial Forms

Function: Replaces glyphs at the beginnings of words with alternate forms designed for this use. This is common in Latin connecting scripts, and required in various non-Latins like Arabic.

Example: In the typeface Ex Ponto, the default t in the word 'type' is replaced with the t.begin form.

Recommended implementation: The init table maps default alphabetic forms to corresponding beginning forms (GSUB lookup type 1).

Application interface: The application is responsible for noting word boundaries. For GIDs at the beginnings of words and found in the init coverage table, the application passes a GID to the feature and gets back a new GID.

UI suggestion: This feature should be active by default.

Script/language sensitivity: Can be used in any alphabetic script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also medi and fina.

Tag: "isol"

Friendly name: Isolated Forms

Function: Replaces the nominal form of glyphs with their isolated forms.

Example: In Arabic, if the Alef is followed by Lam, the default glyph for Alef is replaced with its isolated form.

Recommended implementation: The **isol** table maps default alphabetic forms to corresponding isolated forms (GSUB lookup type 1).

Application interface: For GIDs found in the **isol** coverage table, the application passes a GID to the feature and gets back a new GID for the isolated form.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Can be used in any cursive script.

Feature interaction: This feature overrides the results of all other features. See also init, medi, fina.

Tag: 'ital'

Friendly name: Italics

Function: Some fonts (such as Adobe's Pro Japanese fonts) will have both Roman and Italic forms of some characters in a single font. This feature replaces the Roman glyphs with the corresponding Italic glyphs.

Example: The user would apply this feature to replace B with *B*.

Recommended implementation: The ital table maps the Roman forms in a font to the corresponding Italic forms (GSUB lookup type 1).

Application interface: For GIDs found in the ital coverage table, the application passes the GIDs to the table and gets back one new GID for each.

UI suggestion: When a user selects text and applies an Italic style, an application should check for this feature and use it if present.

Script/language sensitivity: Applies mostly to Latin; but it should be noted that many non-Latin fonts contain Latin as well.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. In CJKV fonts it should activate the kern feature (which would be on anyway in other scripts).

Tag: "jalt"

Friendly name: Justification Alternates

Function: Improves justification of text by replacing glyphs with alternate forms specifically designed for this purpose (they would have less or more advance width as need may be).

Example: In the Arabic script, providing alternate forms for line final glyphs would result in better justification and reduce the use of tatweels (Kashidas). eg. replacing a Swash Kaf with an alternate form.

Recommended implementation: The **jalt** table maps the initial, medial, final or isolated forms to their corresponding alternate forms (GSUB lookup type 3).

Application interface: The application is responsible for noting line ends/boundaries. For GIDs found in the **jalt** coverage table, the application passes a GID to the feature and gets back a new GID.

UI suggestion: This feature could be made active or inactive by default, at the user's preference.

Script/language sensitivity: Can be used in any cursive script.

Feature interaction: If the font contains **init**, **medi**, **fina**, **isol** features, these need to be called prior to calling this feature.

Tag: 'jp78'

Friendly name: JIS78 Forms

Function: This feature replaces default (JIS90) Japanese glyphs with the corresponding forms from the JIS C 6226-1978 (JIS78) specification.

Example: The user would invoke this feature to replace kanji character U+5516 with U+555E.

Recommended implementation: When JIS90 glyphs correspond to JIS78 forms, the jp78 table maps each of those glyphs to their alternates. While many of these substitutions are one-to-one (GSUB lookup type 1), others require a selection from a set (GSUB lookup type 3). The manufacturer may choose to build two tables (one for each lookup type) or only one which uses lookup type 3 for all substitutions.

Application interface: For GIDs found in the jp78 coverage table, the application passes the GIDs to the table and gets back one or more new GIDs. If more than one GID is returned, the application must provide a means for the user to select the one desired. The application may assume that the first glyph in a set is the preferred form, so the font developer should order them accordingly.

NOTE This is a change of character code. Besides the original character code, the application should store the code for the new character.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies only to Japanese.

Feature interaction: This feature is mutually exclusive with all other features, which should be turned off when it's applied, except the palt, vpal, vert and vrt2 features, which may be used in addition.

Tag: 'jp83'

Friendly name: JIS83 Forms

Function: This feature replaces default (JIS90) Japanese glyphs with the corresponding forms from the JIS X 0208-1983 (JIS83) specification.

Example: Because of the Han unification in Unicode, there are no JIS83 glyphs which have distinct Unicode values, so the substitution cannot be described specifically.

Recommended implementation: When JIS90 glyphs correspond to JIS83 forms, the jp83 table maps each of those glyphs to their alternates (GSUB lookup type 1).

Application interface: For GIDs found in the jp83 coverage table, the application passes the GIDs to the table and gets back one or more new GIDs. If more than one GID is returned, the application must provide a means for the user to select the one desired.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies only to Japanese.

Feature interaction: This feature is mutually exclusive with all other features, which should be turned off when it's applied, except the palt, vpal, vert and vrt2 features, which may be used in addition.

Tag: 'jp90'

Friendly name: JIS90 Forms

Function: This feature replaces Japanese glyphs from the JIS78 or JIS83 specifications with the corresponding forms from the JIS X 0208-1990 (JIS90) specification.

Example: The user would invoke this feature to replace kanji character U+555E with U+5516.

Recommended implementation: The jp90 table maps each JIS78 and JIS83 form in a font to JIS90 forms (GSUB lookup type 1). The application stores a record of any simplified forms which resulted from substitutions (the jp78 or jp83 features); for such forms, applying the jp90 feature undoes the previous substitution. When there is no record of a substitution, the application uses the jp90 table to get back to the default form.

Application interface: For GIDs found in the jp90 coverage table, the application passes the GIDs to the table and gets back one new GID for each.

NOTE This is a change of character code. Besides the original character code, the application should store the code for the new character.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies only to Japanese.

Tag: 'jp04'

Friendly name: JIS2004 Forms

Registered by: Adobe

Function: The National Language Council (NLC) of Japan has defined new glyph shapes for a number of JIS characters, which were incorporated into JIS X 0213:2004 as new prototypical forms. The 'jp04' feature is a subset of the 'nlck' feature, and is used to access these prototypical glyphs in a manner that maintains the integrity of JIS X 0213:2004.

Example: The glyph  is replaced by the glyph  .

Recommended implementation: One-for-one substitution of non-JIS X 0213:2004 glyphs by the corresponding JIS X 0213:2004 glyph.

Application interface: For GIDs found in the jp04 coverage table, the application passes the GIDs to the table and gets back one new GID for each.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Used only with Kanji script.

Feature interaction: This feature is exclusive with jp78, jp83, jp90, nlck and similar features. It can be combined with the palt, vpal, vert and vrt2 features.

Tag: 'kern'

Friendly name: Kerning

Function: Adjusts amount of space between glyphs, generally to provide optically consistent spacing between glyphs. Although a well-designed typeface has consistent inter-glyph spacing overall, some glyph combinations require adjustment for improved legibility. Besides standard adjustment in the horizontal direction, this feature can supply size-dependent kerning data via device tables, "cross-stream" kerning in the Y text direction, and adjustment of glyph placement independent of the advance adjustment.

NOTE This feature may apply to runs of more than two glyphs, and would not be used in monospaced fonts. This feature does not apply to text set vertically.

Example: The o is shifted closer to the T in the combination "To."

Recommended implementation: The font stores a set of adjustments for pairs of glyphs (GPOS lookup type 2 or 8). These may be stored as one or more tables matching left and right classes, &/or as individual pairs. Additional adjustments may be provided for larger sets of glyphs (e.g. triplets, quadruplets, etc.) to overwrite the results of pair kerns in particular combinations.

Application interface: The application passes a sequence of GIDs to the kern table, and gets back adjusted positions (XPlacement, XAdvance, YPlacement and YAdvance) for those GIDs. When using the type 2 lookup on a run of glyphs, it's critical to remember to not consume the last glyph, but to keep it available as the first glyph in a subsequent run (this is a departure from normal lookup behavior).

UI suggestion: This feature should be active by default for horizontal text setting. Applications may wish to allow users to add further manually-specified adjustments to suit specific needs and tastes.

Script/language sensitivity: None.

Feature interaction: If 'kern' is activated, 'palt' must also be activated if it exists. (If 'palt' is activated, there is no requirement that 'kern' must also be activated.) May be used in addition to any other feature except those which result in fixed (uniform) advance widths (e.g. fwid, halt, hwid, qwid and twid).

Tag: 'lfbd'

Friendly name: Left Bounds

Function: Aligns glyphs by their apparent left extents at the left ends of horizontal lines of text, replacing the default behavior of aligning glyphs by their origins. This feature is called by the Optical Bounds (opbd) feature above.

Example: Succeeding lines beginning with T, D and W would shift to the left by varying amounts when the text is left-justified and this feature is applied.

Recommended implementation: Values for affected glyphs describe the amount by which the placement and advance width should be altered (GPOS lookup type 1).

Application interface: For GIDs found in the lfbd coverage table, the application passes a GID to the table and gets back a new XPlacement and XAdvance value.

UI suggestion: This feature is called by an application when the user invokes the opbd feature.

Script/language sensitivity: None.

Feature interaction: Should not be applied to glyphs which use fixed-width features (e.g. fwid, halt, hwid, qwid and twid) or vertical features (e.g. vert, vrt2, vpal, valt and vhal). Is called by the opbd feature.

Tag: 'liga'

Friendly name: Standard Ligatures

Function: Replaces a sequence of glyphs with a single glyph which is preferred for typographic purposes. This feature covers the ligatures which the designer/manufacturer judges should be used in normal conditions.

Example: The glyph for ffl replaces the sequence of glyphs f f l.

Recommended implementation: The liga table maps sequences of glyphs to corresponding ligatures (GSUB lookup type 4). Ligatures with more components must be stored ahead of those with fewer components in order to be found. The set of standard ligatures will vary by design and script.

Application interface: For sets of GIDs found in the liga coverage table, the application passes the sequence of GIDs to the table and gets back a single new GID. Full sequences must be passed.

UI suggestion: This feature serves a critical function in some contexts, and should be active by default.

Script/language sensitivity: Applies to virtually all scripts.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: "ljmo"

Friendly name: Leading Jamo Forms

Function: Substitutes the leading jamo form of a cluster.

Example: In Hangul script, the jamo cluster is composed of three parts (leading consonant, vowel, and trailing consonant). When a sequence of leading class jamos are found, their combined leading jamo form is substituted.

Recommended implementation: The **ljmo** table maps the sequence required to convert a series of jamos into its leading jamo form (GSUB lookup type 4).

Application interface: For substitutions defined in the **ljmo** table, the application passes the sequence of GIDs to the feature, and gets back the GID for the leading jamo form.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required for Hangul script when Ancient Hangul writing system is supported.

Feature interaction: This feature overrides the results of all other features.

Tag: 'lnum'

Friendly name: Lining Figures

Function: This feature changes selected figures from oldstyle to the default lining form.

Example: The user invokes this feature in order to get lining figures, which fit better with all-capital text. Various characters designed to be used with figures may also be covered by this feature. In cases where lining figures are the default form, this feature would undo previous substitutions.

Recommended implementation: The lnum table maps each oldstyle figure, and any associated characters to the corresponding lining form (GSUB lookup type 1).

Application interface: For GIDs found in the lnum coverage table, the application passes a GID to the onum table and gets back a new GID. Even if the current figures resulted from an earlier substitution, it may not be

correct to simply revert to the original GIDs, because of interaction with the figure width features, so it's best to use this table.

UI suggestion: This feature should be inactive by default. Users can switch between the lining and oldstyle sets by turning this feature on or off.

NOTE This feature is distinct from the figure width features (pnum and tnum). When the user invokes this feature, the application may wish to inquire whether a change in width is also desired.

Script/language sensitivity: None.

Feature interaction: This feature overrides the results of the Oldstyle Figures feature (onum).

Tag: 'locl'

Friendly name: Localized Forms

Function: Many scripts used to write multiple languages over wide geographical areas have developed localized variant forms of specific letters, which are used by individual literary communities. For example, a number of letters in the Bulgarian and Serbian alphabets have forms distinct from their Russian counterparts and from each other. In some cases the localized form differs only subtly from the script 'norm', in others the forms are radically distinct. This feature enables localized forms of glyphs to be substituted for default forms.

Example: The user applies this feature to text to enable localized Bulgarian forms of Cyrillic letters; alternatively, the feature might enable localized Russian forms in a Bulgarian manufactured font in which the Bulgarian forms are the default characters.

Recommended implementation: For a given Unicode value, the font contains glyphs for two or more locales. The locl table maps GIDs for default forms to GIDs for corresponding localized alternatives. These are one-to-one substitutions (GSUB lookup type 1).

Application interface: Localized forms are associated with specific languages and are activated by language tags. Which glyph is used as the localized form should be determined by the language the user has specified. The user can switch localized forms by selecting a new language, or may enable default forms by switching off the locl feature.

UI suggestion: This feature should be active by default.

Script/language sensitivity: Applies to all scripts and languages; but of course behavior differs by script and language.

Feature interaction: This feature can be used in combination with any other feature. It replaces and extends the earlier locale-specific tags zhcn, zhtw, jajp, kokr and vivn which had been defined for CJKV scripts.

Tag: 'ltra'

Friendly name: Left-to-right glyph alternates

Registered by: Adobe

Function: This feature applies glyphic variants (other than mirrored forms) appropriate for left-to-right text. (For mirrord forms, see 'ltrm'.)

Recommended implementation: These are required to be glyph substitutions, and it is recommended that they be one-to-one (GSUB lookup type 1).

Application interface: See section “Left-to-right and right-to-left text” in the subclause 6.1.4 "Text processing with OFF Layout".

UI suggestion: None

Script/language sensitivity: Left-to-right runs of text.

Feature interaction: This feature is to be applied simultaneously with other pre-shaping features such as 'ccmp' and 'locl'.

Tag: 'ltrm'

Friendly name: Left-to-right mirrored forms

Registered by: Adobe

Function: This feature applies mirrored forms appropriate for left-to-right text. (For left-to-right glyph alternates, see 'ltra'.)

Example: The Old South Arabian script is a case of a strong right-to-left script that can have lines laid out left-to-right, in which case some glyphs would need to be mirrored with the 'ltrm' feature.

Recommended implementation: These are required to be glyph substitutions, and it is recommended that they be one-to-one (GSUB lookup type 1).

Application interface: See section "Left-to-right and right-to-left text" in the subclause 6.1.4 "Text processing with OFF Layout".

UI suggestion: None

Script/language sensitivity: Left-to-right runs of text, also see *Example* above.

Feature interaction: This feature is to be applied simultaneously with other pre-shaping features such as 'ccmp' and 'locl'.

Tag: 'mark'

Friendly name: Mark Positioning

Function: Positions mark glyphs with respect to base glyphs.

Example: In the Arabic script, positioning the Hamza above the Yeh.

Recommended implementation: This feature may be implemented as a MarkToBase Attachment lookup (GPOS LookupType = 4) or a MarkToLigature Attachment lookup (GPOS LookupType = 5).

Application interface: For GIDs found in the **mark** coverage table, the application gets back the positioning or position adjustment values for the mark glyph.

UI suggestion: This feature should be active by default.

Script/language sensitivity: None.

Feature interaction: None.

Tag: "med2"

Friendly name: Medial Form #3

Function: Replaces Alaph glyphs in the middle of Syriac words when the preceding base character cannot be joined to.

Example: When an Alaph is preceded by a Waw, the Alaph would be replaced by an appropriate form. This feature is used only for the Syriac script alaph character.

Recommended implementation: The **med2** table maps default alphabetic forms to corresponding medial forms (GSUB lookup type 5).

Application interface: The application is responsible for noting word boundaries. For GIDs in the middle of words and found in the med2 coverage table, the application passes a GID to the feature and gets back a new GID.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Used only with the Syriac script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also init and fina.

Tag: 'medi'

Friendly name: Medial Forms

Function: Replaces glyphs in the middles of words (i.e. following a beginning and preceding an end) with alternate forms designed for this use.

NOTE This is different from the default form, which is designed for stand-alone use. This is common in Latin connecting scripts, and required in various non-Latins like Arabic.

Example: In the typeface Caflisch Script, the y and p in the word 'type' are replaced by the y.med and p.med forms.

Recommended implementation: The medi table maps default alphabetic forms to corresponding medial forms (GSUB lookup type 1).

Application interface: The application is responsible for noting word boundaries. For GIDs in the middles of words and found in the medi coverage table, the application passes a GID to the feature and gets back a new GID.

UI suggestion: This feature should be active by default.

Script/language sensitivity: Can be used in any alphabetic script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also init and fina.

Tag: 'mgrk'

Friendly name: Mathematical Greek

Function: Replaces standard typographic forms of Greek glyphs with corresponding forms commonly used in mathematical notation (which are a subset of the Greek alphabet).

Example: The user applies this feature to U+03A3 (Sigma), and gets U+2211 (summation).

Recommended implementation: The mgrk table maps Greek glyphs to the corresponding forms used for mathematics (GSUB lookup type 1).

Application interface: For GIDs found in the mgrk coverage table, the application passes a GID to the feature table and gets back a new GID.

NOTE This is a change of semantic value. Besides the original character codes, the application should store the code for the new character.

UI suggestion: This feature should be off by default in most applications. Math-oriented applications may want to activate this feature by default.

Script/language sensitivity: Could apply to any font which includes coverage for the Greek script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: "mkmk"

Friendly name: Mark to Mark Positioning

Function: Positions marks with respect to other marks. Required in various non-Latin scripts like Arabic.

Example: In Arabic, the ligaturised mark Ha with Hamza above it; can also be obtained by positioning these marks relative to one another.

Recommended implementation: This feature may be implemented as a MarkToMark Attachment lookup (GPOS lookup type 6).

Application interface: The application gets back positioning values or positional adjustments for marks.

UI suggestion: This feature should be active by default.

Script/language sensitivity: None.

Feature interaction: None.

Tag: 'mset'

Function: Positions Arabic combining marks in fonts for Windows 95 using glyph substitution

Example: In Arabic, the Hamza is positioned differently when placed above a Yeh Barree as compared to the Alef.

Tag: 'nalt'

Friendly name: Alternate Annotation Forms

Function: Replaces default glyphs with various notational forms (e.g. glyphs placed in open or solid circles, squares, parentheses, diamonds or rounded boxes). In some cases an annotation form may already be present, but the user may want a different one.

Example: The user invokes this feature to get U+3200 (the circled form of 'ga') from U+3131 (hangul 'ga').

Recommended implementation: The nalt table maps GIDs for various standard forms to one or more corresponding annotation forms. While many of these substitutions are one-to-one (GSUB lookup type 1), others require a selection from a set (GSUB lookup type 3). The manufacturer may choose to build two tables (one for each lookup type) or only one which uses lookup type 3 for all substitutions. If more than one form is present, the set of forms for each character should be ordered consistently - both within the font and across the family.

Application interface: For GIDs found in the nalt coverage table, the application passes a GID and gets back a set of new GIDs, then stores the one selected by the user.

UI suggestion: This feature should be inactive by default. The application must provide a means for the user to select the desired form from the set returned by the table. It can note the position of the selected form in a set of alternates, and offer the glyph at that position as the default selection the next time this feature is invoked. In the absence of such prior information, the application may assume that the first glyph in a set is the preferred form, so the font developer should order them accordingly.

Script/language sensitivity: Used mostly in CJKV fonts, but can apply to European scripts.

Feature interaction: This feature is mutually exclusive with all other features, which should be turned off when it's applied, except the vert and vrt2 features, which may be used in addition.

Tag: "nlck"

Friendly name: NLC Kanji Forms

Function: The National Language Council (NLC) of Japan has defined new glyph shapes for a number of JIS characters. The 'nlck' feature is used to access those glyphs.

Example: The glyph  is replaced by the glyph  .

Recommended implementation: One-for-one substitution of non-NLC glyphs by the corresponding NLC glyph.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Used only with Kanji script.

Feature interaction: This feature is exclusive with the 'jp78', 'jp83', 'jp90' and similar features. It can be combined with the 'palt', 'vpal', 'vert' and 'vrt2' features.

Tag: "nukt"

Friendly name: Nukta Forms

Function: Produces Nukta forms in Indic scripts.

Example: In Hindi (Devanagari script), a consonant when combined with a nukta gives its nukta form.

Recommended implementation: The **nukt** table maps the sequence of a consonant followed by a nukta to the consonant's nukta form (GSUB lookup type 4).

Application interface: The application passes the sequence of GIDs (consonant and nukta), to the table, and gets back the GID for the nukta form.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required in Indic scripts.

Feature interaction: This feature overrides the results of all other features.

Tag: 'numr'

Friendly name: Numerators

Function: Replaces selected figures which precede a slash with numerator figures, and replaces the typographic slash with the fraction slash.

Example: In the string 11/17 selected by the user, the application turns the 11 into numerators, and the slash into a fraction slash when the user applies the fraction feature (frac).

Recommended implementation: The numr table maps sets of figures and related characters to corresponding numerator glyphs in the font. It also maps the typographic slash (U+002F) to the fraction slash (U+2044). All mappings are one-to-one (GSUB lookup type 1).

Application interface: For GIDs found in the numr coverage table, the application passes a GID to the table and gets back a new GID.

UI suggestion: This feature should normally be called by an application when the user applies the frac feature.

Script/language sensitivity: None.

Feature interaction: This feature supports frac. It may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'onum'

Friendly name: Oldstyle Figures

Function: This feature changes selected figures from the default lining style to oldstyle form.

Example: The user invokes this feature to get oldstyle figures, which fit better into the flow of normal upper- and lowercase text. Various characters designed to be used with figures may also have oldstyle versions.

Recommended implementation: The onum table maps each lining figure, and any associated characters, to the corresponding oldstyle form (GSUB lookup type 1).

Application interface: For GIDs found in the onum coverage table, the application passes a GID to the onum table and gets back a new GID.

UI suggestion: Users can switch between the lining and oldstyle sets by turning this feature on or off.

NOTE This feature is separate from the figure-width features pnum and tnum. When the user changes figure style, the application may want to query whether a change in width is also desired.

Script/language sensitivity: None.

Feature interaction: This feature overrides the results of the Lining Figures feature (lnum).

Tag: 'opbd'

Friendly name: Optical Bounds

Function: Aligns glyphs by their apparent left or right extents in horizontal setting, or apparent top or bottom extents in vertical setting, replacing the default behavior of aligning glyphs by their origins. Another name for this behavior would be visual justification. The optical edge of a given glyph is only indirectly related to its advance width or bounding box; this feature provides a means for getting true visual alignment.

Example: Succeeding lines beginning with T, D and W would shift to the left by varying amounts when the text is left-justified and this feature is applied. Succeeding lines ending with r, h and y would likewise shift to the right by differing degrees when the text is right-justified and this feature is applied.

Recommended implementation: Values for affected glyphs are defined with a separate record for left, right, top, and bottom. Each record describes the amount by which the placement and advance width should be altered (GPOS lookup type 1).

Application interface: For GIDs found in the opbd coverage table, the application calls one of two related tables, depending on the position of the glyph. For glyphs at the left end of a horizontal line, it calls the lfbd table, for glyphs at the right end of a horizontal line, it calls the rtbd table.

UI suggestion: This feature should be active by default. It effectively changes the line length, so justification algorithms should account for this adjustment.

Script/language sensitivity: None.

Feature interaction: Should not be applied to glyphs which use fixed-width features (e.g. fwid, halt, hwid, qwid and twid) or vertical features (e.g. vert, vrt2, vpal, valt and vhal). Uses lfbd and rtbd features.

Tag: 'ordn'

Friendly name: Ordinals

Function: Replaces default alphabetic glyphs with the corresponding ordinal forms for use after figures. One exception to the follows-a-figure rule is the numero character (U+2116), which is actually a ligature substitution, but is best accessed through this feature.

Example: The user applies this feature to turn 2.o into 2.º (abbreviation for secundo).

Recommended implementation: The ordn table maps various lowercase letters to corresponding ordinal forms in a chained context (GSUB lookup type 6), and the sequence No to the numero character (GSUB lookup type 4).

Application interface: For sets of GIDs found in the clig coverage table, the application passes the sequence of GIDs to the table and gets back new GIDs. Full sequences must be passed.

NOTE This may be a change of semantic value. Besides the original character codes, the application should store the code for the new character.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies mostly to Latin script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'ornm'

Friendly name: Ornaments

Function: This is a dual-function feature, which uses two input methods to give the user access to ornament glyphs (e.g. fleurons, dingbats and border elements) in the font. One method replaces the bullet character with a selection from the full set of available ornaments; the other replaces specific "lower ASCII" characters with ornaments assigned to them. The first approach supports the general or browsing user; the second supports the power user.

Example: The user inputs qwwwwwwwwwe to form the top of a flourished box in Adobe Caslon, or inputs the bullet character, then chooses the thistle dingbat.

Recommended implementation: The ornm table maps all ornaments in a font to the bullet character (GSUB lookup type 3) and each ornament in a font to a corresponding alphanumeric character (GUSB lookup type 1). The manufacturer may choose to build two tables (one for each lookup type) or only one which uses lookup type 3 for all substitutions. As in any one-from-many substitution, alternates present in more than one face should be ordered consistently across a family, so that those alternates can work correctly when switching between family members.

Application interface: When this feature is invoked, the application must note whether the selected text is the bullet character (U+2022) or alphanumeric characters. In the first case, it passes the GID for bullet to the ornm table and gets back a set of GIDs, and gives the user a means to select from among them. In the second case, for GIDs found in the ornm coverage table, it passes GIDs to the ornm table and gets back new GIDs.

UI suggestion: This feature should be inactive by default. When more than one GID is returned (the bullet case), an application could display the forms sequentially in context, or present a palette showing all the forms at once, or give the user a choice between these approaches. Once the user has selected a specific ornament, that one should be the default selection the next time the bullet is typed. In the absence of such prior information, the application may assume that the first ornament in a set is the preferred form, so the font developer should order them accordingly.

Script/language sensitivity: None.

Feature interaction: This feature is mutually exclusive with all other substitution (GSUB) features, which should be turned off when it's applied.

Tag: 'palt'

Friendly name: Proportional Alternate Widths

Function: Respaces glyphs designed to be set on full-em widths, fitting them onto individual (more or less proportional) horizontal widths. This differs from pwid in that it does not substitute new glyphs (GPOS, not GSUB feature). The user may prefer the monospaced form, or may simply want to ensure that the glyph is well-fit and not rotated in vertical setting (Latin forms designed for proportional spacing would be rotated).

Example: The user may invoke this feature in a Japanese font to get Latin, Kanji, Kana or Symbol glyphs with the full-width design but individual metrics.

Recommended implementation: The font specifies alternate metrics for the full-width glyphs (GPOS lookup type 1).

Application interface: For GIDs found in the palt coverage table, the application passes the GIDs to the table and gets back positional adjustments (XPlacement, XAdvance, YPlacement and YAdvance).

UI suggestion: This feature would be off by default.

Script/language sensitivity: Used mostly in CJKV fonts.

Feature interaction: This feature is mutually exclusive with all other glyph-width features (e.g. fwid, halt, hwid, qwid and twid), which should be turned off when it's applied. If palt is activated, there is no requirement that kern must also be activated. If kern is activated, palt must also be activated if it exists.. See also vpal.

Tag:'pcap'

Friendly name: Petite Capitals

Function: Some fonts contain an additional size of capital letters, shorter than the regular smallcaps and whimsically referred to as petite caps. Such forms are most likely to be found in designs with a small lowercase x-height, where they better harmonise with lowercase text than the taller smallcaps (for examples of petite caps, see the Emigre type families Mrs Eaves and Filosofia). This feature turns lowercase characters into petite capitals. Forms related to petite capitals, such as specially designed figures, may be included.

Example: The user enters text as lowercase or mixed case, and gets petite cap text or text with regular uppercase and petite caps.

NOTE Some designers, might extend the petite cap lookups to include uppercase-to-smallcap substitutions, creating a shifting hierarchy of uppercase forms.

Recommended implementation: The pcap table maps lowercase glyphs to the corresponding petite cap forms (GSUB lookup type 1).

Application interface: For GIDs found in the pcap coverage table, the application passes GIDs to the pcap table, and gets back new GIDs. Petite cap substitutions should follow language rules for smallcap (smcp) substitutions.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies only to scripts with both upper- and lowercase forms (e.g. Latin, Cyrillic, Greek).

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'pkna'

Friendly name: Proportional Kana

Function: Replaces glyphs, kana and kana-related, set on uniform widths (half or full-width) with proportional glyphs.

Example: The user may invoke this feature in a Japanese font to get a proportional glyph instead of a corresponding half- or full-width kana glyph.

Recommended implementation: The font contains alternate kana and kana-related glyphs designed to be set on proportional widths (GSUB lookup type 1).

Application interface: For GIDs found in the pkna coverage table, the application passes the GIDs to the table and gets back new GIDs.

UI suggestion: This feature would normally be off by default.

Script/language sensitivity: Generally used only in Japanese fonts.

Feature interaction: This feature is mutually exclusive with all other glyph-width features (e.g. fwid, halt, hwid, palt, pwid, qwid, twid, and vhal), which should be turned off when it's applied. Applying this feature should activate the kern feature.

Tag: 'pnum'

Friendly name: Proportional Figures

Function: Replaces figure glyphs set on uniform (tabular) widths with corresponding glyphs set on glyph-specific (proportional) widths. Tabular widths will generally be the default, but this cannot be safely assumed. Of course this feature would not be present in monospaced designs.

Example: The user may apply this feature to get even spacing for lining figures used as dates in an all-cap headline.

Recommended implementation: In order to simplify associated kerning and get the best glyph design for a given width, this feature should use new glyphs for the figures, rather than only adjusting the fit of the tabular glyphs (although some may be simple copies); i.e. not a GPOS feature. The pnum table maps tabular versions of lining and/or oldstyle figures to corresponding proportional glyphs (GSUB lookup type 1).

Application interface: For GIDs found in the pnum coverage table, the application passes GIDs to the pnum table and gets back new GIDs.

UI suggestion: This feature should be off by default. The application may want to query the user about this feature when the user changes figure style (onum or lnum).

Script/language sensitivity: None.

Feature interaction: This feature overrides the results of the Tabular Figures feature (tnum).

Tag: 'pref'

Friendly name: Pre-base Forms

Function: Substitutes the pre-base form of a consonant.

In some scripts of south or southeast Asia, such as Khmer, the conjoined form of certain consonants is always denoted as a pre-base form. In the case of some scripts of south India, variations in writing conventions exist such that a conjoined Ra consonant may be written as a pre-base form, or a below-base or post-base form. Fonts may be designed to support one or another convention. If a font is designed to support a writing convention in which conjoined Ra is a pre-base form, the Pre-Base Forms feature would be used.

Example: In the Khmer script, the consonant Ra has a pre-base subscript form subscript called Coeng Ra. When the sequence of Coeng followed by Ra, its pre-base form is substituted.

Recommended implementation: The **pref** table maps the sequence required to convert a consonant into its pre-base form (GSUB lookup type 4).

Application interface: For substitutions defined in the **pref** table, the application passes the sequence of GIDs to the table, and gets back the GID for the pre base form of the consonant. When shaping scripts of south India, the application may examine the results of processing this feature to determine if the conjoining consonant form needs to be re-ordered.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Required in Khmer and Myanmar (Burmese) scripts that have pre-base forms for consonants.It is also required for southern Indic scripts that may display a pre-base form of Ra, such as Malayalam or Telugu.

Feature interaction: This feature is used in conjunction with certain other features to derive required forms of certain Indic and southeast Asian scripts. The application is expected to process this feature and certain other features in an appropriate order to obtain the correct set of basic forms for the given script. For Indic scripts, the following features should be applied in order: nukt, akhn, rphf, rkrf, pref, blwf, half, pstf, cjct. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: 'pres'

Friendly name: Pre-base Substitutions

Function: Produces the pre-base forms of conjuncts in Indic scripts. It can also be used to substitute the appropriate glyph variant for pre-base vowel signs.

Example: In the Gujarati (Indic) script, the doubling of consonant Ka requires the first Ka to be substituted by its pre-base form. This in turn ligates with the second Ka. Applying this feature would result in the ligaturised version of the doubled Ka.

Recommended implementation: The **pres** table maps a sequence of consonants separated by the virama (halant), to the ligated conjunct form (GSUB lookup type 4). In the case of pre-base matra substitution, the appropriate matra can be substituted using contextual substitution (GSUB lookup type 5).

Application interface: For substitutions defined in the **pres** table, the application passes the sequence of GIDs to the feature, and gets back the GID for the ligature (or matra as the case may be).

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required in Indic scripts.

Feature interaction: This feature overrides the results of all other features.

Tag: 'pstf'

Friendly name: Post-base Forms

Function: Substitutes the post-base form of a consonant.

Example: In the Gurmukhi (Indic) script, the consonant Ya has a post base form. When the Ya is used as the second consonant in conjunct formation, its post-base form is substituted.

Recommended implementation: The **pstf** table maps the sequence required to convert a consonant into its post-base form (GSUB lookup type 4).

Application interface: For substitutions defined in the **pstf** table, the application passes the sequence of GIDs to the feature, and gets back the GID for the post base form of the consonant.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Required in scripts of south and southeast Asia that have post-base forms for consonants eg: Gurmukhi, Malayalam, Khmer.

Feature interaction: This feature is used in conjunction with certain other features to derive required forms of Indic and other related scripts. The application is expected to process this feature and certain other features in an appropriate order to obtain the correct set of basic forms for the given script. For Indic scripts, the following features should be applied in order: nukt, akhn, rphf, rkrf, pref, blwf, half, pstf, cjct. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: 'psts'

Friendly name: Post-base Substitutions

Function: Substitutes a sequence of a base glyph and post-base glyph, with its ligaturised form.

Example: In the Malayalam (Indic) script, the consonant Va has a post base form. When the Va is doubled to form a conjunct- VVa, the first Va [base] and the post base form that follows it, is substituted with a ligature.

Recommended implementation: The **psts** table maps identified conjunct formation sequences to corresponding ligatures (GSUB lookup type 4).

Application interface: For substitutions defined in the **psts** table, the application passes the sequence of GIDs to the feature, and gets back the GID for the ligature.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Can be used in any alphabetic script. Required in Indic scripts.

Feature interaction: This feature overrides the results of all other features.

Tag: 'pwid'

Friendly name: Proportional Widths

Function: Replaces glyphs set on uniform widths (typically full or half-em) with proportionally spaced glyphs. The proportional variants are often used for the Latin characters in CJKV fonts, but may also be used for Kana in Japanese fonts.

Example: The user may invoke this feature in a Japanese font to get a proportionally-spaced glyph instead of a corresponding half-width Roman glyph or a full-width Kana glyph.

Recommended implementation: The font contains alternate glyphs designed to be set on proportional widths (GSUB lookup type 1).

Application interface: For GIDs found in the pwid coverage table, the application passes the GIDs to the table and gets back new GIDs.

UI suggestion: Applications may want to have this feature active or inactive by default depending on their markets.

Script/language sensitivity: Although used mostly in CJKV fonts, this feature could be applied in European scripts.

Feature interaction: This feature is mutually exclusive with all other glyph-width features (e.g. fwid, halt, hwid, palt, qwid, twid, valt and vhal), which should be turned off when it's applied. Applying this feature should activate the kern feature.

Tag: 'qwid'

Friendly name: Quarter Widths

Function: Replaces glyphs on other widths with glyphs set on widths of one quarter of an em (half an en). The characters involved are normally figures and some forms of punctuation.

Example: The user may apply qwid to place a four-digit figure in a single slot in a column of vertical text.

Recommended implementation: The font may contain alternate glyphs designed to be set on quarter-em widths (GSUB lookup type 1), or it may specify alternate metrics for the original glyphs (GPOS lookup type 1) which adjust their spacing to fit in quarter-em widths.

Application interface: For GIDs found in the qwid coverage table, the application passes the GIDs to the table and gets back either new GIDs or positional adjustments (XPlacement and XAdvance).

UI suggestion: This feature would normally be off by default.

Script/language sensitivity: Generally used only in CJKV fonts.

Feature interaction: This feature is mutually exclusive with all other glyph-width features (e.g. fwid, halt, hwid and twid), which should be turned off when it's applied. It deactivates the kern feature.

Tag: 'rand'

Friendly name: Randomize

Function: In order to emulate the irregularity and variety of handwritten text, this feature allows multiple alternate forms to be used.

Example: The user applies this feature in FF Kosmic to get three forms of f in one word.

Recommended implementation: The rand table maps GIDs for default glyphs to one or more GIDs for corresponding alternates (GSUB lookup type 3).

Application interface: For GIDs found in the rand coverage table, the application passes a GID to the rand table and gets back one or more new GIDs. The application selects one of these either by a pseudo-random

algorithm, or by noting the sequence of IDs returned, storing that sequence, and stepping through that set as the corresponding character code is invoked.

UI suggestion: This feature should be enabled/disabled via a preference setting; "enabled" is the recommended default.

Script/language sensitivity: None.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'rkrf'

Friendly name: Rakar Forms

Function: Produces conjoined forms for consonants with rakar in Devanagari and Gujarati scripts.

In Devanagari and Gujarati scripts, consonant clusters involving Ra following another consonant are denoted by conjoining an alternate form of Ra to the preceding consonant. Depending on the particular syllable, the preceding consonant may be denoted in its full form or as a half form. Because of interactions involving other behaviors of these scripts, a font implementation may need to process substitution lookups for rakar forms and half forms in a particular sequence in order to derive the appropriate display for various sequences. In recommended usage, the Rakar Forms feature is processed before the Half Forms feature; a half form for a given consonant-Ra combination can be derived by subsequent application of the Half Forms feature. This sequential ordering allows for correct display results.

Example: In Hindi (Devanagari script), the conjunct KRa is denoted with a conjunct ligature form.

Recommended implementation: The rkrf table maps the sequence of a consonant (the nominal form only) followed by a virama (halant) followed by Ra (the nominal form) to the corresponding conjoined form (GSUB lookup type 4).

Application interface: For substitution sequences defined in the rkrf table, the application passes the sequence of GIDs to the table, and gets back the GID for the half form.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Required in Devanagari and Gujarati scripts.

Feature interaction: This feature is used in conjunction with certain other features to derive required forms of Indic scripts. The application is expected to process this feature and certain other features in an appropriate order to obtain the correct set of basic forms: nukt, akhn, rphf, rkrf, pref, blwf, half, pstf, cjct. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: 'rlig'

Friendly name: Required Ligatures

Function: Replaces a sequence of glyphs with a single glyph which is preferred for typographic purposes. This feature covers those ligatures, which the script determines as required to be used in normal conditions. This feature is important for some scripts to insure correct glyph formation.

Example: The Arabic character lam followed by alef will always form a ligated lamalef form. This ligated form is a requirement of the script's shaping. The same happens with the Syriac script.

Recommended implementation: The rlig table maps GIDs for default glyphs to one or more GIDs for corresponding alternates (GSUB lookup type 3).

Application interface: The rlig table maps sequences of glyphs to corresponding ligatures (GSUB lookup type 4). Ligatures with more components must be stored ahead of those with fewer components in order to be found. The set of standard ligatures will normally remain constant by script.

UI suggestion: This feature should be active by default. It is recommended that this feature not be turned off to avoid breaking obligatory script shaping.

Script/language sensitivity: Applies to Arabic and Syriac. May apply to some other scripts.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also liga.

Tag: 'rphf'

Friendly name: Reph Form

Function: Substitutes the Reph form for a consonant and halant sequence.

Example: In the Devanagari (Indic) script, the consonant Ra possesses a reph form. When the Ra is a syllable initial consonant and is followed by the virama, it is repositioned after the post base vowel sign within the syllable, and also substituted with a mark that sits above the base glyph.

Recommended implementation: The **rphf** table maps the sequence of default form of Ra and virama to the Reph (GSUB lookup type 4).

Application interface: The application passes the GIDs for Ra and virama to the table and gets back the GID for the reph mark. The application may examine the results of processing other features to determine where in the sequence the reph mark should be re-ordered to.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Required in Indic scripts, eg. Devanagari, Kannada.

Feature interaction: This feature is used in conjunction with certain other features to derive required forms of Indic scripts. The application is expected to process this feature and certain other features in an appropriate order to obtain the correct set of basic forms: nukt, akhn, rphf, rkrf, pref, blwf, half, pstf, cjct. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: 'rtbd'

Friendly name: Right Bounds

Function: Aligns glyphs by their apparent right extents at the right ends of horizontal lines of text, replacing the default behavior of aligning glyphs by their origins. This feature is called by the Optical Bounds (opbd) feature above.

Example: Succeeding lines ending with r, h and y would shift to the right by differing degrees when the text is right-justified and this feature is applied.

Recommended implementation: Values for affected glyphs describe the amount by which the placement and advance width should be altered (GPOS lookup type 1).

Application interface: For GIDs found in the rtbd coverage table, the application passes a GID to the table and gets back a new XPlacement and XAdvance value.

UI suggestion: This feature is called by an application when the user invokes the opbd feature.

Script/language sensitivity: None.

Feature interaction: Should not be applied to glyphs which use fixed-width features (e.g. fwid, halt, hwid, qwid and twid) or vertical features (e.g. vert, vrt2, vpal, valt and vhal). Is called by opbd feature.

Tag: 'rtla'

Friendly name: Right-to-left alternates

Registered by: Adobe

Function: This feature applies glyphic variants (other than mirrored forms) appropriate for right-to-left text. (For mirrored forms, see 'rtlm'.)

Recommended implementation: These are required to be glyph substitutions, and it is recommended that they be one-to-one (GSUB lookup type 1).

Application interface: See section “Left-to-right and right-to-left text” in the subclause 6.1.4 "Text processing with OFF Layout".

UI suggestion: None.

Script/language sensitivity: Right-to-left runs of text.

Feature interaction: This feature is to be applied simultaneously with other pre-shaping features such as 'ccmp' and 'locl'.

Tag: 'rtlm'

Friendly name: Right-to-left mirrored forms

Registered by: Adobe

Function: This feature applies mirrored forms appropriate for right-to-left text *other* than for those characters that would be covered by the character-level mirroring step performed by an OFF layout engine. (For right-to-left glyph alternates, see 'rtla'.)

Example: The ‘rtlm’ feature replaces the glyph for U+2232, CLOCKWISE CONTOUR INTEGRAL, with one in which the integral sign is mirrored but the circular arrow has retained its direction.

Recommended implementation: These are required to be glyph substitutions, and it is recommended that they be one-to-one (GSUB lookup type 1).

Application interface: See section “Left-to-right and right-to-left text” in the subclause 6.1.4 "Text processing with OFF Layout".

UI suggestion: None.

Script/language sensitivity: Right-to-left runs of text.

Feature interaction: This feature is to be applied simultaneously with other pre-shaping features such as 'ccmp' and 'locl'.

Tag: 'ruby'

Friendly name: Ruby Notation Forms

Function: Japanese typesetting often uses smaller kana glyphs, generally in superscripted form, to clarify the meaning of kanji which may be unfamiliar to the reader. These are called ruby, from the old typesetting term for four-point-sized type. This feature identifies glyphs in the font which have been designed for this use, substituting them for the default designs.

Example: The user applies this feature to the kana character U+3042, to get the ruby form for annotation.

Recommended implementation: The font contains alternate glyphs for all kana characters which are enabled for ruby notation. The ruby table maps GIDs for default forms to GIDs for corresponding ruby alternates. These are one-to-one substitutions (GSUB lookup type 1).

Application interface: For GIDs found in the ruby coverage table, the application passes the GIDs for default forms to the table and gets back new GIDs for ruby forms. The application then scales and positions these forms according to its defaults, which may take user parameters.

UI suggestion: This feature should be inactive by default. Applications may offer the user an opportunity to specify the degree of scaling and baseline shift.

Script/language sensitivity: Applies only to Japanese.

Feature interaction: This feature overrides the results of any other feature for the affected characters.

Tag: 'salt'

Friendly name: Stylistic Alternates

Function: Many fonts contain alternate glyph designs for a purely esthetic effect; these don't always fit into a clear category like swash or historical. As in the case of swash glyphs, there may be more than one alternate form. This feature replaces the default forms with the stylistic alternates.

Example: The user applies this feature to Industria to get the alternate form of g.

Recommended implementation: The salt table maps GIDs for default forms to one or more GIDs for corresponding stylistic alternatives. While many of these substitutions are one-to-one (GSUB lookup type 1), others require a selection from a set (GSUB lookup type 3). The manufacturer may choose to build two tables (one for each lookup type) or only one which uses lookup type 3 for all substitutions. As in any one-from-many substitution, alternates present in more than one face should be ordered consistently across a family, so that those alternates can work correctly when switching between family members.

Application interface: For GIDs found in the salt coverage table, the application passes the GIDs to the salt table and gets back one or more new GIDs. If more than one GID is returned, the application must provide a means for the user to select the one desired.

UI suggestion: This feature should be inactive by default. When more than one GID is returned, an application could display the forms sequentially in context, or present a palette showing all the forms at once, or give the user a choice between these approaches. The application may assume that the first glyph in a set is the preferred form, so the font developer should order them accordingly.

Script/language sensitivity: None.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'sinf'

Friendly name: Scientific Inferiors

Function: Replaces lining or oldstyle figures with inferior figures (smaller glyphs which sit lower than the standard baseline, primarily for chemical or mathematical notation). May also replace lowercase characters with alphabetic inferiors.

Example: The application can use this feature to automatically access the inferior figures (more legible than scaled figures).

Recommended implementation: The sinf table maps figures to the corresponding inferior forms (GSUB lookup type 1).

Application interface: For GIDs found in the sinf coverage table, the application passes a GID to the feature and gets back a new GID.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Can apply to nearly any script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'size'

Friendly name: Optical size

Function: This feature stores two kinds of information about the optical size of the font: design size (the point size for which the font is optimized) and size range (the range of point sizes which the font can serve well), as well as other information which helps applications use the size range. The design size is useful for determining proper tracking behavior. The size range is useful in families which have fonts covering several ranges. Additional values serve to identify the set of fonts which share related size ranges, and to identify their shared name.

NOTE Sizes refer to nominal final output size, and are independent of viewing magnification or resolution.

Required implementation:

The Feature table of this GPOS feature contains no lookups; its Feature Parameters field records an Offset from the beginning of the Feature table to an array of five 16-bit unsigned integer values. The size feature must be implemented in all fonts in any family which uses the feature. In this usage, a family is a set of fonts which share a Preferred Family name (name ID 16), or Font Family name (name ID 1) if the Preferred Family name is absent.

- The first value represents the design size in 720/inch units (decipoints). The design size entry must be non-zero. When there is a design size but no recommended size range, the rest of the array will consist of zeros.
- The second value has no independent meaning, but serves as an identifier that associates fonts in a subfamily. All fonts which share a Preferred or Font Family name and which differ only by size range shall have the same subfamily value, and no fonts which differ in weight or style shall have the same subfamily value. If this value is zero, the remaining fields in the array will be ignored.
- The third value enables applications to use a single name for the subfamily identified by the second value. If the preceding value is non-zero, this value must be set in the range 256 - 32767 (inclusive). It records the value of a field in the name table, which must contain English-language strings encoded in Windows Unicode and Macintosh Roman, and may contain additional strings localized to other scripts and languages. Each of these strings is the name an application should use, in combination with the family name, to represent the subfamily in a menu. Applications will choose the appropriate version based on their selection criteria.
- The fourth and fifth values represent the small end of the recommended usage range (exclusive) and the large end of the recommended usage range (inclusive), stored in 720/inch units (decipoints). Ranges must not overlap, and should generally be contiguous.

Example: The size information in Bell Centennial is [60 0 0 0 0]. This tells an application that the fontâs design size is six points, so larger sizes may need proportionate reduction in default inter-glyph spacing. The size information in Minion Pro Semibold Condensed Subhead is [180 3 257 139 240]. These values tell an application that:

- The font's design size is 18 points;
- This font is part of a subfamily of fonts that differ only by the size range which each covers, and which share the arbitrary identifier number 3;

- ID 257 in the name table is the suggested menu name for this subfamily. In this case, the string at name ID 257 is Semibold Condensed;
- This font is the recommended choice from sizes greater than 13.9-point up through 24-points.

Application interface: When the user specifies a size, the application checks for a size feature in the active font. If none is found, the application follows its default behavior. If one is found, the application follows the specified Offset to retrieve the five values.

- *Design size:* Applications which offer size-based tracking have a pre-defined curve which they can apply. By default, this curve should be set to produce no adjustment at the font's design size (first value in the array, in decipoints).
- *Size ranges:* If the second value in the size array is non-zero, the font has a recommended size range. When any such font is selected by the user, the application builds a list of all fonts with this subfamily value and the same Preferred Family name, and notes the size range in the current font. Applications may want to cache the subfamily list at this point. If the specified size falls in the current font's range, the application uses the current font. If not, the application checks the other ranges in the subfamily, and if the specified size falls in one of them, uses that font. If the specified size is not in any range present, the font with the range closest to the specified value is used. If the specified size falls exactly between two ranges, the range with the larger values is used. Since adding or removing fonts from a subfamily may cause reflow, applications should note which fonts are used for which text.

UI suggestion: This feature should be active by default. Applications may want to present the tracking curve to the user for adjustments via a GUI. At start-up, and when fonts are added or removed, applications may want to build a list of fonts with such ranges, and display the filtered subfamily names in their font selection UI, with each filtered name representing the full set of related sizes. Applications may also present a setting which allows the user to select non-default sizes (for example, in the case where final output is intended for on-screen viewing, a smaller optical size will produce better results). In such a case, the font-selection UI should present the unfiltered names. Applications should notify the user if fonts are removed or added from a subfamily with size ranges, and query about desired behavior.

Script/language sensitivity: None. The FeatureParams of all ‘size’ features in the GPOS FeatureList must point to the same set of values.

Feature interaction: None.

Tag: 'smcp'

Friendly name: Small Capitals

Function: This feature turns lowercase characters into small capitals. This corresponds to the common SC font layout. It is generally used for display lines set in Large & small caps, such as titles. Forms related to small capitals, such as oldstyle figures, may be included.

Example: The user enters text as mixed capitals and lowercase, and gets Large & small cap text.

Recommended implementation: The smcp table maps lowercase glyphs to the corresponding small-cap forms (GSUB lookup type 1).

Application interface: For GIDs found in the smcp coverage table, the application passes GIDs to the smcp table, and gets back new GIDs.

NOTE Applications should treat ß (U+00DF) as a pair of s characters, and that the Turkish dotless i maps to the normal small cap I.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies only to bicameral scripts (i.e. those with case differences), such as Latin, Greek, Cyrillic, and Armenian.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. Also see c2sc.

Tag: 'smpl'

Friendly name: Simplified Forms

Function: Replaces 'traditional' Chinese or Japanese forms with the corresponding 'simplified' forms.

Example: The user gets U+53F0 when U+6AAF, U+81FA, or U+98B1 is entered.

Recommended implementation: The smpl table maps each traditional form in a font to a corresponding simplified form (GSUB lookup type 1).

NOTE More than one traditional form may map to a single simplified form.

Application interface: For GIDs found in the smpl coverage table, the application passes the GIDs to the table and gets back one new GID for each.

NOTE This is a change of character code. Besides the original character code, the application should store the code for the new character.

UI suggestion: This feature would be off by default, but could be made the default by a preference setting.

Script/language sensitivity: Applies only to Chinese and Japanese.

Feature interaction: This feature is mutually exclusive with all other features, which should be turned off when it's applied, except the palt, vert and vrt2 features, which may be used in addition; trad and tnam are mutally exclusive, and override the results of smpl.

Tag: 'ss01' - 'ss20'

Friendly name: Stylistic Set 1 - Stylistic Set 20

Function: In addition to, or instead of, stylistic alternatives of individual glyphs (see 'salt' feature), some fonts may contain sets of stylistic variant glyphs corresponding to portions of the character set, e.g. multiple variants for lowercase letters in a Latin font. Glyphs in stylistic sets may be designed to harmonise visually, interract in particular ways, or otherwise work together. Examples of fonts including stylistic sets are Zapfino Linotype and Adobe's Poetica. Individual features numbered sequentially with the tag name convention 'ss01' 'ss02' 'ss03' . 'ss20' provide a mechanism for glyphs in these sets to be associated via GSUB lookup indexes to default forms and to each other, and for users to select from available stylistic sets.

Recommended implementation: An ssXX table maps GIDs for default forms to one GIDs for corresponding stylistic alternatives in each set. Each ssXX feature uses one-to-one (GSUB lookup type 1) substitutions. Font developers may choose to map only from default forms to variants for each stylistic set, or may choose to map between all stylistic sets in each feature, depending on intended user experience. For example, feature 'ss03' might contain lookups mapping variant glyphs from 'ss01' and 'sso2' to corresponding variants in 'ss03', in addition to mapping from default forms.

The FeatureParams field of the Feature Table of these GSUB features may be set to 0, or to an offset to a Feature Parameters table comprising two successive USHORT values, as follows:

- *Version (set to 0):* This corresponds to a "minor" version number. Additional data may be added to the end of this Feature Parameters table in the future.
- *UI Name ID:* The 'name' table name ID that specifies a string (or strings, for multiple languages) for a user-interface label for this feature. The values of uiLabelNameId and sampleTextNameId are expected to be in the font-specific name ID range (256–32767), though that is not a requirement in this Feature Parameters specification. The user-interface label for the feature can be provided in

multiple languages. An English string should be included as a fallback. The string should be kept to a minimal length to fit comfortably with different application interfaces.

Application interface: The application is responsible for counting and enumerating the number of features in the font with tag names of the format 'ss01' to 'ss20', and for presenting the user with an appropriate selection mechanism. For GIDs found in the ssXX coverage table, the application passes the GIDs to the ssXX table and gets back one or more new GIDs.

UI suggestion: This feature should be off by default.

Script/language sensitivity: None.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. After an ssXX feature has been applied, the user may wish to apply glyph-specific features, e.g. 'salt', to individual glyphs in the resulting layout; font developers are responsible for ordering substitution lookups to obtain desired user experience.

Tag: 'subs'

Friendly name: Subscript

Function: The "subs" feature may replace a default glyph with a subscript glyph, or it may combine a glyph substitution with positioning adjustments for proper placement.

Recommended implementation: First, a single or contextual substitution lookup implements the subscript glyph (GSUB lookup type 1). Then, if the glyph needs repositioning, an application may apply a single adjustment, pair adjustment, or contextual adjustment positioning lookup to modify its position.

Application interface: For GIDs found in the subs coverage table, the application passes a GID to the feature and gets back a new GID. This is a change of semantic value. Besides the original character codes, the application should store the code for the new character.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Can apply to nearly any script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'sups'

Friendly name: Superscript

Function: Replaces lining or oldstyle figures with superior figures (primarily for footnote indication), and replaces lowercase letters with superior letters (primarily for abbreviated French titles).

Example: The application can use this feature to automatically access the superior figures (more legible than scaled figures) for footnotes, or the user can apply it to Mssr to get the classic form.

Recommended implementation: The sups table maps figures and lowercase letters to the corresponding superior forms (GSUB lookup type 1).

Application interface: For GIDs found in the sups coverage table, the application passes a GID to the feature and gets back a new GID.

NOTE This can include a change of semantic value. Besides the original character codes, the application should store the code for the new character.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Can apply to nearly any script.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'swsh'

Friendly name: Swash

Function: This feature replaces default character glyphs with corresponding swash glyphs. It should be noted that there may be more than one swash alternate for a given character.

Example: The user inputs the ampersand character when setting text with Poetica with this feature active, and is presented with a choice of the 63 ampersand forms in that face.

Recommended implementation: The swsh table maps GIDs for default forms to those for one or more corresponding swash forms. While many of these substitutions are one-to-one (GSUB lookup type 1), others require a selection from a set (GSUB lookup type 3). The manufacturer may choose to build two tables (one for each lookup type) or only one which uses lookup type 3 for all substitutions. If several styles of swash are present across the font, the set of forms for each character should be ordered consistently.

Application interface: For GIDs found in the swsh coverage table, the application passes the GIDs to the swsh table and gets back one or more new GIDs. If more than one GID is returned, the application must provide a means for the user to select the one desired.

UI suggestion: This feature should be inactive by default. When more than one GID is returned, an application could display the forms sequentially in context, or present a palette showing all the forms at once, or give the user a choice between these approaches. The application may assume that the first glyph in a set is the preferred form, so the font developer should order them accordingly.

Script/language sensitivity: Does not apply to ideographic scripts.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'titl'

Friendly name: Titling

Function: This feature replaces the default glyphs with corresponding forms designed specifically for titling. These may be all-capital and/or larger on the body, and adjusted for viewing at larger sizes.

Example: The user applies this feature in Adobe Garamond to get the titling caps.

Recommended implementation: The titl table maps default forms to corresponding titling forms (GSUB lookup type 1).

Application interface: For GIDs found in the titl coverage table, the application passes the GIDs to the titl table and gets back new GIDs.

UI suggestion: This feature should be off by default.

Script/language sensitivity: None.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'tjmo'

Friendly name: Trailing Jamo Forms

Function: Substitutes the trailing jamo form of a cluster.

Example: In Hangul script, the jamo cluster is composed of three parts (leading consonant, vowel, and trailing consonant). When a sequence of trailing class jamos are found, their combined trailing jamo form is substituted.

Recommended implementation: The **tjmo** table maps the sequence required to convert a series of jamos into its trailing jamo form (GSUB lookup type 4).

Application interface: For substitutions defined in the **tjmo** table, the application passes the sequence of GIDs to the feature, and gets back the GID for the trailing jamo form.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required for Hangul script when Ancient Hangul writing system is supported.

Feature interaction: This feature overrides the results of all other features.

Tag: 'tnam'

Friendly name: Traditional Name Forms

Function: Replaces 'simplified' Japanese kanji forms with the corresponding 'traditional' forms. This is equivalent to the Traditional Forms feature, but explicitly limited to the traditional forms considered proper for use in personal names (as many as 205 glyphs in some fonts).

Example: The user inputs U+4E9C and gets U+4E9E.

Recommended implementation: The tnam table maps simplified forms in a font to corresponding traditional forms which can be used in personal names (GSUB lookup type 1). The application stores a record of any simplified forms which resulted from substitutions (the smpl feature); for such forms, applying the tnam feature undoes the previous substitution.

Application interface: For GIDs found in the tnam coverage table, the application passes the GIDs to the table and gets back new GIDs.

NOTE This is a change of character code. Besides the original character code, the application should store the code for the new character.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies only to Japanese.

Feature interaction: May include some characters affected by the Proportional Alternate Widths feature (palt); trad and tnam are mutually exclusive, and override the results of smpl.

Tag: 'tnum'

Friendly name: Tabular Figures

Function: Replaces figure glyphs set on proportional widths with corresponding glyphs set on uniform (tabular) widths. Tabular widths will generally be the default, but this cannot be safely assumed. Of course this feature would not be present in monospaced designs.

Example: The user may apply this feature to get oldstyle figures to align vertically in a column.

Recommended implementation: In order to simplify associated kerning and get the best glyph design for a given width, this feature should use new glyphs for the figures, rather than only adjusting the fit of the proportional glyphs (although some may be simple copies); i.e. not a GPOS feature. The tnum table maps proportional versions of lining &/or oldstyle figures to corresponding tabular glyphs (GSUB lookup type 1).

Application interface: For GIDs found in the tnum coverage table, the application passes GIDs to the tnum table and gets back new GIDs.

UI suggestion: This feature should be off by default. The application may want to query the user about this feature when the user changes figure style (onum or lnum).

Script/language sensitivity: None.

Feature interaction: This feature overrides the results of the Proportional Figures feature (pnum).

Tag: 'trad'

Friendly name: Traditional Forms

Function: Replaces 'simplified' Chinese hanzi or Japanese kanji forms with the corresponding 'traditional' forms.

Example: The user inputs U+53F0 and is offered a choice of U+6AAF, U+81FA, or U+98B1.

Recommended implementation: The trad table maps each simplified form in a font to one or more traditional forms. While many of these substitutions are one-to-one (GSUB lookup type 1), others require a selection from a set (GSUB lookup type 3). The manufacturer may choose to build two tables (one for each lookup type) or only one which uses lookup type 3 for all substitutions. As in any one-from-many substitution, alternates present in more than one face should be ordered consistently across a family, so that those alternates can work correctly when switching between family members.

Application interface: For GIDs found in the trad coverage table, the application passes the GIDs to the table and gets back one or more new GIDs. If more than one GID is returned, the application must provide a means for the user to select the one desired. The application stores a record of any simplified forms which resulted from substitutions (the smpl feature); for such forms, applying the trad feature undoes the previous substitution.

NOTE This is a change of character code. Besides the original character code, the application should store the code for the new character.

UI suggestion: This feature should be inactive by default. If there's no record of a conversion from traditional to simplified, the user must be offered a set of possibilities from which to select. The application may note the user's choice, and offer it as a default the next time the source simplified character is encountered. In the absence of such prior information, the application may assume that the first glyph in a set is the preferred form, so the font developer should order them accordingly.

Script/language sensitivity: Applies only to Chinese and Japanese.

Feature interaction: May include some characters affected by the Proportional Alternate Widths feature (palt); trad and tnam are mutually exclusive, and override the results of smpl.

Tag: 'twid'

Friendly name: Third Widths

Function: Replaces glyphs on other widths with glyphs set on widths of one third of an em. The characters involved are normally figures and some forms of punctuation.

Example: The user may apply twid to place a three-digit figure in a single slot in a column of vertical text.

Recommended implementation: The font may contain alternate glyphs designed to be set on third-em widths (GSUB lookup type 1), or it may specify alternate metrics for the original glyphs (GPOS lookup type 1) which adjust their spacing to fit in third-em widths.

Application interface: For GIDs found in the twid coverage table, the application passes the GIDs to the table and gets back either new GIDs or positional adjustments (XPlacement and XAdvance).

UI suggestion: This feature would normally be off by default.

Script/language sensitivity: Generally used only in CJKV fonts.

Feature interaction: This feature is mutually exclusive with all other glyph-width features (e.g. fwid, halt, hwid and qwid), which should be turned off when it's applied. It deactivates the kern feature.

Tag: 'unic'

Friendly name: Unicase

Function: This feature maps upper- and lowercase letters to a mixed set of lowercase and small capital forms, resulting in a single case alphabet (for an example of unicase, see the Emigre type family Filosofia). The letters substituted may vary from font to font, as appropriate to the design. If aligning to the x-height, smallcap glyphs may be substituted, or specially designed unicase forms might be used. Substitutions might also include specially designed figures.

Example: The user enters text as uppercase, lowercase or mixed case, and gets unicase text.

Recommended implementation: The unic table maps some uppercase and lowercase glyphs to corresponding unicase forms (GSUB lookup type 1).

Application interface: For GIDs found in the unic coverage table, the application passes GIDs to the unic table, and gets back new GIDs.

UI suggestion: This feature should be off by default.

Script/language sensitivity: Applies only to scripts with both upper- and lowercase forms (e.g. Latin, Cyrillic, Greek).

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'valt'

Friendly name: Alternate Vertical Metrics

Function: Repositions glyphs to visually center them within full-height metrics, for use in vertical setting. Typically applies to full-width Latin glyphs, which are aligned on a common horizontal baseline and not rotated when set vertically in CJKV fonts.

Example: Applying this feature would shift a Roman h down, or y up, from their default full-width positions.

Recommended implementation: The font specifies alternate metrics for the original glyphs (GPOS lookup type 1).

Application interface: For GIDs found in the valt coverage table, the application passes the GIDs to the table and gets back positional adjustments (YPlacement).

UI suggestion: This feature should be active by default in vertical-setting contexts.

Script/language sensitivity: Applies only to scripts with vertical writing modes.

Feature interaction: This feature is mutually exclusive with all other glyph-height features (e.g. vhal and vpal), which should be turned off when it's applied. It deactivates the kern feature.

Tag: "vatu"

Friendly name: Vattu Variants

Function: : In an Indic consonant conjunct, substitutes a ligature glyph for a base consonant and a following vattu (below-base) form of a conjoining consonant, or for a half form of a consonant and a following vattu form.

Example: In the Devanagari (Indic) script, the consonant Ra takes a vattu form, when it is not the syllable initial consonant in a conjunct. This vattu form ligates with the base consonant as well as half forms of consonants.

Recommended implementation: The **vatu** table maps consonant and vattu form combinations to their respective ligatures (GSUB lookup type 4).

Lookups associated with the Vattu Variants feature apply to glyphs derived using the Below-base Forms feature and (for half-form plus vattu ligatures) the Half Forms features. The Below-base Forms feature should be used to derive the nominal vattu form of a consonant; the Vattu Variants feature should only be used to substitute the nominal vattu form and a base consonant or half form with a ligature glyph. If the Rakar Forms feature is used, the Vattu Variants feature is not required.

Application interface: For substitutions defined in the **vatu** table, the application passes the sequence of GIDs to the table, and gets back the GID for the vattu variant ligature.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied in the appropriate contexts, as determined by script-specific processing. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Used in Indic scripts. eg: Devanagari.

Feature interaction: This feature may be used in conjunction with certain other features to derive required forms of Indic scripts. For Indic script implementations that use the Vattu Variants feature, the application is expected to process this feature and certain other features in an appropriate order to obtain the correct set of basic forms: nukt, akhn, rphf, rkrf, pref, blwf, half, pstf, cjct. Other discretionary features for optional typographic effects may also be applied. Lookups for such discretionary features should be processed after lookups for this feature have been processed.

Tag: 'vert'

Friendly name: Vertical Alternates

Function: Replaces default forms with variants adjusted for vertical writing when in vertical writing mode. While most CJKV glyphs remain vertical when set in vertical writing mode, some take a different form (usually rotated and repositioned) for this purpose. Glyphs covered by this feature correspond to the set normally rotated in low-end DTP applications.

Example: In vertical writing mode, the opening parenthesis (U+FF08) is replaced by the rotated form (U+FE35).

Recommended implementation: The font includes rotated versions of the glyphs covered by this feature. The vert table maps the standard forms to the corresponding rotated forms (GSUB lookup type 1). This feature should be the last substitution in the font, and take input from other features.

Application interface: For GIDs found in the vert coverage table, the application passes GIDs to the feature, and gets back new GIDs. See the vrt2 feature description for more details.

UI suggestion: This feature should be active by default when vertical writing mode is on if the vrt2 feature is not present. See the vrt2 feature description for more details, and a discussion of vertical writing in OFF.

Script/language sensitivity: Applies only to scripts with vertical writing capability.

Feature interaction: This is a subset of the vrt2 feature; vrt2 is preferred. May be used in addition to any other feature.

Tag: 'vhal'

Friendly name: Alternate Vertical Half Metrics

Function: Respaces glyphs designed to be set on full-em heights, fitting them onto half-em heights. This differs from valt in that it does not substitute new glyphs.

Example: The user may invoke this feature in a CJKV font to get better fit for punctuation or symbol glyphs without disrupting the monospaced alignment.

Recommended implementation: The font specifies alternate metrics for the full-height glyphs (GPOS lookup type 1).

Application interface: For GIDs found in the vhal coverage table, the application passes the GIDs to the table and gets back positional adjustments (XPlacement, XAdvance, YPlacement and YAdvance).

UI suggestion: This feature would be off by default.

Script/language sensitivity: Used only in CJKV fonts.

Feature interaction: This feature is mutually exclusive with all other glyph-height features (e.g. valt and vpal), which should be turned off when it's applied. It deactivates the kern feature. See also halt.

Tag: "vjmo"

Friendly name: Vowel Jamo Forms

Function: Substitutes the vowel jamo form of a cluster.

Example: In Hangul script, the jamo cluster is composed of three parts (leading consonant, vowel, and trailing consonant). When a sequence of vowel class jamos are found, their combined vowel jamo form is substituted.

Recommended implementation: The **vjmo** table maps the sequence required to convert a series of jamos into its vowel jamo form (GSUB lookup type 4).

Application interface: For substitutions defined in the **vjmo** table, the application passes the sequence of GIDs to the feature, and gets back the GID for the vowel jamo form.

UI suggestion: This feature should be on by default.

Script/language sensitivity: Required for Hangul script when Ancient Hangul writing system is supported.

Feature interaction: This feature overrides the results of all other features.

Tag: 'vkna'

Friendly name: Vertical Kana Alternates

Function: Replaces standard kana with forms that have been specially designed for only vertical writing. This is a typographic optimization for improved fit and more even color. Also see hkna.

Example: Standard full-width kana (hiragana and katakana) are replaced by forms that are designed for vertical use.

Recommended implementation: The font includes a set of specially-designed glyphs, listed in the vkna coverage table. The vkna feature maps the standard full-width forms to the corresponding special vertical forms (GSUB lookup type 1).

Application interface: For GIDs found in the vkna coverage table, the application passes GIDs to the feature, and gets back new GIDs.

UI suggestion: This feature would be off by default.

Script/language sensitivity: Applies only to fonts that support kana (hiragana and katakana).

Feature interaction: Since this feature is only for vertical use, features applying to horizontal behaviors (e.g. kern) do not apply.

Tag: 'vkrn'

Friendly name: Vertical Kerning

Function: Adjusts amount of space between glyphs, generally to provide optically consistent spacing between glyphs. Although a well-designed typeface has consistent inter-glyph spacing overall, some glyph combinations require adjustment for improved legibility. Besides standard adjustment in the vertical direction, this feature can supply size-dependent kerning data via device tables, "cross-stream" kerning in the X text direction, and adjustment of glyph placement independent of the advance adjustment.

NOTE This feature may apply to runs of more than two glyphs, and would not be used in monospaced fonts. This feature applies only to text set vertically.

Example: When the katakana character U+30B9 or U+30D8 is followed by U+30C8 in a vertical setting, U+30C8 is shifted up to fit more evenly.

Recommended implementation: The font stores a set of adjustments for pairs of glyphs (GPOS lookup type 2 or 8). These may be stored as one or more tables matching left and right classes, &/or as individual pairs. Additional adjustments may be provided for larger sets of glyphs (e.g. triplets, quadruplets, etc.) to overwrite the results of pair kerns in particular combinations.

Application interface: The application passes a sequence of GIDs to the kern table, and gets back adjusted positions (XPlacement, XAdvance, YPlacement and YAdvance) for those GIDs. When using the type 2 lookup on a run of glyphs, it's critical to remember to not consume the last glyph, but to keep it available as the first glyph in a subsequent run (this is a departure from normal lookup behavior).

UI suggestion: This feature should be active by default for vertical text setting. Applications may wish to allow users to add further manually-specified adjustments to suit specific needs and tastes.

Script/language sensitivity: None

Feature interaction: If 'vkrn' is activated, 'vpal' must also be activated if it exists. (If 'vpal' is activated, there is no requirement that 'vkrn' must also be activated.) May be used in addition to any other feature except those which result in fixed (uniform) advance heights.

Tag: 'vpal'

Friendly name: Proportional Alternate Vertical Metrics

Function: Respaces glyphs designed to be set on full-em heights, fitting them onto individual (more or less proportional) vertical heights. This differs from valt in that it does not substitute new glyphs (GPOS, not GSUB feature). The user may prefer the monospaced form, or may simply want to ensure that the glyph is well-fit.

Example: The user may invoke this feature in a Japanese font to get Latin, Kanji, Kana or Symbol glyphs with the full-height design but individual metrics.

Recommended implementation: The font specifies alternate heights for the full-height glyphs (GPOS lookup type 1).

Application interface: For GIDs found in the vpal coverage table, the application passes the GIDs to the table and gets back positional adjustments (XPlacement, XAdvance, YPlacement and YAdvance).

UI suggestion: This feature would be off by default.

Script/language sensitivity: Used mostly in CJKV fonts.

Feature interaction: This feature is mutually exclusive with all other glyph-height features (e.g. valt and vhal), which should be turned off when it's applied. If vpal is activated, there is no requirement that vkrn must also be activated. If vkrn is activated then vpal must also be activated if it exists.

Tag: 'vrt2'

Friendly name: Vertical Alternates and Rotation

Function: Replaces some fixed-width (half-, third- or quarter-width) or proportional-width glyphs (mostly Latin or katakana) with forms suitable for vertical writing (that is, rotated 90 degrees clockwise).

NOTE These are a superset of the glyphs covered in the vert table.

ATM/NT 4.1 and the Windows 2000 OTF driver impose the following requirements for an OFF font with CFF outlines to be used for vertical writing: the vrt2 feature must be present in the GSUB table, it must comprises a single lookup of LookupType 1 and LookupFlag 0, and the lookup must have a single subtable. The predecessor feature, vert, is ignored.

A rotated glyph must be designed such that its top side bearing and vertical advance as recorded in the Vertical Metrics ('vmtx') table are identical to the left side bearing and horizontal advance, respectively, of the corresponding upright glyph as recorded in the Horizontal Metrics ('hmtx') table. (The horizontal advance of the rotated glyph may be set to any value, since the glyph is intended only for vertical writing use. The vendor may however set it to head.unitsPerEm, to prevent overlap during font proofing tests, for example.)

Thus, proportional-width glyphs with rotated forms in the vrt2 feature will appear identically spaced in both vertical and horizontal writing. In order for kerning to produce identical results as well, developers must ensure that the Vertical Kerning (vkrn) feature record kern values between the rotated glyphs that are the same as kern values between their corresponding upright glyphs in the Kerning (kern) feature.

Example: Proportional- or half-width Latin and half-width katakana characters are rotated 90 degrees clockwise for vertical writing.

Recommended implementation: The font includes rotated versions of the glyphs covered by this feature. The vrt2 table maps the standard (horizontal) forms to the corresponding vertical (rotated) forms (GSUB lookup type 1). This feature should be the last substitution in the font, and take input from other features.

Application interface: For GIDs found in the vrt2 coverage table, the application passes GIDs to the feature, and gets back new GIDs.

UI suggestion: This feature should be active by default when vertical writing mode is on, although the user must be able to override it.

Script/language sensitivity: Applies only to scripts with vertical writing capability.

Feature interaction: Overrides the vert (Vertical Writing) feature, which is a subset of this one. May be used in addition to any other feature.

Tag: 'zero'

Friendly name: Slashed Zero

Function: Some fonts contain both a default form of zero, and an alternative form which uses a diagonal slash through the counter. Especially in condensed designs, it can be difficult to distinguish between 0 and O (zero and capital O) in any situation where capitals and lining figures may be arbitrarily mixed. This feature allows the user to change from the default 0 to a slashed form.

Example: When setting labels, the user applies this feature to get the slashed 0.

Recommended implementation: The zero table maps the GIDs for the lining forms of zero to corresponding slashed forms (GSUB lookup type 1).

Application interface: For GIDs in the zero coverage table, the application passes a GID to the zero table and gets back a new GID.

UI suggestion: Optimally, the application would store this as a preference setting, and the user could use the feature to toggle back and forth between the two forms. Most applications will want the default setting to disable this feature.

Script/language sensitivity: Does not apply to scripts which use forms other than 0 for zero.

Feature interaction: Applies only to lining figures, so is inactivated by oldstyle figure features (e.g. onum).

6.4.4 Baseline tags

This clause defines the standard OFF Layout baseline tags. A registered baseline tag has a specific meaning when used in the horizontal writing direction (used in the 'BASE' table's HorizAxis table), vertical writing direction (used in the 'BASE' table's VertAxis table), or both, and conveys information to font users about a baseline's use. For example, the "romn" baseline tag is commonly used to identify the baseline to layout Latin text in the horizontal, vertical, or both directions for Latin text layout.

This version of the Tag Registry identifies the baselines . All baseline tags are 4-byte character strings composed of a limited set of ASCII characters in the 0x20-0x7E range. Baseline tags consist of four lowercase letters.

Baseline Tag	Baseline for HorizAxis	Baseline for VertAxis
"hang"	The hanging baseline. This is the horizontal line from which syllables seem to hang in Tibetan script.	The hanging baseline, (which now appears vertical) for Tibetan characters rotated 90 degrees clockwise, for vertical writing mode.
"icfb"	Ideographic character face bottom edge baseline. (See Ideographic Character Face below for usage.)	Ideographic character face left edge baseline. (See clause Ideographic Character Face below for usage.)
"icft"	Ideographic character face top edge baseline. (See Ideographic Character Face below for usage.)	Ideographic character face right edge baseline. (See clause Ideographic Character Face below for usage.)
"ideo"	Ideographic em-box bottom edge baseline. (See clause Ideographic Em-Box below for usage.)	Ideographic em-box left edge baseline. If this tag is present in the VertAxis, the value must be set to 0. (See clause Ideographic Em-Box below for usage.)

"idtp"	Ideographic em-box top edge baseline. (See Ideographic Em-Box below for usage.)	Ideographic em-box right edge baseline. If this tag is present in the VertAxis, the value is strongly recommended to be set to head.unitsPerEm. (See clause Ideographic Em-Box below for usage.)
"math"	The baseline about which mathematical characters are centered.	The baseline about which mathematical characters, when rotated 90 degrees clockwise for vertical writing mode, are centered.
"romn"	The baseline used by simple alphabetic scripts such as Latin, Cyrillic and Greek.	The alphabetic baseline for characters rotated 90 degrees clockwise for vertical writing mode. (This would not apply to alphabetic characters that remain upright in vertical writing mode, since these characters are not rotated.)

Ideographic Em-box

*[The notation **<Axis>.<Baseline Tag>** is used in the following description to mean the baseline tag as defined in the specified axis. For example, **HorizAxis.ideo** means the **ideo** baseline tag as defined in the HorizAxis of the BASE table. See above for a list of registered baseline tags.]*

A font's ideographic em-box is the rectangle that defines a standard escapement around the full-width ideographic glyphs of the font, for both the horizontal and vertical writing directions. It is usually a square, but may be non-square as in the case of fonts used in Japanese newspaper layout that have a vertically condensed design.

The left, right, top and bottom edges of the ideographic em-box are to be determined as follows:

ideoEmboxLeft = 0

If **HorizAxis.ideo** defined:

ideoEmboxBottom = **HorizAxis.ideo**

If **HorizAxis.idtp** defined:

ideoEmboxTop = **HorizAxis.idtp**

Else:

ideoEmboxTop = **HorizAxis.ideo** + head.unitsPerEm

If **VertAxis.idtp** defined:

ideoEmboxRight = **VertAxis.idtp**

Else:

ideoEmboxRight = head.unitsPerEm

If **VertAxis.ideo** defined and non-zero:

Warning: Bad **VertAxis.ideo** value

Else If this is a CJK font:

ideoEmboxBottom = OS/2.sTypoDescender
ideoEmboxTop = OS/2.sTypoAscender
ideoEmboxRight = head.unitsPerEm

Else:

ideoEmbox cannot be determined for this font

Determining whether a font is CJK (Chinese, Japanese, or Korean) or not, as in the second-last "Else" clause above, can be done by checking the CJK-related bits of the OS/2.ulUnicodeRange fields.

NOTE Font designers can specify a **HorizAxis.ideo** baseline in their non-CJK fonts; this can be used by applications when aligning the font with an ideographic font used on the same line of text, when the user has specified ideographic em-box alignment.

The ideographic em-box center baseline is defined as halfway between the ideographic em-box top and bottom baselines in the horizontal axis, and halfway between the ideographic em-box left and right baselines in the vertical axis. These center baselines are defined in whole character units. The division used in the calculation must round to the character unit nearest 0 if needed. Thus, for maximal precision of center baseline placement, vendors should ensure that opposite edges of the ideographic em-box box are an even number of character units apart.

Example:

The values of the ideographic baseline tags for the Kozuka Mincho font family (designed on a 1000-unit em) are:

HorizAxis.ideo = -120; **HorizAxis.idtp** = 880.
Since this describes a square ideographic em-box, it is sufficient to record only the following:
HorizAxis.ideo = -120.
If **HorizAxis.ideo** is not present, then the following will be used for the ideographic em-box bottom and top, since this is a CJK font:
OS/2.sTypoDescender = -120; OS/2.sTypoAscender = 880.

Compatibility notes:

a. Most applications expect the width of full-width ideographs in a CJK font to be exactly one em, thus it is strongly recommended that **VertAxis.idtp**, if present, be set to head.unitsPerEm. (The **idtp** baseline tag was introduced in OpenType 1.3.)

b. While the OFF specification allows for CJK fonts' OS/2.sTypoDescender and OS/2.sTypoAscender fields to specify metrics different from the **HorizAxis.ideo** and **HorizAxis.idtp** in the 'BASE' table, CJK font developers should be aware that existing applications may not read the 'BASE' table at all but simply use the OS/2.sTypoDescender and OS/2.sTypoAscender fields to describe the bottom and top edges of the ideographic em-box. If developers want their fonts to work correctly with such applications, they should ensure that any ideographic em-box values in the 'BASE' table of their CJK fonts describe the same bottom and top edges as the OS/2.sTypoDescender and OS/2.sTypoAscender fields.

c. Applications on platforms other than Windows that don't parse the 'OS/2' table won't have access to the OS/2.sTypoDescender and OS/2.sTypoAscender fields Thus, CJK fonts will typically have the same descender value recorded in hhea.Descender, OS/2.sTypoDescender, and **HorizAxis.ideo** (if present), and the same Ascender value recorded in hhea.Ascender, OS/2.sTypoAscender, and **HorizAxis.idtp** (if present).

See subclause 6 "OFF CJK Font Guidelines" for more information about constructing CJK fonts.

Ideographic character face

*[The notation **<Axis>.<Baseline Tag>** is used in the following description to mean the baseline tag as defined in the specified axis. For example, **HorizAxis.icfb** means the **icfb** baseline tag as defined in the HorizAxis of the BASE table. See above for a list of registered baseline tags.]*

The Ideographic Character Face (ICF), also known as the Average Character Face (ACF), specifies the approximate bounding box of the full-width ideographic and kana glyphs in a CJK font. (This is different from the FontBBox, as described in the PostScript programming language, which is the bounding box of all glyphs in the font.) In Japanese, the term for ICF is *heikin jizura*.

It is typically expressed as a percentage that represents the ratio of the length of an ICF box edge to the length of an ideographic em-box edge, and is conceptualized as a square centered within the ideographic em-box. However, in OFF, the ICF box's left, bottom, right, and top edges are specified as the **VertAxis.icfb**, **HorizAxis.icfb**, **VertAxis.icft**, and **HorizAxis.icft** baselines, respectively, thus giving font designers the flexibility to specify a non-square and/or non-centered ICF box.

Font designers should set the value of the ICF box edges based on how tight or loose they want the font to appear when text is set with no tracking or kerning (*beta gumi* in Japanese). Therefore, the left-over boundary of the ideographic em-box around the ICF box is the default escapement of the font.

Applications can use the ICF box as an alignment tool, to ensure that glyphs touch the edges of the text frame and page objects are visually aligned to text edges. It is also useful for aligning glyphs of different sizes on the same line. In Japanese traditional paper-based workflow, the ICF box was often used for these purposes. It provides optically aligned results that are superior to using the ideographic em-box.

HorizAxis.icfb is the mininum piece of information required to define the ICF, in a CJK font. First, the ideographic em-box dimensions must be calculated as in the clause "Ideographic Em-Box" above. The ICF edges are then calculated in the following order:

If **HorizAxis.icfb** defined:
 icfBottom = **HorizAxis.icfb**
 margin = **HorizAxis.icfb** – ideoEmboxBottom
If HorizAxis.icft defined:
 icfTop = HorizAxis.icft
Else:
 icfTop = ideoEmboxTop - margin
If **VertAxis.icfb** defined:
 icfLeft = **VertAxis.icfb**
Else:
 icfLeft = margin
If **VertAxis.icft** defined:
 icfRight = **VertAxis.icft**
Else:
 icfRight = ideoEmBoxRight - icfLeft
Else:
 ICF cannot be determined for this font

For the last case above, i.e. fonts that don't have ICF information in their 'BASE' table, an application may choose to apply a heuristic such as calculating the bounding box of some or all of the ideographic and kana glyphs, and then averaging its margin with the ideographic em-box.

The ICF center baseline is defined as halfway between the ICF top and bottom baselines in the horizontal axis, and halfway between the ICF left and right baselines in the vertical axis. These center baselines are defined in whole character units. The division used in the calculation must round to the character unit nearest 0 if needed. Thus, for maximal precision of center baseline placement, vendors should ensure that opposite edges of the ICF box are an even number of character units apart.

Example:

The values of the ICF baselines for the Extra Light and Heavy weights of the Kozuka Mincho font family (designed on a 1000-unit em, with ideographic em-box as given in the example in the previous clause) are:

Kozuka Mincho Extra Light:
VertAxis.icfb = 41; **HorizAxis.icfb** = -79;
VertAxis.icft = 959; **HorizAxis.icft** = 839.
Since this describes a square ICF centered in a square ideographic em-box, it is sufficient to record only the following:
HorizAxis.icfb = -79.

Kozuka Mincho Heavy:
VertAxis.icfb = 26; **HorizAxis.icfb** = -94;
VertAxis.icft = 974; **HorizAxis.icft** = 854.
It is sufficient to record only:
HorizAxis.icfb = -94.

It is strongly recommended that each of the edges of the ICF box be equidistant from the corresponding edge of the ideographic em-box. Following this will result in more predictable results in applications that use these values. That is, for fonts based on a square ideographic em-box, the ICF box should be a centered square.

See subclause 6 "OFF CJK Font Guidelines" for more information about constructing CJK fonts.

7 Recommendations for OFF fonts

This clause outlines recommendations for creating OFF fonts.

Byte ordering

All OFF fonts use Motorola-style byte ordering (Big Endian).

'sfnt' version

OFF fonts that contain TrueType outlines should use the value of 1.0 for the sfnt version. OFF fonts containing CFF data should use the tag 'OTTO' as the sfnt version number.

Mixing outline formats

It is not recommended to mix outline formats within a single font. Choose the format that meets your feature requirements.

Filenames

OFF fonts may have the extension .OTF, .TTF, or .TTC, depending on the type of outlines in the font and the presence of OFF layout tables.

- Fonts with CFF data always have an .OTF extension.
- Fonts containing TrueType outlines that have OFF layout tables should use the .OTF extension when backward compatibility is not an issue. Fonts without OFF layout tables, or fonts that have backward compatibility issues should use the .TTF extension. TrueType Collection fonts should have a .TTC extension whether or not the fonts have OFF layout tables present.

Table alignment and length

All tables should be aligned to begin at Offsets which are multiples of four bytes. While this is not required by the TrueType rasterizer, it does prevent ambiguous checksum calculations and greatly speeds table access on some processors.

All tables should be recorded in the table directory with their actual length. To ensure that checksums are calculated correctly, it is suggested that tables begin on LONG word boundries. Any extra space after a table (and before the next LONG word boundry) should be padded with zeros.

First four glyphs in fonts

TrueType outline fonts should have the following four glyphs at the glyph ID indicated.

Glyph ID	Glyph name	Unicode value
0	.notdef	undefined
1	.null	U+0000
2	CR	U+000D
3	Space	U+0020

Additional recommendations:

- Glyph 1 should have no contours and zero advance width.
- Character U+000D (carriage return) should map to a glyph with a positive advance width.
- Character U+0001-001F (misc ASCII control codes) and U+007F (delete) should be mapped to glyph 0 (with some exceptions noted below).
- Characters U+0000 (null), U+0008 (backspace) and U+001D (group separator) should map to glyph 1.
- Characters U+0009 (horizontal tabulation), U+0020 (space) and U+00A0 (no-break space) should map to a glyph with no contours and a positive advance width.
- Characters U+0009 and U+0020 should map to a glyph with the same width.

Shape of .notdef glyph

The .notdef glyph is very important for providing the user feedback that a glyph is not found in the font. This glyph should not be left without an outline as the user will only see what looks like a space if a glyph is missing and not be aware of the active font's limitation.

It is recommended that the shape of the .notdef glyph be either an empty rectangle, a rectangle with a question mark inside of it, or a rectangle with an "X". Creative shapes, like swirls or other symbols, may not be recognized by users as indicating that a glyph is missing from the font and is not being displayed at that location.

'BASE' table

The 'BASE' table allows for different scripts in the font to specify different values for the same baseline tag. This situation could arise when a developer makes a Unicode font, for example, by combining glyphs from fonts that use different baseline systems.

However, glyphs from different scripts in this font may not appear correctly aligned relative to each other when used with applications that either don't support the 'BASE' table or that support it but assume that a particular baseline will not vary across scripts. Furthermore, it is not always possible to determine the script of every glyph in the font, some "weakly-scripted" characters such as punctuation may be used in several scripts, and some glyphs such as ornaments may not have a script at all.

Thus, it is strongly recommended that developers construct their fonts so that all scripts in the 'BASE' table record the same value for a particular baseline if they want their fonts to work as expected in the above situations.

If baselines vary by script, then it is strongly recommended that the vendor add a DFLT script entry to the BASE table, which can be used if the script requested by the client is not matched or if the client does not or can not determine the script.

'cmap' table

When building a Unicode font for Windows, the platform ID should be 3 and the encoding ID should be 1 (this subtable must use cmap format 4). When building a symbol font for Windows, the platform ID should be 3 and the encoding ID should be 0.

When building a font to support surrogate characters i.e. the UCS-4 (4 byte) form of ISO/IEC 10646 (ISO/IEC 10646 UCS-4 contains 2^31 code positions and the Unicode transformation formats UTF-8 and UTF-16 access a subset of these code positions using surrogate characters), use platform ID 3, encoding ID 10 and format 12. Depending on support installed and the content of text being displayed, Windows 2000 may use either the format 4 or format 12 cmap. Therefore the first 64k codepoint to glyph mappings must be **identical** for any font containing both cmap format 4 and format 12. Please note that the content of format 12 subtable, needs to be a super set of the content in the format 4 subtable. The format 4 subtable needs to be included, for backward compatibility needs.

The number of glyphs that may be included in one font is limited to 64k.

Remember that, despite references to 'first' and 'second' subtables, the subtables must be stored in sorted order by platform and encoding ID.

'cvt' table

Should be defined only if required by font instructions.

'fpgm' table

Should be defined only if required by TrueType font instructions.

'glyf' table

The 'glyf' table contains TrueType outline data, and can be optimized by applying Microtype Express compression defined in ISO/IEC 14496-18.

NOTE t is recommended that developers perform this optimization prior to finalizing and adding a digital signature to the font. This is necessary for the creator's signature to remain valid in embedded OFF fonts.

'hdmx' table

This table improves the performance of OFF fonts with TrueType outlines. This table is not necessary at all unless instructions are used to control the "phantom points," and should be omitted if bits 2 and 4 of the flags field in the 'head' table are zero. (See the 'head' table description.) It is recommended that this table be included for fonts with one or more non-linearly scaled glyphs (i.e., bit 2 or 4 of the 'head' table flags field are set).

Device records should be defined for all sizes from 8 through 14 point, and even point sizes from 16 through 24 point. However, the table requires pixel-per-em sizes, which depend on the horizontal resolution of the output device. The records in 'hdmx' should cover both 96 dpi devices (CGA, EGA, VGA) and 300 dpi devices (laser and ink jet printers).

Thus, 'hdmx' should contain entries for the following pixel sizes (PPEM): 11, 12, 13, 15, 16, 17, 19, 21, 24, 27, 29, 32, 33, 37, 42, 46, 50, 54, 58, 67, 75, 83, 92, 100. These values have been rounded to the nearest pixel. For instance, 12 points at 300 dpi would measure 37.5 pixels, but this is rounded down to 37 for this list.

This will add approximately 9,600 bytes to the font file. However, there will be a significant improvement in speed when a client requests advance widths covered by these device records.

If the font includes an 'LTSH' table, the hdmx values are not needed above the linearity threshold.

'head' table

Although historical usage of the **fontRevision** value is varied, the recommended use of the field is to set it as a Fixed 16.16 value, and to report it rounded and zero-padded to three fractional decimal places. Examples: Decimal 1.5 is set as 0x00018000 and is reported as "1.500"; decimal 1.001 is set as 0x00010041 and is reported as "1.001". All data required. If the font has been compressed with Microtype Express compression defined in ISO/IEC 14496-18 this must be indicated in the flags field of the 'head' table.

'hhea' table

All data required. It is suggested that monospaced fonts set numberOfHMetrics to three (see hmtx).

'hmtx' table

All data required. It is suggested that monospaced fonts have three entries in the numberOfHMetrics field. OFF fonts that include CFF data must set numberOfHMetrics equal to the number of glyphs in the font and therefore cannot use the "repeat last width" optimization normally available within the 'hmtx' table.

'kern' table

Should contain a single kerning pair subtable (format 0). Windows will not support format 2 (two-dimensional array of kern values by class); nor multiple tables (only the first format 0 table found will be used) nor coverage bits 0 through 4 (i.e. assumes horizontal data, kerning values, no cross stream, and override). OFF fonts containing CFF data do not support the 'kern' table and should therefore specify kerning data using the 'GPOS' table (LookupType=2).

'loca' table

All data required for fonts with TrueType outlines. We recommend that local Offsets should be word-aligned, in both the short and long formats of this table.

The actual ordering of the glyphs in the font can be optimized based on expected utilization, with the most frequently used glyphs appearing at the beginning of the font file. Additionally, glyphs that are often used together should be grouped together in the file. The will help to minimize the amount of swapping required when the font is loaded into memory.

'LTSH' table

This table improves the performance of OFF fonts with TrueType outlines. The table should be used if bit 2 or 4 of flags in 'head' is set.

'maxp' table

All data required for a font with TrueType outlines. Fonts with CFF data must only fill the numGlyphs field.

'name' table

Platform and encoding ID's in the name table should be consistent with those in the cmap table. If they are not, the font will not load in Windows. 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 containing Roman characters that will be used on the Macintosh, an additional name record is required, specifying platform ID of 1 and encoding ID of 0.

Each set of name records should appear for US English (language ID = 0x0409 for Windows platform records, language ID = 0 for Macintosh records); additional language strings for the Windows platform set of records (platform ID 3) may be added at the discretion of the font vendor.

Remember that, despite references to "first" and "second," the name record must be stored in sorted order (by platform ID, encoding ID, language ID, name ID). The 'name' table platform/encoding IDs must match the 'cmap' table platform/encoding IDs, which is how Windows knows which name set to use.

Name strings

We recommend using name ID's 8-12, to identify manufacturer, designer, description, URL of the vendor, and URL of the designer. URLs must contain the protocol of the site: for example, http:// or mailto: or ftp://. The OFF font properties extension can enumerate this information to the users.

The Subfamily string in the 'name' table should be used for variants of weight (ultra light to extra black) and style (oblique/italic or not). So, for example, the full font name of "Helvetica Narrow Italic" should be defined as Family name "Helvetica Narrow" and Subfamily "Italic". This is so that Windows can group the standard four weights of a font in a reasonable fashion for non-typographically aware applications which only support combinations of "bold" and "italic."

The Full font name string usually contains a concatenation of strings 1 and 2. However, if the font is 'Regular' as indicated in string 2, then use only the family name contained in string 1. This is the font name that Windows will expose to users.

'OS/2' table

All data required. We recommend applying PANOSE values to fonts to improve the user's experience when using the Windows fonts folder or other font management utilities. If the font is a symbol font, the first byte of the PANOSE value must be set to 'decorative.'

sTypoAscender, sTypoDescender and sTypoLineGap

sTypoAscender is used to determine the optimum Offset from the top of a text frame to the first baseline. sTypoDescender is used to determine the optimum Offset from the last baseline to the bottom of the text frame. The value of (sTypoAscender - sTypoDescender) is recommended to equal one em.

While the OFF specification allows for CJK (Chinese, Japanese, and Korean) fonts' sTypoDescender and sTypoAscender fields to specify metrics different from the HorizAxis.ideo and HorizAxis.idtp baselines in the 'BASE' table, CJK font developers should be aware that existing applications may not read the 'BASE' table at all but simply use the sTypoDescender and sTypoAscender fields to describe the bottom and top edges of the ideographic em-box. If developers want their fonts to work correctly with such applications, they should ensure that any ideographic em-box values in the 'BASE' table describe the same bottom and top edges as the sTypoDescender and sTypoAscender fields. See subclause 6 "OFF CJK Font Guidelines" and "Ideographic Em-Box" respectively for more details.

For Western fonts, the Ascender and Descender fields in Type 1 fonts' AFM files are a good source of sTypoAscender and sTypoDescender, respectively. The Minion Pro font family (designed on a 1000-unit em), for example, sets sTypoAscender = 727 and sTypoDescender = -273.

sTypoAscender, sTypoDescender and sTypoLineGap specify the recommended line spacing for single-spaced horizontal text. The baseline-to-baseline value is expressed by:

OS/2.sTypoAscender - OS/2.sTypoDescender + OS/2.sTypoLineGap

sTypoLineGap will usually be set by the font developer such that the value of the above expression is approximately 120% of the em. The application can use this value as the default horizontal line spacing. The Minion Pro font family (designed on a 1000-unit em), for example, sets sTypoLineGap = 200.

'post' table

All information required, although the VM Usage fields may be set to zero. OFF fonts containing CFF outlines use only format 3.0 of the 'post' table. Glyph names are described in the Adobe document "Unicode and Glyph Names" in the informative reference 3 in the bibliography, which specifies glyph naming conventions for all Unicode characters as well as those that don't have standard Unicode values such as certain ligatures or glyphic variants.

NOTE Names for all glyphs must be supplied as it cannot be assumed that all Windows platforms will support the default names supplied on the Macintosh.

'prep' table

Should be defined only if required by the TrueType font instructions.

'VDMX' table

This table improves the performance of OFF fonts with TrueType outlines. It should be present if hints cause the font to scale non-linearly. If not present, the font is assumed to scale linearly. Clipping may occur if values in this table are absent and font exceeds linear height.

8 General recommendations

8.1 Optimized table ordering

OFF fonts with TrueType outlines are more efficient in the Windows operating system when the tables are ordered as follows (from first to last):

head, hhea, maxp, OS/2, hmtx, LTSH, VDMX, hdmx, cmap, fpgm, prep, cvt, loca, glyf, kern, name, post, gasp, PCLT, DSIG

The initial loading of an OFF font containing CFF data will be more efficiently handled if the following sfnt table ordering is used within the body of the sfnt (listed from first to last):

head, hhea, maxp, OS/2, name, cmap, post, CFF, (other tables, as convenient)

8.2 Non-standard (Symbol) fonts

Non-standard fonts such as Symbol or Wingdings™ have special requirements for Windows platforms. These requirements affect the 'cmap,' 'name,' and 'OS/2' tables; the requirements and recommendations for all other tables remain the same.

For non-standard fonts on Windows platforms, however, the 'cmap' and 'name' tables must use platform ID 3 () and encoding ID 0 (Unicode, non-standard character set). Remember that 'name' table encodings should agree with the 'cmap' table. Additionally, the first byte of the PANOSE value in the 'OS/2' table must be set to 'decorative.'

The 'cmap' subtable (platform 3, encoding 0) must use format 4. The character codes should start at 0xF000, which is in the Private Use Area of Unicode. It is suggested to derive the format 4 (encodings by simply adding 0xF000 to the format 0 (Macintosh) encodings.

Under Windows, only the first 224 characters of non-standard fonts will be accessible: a space and up to 223 printing characters. It does not matter where in user space these start, but 0xF020 is suggested. The usFirstCharIndex and usLastCharIndex values in the 'OS/2' table would be set based on the actual minimum and maximum character indices used.

8.3 Device resolutions

Windows makes use of a logical device resolution. The physical resolution of a device is also available, but fonts will be rendered based on the logical resolution. The table below lists some important logical resolutions in dots per inch (Horizontal x Vertical). The most important ratios (in order) are 1:1, 1.67:1 and 1.33:1.

Device	Resolution	Aspect Ratio
CGA	96 x 48	2:1
EGA	96 x 72	1.33:1
VGA	96 x 96	1:1
8514	120 x 120	1:1
Dot Matrix	120 x 72	1.67:1
Laser Printer	300 x 300	1:1
Laser Printer	600 x 600	1:1

8.4 Baseline to baseline distances

The 'OS/2' table fields sTypoAscender, sTypoDescender, and sTypoLineGap free applications from Macintosh- or Windows-specific metrics which are constrained by backward compatibility requirements. The following discussion only pertains to the platform-specific metrics.

The suggested Baseline to Baseline Distance (BTBD) is computed differently for Windows and the Macintosh, and it is based on different OFF metrics. However, if the recommendations below are followed, the BTBD will be the same for both Windows and the Mac.

Windows

The Windows metrics in the table below are returned as part of the logical font data structure.

Windows Metric	OFF Metric
Ascent	usWinAscent
descent	usWinDescent
internal leading	usWinAscent + usWinDescent - unitsPerEm
external leading	MAX(0, LineGap - ((usWinAscent + usWinDescent) - (Ascender - Descender)))

The suggested BTBD = *ascent* + *descent* + *external leading*

It should be clear that the "external leading" can never be less than zero. Pixels above the ascent or below the descent will be clipped from the character; this is true for all output devices.

The usWinAscent and usWinDescent are values from the 'OS/2' table. The unitsPerEm value is from the 'head' table. The LineGap, Ascender and Descender values are from the 'hhea' table.

Macintosh

Ascender and Descender are metrics defined and are not to be confused with the Windows ascent or descent, nor should they be confused with the true typographic ascender and descender that are found in AFM files.

Macintosh Metric	OFF Metric
ascender	Ascender
descender	Descender
Leading	LineGap

The suggested BTBD = *ascent* + *descent* + *leading*

If pixels extend above the ascent or below the descent, the character will be squashed in the vertical direction so that all pixels fit within these limitations; this is true for screen display only.

Making Them Match

If you perform some simple algebra, you will see that the suggested BTBD across both Macintosh and Windows will be identical if and only if:

LineGap >= (yMax - yMin) - (Ascender - Descender)

8.5 Style bits

For backwards compatibility with previous versions of Windows, the macStyle bits in the 'head' table will be used to determine whether or not a font is regular, bold or italic (in the absence of an 'OS/2' table). This is completely independent of the usWeightClass and PANOSE information in the 'OS/2' table, the ItalicAngle in the 'post' table, and all other related metrics. If the 'OS/2' table is present, then the fsSelection bits are used to determine this information.

8.6 Drop-out control

Drop-out control is needed if there is a difference in bitmaps with dropout control on and off. Two cases where drop-out control is needed are when the font is rotated or when the size of the font is at or below 8 ppem. Do not use SCANCTRL unless needed. SCANCTRL or the drop-out control rasterizer should be avoided for Roman fonts above 8 points per em (ppem) when the font is not under rotation. SCANCTRL should not be used for "stretched" fonts (e.g. fonts displayed at non-square aspect ratios, like that found on an EGA).

8.7 Embedded bitmaps

Three tables are used to embed bitmaps in OFF 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. OFF embedded bitmaps are also called 'sbits'.

The behavior of sbits within an OFF font is essentially transparent to the client. A client need not be aware whether the bitmap returned by the rasterizer comes from an sbit or from a scan-converted outline.

The metrics in 'sbit' tables overrule the outline metrics at all sizes where sbits are defined. Fonts with 'hdmx' tables should correct those tables with 'sbit' values.

'Sbit only' fonts, that is fonts with embedded bitmaps but without outline data, are permitted. Care must be taken to ensure that all required OFF tables except 'glyf' and 'loca' are present in such a font. Obviously, such fonts will only be able to return glyphs and sizes for which sbits are defined. These metrics are returned as part of the logical font data structure in the Macintosh platform.

8.8 OFF CJK font guidelines

This clause provides a checklist of links to various CJK-related clauses of the OFF specification. Some items are requirements; others, recommendations:

1. The ideographic em-box of an OFF font will be determined as described in "Ideographic Em-Box" in the Baseline Tags of the OFF Layout Tag Registry. Also see the description for OS/2.sTypoAscender and OS/2.sTypoDescender, and the 'BASE' table recommendation in clause 6.above.

2. CJK font vendors can choose to provide the ideographic character face (ICF) metrics, which applications can use for accurate text alignment. This is described in "Ideographic Character Face" in the Baseline Tags clause of the OFF Layout Tag Registry.

3. All OFF fonts that are used for vertical writing must include a Vertical Header ('vhea') table and a Vertical Metrics ('vmtx') table. It is strongly recommended that CFF OpenType fonts that are used for vertical writing include a Vertical Origin ('VORG') table.

4. If an OFF font with CFF outlines is to be used for vertical writing, Adobe Type Manager/NT 4.1 and the Windows 2000 OTF driver require that a Vertical Rotation ('vrt2') feature be present in the Glyph Substitution ('GSUB') table. See the Feature Tags (subclause 6.4.3) and informative reference [11] in the bibliography for a description of and further requirements for this feature.

5. See the Feature Tags (subclause 6.4.3) and informative reference [11] in the bibliography for descriptions of currently registered OFF layout features, such as Alternate Half Widths ('halt') and Traditional Forms ('trad') that can be specified in the font.