INTERNATIONAL STANDARD

ISO/IEC 14496-22

Fourth edition
2019-01

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 14496-22:2019(E)

COPYRIGHT PROTECTED DOCUMENT

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: 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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

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

This fourth edition cancels and replaces the third edition (ISO/IEC 14496-22:2015), which has been technically revised. It also incorporates the Amendments ISO/IEC 14496-22:2015/Amd.1:2017 and ISO/IEC 14496-22:2015/Amd.2:2017.

The main changes compared to the previous edition are as follows:

— new technology clauses were added;

— many existing clauses, subclauses, tables, figures and annexes were editorially revised.

A list of all parts in the ISO/IEC 14496 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

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, which is based on the OpenType®[1] font format, was originally developed as an extension of the TrueType®[2] font format, adding support for PostScript®[3] Compact Font Format (CFF) 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 Type1) 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.

CFF 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.

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 allows 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.

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 patents.

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights. The holders of these patent rights have assured ISO and IEC that they 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 holders of these patent rights is registered with ISO and IEC.

[1] OpenType® is the trademark of a product supplied by Microsoft. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

[2] TrueType® is the trademark of a product supplied by Apple Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

[3] PostScript® is the trademark of a product supplied by Adobe Systems Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

Information may be obtained from:

Apple Inc.
1 Infinite Loop MS 3-PAT
US-Cupertino, CA 95014-2084
Tel.: +1 408 974 9453
Email: iplaw@apple.com

Microsoft Corporation
Interoperability Group 3460 157th Avenue NE
US-Redmond, WA 98052
Tel.: +1 425 882 80 80

Monotype Imaging Inc.
500 Unicorn Park Drive
US-Woburn, MA 01801
Tel.: +1 781-970-6088
E-mail: vladimir.levantovsky@monotypeimaging.com

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 above. 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 document specifies the Open Font Format (OFF) specification, including the TrueType and Compact Font Format (CFF) outline formats. Many references to both TrueType and PostScript exist throughout this document, as Open Font Format fonts combine the two technologies. The document defines data structures for variaous font tables, and provides the necessary details for developers to build a font rendering and taxt layout/shaping engines in compliance with this document.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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 Coded Character Set (UCS)*

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

ISO/IEC 15948, *Information technology — Computer graphics and image processing — Portble Network Graphics: Functional specification* [4]

IEC 61966-2-1/Amd 1:2003: *Multimedia systems and equipment — Colour measurement and management — Part 2-1: Colour management — Default RGB colour space — sRGB*.

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

Unicode 11.0, http://www.unicode.org/versions/Unicode11.0.0/

Scalable Vector Graphics (SVG) 1.1 (2nd edition), W3C Recommendation, 16 August 2011 http://www.w3.org/TR/SVG11/

IETF BCP 47 specification, “Tags for Identifying Languages”. http://tools.ietf.org/html/bcp47

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

No terms and definitions are listed in this document.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

[4] Also available as a W3C Recommendation (Reference [15]).

3.2 Abbreviated terms

ACF	average character face
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
BMP	[Unicode] basic multilingual plane
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
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 CFF 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.

References to the Universally Coded Character Set and the Unicode standard are used throughout this document; the users of the OFF cannot meet the requirements of this document without strict adherence to these standards.

4.2 Filenames

OFF font files may have the extension .OTF, .TTF, .OTC or .TTC. The extensions .OTC and .TTC should only be used for font collection files. For additional information on filename extension conventions, see subclause 8.4.

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
uint8	8-bit unsigned integer.
int8	8-bit signed integer.
uint16	16-bit unsigned integer.
int16	16-bit signed integer.
uint24	24-bit unsigned integer.
uint32	32-bit unsigned integer.
int32	32-bit signed integer.
Fixed	32-bit signed fixed-point number (16.16)
FWORD	int16 that describes a quantity in font design units.
UFWORD	uint16 that describes a quantity in font design units.
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 table, design-variation axis, script, language system, feature, or baseline
Offset16	Short offset to a table, same as uint16, NULL offset = 0x0000
Offset32	Long offset to a table, same as uint32, NULL offset = 0x00000000

The F2DOT14 format consists of a signed, 2's complement integer and an unsigned fraction. To compute the actual value, take the integer and add the fraction. Examples of 2.14 values are:

Decimal Value	Hex Value	Integer	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

A Tag value is a uint8 array. Each byte within the array shall have a value in the range 0x20 to 0x7E. This corresponds to the range of values of Unicode Basic Latin characters in UTF-8 encoding, which is the same as the printable ASCII characters. As a result, a Tag value can be re-interpreted as a four-character sequence, which is conventionally how they are referred to. Formally, however, the value is a byte array.

When re-interpreted as characters, the Tag value is case sensitive. It shall have one to four non-space characters, padded with trailing spaces (byte value 0x20). A space character cannot be followed by a non-space character.

4.4 Table version numbers

Most tables have version numbers, and the version number for the entire font is contained in the Table Directory. Note that there are five different table version number types, each with its own numbering scheme.

- A single uint16 field. This is used in a number of tables, usually with versions starting at zero (0).
- Separate, uint16 major and minor version fields. This is used in a number of tables, usually with versions starting at 1.0.
- A Fixed field for major/minor version numbers. This is used in the maxp, post and vhea tables.
- A uint32 field with enumerated values.
- A uint32 field with a numeric value. This is used only in the DSIG and meta tables.

Only certain tables use a Fixed value for version, and only for reasons of backward compatibility. Fixed values will not be used in the future for any new tables that may be introduced. 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 version. The representation of a non-zero minor version, however, is not consistent with the normal treatment of Fixed values, in which the lower 16 bits represent a fractional value, N * 2 ^ -16. Rather, tables with non-zero minor version numbers always specify the literal value of the version number. For example, the version number of 'maxp' table version 0.5 is 0x00005000, and that of 'vhea' table version 1.1 is 0x00011000. When Fixed is indicated as the type for a version field, the possible values should be treated as an enumeration of specific values, rather than as a numeric value capable of representing many potential major and minor versions.

The Table Directory uses a uint32 field with an enumeration of defined values that represent four-character tags; see subclause 4.5 (Top-level OFF organization) for details.

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 handle newer versions gracefully.

Minor version number changes always imply format changes that are compatible extensions. If an implementation understands a major version number, then it can safely proceed reading the table. If the minor version is greater than the latest version recognized by the implementation, then the extension fields will be undetectable to the implementation.

For purposes of compatibility, version numbers that are represented using a single uint16 or uint32 value are treated like a minor version number, and any format changes are compatible extensions.

Note that some field values that were undefined or reserved in an earlier revision may be assigned meanings in a minor version change. Implementations should never make assumptions regarding reserved or unassigned values or bits in bit fields, and can ignore them if encountered. When writing font data, tools should always write zero for reserved fields or bits. Validators should warn of any non-zero values for fields or bits that are not defined for the given version against which data is being validated.

If the major version is not recognized, the implementation must not read the table as it can make no assumptions regarding interpretation of the binary data. The implementation should treat the table as missing.

4.5 Top-level OFF organization

4.5.1 Offset table

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 starts 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 font collection, the beginning point of the Offset Table for each font is indicated in the TTCHeader.

Offset Table		
Type	**Name**	**Description**
uint32	sfntVersion	0x00010000 or 0x4F54544F ('OTTO') – see below.
uint16	numTables	Number of tables.
uint16	searchRange	(Maximum power of 2 <= numTables) x 16.
uint16	entrySelector	Log2(maximum power of 2 <= numTables).
uint16	rangeShift	NumTables x 16 - searchRange.

OFF fonts that contain TrueType outlines should use the value of 0x00010000 for sfntVersion. OFF fonts containing CFF data (version 1 or 2) should use 0x4F54544F ('OTTO', when re-interpreted as a Tag) for sfntVersion.

4.5.2 Table directory

The Offset Table is followed immediately by the Table Record entries. Entries in the Table Record shall 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**
Tag	tableTag	Table identifier.
uint32	checksum	CheckSum for this table.
Offset32	Offset	Offset from beginning of TrueType font file.
uint32	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.

Table tags are the names given to tables in the OFF font file. For requirements of Tag values, see subclause 4.3 (Data types).

Some tables have an internal structure with subtables located at specified offsets, and as a result, it is possible to construct a font with data for different tables interleaved. In general, tables should be arranged contiguously without overlapping the ranges of distinct tables. In any case, however, table lengths measure a contiguous range of bytes that encompasses all of the data for a table. This applies to any subtables as well as to top-level tables.

4.5.3 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:

```
uint32
CalcTableChecksum(uint32 *Table, uint32 Length)
{
uint32 Sum = 0L;
uint32 *Endptr = Table+((Length+3) & ~3) / sizeof(uint32);

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

NOTE This function implies that the length of a table is 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 boundaries, and any remaining space between tables is padded with zeros. The length of all tables would 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:

- Set the checkSumAdjustment to 0.
- Calculate the checksum for all the tables including the 'head' table and enter that value into the table directory.
- Calculate the checksum for the entire font.
- Subtract that value from value 0xB1B0AFBA.
- 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 Font collections

4.6.1 The Font Collection overview

A Font Collection (either TTC or OTC, formerly known as TrueType Collection) is a means of delivering multiple OFF font resources in a single file structure. Font collections containing either TrueType or CFF outlines (TTC or OTC) are most useful when the fonts to be delivered together share many glyphs in common. By allowing multiple fonts to share glyph sets and other common font tables, font collections 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. Font collections were defined to solve this problem.

NOTE Even though the original definition of TrueType Collection (as part of the TrueType specification) was intended to be used with fonts containing TrueType outlines, this is no longer strictly the case. TTC files may contain various types of outlines (or a mix of them), regardless of whether or not fonts have layout tables present. For backward compatibility and simplicity, the description of the font collection file structure is using the term "TrueType Collection" though it is understood that it is used to identify a generic font collection structure containing any type of outline tables.

4.6.2 The Font Collection file structure

A font collection file consists of a single TTC Header table, one or more Offset Tables with Table Directories (each corresponding to a different font resource), and a number of OFF tables. The TTC Header shall be located at the beginning of the TTC file.

The TTC file shall contain a complete Offset Table and Table Directory for each font resource. 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.

4.6.3 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	Font Collection ID string: 'ttcf' (used for fonts with CFF or CFF2 outlines, as well as TrueType outlines)
uint16	majorVersion	Major version of the TTC Header, = 1.

uint16	minorVersion	Minor version of the TTC Header, = 0.
uint32	numFonts	Number of fonts in TTC
Offset32	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'
uint16	majorVersion	Major version of the TTC Header, = 2.
uint16	minorVersion	Minor version of the TTC Header, = 0.
uint32	numFonts	Number of fonts in TTC
Offset32	offsetTable[numFonts]	Array of offsets to the OffsetTable for each font from the beginning of the file
uint32	ulDsigTag	Tag indicating that a DSIG table exists, 0x44534947 ('DSIG') (null if no signature)
uint32	dsigLength	The length (in bytes) of the DSIG table (null if no signature)
uint32	dsigOffset	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

5.2.1 List of required tables

Whether TrueType or CFF 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

5.2.2 cmap – Character to glyph index mapping table

This table defines the mapping of character codes to the glyph index values used in the font. It may contain more than one subtable, in order to support more than one character encoding scheme.

5.2.2.1 Table overview

This table defines the mapping of character codes to a default glyph index. Different subtables may be defined that each contain mappings for different character encoding schemes. The table header indicates the character encodings for which subtables are present.

Regardless of the 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.

Each subtable is in one of seven possible formats and begins with a format code indicating the format used. The first four formats — formats 0, 2, 4 and 6 — were originally defined prior to Unicode 2.0. These formats allow for 8-bit single-byte, 8-bit multi-byte, and 16-bit encodings. With the introduction of supplementary planes in Unicode 2.0, the Unicode addressable code space extends beyond 16 bits. To accommodate this, three additional formats were added — formats 8, 10 and 12 — that allow for 32-bit encoding schemes.

Other enhancements in Unicode led to the addition of other subtable formats. Subtable format 13 allows for an efficient mapping of many characters to a single glyph; this is useful for “last-resort” fonts that provide fallback rendering for all possible Unicode characters with a distinct fallback glyph for different Unicode ranges. Subtable format 14 provides a unified mechanism for supporting Unicode variation sequences.

NOTE The 'cmap' table version number remained at 0x0000 for fonts that make use of the newer subtable formats.

'cmap' Header

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

Type	Name	Description
uint16	Version	Table version number (0)
uint16	numTables	Number of encoding tables that follow
EncodingRecord	encodingRecords[numTables]	

Encoding records and encodings

The array of encoding records specify particular encoding and the offset to the subtable for each encoding.

Encoding Record:

Type	Name	Description
uint16	platformID	Platform ID.
uint16	encodingID	Platform-specific encoding ID.
Offset32	offset	Byte offset from beginning of table to the subtable for this encoding

The platform ID and platform-specific encoding ID in the encoding record are used to specify a particular character encoding. In the case of the Macintosh platform, a language field within the mapping subtable is also used for this purpose.

The encoding record entries in the 'cmap' header 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.

Complete details on platformIDs, and platform-specific encoding and language IDs are defined in subclause 5.2.7. Some specific details applicable to the 'cmap' table are provided here.

Unicode platform (platform ID = 0)

Unicode Variation Sequences supported by the font should be specified in the 'cmap' table using a format 14 subtable. A format 14 subtable must only be used under platform ID 0 and encoding ID 5.*Macintosh platform (platform ID = 1)*

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 platform (platform ID = 3)

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.

Microsoft strongly recommends using a BMP Unicode 'cmap' for all fonts. However, other non-Unicode encodings are also used in existing fonts with the Windows platform. The following are encoding IDs defined for the Windows platform:

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

Fonts that support Unicode BMP characters on Windows platform must have a Format 4 'cmap' subtable for platform ID 3, platform-specific encoding 1.

Fonts that support Unicode supplementary-plane characters on Windows platform must have a Format 12 subtable for platform ID 3, encoding ID 10. To ensure backward compatibility with older software and devices, a format 4 subtable for platform ID 3, encoding ID 1 is also required. The characters supported in the format 4 subtable must be a subset of the characters in the format 12 subtable and should include all of the Unicode BMP characters supported by the font.

Custom platform (Platform ID = 4) and 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 that the 'cmap' subtable shall use Format 0 or 6 for its subtable, and the encoding shall be identical to the CFF's encoding.

This 'cmap' encoding is not required. It provides a compatibility mechanism for non-Unicode applications that use the font as if it were Windows ANSI encoded. Non-Windows ANSI Type 1 fonts, such as Cyrillic and Central European fonts, that Adobe shipped in the past had “0” (Windows ANSI) recorded in the CharSet field of the .PFM file; ATM for Windows 9x ignores the CharSet altogether. Adobe provides this compatibility 'cmap' encoding in every OTF converted from a Type1 font in which the Encoding is not StandardEncoding.

5.2.2.2 cmap subtable formats

5.2.2.2.1 Use of the language field in 'cmap' subtables

The language 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.

5.2.2.2.2 Format 0: Byte encoding table

This is the Macintosh platform standard character to glyph index mapping table which is available via the Reference [7].

'cmap' Subtable Format 0:

Type	Name	Description
uint16	format	Format number is set to 0.
uint16	length	This is the length in bytes of the subtable.

uint16	language	For requirements on use of the language field, see subclause 5.2.2.2.1.
uint8	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.

5.2.2.2.3 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.

'cmap' Subtable Format 2:

Type	Name	Description
uint16	format	Format number is set to 2.
uint16	length	This is the length in bytes of the subtable.
uint16	language	For requirements on use of the language field, see subclause 5.2.2.2.1.
uint16	subHeaderKeys[256]	Array that maps high bytes to subHeaders: value is subHeader index * 8.
SubHeader	subHeaders[ ]	Variable-length array of subHeader structures.
uint16	glyphIndexArray[ ]	Variable-length array containing subarrays used for mapping the low byte of 2-byte characters.

A subHeader is structured as follows:

SubHeader Record:

Type	Name	Description
uint16	firstCode	First valid low byte for this subHeader.
uint16	entryCount	Number of valid low bytes for this subHeader.
int16	idDelta	See text below.
uint16	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.

5.2.2.2.4 Format 4: Segment mapping to delta values

This is the standard character-to-glyph-index mapping table for the Windows platform for fonts that support Unicode BMP characters. See Windows platform (platform ID = 3) in subclause 5.2.2.1 for additional details regarding subtable formats for Unicode encoding on the Windows platform.

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 shall occur in the following order:

a) A four-word header gives parameters for an optimized search of the segment list;

b) Four parallel arrays describe the segments (one segment for each contiguous range of codes);

c) A variable-length array of glyph IDs (unsigned words).

'cmap' Subtable Format 4:

Type	Name	Description
uint16	format	Format number is set to 4.
uint16	length	This is the length in bytes of the subtable.
uint16	language	For requirements on use of the language field, see subclause 5.2.2.2.1.
uint16	segCountX2	2 x segCount.
uint16	searchRange	2 x (2**floor(log2(segCount)))
uint16	entrySelector	log2(searchRange/2)
uint16	rangeShift	2 x segCount - searchRange
uint16	endCode[segCount]	End characterCode for each segment, last=0xFFFF.
uint16	reservedPad	Set to 0.
uint16	startCode[segCount]	Start character code for each segment.
int16	idDelta[segCount]	Delta for all character codes in segment.
uint16	idRangeOffset[segCount]	Offsets into glyphIdArray or 0
uint16	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 x (largest power of 2 <=39))
entrySelector	5	(= $\log_2$ (32))
rangeShift	14	(= 2 x 39 – 64)

Each segment is described by a startCode and endCode, along with an idDelta and an idRangeOffset, which are used for mapping the character codes in the segment. The segments are sorted in order of increasing endCode values, and the segment values are specified in four parallel arrays. You search for the first endCode that is greater than or equal to the character code you want to map. If the corresponding startCode is less than or equal to the character code, then you use the corresponding idDelta and idRangeOffset to map the character code to a glyph index (otherwise, the missingGlyph is returned). For the search to terminate, the final startCode and endCode values shall 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:

```
glyphId = *(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

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

5.2.2.2.5 Format 6: Trimmed table mapping

'cmap' Subtable Format 6:

Type	Name	Description
uint16	format	Format number is set to 6.
uint16	length	This is the length in bytes of the subtable.
uint16	language	For requirements on use of the language field, see subclause 5.2.2.2.1.
uint16	firstCode	First character code of subrange.
uint16	entryCount	Number of character codes in subrange.
uint16	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.

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

Format 8 is similar to format 2 in that it provides for mixed-length character codes. Instead of allowing for 8- and 16-bit character codes, however, it allows for 16- and 32-bit character codes.

If a font contains Unicode supplementary-plane characters (U+10000 to U+10FFFF), then it's likely that it will also include Unicode BMP characters (U+0000 to U+FFFF) as well. Hence, there is a need to map a mixture of 16-bit and 32-bit character 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. (Since the Unicode code space extends only to U+1FFFFF, a potential conflict exists only for characters U+0000 to U+001F, which are non-printing control characters.) 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.

'cmap' Subtable Format 8:

Type	Name	Description
uint16	format	Subtable format; set to 8.
uint16	reserved	Reserved; set to 0
uint32	length	Byte length of this subtable (including the header)
uint32	language	For requirements on use of the language field, see subclause 5.2.2.2.1.

uint8	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
uint32	numGroups	Number of groupings which follow
SequentialMapGroup	groups[numGroups]	Array of SequentialMapGroup records.

Each sequential map group record specifies a character range and the starting glyph ID mapped from the first character. Glyph IDs for subsequent characters follow in sequence.

SequentialMapGroup Record:

Type	Name	Description
uint32	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
uint32	endCharCode	Last character code in this group; same condition as listed above for the startCharCode
uint32	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 shall be sorted by increasing startCharCode. A group's endCharCode shall 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 was created to support Unicode supplementary-plane characters, it's not widely supported or used. Also, no character encoding other than Unicode uses mixed 16-/32-bit characters. The use of this format is discouraged.

5.2.2.2.7 Format 10: Trimmed array

Format 10 is similar to format 6, in that it defines a trimmed array for a tight range of character codes. It differs, however, in that is uses 32-bit character codes.

'cmap' Subtable Format 10:

Type	Name	Description
uint16	format	Subtable format; set to 10.
uint16	reserved	Reserved; set to 0
uint32	length	Byte length of this subtable (including the header)
uint32	language	For requirements on use of the language field, see subclause

		5.2.2.2.1.
uint32	startCharCode	First character code covered
uint32	numChars	Number of character codes covered
uint16	glyphs[]	Array of glyph indices for the character codes covered

This format is not widely used and is not supported on Windows platforms. It would be most suitable for fonts that support only a contiguous range of Unicode supplementary-plane characters, but such fonts are rare.

5.2.2.2.8 Format 12: Segmented coverage

This is the standard character-to-glyph-index mapping table for the Windows platform for fonts supporting Unicode supplementary-plane characters (U+10000 to U+10FFFF). See Windows platform (platform ID = 3) in subclause 5.2.2.1 for additional details regarding subtable formats for Unicode encoding on the Windows platform.

Format 12 is similar to format 4 in that it defines segments for sparse representation in 4-byte character space. It differs, however, in that it uses 32-bit character codes.

'cmap' Subtable Format 12:

Type	Name	Description
uint16	format	Subtable format; set to 12.
uint16	reserved	Reserved; set to 0
uint32	length	Byte length of this subtable (including the header)
uint32	language	For requirements on use of the language field, see subclause 5.2.2.2.1.
uint32	numGroups	Number of groupings which follow
SequentialMapGroup	groups[numGroups]	Array of SequentialMapGroup records.

The sequential map group record is the same format as is used for the format 8 subtable. The qualifications regarding 16-bit character codes does not apply here, however, since characters codes are uniformly 32-bit.

SequentialMapGroup Record:

Type	Name	Description
uint32	startCharCode	First character code in this group
uint32	endCharCode	Last character code in this group
uint32	startGlyphID	Glyph index corresponding to the starting character code

Groups shall be sorted by increasing startCharCode. A group's endCharCode shall 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.

5.2.2.2.9 Format 13: Many-to-one range mappings

This subtable provides for situations in which the same glyph is used for hundreds or even thousands of consecutive characters spanning across multiple ranges of the code space. This subtable format may be useful for “last resort” fonts, although these fonts may use other suitable subtable formats as well. (For "last resort" fonts, see also the 'head' table flags in subclause 5.2.3, flags: bit 14).

NOTE Subtable formats 13 has the same data structure as format 12; it differs only in the interpretation of the startGlyphID/glyphID fields.

'cmap' Subtable Format 13:

Type	Name	Description
uint16	format	Subtable format; set to 13.
uint16	reserved	Reserved; set to 0
uint32	length	Byte length of this subtable (including the header)
uint32	language	For requirements on use of the language field, see subclause 5.2.2.2.1.
uint32	numGroups	Number of groupings which follow
ConstantMapGroup	groups[numGroups]	Array of ConstantMapGroup records.

The constant map group record has the same structure as the sequential map group record, with start and end character codes and a mapped glyph ID. However, the same glyph ID applies to all characters in the specified range rather than sequential glyph IDs.

ConstantMapGroup Record:

Type	Name	Description
uint32	startCharCode	First character code in this group
uint32	endCharCode	Last character code in this group
uint32	glyphID	Glyph index to be used for all the characters in the group's range.

5.2.2.2.10 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 BMP subtable or BMP + supplementary-planes 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.

*'cmap' Subtable Format 14:*Format 14 header		
Type	**Name**	**Description**
uint16	format	Subtable format; set to 14.
uint32	length	Byte length of this subtable (including this header)
uint32	numVarSelectorRecords	Number of variation Selector Records
VariationSelector	varSelector[numVarSelectorRecords]	Array of VariationSelector records.

Each variation selector records specifies a variation selector character, and offsets to “default” and “non-default” tables used to map variation sequences using that variation selector.

Variation Selector Record		
Type	**Name**	**Description**
uint24	varSelector	Variation selector
Offset32	defaultUVSOffset	Offset from the start of the format 14 subtable to Default UVS Table. May be 0.
Offset32	nonDefaultUVSOffset	Offset from the start of the format 14 subtable 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’.

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**
uint32	numUnicodeValueRanges	Number of ranges that follow
UnicodeRange	ranges[numUnicodeValueRanges]	Array of UnicodeRange records.

Each Unicode range record specifies a contiguous range of Unicode values.

UnicodeRange Record		
Type	**Name**	**Description**
uint24	startUnicodeValue	First value in this range
uint8	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’) shall 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**
uint32	numUVSMappings	Number of UVS Mappings that follow
UVSMapping	uvsMappings[numUVSMappings]	Array of UVSMapping records.

Each UVSMapping record provides a glyph ID mapping for one base Unicode character, when that base character is used in a variation sequence with the current variation selector.

UVSMapping Record		
Type	**Name**	**Description**
uint24	unicodeValue	Base Unicode value of the UVS
uint16	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

5.2.3 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
uint16	majorVersion	Major version number of the font header table – set to 1.
uint16	minorVersion	Minor version number of the font header table – set to 0.
Fixed	fontRevision	Set by font manufacturer.
uint32	checkSumAdjustment	To compute: set it to 0, sum the entire font as uint32, then store 0xB1B0AFBA - sum. If the font is used as a component in a font collection file, the value of this field will be invalidated by changes to the file structure and font table directory, and must be ignored.
uint32	magicNumber	Set to 0x5F0F3CF5.
uint16	flags	Bit 0: Baseline for font at y=0; Bit 1: Left sidebearing point at x=0 (relevant only for TrueType rasterizers) – see the note below regarding variable fonts; 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); Bit 5: This bit is not used in OFF, and should not be set in order to ensure compatible behavior on all platforms. If set, it may result in different behavior for vertical layout in some platforms. (See 'head' table specification of "Apple's TrueType Reference Manual" [7] for details regarding behavior in Apple platforms.) Bits 6-10: These bits are not used in OFF and should always be cleared. (See 'head' table specification of "Apple's TrueType Reference Manual" [7] for details regarding legacy use in Apple platforms.) Bit 11: Font data is 'lossless' as a result of having been subjected to optimizing transformation and/or compression (such as font compression mechanisms defined by ISO/IEC 14496-18, MicroType®[5] Express, WOFF2 [24], or similar) where the original font functionality and features are retained but the binary compatibility between input and output font files is not guaranteed. As a result of the applied transform, the 'DSIG' Table may also be invalidated. Bit 12: Font converted (produce compatible metrics) Bit 13: Font optimized for ClearType®[6]. Note, fonts that rely on embedded bitmaps (EBDT) for rendering should not be considered optimized for ClearType, and therefore

[5] MicroType® is the trademark of a product supplied by Monotype Imaging Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

[6] ClearType® is the trademark of a product supplied by Microsoft. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

		should keep this bit cleared. Bit 14: Last Resort font. If set, indicates that the glyphs encoded in the cmap subtables are simply generic symbolic representations of code point ranges and don't truly represent support for those code points. If unset, indicates that the glyphs encoded in the cmap subtables represent proper support for those code points Bit 15: Reserved, set to 0
uint16	unitsPerEm	Set to a value from 16 to 16384. Any value in this range is valid. In fonts that have TrueType outlines, a power of 2 is recommended as this allows performance optimization in some rasterizers.
LONGDATETIME	created	Number of seconds since 12:00 midnight that started January 1st, 1904, in GMT/UTC time zone. 64-bit integer
LONGDATETIME	modified	Number of seconds since 12:00 midnight that started January 1st, 1904, in GMT/UTC time zone. 64-bit integer
int16	xMin	For all glyph bounding boxes.
int16	yMin	For all glyph bounding boxes.
int16	xMax	For all glyph bounding boxes.
int16	yMax	For all glyph bounding boxes.
uint16	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).
uint16	lowestRecPPEM	Smallest readable size in pixels.
int16	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]
int16	indexToLocFormat	0 for short offsets, 1 for long.
int16	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.

Note that, in a variable font with TrueType outlines, the left side bearing for each glyph must equal xMin, and bit 1 in the flags field must be set. Also, bit 5 must be cleared in all variable fonts. For general information on OFF Font variations, see subclause 7.1.

5.2.4 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 shall be set to 0.

Type	Name	Description
uint16	majorVersion	Major version number of the horizontal header table – set to 1.
uint16	minorVersion	Minor version number of the horizontal header table – set to 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 in some legacy platform implementations.
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)).
int16	caretSlopeRise	Used to calculate the slope of the cursor (rise/run); 1 for vertical.
int16	caretSlopeRun	0 for vertical.
int16	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
int16	(reserved)	set to 0
int16	(reserved)	set to 0
int16	(reserved)	set to 0
int16	(reserved)	set to 0

int16	metricDataFormat	0 for current format.
uint16	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.

'hhea' Table and OFF Font Variations

In a variable font, various font-metric values within the horizontal header table may need to be adjusted for different variation instances. Variation data for 'hhea' entries can be provided in the metrics variations ('MVAR') table. Different 'hhea' entries are associated with particular variation data in the 'MVAR' table using value tags, as follows:

'hhea' entry	Tag
caretOffset	'hcof'
caretSlopeRise	'hcrs'
caretSlopeRun	'hcrn'

For general information on OFF Font variations, see subclause 7.1.

5.2.5 hmtx – Horizontal metrics

Glyph metrics used for horizontal text layout include glyph advance widths, side bearings and X-direction min and max values (xMin, xMax). These are derived using a combination of the glyph outline data ('glyf', 'CFF ' or 'CFF2') and the horizontal metrics table. The horizontal metrics ('hmtx') table provides glyph advance widths and left side bearings.

In a font with TrueType outline data, the 'glyf' table provides xMin and xMax values, but not advance widths or side bearings. The advance width is always obtained from the 'hmtx' table. In some fonts, depending on the state of flags in the 'head' table, the left side bearings may be the same as the xMin values in the 'glyf' table, though this is not true for all fonts. (See the description of bit 1 of the flags field in the 'head' table.) For this reason, left side bearings are provided in the 'hmtx' table. The right side bearing is always derived using advance width and left side bearing values from the 'hmtx' table, plus bounding-box information in the glyph description — see below for more details.

In a variable font with TrueType outline data, the left side bearing value in the 'hmtx' table must always be equal to xMin (bit 1 of the 'head' flags field must be set). Hence, these values can also be derived directly from the 'glyf' table. Note that these values apply only to the default instance of the variable font: non-default instances may have different side bearing values. These can be derived from interpolated “phantom point” coordinates using the 'gvar' table (see below for additional details), or by applying variation data in the 'HVAR' table to default-instance values from the 'glyf' or 'hmtx' table.

In a font with CFF version 1 outline data, the 'CFF ' table does include advance widths. These values are used by PostScript processors, but are not used in OFF layout. In an OFF context, the 'hmtx' table is required and must be used for advance widths. Note that fonts in a Font Collection file that share a 'CFF ' table may specify different advance widths in font-specific 'hmtx' tables for a particular glyph index. Also note that the 'CFF2' table does not include advance widths. In addition, for either 'CFF ' or 'CFF2' data, there are no explicit xMin and xMax values; side bearings are implicitly contained within the CharString data, and can be obtained from the the CFF / CFF2 rasterizer. Some layout engines may use left side bearing values in the 'hmtx' table, however; hence, font production tools should ensure that the lsb values in the 'hmtx' table match the implicit xMin values reflected in the CharString data. In a variable font with CFF2 outline data, left side bearing and advance width values for non-default instances should be obtained by combining information from the 'hmtx' and 'HVAR' tables.

The table uses a longHorMetric record to give the advance width and left side bearing of a glyph. Records are indexed by glyph ID. As an optimization, the number of records can be less than the number of glyphs, in which case the advance width value of the last record applies to all remaining glyph IDs. This can be useful in monospaced fonts, or in fonts that have a large number of glyphs with the same advance width (provided the glyphs are ordered appropriately). The number of longHorMetric records is determined by the numberOfHMetrics field in the 'hhea' table.

If the longHorMetric array is less than the total number of glyphs, then that array is followed by an array for the left side bearing values of the remaining glyphs. The number of elements in the left side bearing will be derived from numberOfHMetrics plus the numGlyphs field in the 'maxp' table.

Horizontal Metrics Table:

Type	Name	Description
longHorMetric	hMetrics [numberOfHMetrics]	Paired advance width and left side bearing values for each glyph. Records are indexed by glyph ID.
int16	leftSideBearing [numGlyphs - numberOfHMetrics]	Left side bearings for glyph IDs greater than or equal to numberOfHMetrics.

longHorMetric Record:

Type	Name	Description
uint16	advanceWidth	Advance width, in font design units.
int16	lsb	Glyph left side bearing, in font design units.

In a font with TrueType outlines, xMin and xMax values for each glyph are given in the 'glyf' table. The advance width (“aw”) and left side bearing (“lsb”) can be derived from the glyph “phantom points”, which are computed by the TrueType rasterizer; or they can be obtained from the 'hmtx' table. In a font with CFF or CFF2 outlines, xMin (= left side bearing) and xMax values can be obtained from the CFF / CFF2 rasterizer. From those values, the right side bearing (“rsb”) is calculated as follows:

rsb = aw - (lsb + xMax - xMin)

If pp1 and pp2 are TrueType phantom points used to control lsb and rsb, their initial position in the X-direction is calculated as follows:

pp1 = xMin - lsb
pp2 = pp1 + aw

5.2.6 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	version	0x00005000 for version 0.5 (Note the difference in the representation of a non-zero fractional part, in Fixed numbers.)
uint16	numGlyphs	The number of glyphs in the font.

Version 1.0

Type	Name	Description
Fixed	version	0x00010000 for version 1.0.
uint16	numGlyphs	The number of glyphs in the font.
uint16	maxPoints	Maximum points in a non-composite glyph.
uint16	maxContours	Maximum contours in a non-composite glyph.
uint16	maxCompositePoints	Maximum points in a composite glyph.
uint16	maxCompositeContours	Maximum contours in a composite glyph.
uint16	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.
uint16	maxTwilightPoints	Maximum points used in Z0.
uint16	maxStorage	Number of Storage Area locations.
uint16	maxFunctionDefs	Number of FDEFs, equal to the highest function number + 1.
uint16	maxInstructionDefs	Number of IDEFs.
uint16	maxStackElements	Maximum stack depth across Font Program ('fpgm' table), CVT Program ('prep' table) and all glyph instructions (in the 'glyf' table).
uint16	maxSizeOfInstructions	Maximum byte count for glyph instructions.
uint16	maxComponentElements	Maximum number of components referenced at "top level" for any composite glyph.
uint16	maxComponentDepth	Maximum levels of recursion; 1 for simple components.

5.2.7 name – Naming table

5.2.7.1 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 refer to them using a language-independent name ID. In addition to language variants, the table also allows for platform-specific character-encoding variants. Clients that need a particular string can look it up by its platform ID, encoding ID, language ID and name ID. Note that different platforms may have different requirements for the encoding of strings.

Many newer platforms can use strings intended for different platforms if a font does not include strings for that platform. Some applications might display incorrect strings, however, if strings for the current platform are not included.

Naming table header

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. Both formats include variable-size string-data storage, and an array of name records that are used to identify the type of string (name ID), platform, encoding and language variants of the string, and the location within the storage.

Naming table format 0

The format 0 naming table is organized as follows:

Type	Name	Description
uint16	format	Format selector (=0).
uint16	count	Number of name records.
Offset16	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.

Format 0 differs from format 1 in regard to handling of language identification: it uses only numeric language IDs, which generally are values less than 0x8000 and have platform-specific interpretations. See subclause 5.2.7.2 for more details.

Naming table format 1

The format 1 naming table adds additional elements, as follows:

Type	Name	Description
uint16	format	Format selector (=1).
uint16	count	Number of name records.
Offset16	stringOffset	Offset to start of string storage (from start of table).
NameRecord	nameRecord[count]	The name records where *count* is the number of records.
uint16	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 (LangTagRecord) 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
uint16	length	Language-tag string length (in bytes).
uint16	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 shall be as specified in the IETF specification BCP 47. This provides tags such as "en", "fr-CA" and "zh-Hant" to identify languages, including dialects, written form and other language variants.

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 identity 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.

5.2.7.2 Name records

Each string in the string storage is referenced by a name record. The name record has a multi-part key, to identify the logical type of string and its language or platform-specific implementation variants, plus the location of the string in the string storage.

Each *NameRecord* is organized as follows:

Type	Name	Description
uint16	platformID	Platform ID.
uint16	encodingID	Platform-specific encoding ID.
uint16	languageID	Language ID.
uint16	nameID	Name ID.
uint16	length	String length (in bytes).
Offset16	Offset	String offset from start of storage area (in bytes).

The name ID identifies a logical string category, such as family name or copyright. Name IDs are the same for all platforms and languages; these are described in detail below. The other three elements of the key allow for platform-specific implementations: a platform ID, a platform-specific encoding ID, and a language ID.

As with encoding records in the 'cmap' table, name records shall be sorted first by platform ID, then by encoding ID, then language ID, and then, in addition, by name ID. Descriptions of the various IDs follow.

5.2.7.3 Platform, encoding and language IDs

The platform, encoding and language IDs of a name record allow for platform-specific implementations. Different platforms can support different encodings, and different languages. All encoding IDs are platform-

specific. Language IDs are similarly platform-specific, except in the case of IDs used in conjuction with the language-tag mechanism of naming table format 1, described above.

NOTE Platform IDs, platform-specific encoding IDs and, in some cases, platform-specific language IDs are also used in the 'cmap' table (see subclause 5.2.2)

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. A format 0 naming table shall use only language ID values less than 0x8000 from the platform-specific enumerations given below. (Exceptions to this requirement are permitted, however, in the case of user-defined platforms — platform IDs 240 to 255.) Values greater than or equal to 0x8000 can be used in a format 1 naming table in conjunction with language-tag records, as described above. Not all platforms have platform-specific language IDs, and not all platforms support language-tag records.

Platform IDs

The following platform IDs are defined:

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

Platform ID 2 (ISO) has been deprecated. It was intended to represent ISO/IEC 10646, as opposed to Unicode. It is redundant, however, since both standards have identical character code assignments.

Platform ID values 240 through 255 are reserved for user-defined platforms, and shall never be assigned to a registered platform.

Platform-specific encoding and language IDs: Unicode platform (platform ID = 0)

The following encoding IDs are defined for use with the Unicode platform:

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 (cmap subtable formats 0, 4, 6).
4	Unicode 2.0 and onwards semantics, Unicode full repertoire (cmap subtable formats 0, 4, 6, 10, 12).
5	Unicode Variation Sequences (cmap subtable format 14).
6	Unicode full repertoire (cmap subtable formats 0, 4, 6, 10, 12, 13).

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.

A new encoding ID for the Unicode platform is also sometimes assigned when new cmap subtable formats are added to the specification, so as to allow for compatibility with existing parsers. For example, when cmap subtable formats 10 and 12 were added to the specification, encoding ID 4 was added as well, and when cmap subtable format 13 was added to the specification, encoding ID 6 was added. The cmap subtable formats listed in the table above are the only ones that may be used for the corresponding encoding ID.

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.

Platform-specific encoding and language IDs: Windows platform (platform ID= 3)

Windows encoding IDs

The following encoding IDs are defined for use with the Windows platform:

Platform ID	Encoding ID	Description
3	0	Symbol
3	1	Unicode BMP
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 full repertoire

When building a Unicode font for Windows, the platform ID should be 3 and the encoding ID should be 1, and the referenced string data must be encoded in UTF-16BE. When building a symbol font for Windows, the platform ID should be 3 and the encoding ID should be 0, and the referenced string data must be encoded in UTF-16BE.

Windows language IDs

The following are platform-specific language IDs assigned by Microsoft:

Primary Language	Region	Language ID (hexadecimal)
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 Monaco	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
Odia (formerly 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	0488
Yakut	Russia	0485
Yi	PRC	0478
Yoruba	Nigeria	046A

Platform-specific encoding and language IDs: Macintosh platform (platform ID = 1)

Macintosh encoding IDs (script manager codes)

The following encoding IDs are defined for use with the Macintosh platform:

Encoding ID	Script	Encoding ID	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

The following are platform-specific language ID's assigned by Apple:

Language ID	Language	Language ID	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	143	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)

Platform-specific encoding and language IDs: ISO platform (platform ID=2) [Deprecated]

The following encoding IDs are defined for use with ISO platform:

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 could potentially be used for encodings in the 'cmap' table but not for strings in the 'name' table. Note that use of the ISO platform in the 'cmap' table is deprecated.

Platform-specific encoding and language IDs: Custom platform (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.

5.2.7.4 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. This family name is assumed to be shared among fonts that differ only in weight or style (regular, italic, bold, bold). Font Family name is used in combination with Font Subfamily name (name ID 2). Some applications that use this pair of names assume that a Font Family name is shared by at most four fonts that form a font style-linking group: regular, italic, bold, and bold italic. To be compatible with the broadest range of platforms and applications, fonts should limit use of any given Font Family name in this manner. (This four-way distinction should also be reflected in OS/2.fsSelection bit settings.) For fonts within an extended typographic family that fall outside this four-way distinction, the distinguishing attributes should be reflected in the Font Family name so that those fonts appear as a separate font family. For example, the Font Family name for the Arial Narrow font is “Arial Narrow”; the Font Family name for the Arial Black font is “Arial Black”. (Note that, in such cases, name ID 16 should also be included with a shared name that reflects the full, typographic family.)
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 variants only. A font with no distinctive weight or style (e.g. medium weight, not italic and OS/2.fsSelection bit 6 set) should have the string "Regular" as the Font Subfamily name (for English language). Font Subfamily name is used in combination with Font Family name (name ID 1). Some applications that use this pair of names assume that a Font Family name is shared by at most four fonts that form a font style-linking group. These four fonts may have Subfamily name values that reflect various weights or styles, with four-way “Bold” and “Italic” style-linking relationships indicated using OS/2.fsSelection bits 0, 5 and 6. Within an extended typographic family that includes fonts beyond regular, bold, italic, or bold italic, distinctions are made in the Font Family name, so that fonts appear to be in separate families. In some cases, this may lead to specifying a Subfamily name of “Regular” for a font that might not otherwise be considered a regular font. For example, the Arial Black font has a Font Family name of “Arial Black” and a Subfamily name of “Regular”. (Note that, in such cases, name IDs 16 and 17 should also be included, using a shared value for name ID 16 that reflects the full typographic family, and values for name ID 17 that appropriately reflect the actual design variant of each font.)

3	Unique font identifier
4	Full font name that reflects all family and relevant subfamily descriptors. The full font name is generally a combination of name IDs 1 and 2, or of name IDs 16 and 17, or a similar human-readable variant. For fonts in extended typographic families (that is, families that include more than regular, italic, bold, and bold italic variants), values for name IDs 1 and 2 are normally chosen to provide compatibility with certain applications that assume a family has at most four style-linked fonts. In that case, some fonts may end up with a Subfamily name (name ID 2) of “Regular” even though the font would not be considered, typographically, a regular font. For such non-regular fonts in which name ID 2 is specified as “Regular”, the “Regular” descriptor would generally be omitted from name ID 4. For example, the Arial Black font has a Font Family name (name ID 1) of “Arial Black” and a Subfamily name (name ID 2) of “Regular”, but has a full font name (name ID 4) of “Arial Black”. Note that name IDs 16 and 17 should also be included in these fonts, and that name ID 4 would typically be a combination of name IDs 16 and 17, without needing any additional qualifications regarding “Regular”.
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. When translated to ASCII, the name string must be no longer than 63 characters and restricted to the printable ASCII subset, codes 33-126, except for the 10 characters '[', ']', '(', ')', '{', '}', '<', '>', '/', '%'. In a CFF OT font, there is no requirement that this name be the same as the font name in the CFF’s Name INDEX. Thus, the same CFF may be shared among multiple font components in an OFF Font Collection. See subclause 8.20 for additional information.
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	Typographic Family name: The typographic family grouping doesn't impose any constraints on the number of faces within it, in contrast with the 4-style family grouping (ID 1), which is present both for historical reasons and to express style linking groups. If name ID 16 is absent, then name ID 1 is considered to be the typographic family name. (In earlier versions of the specification, name ID 16 was known as "Preferred Family".)
17	Typographic Subfamily name: This allows font designers to specify a subfamily name within the typographic family grouping. This string must be unique within a particular typographic family. If it is absent, then name ID 2 is considered to be the typographic subfamily name. (In earlier versions of the specification, name ID 17 was known as "Preferred Subfamily".)
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. When translated to ASCII, this name string must be restricted to the printable ASCII subset, codes 33 through 126, except for the 10 characters: '[', ']', '(', ')', '{', '}', '<', '>', '/', '%'. See subclause 8.20 in "Recommendations for OFF fonts" for additional information.
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.)
23	Light Background Palette. This ID, if used in the CPAL table’s Palette Labels Array, specifies that the corresponding color palette in the CPAL table is appropriate to use with the font when displaying it on a light background such as white. Name table strings for this ID specify the user interface strings associated with this palette.
24	Dark Background Palette. This ID, if used in the CPAL table’s Palette Labels Array, specifies that the corresponding color palette in the CPAL table is appropriate to use with the font when displaying it on a dark background such as black. Name table strings for this ID specify the user interface strings associated with this palette.

25	Variations PostScript Name Prefix. If present in a variable font, it may be used as the family prefix in the PostScript Name Generation for Variation Fonts algorithm. The character set is restricted to ASCII-range uppercase Latin letters, lowercase Latin letters, and digits. All name strings for name ID 25 within a font, when converted to ASCII, must be identical. See Reference [27] for reasons to include name ID 25 in a font, and for examples. For general information on OFF Font variations, see subclause 7.1.

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

The key information for this table for Microsoft fonts relates to the use of name IDs 1, 2, 4, 16 and 17. Note that some newer applications will use name IDs 16 and 17, while some legacy applications require name IDs 1 and 2 and also assume certain limitations on these values (see descriptions of name IDs 1 and 2 above). Fonts should include all of these strings for the broadest application compatibility. To better understand how to set values for these name IDs, 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 usWeightClass	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 naming table strings for the Windows platform (platform ID 3) must be encoded in UTF-16BE. Strings for the Macintosh platform (platform ID 1) use platform-specific single- or double-byte encodings.

Note that, for a typographic family that includes member faces that differ from Regular in relation to attributes other than weight, width or slope, there may 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

The following are examples of how these strings might be defined, based on Times New Roman Bold:

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, unique, human readable name of the font. This name is used by Windows. *Times New Roman Bold*

5. Release and version information from the font vendor. *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.

23. Light background palette name. No name string present, since this is not a color font.

24. Dark background palette name. No name string present, since this is not a color font.

25. Variations PostScript name prefix. No name string present, since this is not a variable font.

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

5.2.8 OS/2 – Global font information table

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

OS/2 Table formats

Six versions of the OS/2 table have been defined: versions 0 to 5. The format of version 5 is as follows:

Type	Name of Entry	Comments
uint16	Version	0x0000, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005
int16	xAvgCharWidth	
uint16	usWeightClass	
uint16	usWidthClass	
uint16	fsType	
int16	ySubscriptXSize	
int16	ySubscriptYSize	
int16	ySubscriptXOffset	
int16	ySubscriptYOffset	
int16	ySuperscriptXSize	
int16	ySuperscriptYSize	
int16	ySuperscriptXOffset	
int16	ySuperscriptYOffset	
int16	yStrikeoutSize	
int16	yStrikeoutPosition	
int16	sFamilyClass	
uint8	Panose[10]	
uint32	ulUnicodeRange1	Bits 0-31
uint32	ulUnicodeRange2	Bits 32-63 version 0x0001 and later
uint32	ulUnicodeRange3	Bits 64-95 version 0x0001 and later
uint32	ulUnicodeRange4	Bits 96-127 version 0x0001 and later
Tag	achVendID[4]	
uint16	fsSelection	
uint16	usFirstCharIndex	
uint16	usLastCharIndex	
int16	sTypoAscender	
int16	sTypoDescender	
int16	sTypoLineGap	
uint16	usWinAscent	
uint16	usWinDescent	
uint32	ulCodePageRange1	Bits 0-31 version 0x0001 and later
uint32	ulCodePageRange2	Bits 32-63 version 0x0001 and later
int16	sxHeight	version 0x0002 and later
int16	sCapHeight	version 0x0002 and later
uint16	usDefaultChar	version 0x0002 and later
uint16	usBreakChar	version 0x0002 and later
uint16	usMaxContext	version 0x0002 and later

uint16	usLowerOpticalPointSize	version 0x0005 and later
uint16	usUpperOpticalPointSize	version 0x0005 and later

All versions are supported, but use of version 4 or later is recommended. For descriptions of older OS/2 table formats see Annex B. OS/2 field details are provided below.

5.2.8.1 version

Format: uint16

Units: n/a

Title: OS/2 table version number.

Description: The version number for this OS/2 table, = 5.

Comments: The version number allows for identification of the precise contents and layout for the OS/2 table.

5.2.8.2 xAvgCharWidth

Format: int16

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 A.

5.2.8.3 usWeightClass

Format: uint16

Title: Weight class.

Description: Indicates the visual weight (degree of blackness or thickness of strokes) of the characters in the font. Values from 1 to 1000 are valid.

Comments: usWeightClass values use the same scale as the 'wght' axis that is used in the 'fvar' table of variable fonts and in the 'STAT' table. While integer values from 1 to 1000 are supported, some legacy platforms may have limitations on supported values. The following are commonly set values:

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

5.2.8.4 usWidthClass

Format: uint16

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.

The valid usWidthClass values are shown in the following table. Note that the usWidthClass values are related to but distinct from the scale for the 'wdth' axis that is used in the 'fvar' table of variable fonts and in the 'STAT' table. The "% of normal" column in the following table provides a mapping from usWidthClass values 1 – 9 to 'wdth' values.

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

5.2.8.5 fsType

Format: uint16

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	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.

5.2.8.6 ySubscriptXSize

Format: int16

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.

5.2.8.7 ySubscriptYSize

Format: int16

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.

5.2.8.8 ySubscriptXOffset

Format: int16

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.

5.2.8.9 ySubscriptYOffset

Format: int16

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.

5.2.8.10 ySuperscriptXSize

Format: int16

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.

5.2.8.11 ySuperscriptYSize

Format: int16

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.

5.2.8.12 ySuperscriptXOffset

Format: int16

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.

5.2.8.13 ySuperscriptYOffset

Format: int16

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.

5.2.8.14 yStrikeoutSize

Format: int16

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.

5.2.8.15 yStrikeoutPosition

Format: int16

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.

5.2.8.16 sFamilyClass

Format: int16

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 A. 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.

5.2.8.17 Panose

Format: uint8[10]

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'.

Comments: The PANOSE definition contains ten digits each of which currently describes up to sixteen variations. Windows uses bFamilyType, bSerifStyle and bProportion in the font mapper to determine family type. It also uses bProportion to determine if the font is monospaced. If the font is a symbol font, the first byte of the PANOSE number (bFamilyType) must be set to “pictorial”. The specification for assigning PANOSE values [14] can be found in bibliography.

Type	Name
uint8	bFamilyType
uint8	bSerifStyle
uint8	bWeight
uint8	bProportion
uint8	bContrast
uint8	bStrokeVariation
uint8	bArmStyle
uint8	bLetterform
uint8	bMidline
uint8	bXHeight

5.2.8.18 UnicodeRange

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

Format: uint32[4] - total 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 BMP) and platform 3, encoding ID 10 (Microsoft platform, Unicode full repertoire). If a bit is set (1) then the Unicode ranges assigned to that bit are 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.

NOTE Avalable bits were exhausted after Unicode 5.1. There are many additional ranges supported in the current version of Unicode that are not supported by this field in the OS/2 table. See the 'dlng' and 'slng' tags in the Metadata table for an alternate mechanism to declare what scripts or languages a font can support or is designed for.

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)	F0000-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.

5.2.8.19 achVendID

Format: 4-byte Tag

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 bibliolgraphy.

5.2.8.20 fsSelection

Format: uint16.

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)
10 – 15		<reserved>	Reserved; set to 0.

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 5 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.

Bit 15 is permanently reserved. It has been used in some legacy implementations and may result in special behavior in some implementations. Use of this bit is deprecated.

5.2.8.21 usFirstCharIndex

Format: uint16

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.

5.2.8.22 usLastCharIndex

Format: uint16

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.

5.2.8.23 sTypoAscender

Format: int16

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 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 in Clause 7 for more on this field.

5.2.8.24 sTypoDescender

Format: int16

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 in Clause 7 for more on this field.

5.2.8.25 sTypoLineGap

Format: int16

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 sTypoLineGap 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). These new metrics, when combined with the character design widths, will allow applications to lay out documents in a typographically correct and portable fashion.

5.2.8.26 usWinAscent

Format: uint16

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.

5.2.8.27 usWinDescent

Format: uint16

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.

5.2.8.28 CodePageRange

ulCodePageRange1 Bits 0-31
ulCodePageRange2 Bits 32-63

Format: uint32[2] - total 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, Unicode BMP). If the font file is encoding ID 0, then the Symbol Character Set bit should be set.

If a given 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
32-46		Reserved for OEM
47		Reserved
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

5.2.8.29 sxHeight

Format: int16

Description: This metric specifies the distance between the baseline and the approximate height of non-ascending lowercase letters measured in font design units. 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.

5.2.8.30 sCapHeight

Format: int16

Description: This metric specifies the distance between the baseline and the approximate height of uppercase letters measured in font design units. 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.

5.2.8.31 usDefaultChar

Format: uint16

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.

5.2.8.32 usBreakChar

Format: uint16

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.

5.2.8.33 usMaxContext

Format: uint16

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.

5.2.8.34 usLowerOpticalPointSize

Format: uint16

Units: TWIPs

Description: This field is used for fonts with multiple optical styles.

This value is the lower value of the size range for which this font has been designed. The units for this field are TWIPs (one-twentieth of a point, or 1440 per inch). The value is inclusive — meaning that that font was designed to work best at this point size through, but not including, the point size indicated by usUpperOpticalPointSize. When used with other optical fonts that set usLowerOpticalPointSize and usUpperOpticalPointSize, it would be expected that another font has this same value as this entry in the usUpperOpticalPointSize field, unless this font is designed for the lowest size range. The smallest font in an optical size set should set this value to 0. When working across multiple optical fonts, there should be no intentional gaps or overlaps in the ranges. usLowerOpticalPointSize must be less than usUpperOpticalPointSize. The maximum valid value is 0xFFFE.

For fonts that were not designed for multiple optical styles, this field should be set to 0 (zero) and the corresponding usUpperOpticalPointSize set to 0xFFFF.

NOTE Use of this field has been superseded by the 'STAT' table. See subclause 9.9 for more information.

5.2.8.35 usUpperOpticalPointSize

Format: uint16

Units: TWIPs

Description: This field is used for fonts with multiple optical styles.

This value is the upper value of the size range for which this font has been designed. The units for this field are TWIPs (one-twentieth of a point, or 1440 per inch). The value is exclusive — meaning that that font was designed to work best below this point size down to the usLowerOpticalPointSize threshold. When used with other optical fonts that set usLowerOpticalPointSize and usUpperOpticalPointSize, it would be expected that another font has this same value as this entry in the usLowerOpticalPointSize field, unless this font is designed for the highest size range. The largest font in an optical size set should set this value to 0xFFFF, which is interpreted as infinity. When working across multiple optical fonts, there should be no intentional or overlaps left in the ranges. usUpperOpticalPointSize must be greater than usLowerOpticalPointSize. The minimum valid value for this field is 2 (two). The largest possible inclusive point size represented by this field is 3276.65 points, any higher values would be represented as infinity.

For fonts that were not designed for multiple optical styles, this field should be set to 0xFFFF and the corresponding usLowerOpticalPointSize set to 0 (zero).

NOTE Use of this field has been superseded by the 'STAT' table. See subclause 9.9 for more information.

OS/2 Table and Font Variations

In variable fonts, default line metrics should always be set using the sTypoAscender, sTypoDescender and sTypoLinGap values, and the USE_TYPO_METRICS flag in the fsSelection field should be set. The ascender, descender and lineGap fields in the 'hhea' table should be set to the same values as sTypoAscender, sTypoDescender and sTypoLinGap. The usWinAscent and usWinDescent fields should be used to specify a recommended clipping rectangle.

In a variable font, various font-metric values within the OS/2 table may need to be adjusted for different variation instances. Variation data for OS/2 entries can be provided in the metrics variations ('MVAR') table. Different OS/2 entries are associated with particular variation data in the 'MVAR' table using value tags, as follows:

OS/2 entry	Tag
sCapHeight	'cpht'
sTypoAscender	'hasc'
sTypoDescender	'hdsc'
sTypoLineGap	'hlgp'
sxHeight	'xhgt'
usWinAscent	'hcla'
usWinDescent	'hcld'
yStrikeoutPosition	'stro'
yStrikeoutSize	'strs'
ySubscriptXOffset	'sbxo'
ySubScriptXSize	'sbxs'
ySubscriptYOffset	'sbyo'
ySubscriptYSize	'sbys'
ySuperscriptXOffset	'spxo'
ySuperscriptXSize	'spxs'
ySuperscriptYOffset	'spyo'
ySuperscriptYSize	'spys'

NOTE 1 The usWeightClass and usWidthClass values are not adjusted by variation data since these correspond to 'wght' and 'wdth' variation axes that can be used to define a font's variation space. Appropriate usWeightClass and usWidthClass values for a variation instance can be derived from 'wght' and 'wdth' user coordinates that are used to select a particular variation instance. For 'wdth' values greater than 200, the usWidthClass value is clamped to 9. See the discussion of the 'wght' and 'wdth' axes in the "Design-variation axis tags registry" for details on the relationship between these OS/2 fields and the corresponding design axes.

NOTE 2 The usLowerOpticalPointSize and usUpperOpticalPointSize values are not adjusted by variation data. These values (now superseded by the 'STAT' table) are used to indicate a range of sizes for which a given font has been designed. It is assumed that variation that targets different sizes will implemented using the 'opsz' variation axis. If a variable font supports 'opsz' as an axis of variation, then the usLowerOpticalPointSize and usUpperOpticalPointSize fields can be set to the same values as the minValue and maxValue fields for the 'opsz' axis in the 'fvar' table.

To have variable line metrics in a variable font, the 'hasc', 'hdsc' and 'hlgp' value tags should be used in the 'MVAR' table to vary the ascender, descender and line gap values from defaults specified in the sTypoAscender, sTypoDescender and sTypoLinGap fields. The 'hcla' and 'hcld' value tags can be used in addition to vary the size of a clipping region from the default specified in the winAscent and winDescent fields. Other metrics can be varied using value tags listed above.

For general information on OFF Font variations, see the subclause 7.1.

5.2.9 Font class parameters

NOTE See Annex A for details.

5.2.10 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 Reference [3].

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.
uint32	isFixedPitch	Set to 0 if the font is proportionally spaced, non-zero if the font is not proportionally spaced (i.e. monospaced).
uint32	minMemType42	Minimum memory usage when an OFF font is downloaded.
uint32	maxMemType42	Maximum memory usage when an OFF font is downloaded.
uint32	minMemType1	Minimum memory usage when an OFF font is downloaded as a Type 1 font.
uint32	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 Reference [7].

5.2.10.1 Version 1.0

This version is used in order to supply PostScript glyph names when the font file contains exactly the 258 glyphs in the standard Macintosh TrueType font file (see 'post' Format 1 in Reference [2] for a list of the 258 Macintosh glyph names), and the font does not otherwise supply glyph names. As a result, the glyph names are taken from the system with no storage required by the font.

5.2.10.2 Version 2.0

This is the version required in order to supply PostScript glyph names for fonts which do not supply them elsewhere. A version 2.0 'post' table can be used in fonts with TrueType or CFF version 2 outlines.

Type	Name	Description
uint16	numGlyphs	Number of glyphs (this should be the same as numGlyphs in 'maxp' table).
uint16	glyphNameIndex[numGlyphs].	This is not an offset, but is the ordinal number of the glyph in 'post' string tables.
int8	names[numberNewGlyphs]	Glyph names with length bytes [variable] (a Pascal string).

This font file contains glyphs not in the standard Macintosh set or the ordering of the glyphs in the font file differs from the standard Macintosh set. 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 65535, 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.

If you do not want to associate a PostScript name with a particular glyph, use index number 0 which points the name .notdef.

5.2.10.3 Version 2.5

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

5.2.10.4 Version 3.0

This version makes it possible to create a font that is not burdened with a large 'post' table set of glyph names. A version 3.0 'post' table can be used by OFF fonts with TrueType or CFF (version 1 or 2) data.

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.*

5.2.10.5 'post' Table and OFF Font Variations

In a variable font, various font-metric values within the 'post' table may need to be adjusted for different variation instances. Variation data for 'post' entries can be provided in the table in subclause 7.3.6. Different 'post' entries are associated with particular variation data in the 'MVAR' table using value tags, as follows:

'post' entry	Tag
underlinePosition	'undo'
underlineThickness	'unds'

NOTE The italicAngle value is not adjusted by variation data since this corresponds to the 'slnt' variation axis that can be used to define a font's variation space. Appropriate post.italicAngle values for a variation instance can be derived from the 'slnt' user coordinates that are used to select a particular variation instance. See the discussion of the 'slnt' axis in subclause 7.2.4 for details on the relationship between italicAngle and the 'slnt' axis.

For general information on OFF Font Variations, see subclause 7.1.

5.3 Tables related to TrueType outlines

5.3.1 List of TrueType outlines tables

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

Tag	Name
cvt	Control Value Table (optional table)
fpgm	Font program (optional table)
glyf	Glyph data
loca	Index to location
prep	CV Program (optional table)
gasp	Grid-fitting/Scan-conversion (optional table)

5.3.2 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.

5.3.3 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
uint8[*n*]	Instructions. *n* is the number of uint8 items that fit in the size of the table.

5.3.4 glyf – Glyf data

5.3.4.1 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.

Table organization

The 'glyf' table is comprised of a list of glyph data blocks, each of which provides the description for a single glyph. Glyphs are referenced by identifiers (glyph IDs), which are sequential integers beginning at zero. The total number of glyphs is specified by the numGlyphs field in the 'maxp' table. The 'glyf' table does not include any overall table header or records providing offsets to glyph data blocks. Rather, the 'loca' table provides an array of offsets, indexed by glyph IDs, which provide the location of each glyph data block within the 'glyf' table. Note that the 'glyf' table must always be used in conjunction with the 'loca' and 'maxp' tables. The size of each glyph data block is inferred from the difference between two consecutive offsets in the 'loca' table (with one extra offset provided to give the size of the last glyph data block). As a result of the 'loca' format, glyph data blocks within the 'glyf' table must be in glyph ID order.

Glyph headers

Each glyph begins with the following header.

Glyph Header

Type	Name	Description
int16	numberOfContours	If the number of contours is greater than or equal to zero, this is a simple glyph; if negative, this is a composite glyph – the value -1 should be used for composite glyphs.
int16	xMin	Minimum x for coordinate data.
int16	yMin	Minimum y for coordinate data.
int16	xMax	Maximum x for coordinate data.
int16	yMax	Maximum y for coordinate data.

NOTE 1 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.

NOTE 2 The glyph descriptions do not include side bearing information. Left side bearings are provided in the 'hmtx' table, and right side bearings are inferred from the advance width (also provided in the 'hmtx' table) and the bounding box coordinates provided in the 'glyf' table. For vertical layout, top side bearings are provided in the 'vmtx' table, and bottom side bearings are inferred. The rasterizer will generate a representation of side bearings in the form of “phantom” points, which are added as four additional points at the end of the glyph description and which can be referenced and manipulated by glyph instructions. See the chapter "Instructing TrueType Glyphs", as specified by the "TrueType Instruction Set", for more background on phantom points.

5.3.4.1.1 Simple glyph description

This is the table information needed if numberOfContours is greater than zero, that is, a glyph is not a composite. Note that point numbers are base-zero indices that are numbered sequentially across all of the contours for a glyph; that is, the first point number of each contour (except the first) is one greater than the last point number of the preceding contour.

Simple Glyph table

Type	Name	Description
uint16	endPtsOfContours[numberOfContours]	Array of point indices for the last point of each contour; in increasing numeric order.
uint16	instructionLength	Total number of bytes for instructions. If instructionLength is zero, no instructions are present for this glyph, and this field is followed directly by the flags field.
uint8	instructions[instructionLength]	Array of instructions for each glyph; shall be as specified in the TrueType Instruction Set.
uint8	flags[*variable*]	Array of flag elements. See below for details regarding the number of flag array elements.
uint8 or int16	xCoordinates[*variable*]	Contour point x-coordinates. See below for details regarding the number of coordinate array elements. Coordinate for the first point is relative to (0,0); others are relative to previous point.
uint8 or int16	yCoordinates[*variable*]	Contour point y-coordinates. See below for details regarding the number of coordinate array elements. Coordinate for the first point is 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. Coordinate values are in font design units, as defined by the unitsPerEm field in the 'head' table. Note that smaller unitsPerEm values will make it more likely that delta-x and delta-y values can fit in a smaller representation (8-bit rather than 16-bit), though with a trade-off in the level or precision that can be used for describing an outline.

Each element in the flags array is a single byte, each of which has multiple flag bits with distinct meanings, as shown below.

In logical terms, there is one flag byte element, one x-coordinate, and one y-coordinate for each point. Note, however, that the flag byte elements and the coordinate arrays use packed representations. In particular, if a logical sequence of flag elements or sequence of x- or y-coordinates is repeated, then the actual flag byte element or coordinate value can be given in a single entry, with special flags used to indicate that this value is repeated for subsequent logical entries. The actual stored size of the flags or coordinate arrays must be determined by parsing the flags array entries. See the flag descriptions below for details.

Simple Glyph flags

Mask	Name	Description
0x01	ON_CURVE_POINT	Bit 0: If set, the point is on the curve; otherwise, it is off the curve.
0x02	X_SHORT_VECTOR	Bit 1: If set, the corresponding x-coordinate is 1 byte long. If not set, it is two bytes long. For the sign of this value, see the description of the X_IS_SAME_OR_POSITIVE_X_SHORT_VECTOR flag.
0x04	Y_SHORT_VECTOR	Bit 2: If set, the corresponding y-coordinate is 1 byte long. If not set, it is two bytes long. For the sign of this value, see the description of the Y_IS_SAME_OR_POSITIVE_Y_SHORT_VECTOR flag.

0x08	REPEAT_FLAG	Bit 3: If set, the next byte (read as unsigned) specifies the number of additional times this flag byte is to be repeated in the logical flags array – that is, the number of additional logical flag entries inserted after this entry. (In the expanded logical array this bit is ignored.) In this way, the number of flags listed can be smaller than the number of points in the glyph description.
0x10	X_IS_SAME_OR_POSITIVE_X_SHORT_VECTOR	Bit 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 X_SHORT_VECTORt is not set and this bit is set, then the current x-coordinate is the same as the previous x-coordinate. If X_SHORT_VECTOR is not set and this bit is also not set, the current x-coordinate is a signed 16-bit delta vector.
0x20	Y_IS_SAME_OR_POSITIVE_Y_SHORT_VECTOR	Bit 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 Y_SHORT_VECTOR is not set and this bit is set, then the current y-coordinate is the same as the previous y-coordinate. If Y_SHORT_VECTOR is not set and this bit is also not set, the current y-coordinate is a signed 16-bit delta vector.
0x40	OVERLAP_SIMPLE	Bit 6: If set, contours in the glyph description may overlap. Use of this flag is not required in OFF – that is, it is valid to have contours overlap without having this flag set. It may affect behaviors in some platforms, however. (See Apple's TrueType Reference Manual [7] for details regarding behavior in Apple platforms.) When used, it must be set on the first flag byte for the glyph. See additional details below.
0x80	Reserved	Bits 7 is reserved:set to zero.

A non-zero-fill algorithm is needed to avoid dropouts when contours overlap. This can be particularly relevant for variable fonts, which often make use of overlapping contours. The OVERLAP_SIMPLE flag is used by some rasterizer implementations to ensure that a non-zero-fill algorithm is used rather than an even-odd-fill algorithm. Note that some implementations might use this flag specifically in non-variable fonts, but not in variable fonts. Implementations that always use a non-zero-fill algorithm will ignore this flag. This flag can be used in order to provide broad interoperability of fonts — particularly non-variable fonts — when glyphs have overlapping contours. Tools that generate static-font data for a specific instance of a variable font should either set this flag when contours in the derived glyph data are overlapping, or else should merge contours to remove overlap of separate contours.

NOTE The OVERLAP_COMPOUND flag, described below, has a similar purpose in relation to composite glyphs. The same considerations described for the OVERLAP_SIMPLE flag also apply to the OVERLAP_COMPOUND flag.

5.3.4.1.2 Composite glyph description

If numberOfContours is negative, a composite glyph description is used.

A composite glyph starts with two uint16 values ("flags" and "glyphIndex", i.e. the index of the first contour in this composite glyph); the data then varies according to "flags").

Composite Glyph table

Type	Name	Description
uint16	Flags	component flag
uint16	glyphIndex	glyph index of component
uint8, int8, uint16 or int16	Argument1	x-Offset for component or point number; type depends on bits 0 and 1 in component flags
uint8, int8, uint16 or int16	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 {
    uint16 flags;
    uint16 glyphIndex;
    if ( flags & ARG_1_AND_2_ARE_WORDS) {
            (int16 or FWord) argument1;
            (int16 or FWord) argument2;
    } else {
            uint16 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){
    uint16 numInstr
    uint8 instr[numInstr]
```

Argument1 and argument2 can be either x and y offsets to be added to the glyph (the ARGS_ARE_XY_VALUES flag is set), or two point numbers (the ARGS_ARE_XY_VALUES flag is not set). 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 font design units.

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 linear transformation of the X and Y coordinates by specifying a 2 × 2 matrix. This could be used for 90-degree rotations of the glyph components, for example.

The following composite glyph flags are defined:

Composite Glyph flags

Mask	Flags	Description
0x0001	ARG_1_AND_2_ARE_WORDS	Bit 0: If this is set, the arguments are words; otherwise, they are bytes.
0x0002	ARGS_ARE_XY_VALUES	Bit 1: If this is set, the arguments are xy values; otherwise, they are points.
0x0004	ROUND_XY_TO_GRID	Bit 2: For the xy values if the preceding is true.
0x0008	WE_HAVE_A_SCALE	Bit 3: This indicates that there is a simple scale for the component. Otherwise, scale = 1.0.
0x0020	MORE_COMPONENTS	Bit 5: Indicates at least one more glyph after this one.
0x0040	WE_HAVE_AN_X_AND_Y_SCALE	Bit 6: The x direction will use a different scale from the y direction.
0x0080	WE_HAVE_A_TWO_BY_TWO	Bit 7: The bit WE_HAVE_A_TWO_BY_TWO allows for linear transformation of the X and Y coordinates by specifying a 2 × 2 matrix. This could be used for scaling and 90° rotations of the glyph components, for example.
0x0100	WE_HAVE_INSTRUCTIONS	Bit 8: Following the last component are instructions for the composite character.
0x0200	USE_MY_METRICS	Bit 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.
0x0400	OVERLAP_COMPOUND	Bit 10: If set, the components of this compound glyph overlap. Use of this flag is not required in OFF — that is, it is valid to have components overlap without having this flag set. It may affect behaviors in some platforms, however. (See Apple's TrueType Reference Manual [7] for details regarding behavior in Apple platforms.) When used, it must be set on the flag word for the first component. See additional remarks, above, for the similar OVERLAP_SIMPLE flag used in simple-glyph descriptions.
0x0800	SCALED_COMPONENT_OFFSET	Bit 11: Composite designed to have the component offset scaled.
0x1000	UNSCALED_COMPONENT_OFFSET	Bit 12: Composite designed not to have the component offset scaled.
0xE010	RESERVED	Bits 4, 13, 14 and 15 are reserved: set to 0.

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.

The SCALED_COMPONENT_OFFSET and UNSCALED_COMPONENT_OFFSET flags are used to determine how x and y offset values are to be interpreted when the component glyph is scaled. If the SCALED_COMPONENT_OFFSET flag is set, then the x and y offset values are deemed to be in the component glyph's coordinate system, and the scale transformation is applied to both values. If the UNSCALED_COMPONENT_OFFSET flag is set, then the x and y offset values are deemed to be in the current glyph's coordinate system, and the scale transformation is not applied to either value. If neither flag is set, then the rasterizer will apply a default behavior. On Microsoft and Apple platforms, the default behavior is the same as when the UNSCALED_COMPONENT_OFFSET flag is set; this behavior is recommended for all rasterizer implementations. If a font has both flags set, this is invalid; the rasterizer should use its default behavior for this case.

5.3.5 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 shall 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
Offset16	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
Offset32	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.

5.3.6 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
uint8[*n*]	Set of instructions executed whenever point size or font or transformation change. *n* is the number of uint8 items that fit in the size of the table.

5.3.7 gasp – Grid-fitting and scan-conversion procedure table

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' Header

Type	Name	Description
uint16	version	Version number (set to 0 or 1)
uint16	numRanges	Number of records to follow
GaspRange	gaspRanges[numRanges]	Sorted by ppem

The array of GaspRange records provides recommended behaviors for various ppem sizes:

GaspRange Record

Type	Name	Description
uint16	rangeMaxPPEM	Upper limit of range, in PPEM
uint16	rangeGaspBehavior	Flags describing desired rasterizer behavior.

There are fourRangeGaspBehavior flags defined.

RangeGaspBehavior flags

Mask	Name	Description
0x0001	GASP_GRIDFIT	Use gridfitting
0x0002	GASP_DOGRAY	Use grayscale rendering
0x0004	GASP_SYMMETRIC_GRIDFIT	Use gridfitting with ClearType symmetric smoothing Only supported in version 1 of 'gasp' table
0x0008	GASP_SYMMETRIC_ SMOOTHING	Use smoothing along multiple axes with ClearType® Only supported in version 1 of 'gasp' table
0xFFF0	Reserved	Reserved flags – set to 0.

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.

'gasp' Table and OFF Font Variations

In a variable font, the threshold sizes at which rasterizer behaviors are changed may need to be adjusted for different variation instances. Variation data for adjusting the rangeMaxPPEM value of up to ten GaspRange records can be provided in the metrics variations ('MVAR') table, referenced using value tags 'gsp0' to 'gsp9'. Note that the rasterizer behavior for a given GaspRange record cannot be changed for different variation instances; only the rangeMaxPPEM value can be adjusted.

The last GASPRANGE record in a ‘gasp’ table is assumed to have a rangeMaxPPEM value of 0xFFFF (effectively infinity). The rangeMaxPPEM value of the last record is never adjusted for different instances; the number of value records in the 'MVAR' table that are associated with 'gasp' entries must never be more than numRanges minus one.

For general information on OFF Font Variations, see subclause 7.1.

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

5.4 Tables related to CFF outlines

5.4.1 List of CFF outline tables

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

Tag	Name
CFF	Compact Font Format 1.0
CFF2	Compact Font Format 2.0
VORG	Vertical Origin (optional table)

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

5.4.2 CFF – Compact Font Format (version 1) table

This table contains a Compact Font Format version 1 representation (also known as 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].

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

The Name INDEX in the CFF must contain only one entry; that is, there must be only one font in the CFF FontSet. It is not a requirement that this name be the same as Name ID 6 in the OFF font's 'name' table. Note that, in an OFF Font Collection file, a single CFF table can be shared accross multiple fonts; names used by applications must be those provided in the 'name' table, not the Name INDEX entry. The CFF Top DICT must specify a CharstringType value of 2.

The numGlyphs field in the 'maxp' table must be the same as the number of entries in the CFF's CharStrings INDEX. The OFF glyph index is the same as the CFF glyph index for all glyphs in the font.

5.4.3 CFF2 – Compact Font Format (version 2) table

5.4.3.1 Overview

This document describes the CFF2 format. Like the CFF version 1 format, CFF2 allows efficient storage of glyph outlines and metadata. The CFF2 format differs from CFF version 1 in that it cannot be used as a stand-alone font program: it is intended for use only in the context of an OFF font as an SFNT table with the tag 'CFF2', and depends on data in other OFF tables. All the data from the version 1 format that is duplicated by data in other tables, or which is not used in the context of an OFF font, is removed.

Another important difference is that the CFF2 format adds new operators that allow CFF2 to represent the data for a variable font: a font that includes representations for several different variants of each glyph, which can be blended to produce an intermediate instance. See subclause 7.1, for a general description of variable fonts and for a complete list of the tables required to support a variable font.

Finally, the CFF2 format requires the use of CFF2 CharStrings rather than Type 2 CharStrings. This CharString format, like the CFF2 format itself, has removed operators to reduce file size, and added new operators to support variable fonts. See subclause 5.4.2.8 for additional information on CFF2 CharStrings.

For a complete description of the differences between CFF format version 1 and CFF2, see "CFF2 changes From CFF 1.0" [28].

5.4.3.2 Data layout

Conceptually, the binary data is organized as a number of separate data structures. The overall layout within the binary data is shown in following table. The first three structures occupy fixed locations. The remainder are reached via offsets, and their ordering can be changed.

CFF Data layout

Entry	Comments
Header	Fixed location
Top DICT	Fixed location
Global Subr INDEX	Fixed location
VariationStore	-
FDSelect	Present only if there is more than one Font DICT in the Font DICT INDEX.
Font DICT INDEX	-
Array of Font DICT	Included in Font DICT INDEX
Private DICT	one per Font DICT

An annotated example of a CFF2 table can be found in the Example CFF2 Font [29].

5.4.3.3 Data types

This subclause describes data representation and types used by the CFF2 format.

All multi-byte numeric data and offset fields are stored in big-endian byte order (high byte low offset) and do not honor any alignment restrictions. This leads to a format that is free from padding bytes.

Data objects are often specified by byte offsets that are relative to some reference point within the CFF2 data. These offsets are 1 to 4 bytes in length. The reference position for the offset is indicated in each case.

The data types used in the CFF2 table are shown in the following table:

CFF Data Types

Name	Range	Description
int8	0 to 255	1-byte unsigned number
uint16	0 to 65535	2-byte unsigned number
uint32	0 to 4294967296	4-byte unsigned number
Offset	varies	1, 2, 3, or 4 byte offsets (specified by OffSize field in an Index table)
OffSize	1 to 4	1-byte unsigned number specifies the size of an Offset field or fields

This document describes data structures by listing field types, names, and descriptions. Data structures may be given a type name and subsequently described. Arrays of objects are indicated by the usual square bracket convention enclosing the array length.

The majority of CFF2 data is contained by either of two data structures called DICT and INDEX which are described below.

5.4.3.4 DICT data

Font dictionary data comprising key-value pairs is represented in a compact tokenized format that is similar to that used to represent CharStrings. Dictionary keys are encoded as 1- or 2-byte operators and dictionary values are encoded as variable-size numeric operands. An operator is preceded by the operand(s) that specify its value. A DICT is simply a sequence of operand(s) / operator bytes concatenated together. There are three structures that use the DICT Data format: Top DICT, Font DICT and Private DICT. A list of DICT operators for each of these may be found in subclauses 5.4.2.7, 5.4.2.10, and 5.4.2.11. A summary of all DICT operators is provided in Annex E.

A number of integer operand types of varying sizes are defined and are encoded as shown in the table below (first byte of operand is b0, second is b1, and so on).

Operand Encoding

Size	b0 range	Value range	Value calculation
1	32 to 246	-107 to +107	b0 - 139
2	247 to 250	+108 to +1131	(b0 - 247) * 256 + b1 + 108
2	251 to 254	-1131 to -108	-(b0 - 251) * 256 - b1 - 108
3	28	-32768 to +32767	b1<<8 | b2
5	29	-(2^31) to +(2^31-1)	b1<<24 | b2<<16 | b3<<8 | b4

NOTE The 1- , 2-, and 3- byte integer formats are identical to those used by Type 2 CharStrings.

Examples of the integer formats are shown in the table below:

Integer Format Examples

Value	Encoding
0	8b
100	ef
-100	27
1000	fa 7c
10000	1c 27 10
-10000	1c d8 f0
100000	1d 00 01 86 a0
-100000	1d ff fe 79 60

A real number operand is provided in addition to integer operands. This operand begins with a byte value of 30 followed by a variable-length sequence of bytes. Each byte is composed of two 4-bit nibbles as defined in the table below. The first nibble of a pair is stored in the most significant 4 bits of a byte and the second nibble of a pair is stored in the least significant 4 bits of a byte.

Nibble Definitions

Nibble Value	Represents
0 to 9	0 to 9
a	. (decimal point)
b	E
c	E–
d	<reserved>
e	– (minus)
f	end of number

A real number is terminated by one (or two) 0xf nibbles so that it is always padded to a full byte. Thus, the value –2.25 is encoded by the byte sequence (1e e2 a2 5f) and the value 0.140541E–3 by the sequence (1e 0a 14 05 41 c3 ff).

Operators and operands may be distinguished by inspection of their first byte. Values 28, 29, 30, and 32 to 254 specify operands (numbers). All other values either specify an operator or are reserved. The maximum number of operands which may precede an operator is set by the current stack limit.

An operator may have one or more operands of the types shown in the table below:

Operand Types

Type	Description
number	Integer or real number
array	One or more numbers
delta	A number or a delta-encoded array of numbers (see below)

The length of array or delta types is determined by counting the operands preceding the operator. The second and subsequent numbers in a delta are encoded as the difference between successive values. For example, an array a_0, a_1, ..., a_n would be encoded as: a_0 (a_1-a_0) (a_2-a_1) ..., (a_n-a_{n-1}).

Two-byte operators have an initial escape byte of 12.

Further compaction of dictionary data is achieved by establishing default values for various DICT keys. For those keys that have a default value the absence of the corresponding operator in a DICT implies a key should take its default value.

5.4.3.5 INDEX data

An INDEX is an array of variable-sized objects. It comprises a header, an offset array, and object data. The offset array specifies offsets within the object data. An object is retrieved by indexing the offset array and fetching the object at the specified offset. The object's length can be determined by subtracting its offset from the next offset in the offset array. An additional offset is added at the end of the offset array so the length of the last object may be determined. The INDEX format is shown below.

INDEX Format

Type	Name	Description
uint32	count	Number of objects stored in INDEX
OffSize	offSize	Offset array element size
Offset	offset [count+1]	Offset array - offsets are from the start of the object data.
uint8	data [<varies>]	Object data

Offsets in the offset array are relative to the byte that precedes the object data. Therefore the first element of the offset array is always 1. (This ensures that every object has a corresponding offset which is always nonzero and permits the efficient implementation of dynamic object loading.)

An empty INDEX is represented by a count field with a 0 value and no additional fields. Thus, the total size of an empty INDEX is 4 bytes.

NOTE An INDEX may be skipped by jumping to the offset specified by the last element of the offset array.

5.4.3.6 Header

The binary data begins with a header having the format shown below:

Header Format

Type	Name	Description
uint8	majorVersion	Format major version. Set to 2.
uint8	minorVersion	Format minor version. Set to zero.
uint8	headerSize	Header size (bytes)
uint16	topDictLength	Length of Top DICT structure in bytes.

The headerSize field must be used when locating the start of the Top DICT data. It is provided so that future versions of the format may introduce additional data between the topDictLength field and the Top DICT data in a manner that is compatible with older implementations.

5.4.3.7 Top DICT data

This is the top-level DICT of the CFF2 table. The names of the Top DICT operators and default values (where applicable) are shown in the table below:

Top DICT Operator Entries

Name	Value	Operand(s)	Default	Notes
FontMatrix	12 7	array	0.001 0 0 0.001 0 0	
CharStrings	17	number	–	CharStrings INDEX offset, from start of the CFF2 table.(0)
FDArray	12 36	number	–	Font DICT (FD) INDEX offset, from start of the CFF2 table.
FDSelect	12 37	number	–	FDSelect structure offset, from start of the CFF2 table.
vstore	24	number	–	VariationStore structure offset, from start of the CFF2 table.

The Top DICT FontMatrix operator is required if the unitsPerEm value in the 'head' table is other than 1000. If unitsPerEm is 1000, then the FontMatrix operator may be omitted. When included, the FontMatrix operand array must be [1/unitsPerEm 0 0 1/unitsPerEm 0 0]. The default values shown above assume that unitsPerEm is 1000.

The FDSelect operator and the structure it points to are required if the Font DICT INDEX contains more than one Font DICT, else it must be omitted.

The vstore operator and the data it points to are required if variation data is present, and must be omitted if there is no variation data.

Operators in Top DICT, Font DICTs, Private DICTs and CharStrings may be preceded by up to a maximum of 513 operands.

5.4.3.8 CharStrings INDEX

The CharStrings INDEX is an INDEX structure that contains all of the CFF2 glyphs in the font. EachCharString provides a definition of a glyph and is accessed by glyph index ("GID"). The first CharString (GID 0) must be the .notdef glyph. The number of glyphs defined in the CFF2 table may be determined from the CharString INDEX count field. The value of this field shall match the value of the numGlyphs field in the 'maxp' table.

The format of the CharString data for CFF2 data, and therefore the method of interpretation, is the CFF2 CharString format. This is based on the Type 2 CharString format, and differs only in that some operators are added, and many are removed. See Annex D "The CFF2 CharString format" for details. The major changes are as follows:

- CFF2 CharStrings do not contain a value for advance width.
- For CFF2 tables, the fill rule for CharStrings must always be the nonzero winding number rule, rather than the even-odd rule. This is required in order to support variable font data, in which it is not practical to enforce removal of overlaps between paths.
- The stack depth is increased from 48 to 513.
- The CharString operator set is extended in CFF2 to include the **blend** (16) and **vsindex** (15) operators. These operators work as described below, in relation to equivalent CFF2 Private DICT operators, in subclause 5.4.2.12. Note, however, that the operator codes for these operators when used in CharStrings are different from the operator codes for the equivalent CFF2 Private DICT operators.
- The Type 2 operators **endchar** and **return** are removed.
- The Type 2 logic, storage, and math operators are removed.

The CFF2 format does not contain glyph names or CID values for glyph tags. Glyph that provide some semantic content can be useful for debugging, however, and can also be used as a last resort for deriving encoding information. Glyph tags for CFF2 tables can be represented PostScript glyph names in a version 2.0 'post' table. Glyph names add to the size of a font and are optional. Alternatively, the font can use a version 3.0 'post' table, which omits glyph names.

5.4.3.9 Local and Global Subr INDEXes

A subroutine ("subr") is typically a sequence of CharString bytes representing a sub-program that is used in more than one place in a font's CharString data. A subr may be stored once but referenced many times from within one or more CharStrings by the use of a call-subroutine operator that takes as an operand the number of the subr to be called.

Some subrs are *local*; that is, they are contained within a Private DICT and accessible by the set of CharStrings associated with the Private Dict. Local subrs are contained within an INDEX structure; the offset of the INDEX within the Private DICT is specified using the **Subrs** operator. A CharString references a local subr in its Private DICT by means of the **callsubr** operator.

Subrs can also be *global*, accessible to any CharString within the font. Global subrs are stored in the Global Subrs INDEX, which follows the Top DICT data. A font might not have any global subrs, in which case the Global Subrs INDEX is empty. A CharString references a global subr by means of the **callgsubr** operator.

Subr numbers are skewed by a number called the "subr number bias" that is calculated from the count of the subroutines in either the local or global subr INDEXes. The bias is calculated as follows:

```
uint16 bias;
uint16 nSubrs = subrINDEX.count;
if (nSubrs < 1240)
    bias = 107;
else if (nSubrs < 33900)
    bias = 1131;
else
    bias = 32768;
```

For correct subr selection the calculated bias must be added to the subr number operand before accessing the appropriate subr INDEX. This technique allows subr numbers to be specified using negative as well as positive numbers, thereby fully utilizing the available number ranges and thus saving space.

5.4.3.10 Font DICT INDEX, Font DICTs and FDSelect

The Font DICT INDEX contains one or more Font DICT structures. Unlike CID-keyed Type 1 fonts, the Font DICT INDEX may contain more than 256 Font DICTs.

A Font DICT is used for hinting, variation or subroutine (subr) data used by CharStrings. A font can have one Font DICT, which would apply to all CharStrings, or it can have multiple Font DICTs, each applicable to some set of CharStrings. The actual hinting or other data is contained in a Private DICT. Each Font DICT structure provides a reference to a Private DICT.

Font DICT Operator Entries

Name	Value	Operand(s)	Default	Notes
Private	18	number number	–	Private DICT size and offset, from start of the CFF2 table.

The use of a Font DICT FontMatrix operator is not required in CFF2 fonts, and is deprecated.

If there are multiple Font DICTs, an FDSelect table is used to provide information about which Font DICT ("FD") is used for which glyphs. An FDSelect is used only if there are multiple Font DICTs.

The location of the FDSelect table is given as the operand of the **FDSelect** operator in the Top DICT. An FDSelect table associates a Font DICT with a glyph by specifying an FD index for that glyph. The FD index is used to access one of the Font DICTs stored in the Font DICT INDEX. Three formats are currently defined, as shown in the following tables.

FDSelect Format 0

Type	Name	Description
uint8	format	Set to 0
uint8	fds [nGlyphs]	FD selector array

Each element of the fds array represents the FD index of a Font DICT in the FDArray. This format should be used when the FD indices are in a fairly random order. The number of glyphs (nGlyphs) is the value of the count field in the CharStrings INDEX.

FDSelect Format 3

Type	Name	Description
uint8	format	Set to 3
uint16	nRanges	Number of ranges
Range3	range3 [nRanges]	Array of Range3 records (see below)
uint16	sentinel	Sentinel GID

The format of a Range3 record is as follows:

Range3 Record Format

Type	Name	Description
uint16	first	First glyph index in range
uint8	fd	FD index for all glyphs in range

Each Range3 describes a group of sequential GIDs that have the same FD index. Each range includes GIDs from the first GID in the range record up to, but not including, the first GID of the next range record. Records in the Range3 array must be in increasing order of first GIDs. The first range must have a first GID of 0. A sentinel GID follows the last range element and serves to delimit the last range in the array. The sentinel GID is set equal to the number of glyphs in the font. That is, its value is 1 greater than the last GID in the font. This format is particularly suited to FD indexes that are well ordered (the usual case).

FDSelect Format 4

Type	Name	Description
uint8	format	Set to 4
uint32	nRanges	Number of ranges
Range4	range4 [nRanges]	Array of Range4 records (see below)
uint32	sentinel	Sentinel GID

Format 4 differs from Format 3 only in that it accommodates more than 65536 glyphs by using a uint32 type for the nRanges and sentinel fields, and a Range4 record array.

The format of a Range4 record is as follows:

Range4 Record Format

Type	Name	Description
uint32	first	First glyph index in range
uint16	fd	FD index for all glyphs in range

The Range4 format differs from the Range3 only in that it accommodates more than 65536 glyphs, by using a uint32 type for the first GID field and a uint16 field for the FD index.

NOTE While FDSelect format 4 allows for more than 65536 glyphs, other parts of the OFF format, such as the numGlyphs field of the 'maxp' table, are still constrained to 65536 glyphs.

5.4.3.11 Private DICT data

The names of the Private DICT operators (shown in table below) are, where possible, the same as the corresponding Type 1 dict keys. Operators that have no corresponding Type 1 dict key are indicated with a note.

Name	Value	Operand(s)	Default	Notes
BlueValues	6	delta	–	
OtherBlues	7	delta	–	
FamilyBlues	8	delta	–	
FamilyOtherBlues	9	delta	–	
BlueScale	12 9	number	0.039625	
BlueShift	12 10	number	7	
BlueFuzz	12 11	number	1	
StdHW	10	number	–	
StdVW	11	number	–	
StemSnapH	12 12	delta	–	
StemSnapV	12 13	delta	–	
LanguageGroup	12 17	number	0	
ExpansionFactor	12 18	number	0.06	
vsindex	22	number	0	itemVariationData index in the VariationStore structure.
Blend	23	delta, number of blends	–	Leaves 'number of blends' values on the operand stack.
Subrs	19	number	–	Offset to local subrs INDEX, from start of Private DICT.

The local subrs offset is relative to the beginning of the Private DICT data.

The OtherBlues and FamilyOtherBlues operators must occur after the BlueValues and FamilyBlues operators, respectively.

A Private DICT is required, but may be specified as having a size of 0 if there are no non-default values to be stored.

5.4.3.12 Extensions for font variations

In order to support glyph variation data in CFF2 tables, three new operators are added in CFF2 format: **vsindex**, **blend**, and **vstore**.

A variable font holds data representing the equivalent of several distinct design variations, and uses algorithms for interpolation — or *blending* — between these designs to derive a continuous range of design instances. This allows an entire family of fonts to be represented by a single variable font. For example, a variable font may contain data equivalent to Light and Heavy designs from a family, which can then be interpolated to derive instances for any weight in a continuous range between Light and Heavy.

For general background on OFF font variations, details on the tables used to support a variable font, terminology, and a specification of the interpolation algorithm used to blend values to derive specific design instances see subclause 7.1.

Outline data for a variable font in the CFF2 format are built much like a non-variable CFF2 table would be built, with exactly the same structure and operators as would be used for the default design representation. However, wherever a value occurs in the default design, the single value for the one design is supplemented with a set of delta values, followed by the **blend** operator. (For efficiency, a single **blend** operator may follow a series of such delta sets, rather than after each individual set.) Unlike other DICT operators, **blend** does not clear the stack when it is processed. The result of the **blend** operator remains on the stack to be processed by the following operator.

Within a variable font, different glyphs can use different sets of regions and associated delta values for the blending operation. When processing a given glyph, the interpreter must determine which set to use. These sets are stored in the CFF2 table in an *ItemVariationStore* structure. The ItemVariationStore contains one or more *ItemVariationData subtables*, each of which contains a list of Variation Regions. The first ItemVariationData subtable (index 0) is used by default, when no other subtable has been specified. When an ItemVariationData subtable other than the default is needed for a set of delta values, the **vsindex** operator is used. When this operator is used in a Private DICT to set a non-default itemVariationData index, this then becomes the default Item Variation Data index for not only the Private DICT, but also for all CharStrings that reference that Private DICT.When the **vsindex** operator is used in a CharString, it supersedes any **vsindex** from the private DICT. All private DICTs and CharStrings in a CFF2 table share the same ItemVariationStore.

5.4.3.12.1 Syntax for font variations support operators

vsindex |- ivs **vsindex** (22) |-

Selects the ItemVariationData subtable to be used for blending; the ivs argument is the ItemVariationData index. When used, **vsindex** must precede the **blend** operator.

Note that the operator code, 22, is different from the equivalent CharStrings operator. This operator may be used only in a Private DICT.

blend num(0)…num(n-1), delta(0,0)…delta(k-1,0), delta(0,1)…delta(k-1,1) … delta(0,n-1)…delta(k-1,n-1) n **blend** (23) val(0)…val(n-1)

For *k* regions, produces *n* interpolated result value(s) from *n*(k + 1)* operands. For more information and examples, see the description of the equivalent CharString operator in Annex D (D.4.5).

Note that the operator code, 23, is different from the equivalent CharStrings operator. This operator may only be used in a Private DICT.

vstore |- offset **vstore** (24) |-

Provides the offset to the VariationStore data in the CFF2 table. This operator may only be used in the Top DICT.

5.4.3.12.2 VariationStore data contents

The VariationStore data is comprised of two parts: a uint16 field that specifies a length, followed by an Item Variation Store structure of the specified length. The Item Variation Store format is specified in subclause 7.2. A brief description of the format as used within the CFF2 table follows.

To support variation of glyphs or other font data, the information used is comprised of default values for the particular data item, a set of delta adjustment values used to modify the default value, and a set of regions within the font's variation space over which the different delta values apply. The Item Variation Store format is designed to accommodate both the set of regions and the delta values. Within the CFF2 table, the Item Variation Store is used to represent the different regions, but the delta values are interleaved within the CharStrings where they are used.

An Item VariationStore contains two important lists. The first list is data that describes the region of influence in variation space for each design that is used in the variable font. Each of these is called a *Variation Region*. The entire list of Variation Regions is called a *Variation Region List*.

The second list is an array of ItemVariationData structures that each specify a set of Variation Regions, as a list of indices into the Variation Region List. This allows different glyphs to have delta values that apply to different sets of regions. There is often only one itemVariationData structure, and hence only one set of regions that is used by all glyphs. If more than one set of regions are needed, then an itemVariationData structure is added to define each set. The **vsindex** operator may be used in a Private DICT to set the itemVariationData index for all glyphs which reference the Private DICT, or it may be used in specific CFF2 CharStrings when a CharString needs to use a different ItemVariationData structure than is specified in the Private DICT.

An example of a Variation Store structure in a CFF2 table can be seen in the Example CFF2 Font [29].

5.4.4 VORG – Vertical origin table

This optional 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:

a) directly from the 'VORG', or:

b) 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
uint16	majorVersion	Major version (starting at 1). Set to 1.
uint16	minorVersion	Minor version (starting at 0). Set to 0.
int16	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.
uint16	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
uint16	glyphIndex	Glyph index.
int16	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 Table for SVG glyph outlines

5.5.1 SVG – The SVG (Scalable Vector Graphics) table

OFF fonts with either TrueType or CFF outlines may also contain an optional ‘SVG ’ table, which allows some or all glyphs in the font to be defined with color, gradients, or animation.

This table contains SVG descriptions for some or all of the glyphs in the font, which shall be as specified in the Scalable Vector Graphics (SVG) 1.1 (2nd edition), W3C Recommendation. For every SVG glyph description, there must also exist a corresponding CFF or TT glyph description in the font.

SVG Main Header

Type	Name	Description
uint16	version	Table version (starting at 0). Set to 0.
Offset32	svgDocIndexOffset	Offset (relative to the start of the SVG table) to the SVG Documents Index. Must be non-zero.
uint32	reserved	Set to 0.

SVG Document Index

The SVG Document Index is a set of SVG documents, each of which defines one or more glyph descriptions.

Type	Name	Description
uint16	numEntries	Number of SVG Document Index Entries. Must be non-zero.
SVG Document Index Entry	entries[numEntries]	Array of SVG Document Index Entries.

SVG Document Index Entry

Each SVG Document Index Entry specifies a range [startGlyphID, endGlyphID], inclusive, of glyph IDs and the location of its associated SVG document in the SVG table.

Type	Name	Description
uint16	startGlyphID	The first glyph ID in the range described by this index entry.
uint16	endGlyphID	The last glyph ID in the range described by this index entry. Must be >= startGlyphID.
Offset32	svgDocOffset	Offset from the beginning of the SVG Document Index to an SVG document. Must be non-zero.
uint32	svgDocLength	Length of the SVG document. Must be non-zero.

Index entries must be arranged in order of increasing startGlyphID. The startGlyphID of an index entry must be greater than the endGlyphID of the previous index entry, if any.

While SVG 1.1 [16] requires in addition to plain text encoding that conforming SVG implementations shall correctly support gzip-encoded [RFC1952] and deflate-encoded [RFC1951] data streams, this document requires that the SVG documents be either plain-text or gzip-encoded [RFC1952]. The encoding of the (uncompressed) SVG document must be UTF-8. In both cases, svgDocLength encodes the length of the encoded data, not the decoded document.

For further details about the content of the SVG documents, see “Glyph Identifiers” and the following sections below.

5.5.2 Color Palettes

The SVG glyph descriptions may contain color variables whose values are obtained either from one of the various color palettes in the Color Palette (CPAL) table or by other means, for example values specified by the user. The first color palette shall be the default one. It is strongly recommended that the default values for the color variables in the SVG documents be set to the same values as in the first color palette table, for implementations that may not support color palettes.

Color variables are made available for use in the SVG glyph descriptions by the font engine setting CSS custom properties [18] in a User Agent style sheet. The custom property names are of the form "--color<num>", where <num> is a parameter index in the range [0, numPaletteEntries-1], inclusive, expressed as a non-zero- padded decimal number. numPaletteEntries is defined in the CPAL table. See the “Glyph rendering” section below for exactly how the values are to be passed in to the SVG document.

Font engines that support the SVG table and color palettes are strongly suggested to implement the CSS Custom Properties for Cascading Variables specification [18], as this is required for the palette entries to be passed into the SVG document.

Note that the SVG glyph descriptions are able to express their own explicit or “hard-coded” colors as well. These are not related to color variables and thus do not vary by palette selection. For example, a font designer may want the teardrop on a crying emoji always to be blue (this is “hard-coded”) but the rest of the emoji regulated by color variables, with the skin of the face having a default value of the classic “smiley face”

yellow (default both in the SVG glyph description itself – see the var(--color0, yellow) example below – and in the default color palette).

5.5.3 Glyph Identifiers

For each glyph ID in an SVG Document Index Entry's [startGlyphID, endGlyphID] range, inclusive, the associated SVG document shall contain an element with id "glyph<glyphID>", where <glyphID> is the glyph ID expressed as a non–zero-padded decimal value. This element functions as the SVG glyph description for the glyph ID.

For example, say a font with maxp.numGlyphs=100 has SVG glyph definitions only for its last 5 glyphs. The last SVG glyph definition has its own SVG document, but the rest share an SVG document (say, to take advantage of shared graphical elements). There will be two index entries, the first with glyph ID range [95, 98] and the second with glyph ID range [99, 99]. The SVG document referenced by the first index entry will contain elements with id "glyph95", "glyph96", "glyph97", and "glyph98". The SVG document referenced by the second index entry will contain an element with id "glyph99".

5.5.4 Glyph Semantics and Metrics

The glyph descriptions in the SVG documents are considered to be the SVG versions of the glyphs with the corresponding IDs in the CFF or glyf table. They are designed on an em specified in the head table's unitsPerEm field, as with CFF and TrueType glyphs. SVG glyph definitions will be in SVG's y-down coordinate system, upright, with the default baseline at y = 0. For example, the top of a capital letter may be at y = -800, and the bottom at y = 0 (see subclause 5.5.6). It is the font engine's responsibility to translate this to the coordinate system of the rest of the OT tables and the coordinate system of the graphics environment.

Glyph semantics are expressed in the usual way (cmap table followed by GSUB). Glyph metrics such as horizontal and vertical advances are specified in the OFF metrics tables (hmtx and vmtx), and glyph positioning adjustments by the GPOS or kern table.

As with CFF glyphs, no explicit glyph bounding boxes are recorded. The "ink" bounding box of the rendered SVG glyph should be used if a bounding box is desired; this box may be different for animated vs static renderings of the glyph.

Note that the glyph advances are static and not able to be made variable or animated.

5.5.5 Glyph Rendering

The SVG glyph descriptions may be rendered statically or with animation enabled. Note that static rendering is done by not running any animations in the SVG document; this is different from running the document with animations running but at a snapshot time of zero seconds. Some clients may choose not to support – or may not be able to support – animation. Clients that support animation may still request, in certain cases, that the glyph be rendered statically, e.g. for printing to paper.

The font engine shall apply the following user agent style sheet (or implement its functional equivalent) to SVG documents processed from the SVG table:

```
@namespace svg url(http://www.w3.org/2000/svg);
svg|text, svg|foreignObject {
     display: none !important;
}

:root {
   fill: context-fill;
   fill-opacity: context-fill-opacity;
   stroke: context-stroke;
   stroke-opacity: context-stroke-opacity;
   stroke-width: context-value;
   stroke-dasharray: context-value;
   stroke-dashoffset: context-value;
}
```

In addition, if the font engine supports color palettes, and color palette values are provided, the user agent style sheet must include CSS Custom Property declarations for the color variables. This is done by including 'numPaletteEntries' (defined in the CPAL table) declarations in the :root rule of the form:

```
--color<num>: <color>;
```

where <num> is each of the values from zero to numPaletteEntries–1, inclusive, expressed as a non-zero-padded decimal number; and <color> is the <num> index within the desired Color Palette, expressed in SVG's <color> format. An example entry in the style sheet is:

```
--color0: rgba(255,0,0, 0.5);
```

and the corresponding usage in an SVG glyph description could be something like:

```
<path fill="var(--color0, yellow)" d="..."/>
```

where 'yellow' defines a default color to be used when color0 variable is not defined.

Note that SVG's context-fill value may be used in the glyph descriptions to denote the current text color.

The font engine must support at least version 1.1 of the SVG specification (exceptions are noted in the section on glyph rendering restrictions). The version attribute in the <svg> element is present in the SVG 1.1 and 1.2 specifications, but not in SVG 2. Thus, the SVG document may not always have a version field specified. Given this approach to versioning in SVG, and given that not all implementations may support all of SVG (whether 1.1 or 2), font designers should restrict their SVG, as a practical matter, to whatever is supported by SVG-in-OT implementations they care about. Targeting the capabilities of SVG 1.1 would be the approach most likely to result in cross-implementation consistency.

The following new values for any CSS property that takes an SVG paint value shall be supported:

context-fill

context-stroke

These values mean the same paint as the computed value of the 'fill' or 'stroke' property, respectively, of the context element, which is the element in the outer document that is using the SVG glyphs. If the referenced paint is a gradient or a pattern, then the coordinate space to use and the object used for any 'objectBoundingBox'-relative values are the same as those of the context element.

The following new values for the 'fill-opacity', 'stroke-opacity' and 'opacity' CSS properties shall be supported:

context-fill-opacity

context-stroke-opacity

These values mean the same as the computed value of the 'fill-opacity' or 'stroke-opacity' property, respectively, of the context element.

The following new value for the 'stroke-width', 'stroke-dasharray' and 'stroke-dashoffset' CSS properties shall be supported:

context-value

This value means the same as the computed value of the corresponding property of the context element, but scaled so that it has the same size when used in the coordinate system of the root <svg> element of the SVG glyph document. For example, if the context element has 'stroke-width' set to 2px and the SVG glyph document is rendered with a coordinate system such that 2048 user units corresponds to 16px in the context element's coordinate space, then using context-value for 'stroke-width' in the glyph definition will have the same visual effect as using 256 user units.

Font engines that support SVG glyphs are strongly suggested to implement the context-fill, context-fill-opacity, context-stroke, context-stroke-opacity and context-value property values according to the definitions found in SVG 2 [19], as these definitions may be more precise than those described in this document above.

Security considerations and other glyph rendering restrictions

Processing of SVG glyph documents shall be done with script execution, external references and interactivity disabled. If the font engine is rendering SVG glyphs with animation, then declarative animations shall be enabled; if it is rendering glyphs statically, then declarative animations shall be disabled.

These requirements correspond to the "secure animated" and "secure static" processing modes that the SVG Integration document [17] requires font documents to be run in.

In addition, any SVG <text> and <foreignObject> elements within a glyph description must be ignored and not rendered (see the corresponding rules in the User Agent style sheet above).

Text Layout Process

An implementation that supports the SVG table first does layout in the usual manner, using the cmap, GSUB, hmtx, and other OFF layout tables. This results in a list of glyph IDs arranged at particular x,y positions on the surface (along with the appropriate scale/rotation matrices). At this point, for each such glyph ID, if an SVG glyph description is available for it, it is rendered (in static or animated mode, as appropriate and if supported by the engine); otherwise, the CFF or TT glyph description must be rendered. Since the glyph advances are the same in either case, and not allowed to be animated, switching between SVG and CFF/TT rendering, or between animated and static SVG, should not require re-layout of lines (unless line layout depends on ink bounding boxes).

5.5.6 SVG glyph examples

SVG glyph descriptions must be defined in SVG's own y-down coordinate system, upright, with the default baseline at y=0. It is *always* the font engine's responsibility to translate this into the coordinate system of the rest of the OFF font rendering environment.

The SVG code in these examples is presented exactly as could be used in the SVG documents of an OFF font with SVG glyph outlines. The code is not optimized for brevity.

Example: Glyph specified directly in expected position

```
<svg id="glyph7" version="1.1" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <linearGradient id="grad" x1="0%" y1="0%" x2="0%" y2="100%">
      <stop offset="0%" stop-color="darkblue" stop-opacity="1" />
      <stop offset="100%" stop-color="#00aab3" stop-opacity="1" />
    </linearGradient>
  </defs>
  <rect x="100" y="-430" width="200" height="430" fill="url(#grad)" />
  <rect x="100" y="-635" width="200" height="135" fill="darkblue" />
</svg>
```

In this example, the letter "i" is drawn directly in the +x –y quadrant of the SVG coordinate system, upright, with its baseline on the x axis, exactly where the OFF font engine expects it to be.

Example: Glyph shifted up with viewBox

```
<svg id="glyph7" version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 1000
1000 1000">
  <defs>
    <linearGradient id="grad" x1="0%" y1="0%" x2="0%" y2="100%">
```

```
      <stop offset="0%" stop-color="darkblue" stop-opacity="1" />
      <stop offset="100%" stop-color="#00aab3" stop-opacity="1" />
    </linearGradient>
  </defs>
  <rect x="100" y="570" width="200" height="430" fill="url(#grad)" />
  <rect x="100" y="365" width="200" height="135" fill="darkblue" />
</svg>
```

In this example, the glyph description of the letter “i” is first specified in the +x +y quadrant of the SVG coordinate system, upright, with its baseline along y=1000 in the SVG coordinate system. (This may be the natural way SVG illustrating software positioned it.) A viewBox in the <svg> element is then used to shift it upwards by 1000 units, to end up in the position where the OFF font engine expects it to be.

The diagram is the same as in the above example.

Example: Common elements shared across glyphs in same SVG doc

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs>
    <linearGradient id="grad" x1="0%" y1="0%" x2="0%" y2="100%">
      <stop offset="0%" stop-color="darkblue" stop-opacity="1" />
      <stop offset="100%" stop-color="#00aab3" stop-opacity="1" />
    </linearGradient>
    <g id="i-base">
      <rect x="100" y="570" width="200" height="430" fill="url(#grad)" />
    </g>
  </defs>
  <g id="glyph2" transform="translate(0,-1000)">
    <use xlink:href="#i-base" />
  </g>
  <g id="glyph13" transform="translate(0,-1000)">
    <use xlink:href="#i-base" />
    <rect x="100" y="365" width="200" height="135" fill="darkblue" />
  </g>
  <g id="glyph14" transform="translate(0,-1000)">
    <use xlink:href="#i-base" />
    <polygon fill="darkblue" points="120,500 280,500 435,342 208,342 "/>
  </g>
</svg>
```

In this example, the base of the letter 'i' is shared across three glyphs, and has identifier “i-base” in the <defs> section. It represents the dotless 'i' in glyph ID 2. Glyph ID 13 adds a dot on top. Glyph ID 14 adds an acute accent on top. The diagram above shows glyph IDs 2, 13, and 14, from left to right.

Note that glyph IDs 3–12 can be defined in one or more separate SVG docs, and still allow glyph IDs 2, 13, and 14 to share the same SVG doc. For example:

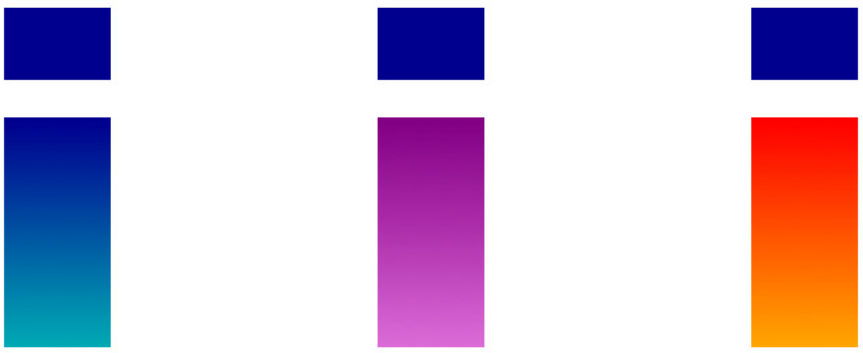
SVG Document Index: numEntries=5
…
entries[2]: { startGlyphID = 2, endGlyphID = 2, svgDocOffset/Length point to svgDoc0 }
entries[3]: { startGlyphID = 3, endGlyphID = 12, svgDocOffset/Length point to svgDoc1 }
entries[4]: { startGlyphID = 13, endGlyphID = 14, svgDocOffset/Length point to svgDoc0 }

Example: Specifying current text color in glyphs

```
<svg id="glyph7" version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 1000
1000 1000">
  <defs>
    <linearGradient id="grad" x1="0%" y1="0%" x2="0%" y2="100%">
      <stop offset="0%" stop-color="darkblue" stop-opacity="1" />
      <stop offset="100%" stop-color="#00aab3" stop-opacity="1" />
    </linearGradient>
  </defs>
  <rect x="100" y="570" width="200" height="430" fill="url(#grad)" />
  <rect x="100" y="365" width="200" height="135" fill="context-fill" />
</svg>
```

Here the “darkblue” color of the dot above the “i” in the “Glyph shifted up with viewBox” example is replaced by “context-fill”. The diagram above shows the glyph when the fill color of the context element (i.e. the text color) is set to black (left) and red (right).

Example: Specifying color palette variables in glyphs



```
<svg id="glyph7" version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 1000
1000 1000">
  <defs>
    <linearGradient id="grad" x1="0%" y1="0%" x2="0%" y2="100%">
      <stop offset="0%" stop-color="var(--color0,darkblue)" stop-opacity="1" />
      <stop offset="100%" stop-color="var(--color1,#00aab3)" stop-opacity="1" />
    </linearGradient>
  </defs>
  <rect x="100" y="570" width="200" height="430" fill="url(#grad)" />
  <rect x="100" y="365" width="200" height="135" fill="darkblue" />
</svg>
```

This example is the duplicate of the “Glyph shifted up with viewBox” example, but with the stop colors of the linear gradient controlled by color variables --color0 and --color1, which are provided by the font engine to the SVG renderer via a user agent style sheet (or its functional equivalent).

The color palettes (CPAL) table in this font specifies two palettes, each with two color entries. Here is a description of the CPAL palettes, with alpha assumed to be 0xFF for all colors:

palette[0]: { darkblue, #00aab3 }

palette[1]: { purple, orchid }

The first item in the diagram above shows the first color palette applied to the glyph, which is done by the font engine passing the following user agent style sheet to the SVG renderer:

```
:root {
  --color0: darkblue;
  --color1: #00aab3;
}
```

The second item in the diagram shows the second color palette applied to the glyph, using the style sheet:

```
:root {
  --color0: purple;
  --color1: orchid;
}
```

Note that the dot is still dark blue, since this is hard coded in the glyph description and not controlled by a color variable.

The last item in the diagram shows the following user-selected colors applied to the glyph via the color variables:

```
:root {
  --color0: red;
  --color1: orange;
}
```

If --color0 and --color1 aren’t defined by the font engine, however, then the default values provided in the stop-colors (darkblue and #00aab3, respectively) are used. Note that these are in fact the same colors as in the first (default) CPAL color palette, which means the glyph will render as in the first item in the diagram. This way, the glyph renders with the same colors by default, whether or not the font engine supports the CPAL.

Example: Embedding a PNG in an SVG glyph

```
<svg id="glyph2" version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" viewBox="0 1000 1000 1000">
  <image x="100" y="365" width="200" height="635"
  xlink:href="
iVBORw0KGgoAAAANSUhEUgAAAMgAAAJ7CAYAAACmmd5sAAAFZklEQVR42u3XsQ3D
MBAEQUpw9ypahrMPGGwiwcFMCQQW9zzWuu4FbJ2eAAQCAgGBgEBAICAQEAgIBAQC
CAQEAgIBgYBAQCAgEBAICAQEAggEBAICAYGAQEAgIBAQCAgEEAgIBAQCAgGBgEBA
ICAQEAgIBBAICAQEAgIBgYBAQCAgEBAIIBAQCAgEBAICAYGAQEAgIBAQCCAQEAgI
BAQCAgGBgEBAICAQQCAgEBAICAQEAgIBgYBAQCAgEEAgIBAQCAgEBAICAYGAQEAg
IBBPAAIBgYBAQCAgEBAICAQEAgIBBAICAYGAQEAgIBAQCAgEBAICAQQCAgGBgEBA
ICAQEAgIBAQCCAQEAgIBgYBAQCAgEBAICAQEAggEBAICAYGAQEAgIBAQCAgEAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA4DHHWtftGWDv80sE2Ds9AQgEBAL+IPBuIAoBJxYIBAQCPukgEHBigUBAIOAP
AlgQiAtiQsCCgEDAJx0sCFgQsCAgEHBigQUB5oKYELAgIBDwSQcLAhYELAgIBJxY
YEEACwItEIWAEwucWGBBwIKABQGBgBMLLAhYEMCCQFwQEwJOLHBigQUBCwICAScW
WBCwIGBBAIFAPbHcWGBBwCcdLAgIBJxYYEHAgoAFAYEA88RyY4EFAZ90sCAgEBAI
+IOAQMCJBQIBBALxD+ITAj7p4MQCgYBAwB8EBAJOLBAICATwB4EYiELAiQUCAYGA
TzoIBJxYIBAQCPiDABYE4oKYELAgIBDwSQcLAhYELAgIBJxYYEGAuSAmBCwICAR8
0sGCgAUBCwICAScWWBDAgkALRCHgxAInFlgQsCBgQUAg4MQCCwIWBLAgEBfEhIAT
C5xYYEHAgoBAwIkFFgQsCFgQQCBQTyw3FlgQ8EkHCwICAScWWBCwIGBBQCDAPLHc
WGBBwCcdLAgIBAQC/iAgEHBigUAAgUD8g/iEgE86OLFAICAQ8AcBgYATCwQCAgH8
QSAGohBwYoFAQCDgkw4CAScWCAQEAv4ggAWBuCAmBCwICAR80sGCgAUBCwICAScW
WBBgLogJAQsCAgGfdLAgYEHAgoBAwIkFFgSwINACUQg4scCJBRYELAhYEBAIOLHA
goAFASwIxAUxIeDEAicWWBCwICAQcGKBBQELAhYEEAjUE8uNBRYEfNLBgoBAwIkF
FgQsCFgQEAgwTyw3FlgQ8EkHCwICAYGAPwgIBJxYIBBAIBD/ID4h4JMOTiwQCAgE
/EFAIODEAoGAQAB/EIiBKAScWCAQEAj4pINAwIkFAgGBgD8IYEEgLogJAQsCAgGf
dLAgYEHAgoBAwIkFFgSYC2JCwIKAQMAnHSwIWBCwICAQcGKBBQEsCLRAFAJOLHBi
gQUBCwIWBAQCTiywIGBBAAsCcUFMCDixwIkFFgQsCAgEnFhgQcCCgAUBBAL1xHJj
gQUBn3SwICAQcGKBBQELAhYEBALME8uNBRYEfNLBgoBAQCDgDwICAScWCAQQCMQ/
iE8I+KSDEwsEAgIBfxAQCDixQCAgEMAfBGIgCgEnFggEBAI+6SAQcGKBQEAg4A8C
WBCIC2JCwIKAQMAnHSwIWBCwICAQcGKBBQHmgpgQsCAgEPBJBwsCFgQsCAgEnFhg
QQALAi0QhYATC5xYYEHAgoAFAYGAEwssCFgQwIJAXBATAk4scGKBBQELAgIBJxZY
ELAgYEEAgUA9sdxYYEHAJx0sCAgEnFhgQcCCgAUBgQDzxHJjgQUBn3SwICAQEAj4
g4BAwIkFAgEEAvEP4hMCPungxAKBgEDgH3wBrUwJtCBGuc0AAAAASUVORK5CYII=
  "/>
</svg>
```

In this example, the PNG is embedded using SVG's <image> element. The use case for this is bitmap lettering artwork that needs to be packaged into an OT-SVG font.

5.6 Tables related to bitmap glyphs

5.6.1 List of 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 Adobe Type Manager 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
CBDT	Color bitmap data
CBLC	Color bitmap location data

5.6.2 EBDT – Embedded bitmap data table

The 'EBDT' table is used to embed monochrome or grayscale bitmap glyph data. It is used together with the 'EBLC' table, which provides embedded bitmap locators, and the 'EBSC' table, which provides embedded bitmap scaling information.

5.6.2.1 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.Table structure

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
uint16	majorVersion	Major version of the EBDT table, = 2.
uint16	minorVersion	Minor version of the EBDT table, = 0.

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 BitmapSize table within the 'EBLC' table.

BigGlyphMetrics

Type	Name
uint8	height
uint8	width
int8	horiBearingX
int8	horiBearingY
uint8	horiAdvance
int8	vertBearingX
int8	vertBearingY
uint8	vertAdvance

SmallGlyphMetrics

Type	Name
uint8	height
uint8	width
int8	bearingX
int8	bearingY
uint8	advance

5.6.2.2 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.

5.6.2.2.1 Format 1: small metrics, byte-aligned data

Type	Name	Description
SmallGlyphMetrics	smallMetrics	Metrics information for the glyph
VARIABLE	imageData	Byte-aligned bitmap data

Glyph bitmap format 1 consists of small metrics records (either horizontal or vertical depending on the flags field of the BitmapSize table within 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.

5.6.2.2.2 Format 2: small metrics, bit-aligned data

Type	Name	Description
SmallGlyphMetrics	smallMetrics	Metrics information for the glyph
VARIABLE	imageData	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.

5.6.2.2.3 Format 3: (obsolete)

This format is not supported in OFF.

5.6.2.2.4 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.

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

Type	Name	Description
VARIABLE	imageData	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.

5.6.2.2.6 Format 6: big metrics, byte-aligned data

Type	Name	Description
BigGlyphMetrics	bigMetrics	Metrics information for the glyph
VARIABLE	imageData	Byte-aligned bitmap data

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

5.6.2.2.7 Format7: big metrics, bit-aligned data

Type	Name	Description
BigGlyphMetrics	bigMetrics	Metrics information for the glyph
VARIABLE	imageData	Bit-aligned bitmap data

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

EbdtComponent Record

The EbdtComponent record is used in glyph bitmap data formats 8 and 9.

Type	Name	Description
uint16	glyphID	Component glyph ID
int8	xOffset	Position of component left
int8	yOffset	Position of component top

The EbdtComponent record contains the glyph ID of the component, which can be used to look up the location of component glyph data in the 'EBLC' table, as well as xOffset and yOffset values which specify 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.

5.6.2.2.8 Format 8: small metrics, component data

Type	Name	Description
SmallGlyphMetrics	smallMetrics	Metrics information for the glyph
uint8	pad	Pad to short boundary
uint16 EbdtComponent	numComponents components[numComponents]	Number of components Array of EbdtComponent records

5.6.2.2.9 Format 9: big metrics, component data

Type	Name	Description
BigGlyphMetrics	bigMetrics	Metrics information for the glyph
uint16	numComponents	Number of components
EbdtComponent	components[numComponents]	Array of EbdtComponent records

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.

5.6.3 EBLC – Embedded bitmap location table

The EBLC provides embedded bitmap locators. It is used together with the EDBT table, which provides embedded, monochrome or grayscale bitmap glyph data, and the EBSC table, which provided 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.

5.6.3.1 Table structure and data types

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
uint16	majorVersion	Major version of the EBLC table, = 2.
uint16	minorVersion	Minor version of the EBLC table, = 0.
uint32	numSizes	Number of BitmapSize tables.

The EblcHeader is followed immediately by the BitmapSize table array(s). The numSizes field in the EblcHeader indicates the number of BitmapSize tables in the array. Each strike is defined by one BitmapSize table.

BitmapSize table

Type	Name	Description
Offset32	indexSubTableArrayOffset	Offset to IndexSubtableArray, from beginning of EBLC.
uint32	indexTablesSize	Number of bytes in corresponding index subtables and array.
uint32	numberOfIndexSubTables	There is an index subtable for each range or format change.
uint32	colorRef	Not used; set to 0.
SbitLineMetrics	hori	Line metrics for text rendered horizontally.
SbitLineMetrics	vert	Line metrics for text rendered vertically.
uint16	startGlyphIndex	Lowest glyph index for this size.
uint16	endGlyphIndex	Highest glyph index for this size.
uint8	ppemX	Horizontal pixels per em.
uint8	ppemY	Vertical pixels per em.
uint8	bitDepth	The following bitDepth values are supported, as described below: 1, 2, 4, and 8.
int8	flags	Vertical or horizontal (see Bitmap Flags)

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.

5.6.3.2 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
int8	ascender
int8	descender
uint8	widthMax
int8	caretSlopeNumerator
int8	caretSlopeDenominator
int8	caretOffset
int8	minOriginSB
int8	minAdvanceSB
int8	maxBeforeBL
int8	minAfterBL
int8	pad1
int8	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 IDs in the strike, but a strike does not necessarily contain bitmaps for all glyph IDs 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.

Bit Depth

The bitDepth field of the BitmapSize table is used to specify the number of levels of gray used in the embedded bitmaps. The the following values are supported.

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

Mask	Name	Description
0x01	HORIZONTAL_METRICS	Horizontal
0x02	VERTICAL_METRICS	Vertical
0xFC	Reserved	Reserved for future use – set to 0.

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

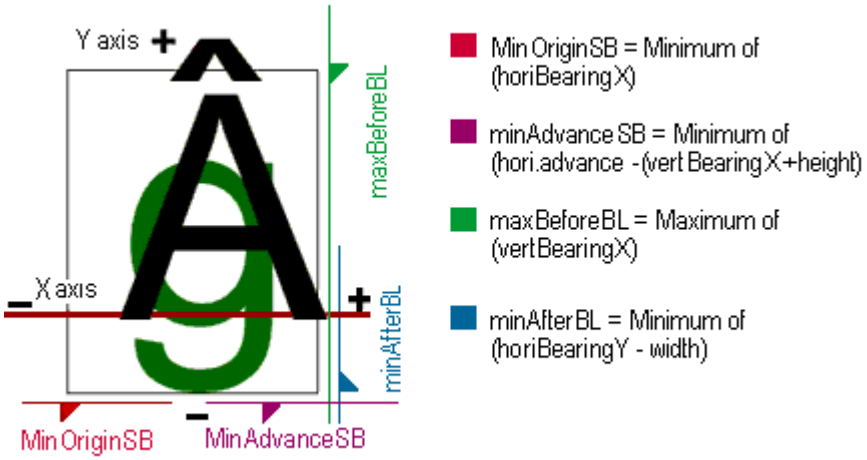


Figure 5.1 – Horizontal text

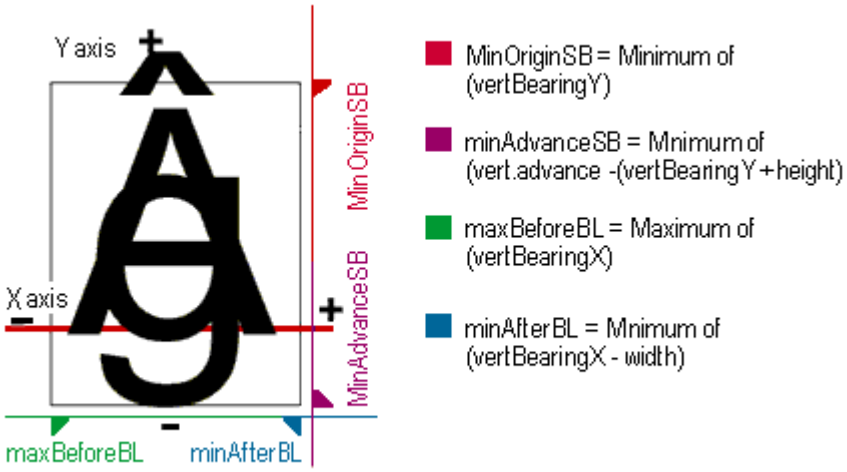


Figure 5.2 – Vertical text

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 BitmapSize table.

BigGlyphMetrics

Type	Name
uint8	height
uint8	width
int8	horiBearingX
int8	horiBearingY
uint8	horiAdvance
int8	vertBearingX
int8	vertBearingY
uint8	vertAdvance

SmallGlyphMetrics

Type	Name
uint8	height
uint8	width
int8	bearingX
int8	bearingY
uint8	advance

The following diagram illustrates the meaning of the glyph metrics.

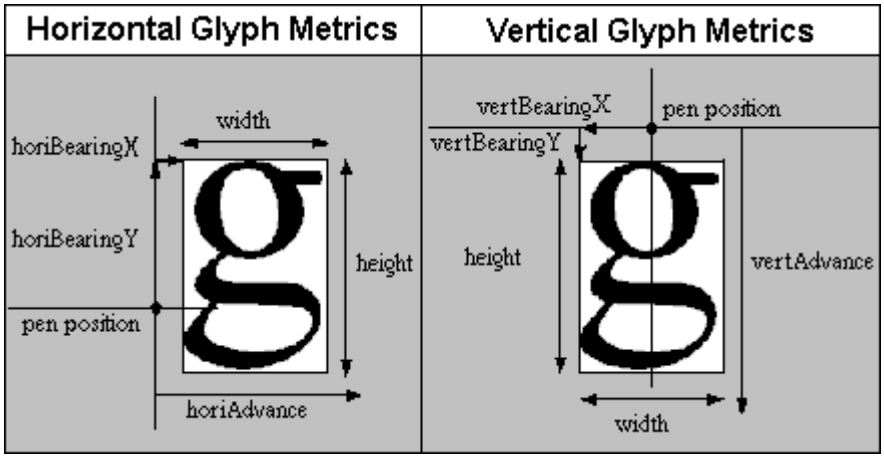


Figure 5.3 – Glyph metrics

The BitmapSize table for each strike contains the offset to an array of IndexSubTableArray elements. Each element describes a glyph ID range and an offset to the IndexSubTable for that range. This allows a strike to contain multiple glyph ID ranges and to be represented in multiple index formats if desirable.

IndexSubTableArray

Type	Name	Description
uint16	firstGlyphIndex	First glyph ID of this range
uint16	lastGlyphIndex	Last glyph ID of this range (inclusive)
Offset32	additionalOffsetToIndexSubtable	Add to indexSubTableArrayOffset to get offset from beginning of 'EBLC'

After determining the strike, the rasterizer searches this array for the range containing the given glyph ID. 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
uint16	indexFormat	Format of this IndexSubTable
uint16	imageFormat	Format of 'EBDT' image data
Offset32	imageDataOffset	Offset to image data in 'EBDT' table

IndexSubTables

There are currently five different formats used for the IndexSubTable, depending upon the size and type of bitmap data in the glyph ID 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.
Offset32	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
uint32	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.
Offset16	offsetArray[]	offsetArray[glyphIndex] + imageDataOffset = glyphData sizeOfArray = (lastGlyph – firstGlyph + 1) + 1 + 1, pad if needed

IndexSubTable4: variable-metrics glyphs with sparse glyph IDs

Type	Name	Description
IndexSubHeader	header	Header info.
uint32	numGlyphs	Array length
GlyphIdOffsetPair	glyphArray[]	One per glyph; sizeOfArray = numGlyphs+1

GlyphIdOffsetPair record:

Type	Name	Description
uint16	glyphID	Glyph ID of glyph present.
Offset16	offset	Location in EBDT

IndexSubTable5: constant-metrics glyphs with sparse glyph IDs

Type	Name	Description
IndexSubHeader	header	Header info.
uint32	imageSize	All glyphs have the same data size
BigGlyphMetrics	bigMetrics	All glyphs have the same metrics
uint32	numGlyphs	Array length
uint16	glyphIDArray[]	One per glyph, sorted by glyph ID; 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 ID 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 IDs are handled best by formats 1, 2, and 3 which store an offset for every glyph ID in the range. Very sparse ranges of glyph IDs should use format 4 or 5 which explicitly call out the glyph IDs 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 uint32's while format 3 uses uint16'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 (uint32) 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 uint16. When there are an odd number of elements in these arrays it is necessary to add an extra padding element to maintain proper alignment.

5.6.4 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 glyph 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 is used together with the EBDT table, which provides embedded monochrome or grayscale bitmap data, and the EBLC table, which provides embedded bitmap locators.

EBSC Header

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

Type	Name	Description
uint16	majorVersion	Major version of the EBSC table, = 2.
uint16	minorVersion	Minor version of the EBSC table, = 0.
uint32	numSizes	

The header is followed immediately by the BitmapScale table array. The numSizes field in the header indicates the number of BitmapScale tables in the array. Each strike is defined by one BitmapScale table.

BitmapScale Table

Type	Name	Description
SbitLineMetrics	hori	line metrics
SbitLineMetrics	vert	line metrics
uint8	ppemX	target horizontal pixels per Em
uint8	ppemY	target vertical pixels per Em
uint8	substitutePpemX	use bitmaps of this size
uint8	substitutePpemY	use bitmaps of this size

The line metrics have the same meaning as those in the BitmapSize table, 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.

5.6.5 CBDT – Color bitmap data table

The 'CBDT' table is used to embed color bitmap glyph data. It is used together with the 'CBLC' table (see subclause 5.6.6), which provides embedded bitmap locators. The formats of these two tables are backward compatible with the 'EBDT' (subclause 5.6.2) and 'EBLC' (subclause 5.6.3) tables used for embedded monochrome and grayscale bitmaps.

5.6.5.1 Table structure

The 'CBDT' table begins with a header containing simply the table version number.

Type	Name	Description
uint16	majorVersion	Major version of the CBDT table, = 3.
uint16	minorVersion	Miinor version of the CBDT table, = 0.

The rest of the 'CBDT' table is a collection of bitmap data. The data can be presented in three possible formats, indicated by information in the 'CBLC' 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 subtables; byte alignment is sufficient.

Uncompressed color bitmaps

The value ‘32’ of the bitDepth field of BitmapSize table struct defined in the CBLC table, to identify color bitmaps with 8-bit blue/green/red/alpha (“BGRA”) channels per pixel, encoded in that order for each pixel. The color channels represent *pre-multiplied* color and shall encode colors in the sRGB colorspace, as specified in IEC 61966-2-1/Amd 1:2003. For example, the color “full-green with half translucency” is encoded as \x00\x80\x00\x80, and *not* \x00\xFF\x00\x80.

All bitmap image data formats defined in the EBDT / EBLC tables are valid for use with the CBDT / CBLC tables.

Compressed color bitmaps

Images for each individual glyph are stored as straight PNG data, and shall be as specified in ISO/IEC 15948. Only the following chunks are allowed in such PNG data: IHDR, PLTE, tRNS, sRGB, IDAT, and IEND. If other chunks are present, the behavior is undefined. The image data shall be in the sRGB colorspace, regardless of color information that may be present in other chunks in the PNG data. The individual images must have the same size as expected by the table in the bitmap metrics.

Glyph Metrics

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 BitmapSize tables within the 'CBLC' table.

BigGlyphMetrics

Type	Name
uint8	height
uint8	width
int8	horiBearingX
int8	horiBearingY
uint8	horiAdvance
int8	vertBearingX
int8	vertBearingY
uint8	vertAdvance

SmallGlyphMetrics

Type	Name
uint8	height
uint8	width
int8	bearingX
int8	bearingY
uint8	advance

5.6.5.2 Glyph bitmap data formats

In addition to nine different formats already defined for glyph bitmap data in the EBDT table, there are three new formats described below.

5.6.5.2.1 Format 17: small metrics, PNG image data

Type	Name	Description
smallGlyphMetrics	glyphMetrics	Metrics information for the glyph
uint32	dataLen	Length of data in bytes
VARIABLE	data	Raw PNG data

5.6.5.2.2 Format 18: big metrics, PNG image data

Type	Name	Description
BigGlyphMetrics	glyphMetrics	Metrics information for the glyph
uint32	dataLen	Length of data in bytes
VARIABLE	data	Raw PNG data

5.6.5.2.3 Format 19: metricsin CBLC, PNG image data

Type	Name	Description
uint32	dataLen	Length of data in bytes
VARIABLE	data	Raw PNG data

Scaling behavior

Applications using these glyphs may need to scale them to fit the raster size available for display. How this scaling takes place is application dependent. It is recommended that where possible the application downscale using the closest sized bitmap that is larger than the desired end raster size.

5.6.6 CBLC – Color bitmap location table

The 'CBLC' table provides embedded bitmap locators. It is used together with the 'CBDT' table (subclause 5.6.5), which provides embedded, color bitmap glyph data. The formats of these two tables are backward compatible with the 'EBDT' (subclause 5.6.2) and 'EBLC' (subclause 5.6.3) tables used for embedded monochrome and grayscale bitmaps.

The 'CBLC' 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 'CBDT' table.

CblcHeader

Type	Name	Description
uint16	majorVersion	Major version of the 'CBLC' table, = 3.
uint16	minorVersion	Minor version of the 'CBLC' table, = 0.
uint32	numSizes	Number of BitmapSize tables

The CblcHeader is followed immediately by the BitmapSize table array(s). The numSizes in the cblcHeader indicates the number of BitmapSize tables in the array. Each strike is defined by one BitmapSize table.

BitmapSize table

Type	Name	Description
Offset32	indexSubTableArrayOffset	Offset to index subtable from beginning of CBLC.
uint32	indexTablesSize	Number of bytes in corresponding index subtables and array
uint32	numberOfIndexSubTables	There is an index subtable for each range or format change
uint32	colorRef	Not used; set to 0.
SbitLineMetrics	hori	Line metrics for text rendered horizontally
SbitLineMetrics	vert	Line metrics for text rendered vertically
uint16	startGlyphIndex	Lowest glyph index for this size
uint16	endGlyphIndex	Highest glyph index for this size
uint8	ppemX	Horizontal pixels per Em
uint8	ppemY	Vertical pixels per Em
uint8	bitDepth	In addtition to already defined bitDepth values 1, 2, 4, and 8 supported by existing implementations, the value of 32 is used to identify color bitmaps with 8 bit per pixel RGBA channels.
int8	flags	Vertical or horizontal (see Bitmap Flags section of subclause 5.6.3)

The indexSubTableArrayOffset is the offset from the beginning of the 'CBLC' table to the IndexSubTableArray. Each strike has one of these arrays to support various formats and discontiguous ranges of bitmaps. The indexTablesSize is the total number of bytes in the IndexSubTableArray and the associated IndexSubTables. The numberOfIndexSubTables is a count of the IndexSubTables for this strike.

The rest of the CBLC table structure is identical to one already defined for EBLC, see subclause 5.6.3 for details.

5.6.7 sbix – Standard bitmap graphics table

This table provides access to bitmap data in a standard graphics format, such as PNG, JPEG or TIFF.

The 'sbix' table has functionality somewhat similar to the 'EBDT' table in that both provide bitmap data for glyph presentation. They are different in three important respects, however. First, whereas the 'EBDT' table supports only black/white or grayscale bitmaps, the 'sbix' table supports color bitmaps. Secondly, whereas the 'EBDT' table uses formats specific to the OFF specification, the 'sbix' table uses standard bitmap graphics formats that are in common use. Thirdly, whereas the 'EBDT' table must be used in conjunction with other tables ('EBLC' and 'EBSC') for processing the bitmap data, the 'sbix' table contains complete data required for processing bitmaps.

A font that includes an 'sbix' table may also include outline glyph data in a 'glyf' or 'CFF ' table. An 'sbix' table can provide bitmap data for all glyph IDs, or for only a subset of glyph IDs. A font can also include different bitmap data for different sizes (“strikes”), and the glyph coverage for one size can be different from that for another size.

5.6.7.1 Header

'sbix' Header:

Type	Name	Description
uint16	version	Table version number – set to 1
uint16	flags	Bit 0: Set to 1. Bit 1: Draw outlines. Bits 2 to 15: reserved (set to 0).
uint32	numStrikes	Number of bitmap strikes.
Offset32	strikeOffsets[numStrikes]	Offsets from the beginning of the 'sbix' table to data for each individual bitmap strike.

For historical reasons, bit 0 of the flags field should always be set to 1.

If bit 1 of the flags field is clear, then the application is instructed to draw only the bitmaps for each glyph supported in the 'sbix' table. If bit 1 is set, then the application is instructed to draw the bitmap and the outline, in that order (that is, with the outline overlaid on top of the bitmap). If the 'sbix' table does not contain bitmap data for a glyph, then the outline is always drawn, regardless of the state of bit 1.

NOTE Application support for bit 1 of the flags field is optional. To ensure the best compatibility, set this bit to 0.

5.6.7.2 Strikes

Each strike includes a header and the glyph bitmap data. The header has the following format:

Type	Name	Description
uint16	ppem	The PPEM size for which this strike was designed.
uint16	ppi	The device pixel density (in PPI) for which this strike was designed. (E.g., 96 PPI, 192 PPI.)
Offset32	glyphDataOffsets[numGlyphs+1]	Offset from the beginning of the strike data header to bitmap data for an individual glyph ID.

The glyphDataOffset array includes offsets for every glyph ID, plus one extra. The number of glyphs is determined from the 'maxp' table. The length of the bitmap data for each glyph is variable, and can be determined from the difference between two consecutive offsets. Hence, the length of data for glyph N is glyphDataOffset[N+1] – glyphDataOffset[N]. If this is zero, there is no bitmap data for that glyph in this strike. There is one extra offset in the array in order to provide the length of data for the last glyph.

NOTE The length of data for non-printing glyphs, such as space, should always be zero.

A strike does not need to include data for every glyph, and does not need to include data for the same set of glyphs as other strikes. If the application is using bitmap data to draw text and there is bitmap data for a glyph in *any* strike, then the glyph must be drawn using a bitmap from *some* strike. If the exact size is not available, implementations may choose a bitmap based on the closest available larger size, or the closest available integer-multiple larger size, or on some other basis. The only cases in which a glyph is not drawn using a bitmap are if the application has not requested that text be drawn using bitmap data or if there is no bitmap data for the glyph in any strike.

Each strike targets a specific PPEM size and device pixel density (PPI). Thus, a font can contain two strikes for the same PPEM but different pixel densities, or two strikes for the same pixel density but different PPEMs. Note that some platforms may not support targeting of strikes for particular pixel densities.

5.6.7.3 Glyph data

The data for each glyph includes a header and the actual, embedded graphic data, with the following format:

Type	Name	Description
int16	originOffsetX	The horizontal (x-axis) offset from the left edge of the graphic to the glyph's origin. That is, the x-coordinate of the point on the baseline at the left edge of the glyph.
int16	originOffsetY	The vertical (y-axis) offset from the bottom edge of the graphic to the glyph's origin. That is, the y-coordinate of the point on the baseline at the left edge of the glyph.
Tag	graphicType	Indicates the format of the embedded graphic data: one of 'jpg ', 'png ' or 'tiff', or the special format 'dupe'.
uint8	data[]	The actual embedded graphic data. The total length is inferred from sequential entries in the glyphDataOffsets array and the fixed size (8 bytes) of the preceding fields.

The graphicType field indicates the format of the embedded graphic data. Three standard formats are supported: JPEG, PNG and TIFF; these are indicated using tag values 'jpg ', 'png ' and 'tiff', respectively.

The special graphicType of 'dupe' indicates that the data field contains a two-byte, big-endian glyph ID. The bitmap data for the indicated glyph should be used for the current glyph.

NOTE Apple's specification for TrueType fonts allows for a graphicType tag value of 'pdf ' or 'mask'. These values are not supported in the OFF specification, however.

5.6.7.4 Table dependencies

The glyph count is derived from the 'maxp' table. Advance and side-bearing glyph metrics are stored in the 'hmtx' table for horizontal layout, and the 'vmtx' table for vertical layout.

5.7 Optional tables

Tag	Name
DSIG	Digital signature
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
COLR	Color table
CPAL	Color palette table

5.7.1 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**
uint32	version	Version number of the DSIG table (0x00000001)
uint16	numSignatures	Number of signatures in the table
uint16	flags	permission flags Bit 0: cannot be resigned Bits 1-7: Reserved (Set to 0)
SignatureRecord	signatureRecords[munSignatures]	Array of signature records

The version of the DSIG table is expressed as a uint32, 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 has an array of signature records, which specifying the format and offset of signature blocks:

SignatureRecord		
Type	Name	Description
uint32	format	Format of the signature
uint32	length	Length of signature in bytes
Offset32	offset	Offset to the signature block from the beginning of the table

Signatures are contained in one or more signature blocks. Signature blocks may have various formats; currently one format is defined. The format identifier specifies both the format of the signature block, as well as the hashing algorithm used to create and authenticate the signature:

Signature Block Format 1		
Type	Name	Description
uint16	reserved1	Reserved for future use; set to zero.
uint16	reserved2	Reserved for future use; set to zero.
uint32	signatureLength	Length (in bytes) of the PKCS#7 packet in the signature field.
uint8	signature[signatureLength]	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.

5.7.2 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. See Clause 7 "Recommendations for OFF Fonts" for more detail.

The table begins as follows:

hdmx Header		
Type	**Name**	**Description**
uint16	version	Table version number (0)
int16	numRecords	Number of device records.
int32	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**
uint8	pixelSize	Pixel size for following widths (as ppem).
uint8	maxWidth	Maximum width.
uint8	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.

5.7.3 kern – Kerning

The kerning table contains the values that control the inter-character spacing for the glyphs in a font.

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
uint16	version	Table version number (0)
uint16	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
uint16	version	Kern subtable version number (0)
uint16	length	Length of the subtable, in bytes (including this header).
uint16	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

Format 0 is the only subtable format supported by Windows.

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
uint16	nPairs	This gives the number of kerning pairs in the table.
uint16	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.
uint16	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).
uint16	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
uint16	left	The glyph index for the left-hand glyph in the kerning pair.
uint16	right	The glyph index for the right-hand glyph in the kerning pair.
FWORD	value	The kerning value for the above pair, in font design units. 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
uint16	rowWidth	The width, in bytes, of a row in the table.
Offset16	leftClassTable	Offset from beginning of this subtable to left-hand class table.
Offset16	rightClassTable	Offset from beginning of this subtable to right-hand class table.
Offset16	array	Offset from beginning of this subtable to the start of the kerning array.

Each class table has the following header:

Type	Field	Description
uint16	firstGlyph	First glyph in class range.
uint16	nGlyphs	Number of glyph in class range.

This header is followed by nGlyphs number of class values, which are in uint16 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.

5.7.4 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
uint16	version	Version number (starts at 0).
uint16	numGlyphs	Number of glyphs (from "numGlyphs" in 'maxp' table).
uint8	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 have yPels = 1; i.e., always scales linearly.

5.7.5 MERG – Merge table

The 'MERG' table enables a font to specify whether antialias filtering of glyphs within a glyph run can be performed separately for each glyph, or whether certain glyph pairs or sequences should be composed together – or *merged* – before antialiasing is performed.

When glyphs are composed together after antialiasing has been performed, that can result in rendering artifacts in some cases in which glyphs touch or overlap. (This is true of any per-primitive antialiasing.) Merging glyphs together before antialiasing eliminates those artifacts, but it also adds a significant performance cost. A font can use the MERG table to indicate *specific* glyph pairs or sequences for which pre-antialias merging is required in order to avoid the risk of rendering artifacts, while implicitly declaring that *other* glyph pairs or sequences *do not* require pre-antialias merging.

NOTE 1 Hereafter, “merging” will be used to refer to composing of glyphs together prior to antialias filtering.

NOTE 2 Some implementations may use caching of glyph-rendering results as a means of performance optimization. If merging of glyph sequences is not required, then the cached renderings may be composed without a need to render glyphs or the glyph sequence again.

The approach used is to give a positive declaration of cases in which merging should be performed. If a 'MERG' table is provided but there no declarations are made for any pairs, then the intended interpretation is that no merging is necessary. (Some implementations may still merge glyphs before antialiasing, however.) If no 'MERG' table is provided, then implementations should always merge glyphs before filtering in order to avoid artifacts.

Data is provided for *pairs* of glyph classes. The first and second glyph elements in a pair correspond to logical ordering of glyphs in a run. Since glyphs are processed in logical order but may be presented in visual left-to-right or right-to-left order, it is possible to give separate recommendations for either left-to-right or right-to-left order.

In some cases, it may be necessary to consider interaction of *sequences* of more than two glyphs rather than simply a pair. The data format allows the font developer to specify sequences that need to be treated together in merging; this is explained further below.

Determination of whether or not merging should be done is a design consideration on the part of the font developer. Glyphs may touch or overlap, but there might not be any perceptible artifacts in visual results. If the designer determines that merging for a particular pair or sequence is not needed to provide adequate visual results, then there will be performance benefits from *not declaring that pair or sequence* as requiring merging.

Note that different platforms may support different rendering techniques, and may or may not support the 'MERG' table. On some platforms, glyphs sequences might always be merged before antialiasing is performed, regardless of whether 'MERG' data is provided in the font. Similarly, in some other platforms, glyphs might always be composed *after* antialiasing is performed, regardless of whether 'MERG' data is provided in the font. Font developers should consult developer documentation for the different platforms on which fonts will be used to determine what benefits a 'MERG' table may provide, and should evaluate rendered results on relevant platforms to determine which glyph pairs or sequences should be declared as requiring merging.

To construct a 'MERG' table, the first step is to classify glyphs based on desired merging behavior such that each glyph has an associated *merge class* (represented by a zero-based index). The system of classification and number of classes will depend on the font and the font developer's discretion, but could take into account such properties as a glyph's general shape and whether it connects to other glyphs. After assigning glyphs into classes, one then selects a recommended merging behavior for each *pair* of merge classes.

NOTE 3 The number of pairs for which data is provided is the square of the number of merge classes. Therefore, the number of merge classes should be as small as possible.

5.7.5.1 Grouping of glyphs

In some cases, a sequence of glyphs may need to be treated as a unit for purposes of merging. For example, a glyph for a combining accent might not typographically interact at all with following glyphs, yet it might come in logical order between glyphs that do interact and that may require merging. This is illustrated in the following Figure 5.4 (assume left-to-right visual order).

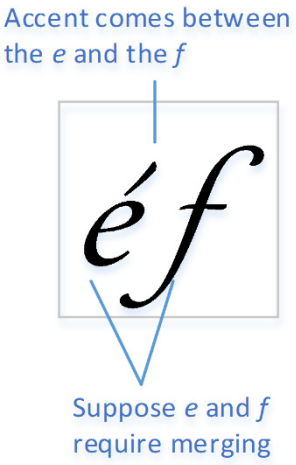


Figure 5.4: Accent glyph logically between two typographically-interacting glyphs

In this case, the accent and the base "e" glyph can be treated as a unit for purposes of evaluating the required merge behavior with the following "f" glyph. Note that, if merging of the "e" and "f" glyphs is required, then the accent will also need to be included in the sequence that gets antialiased together.

The merge data entries that specify how pairs of glyphs should be handled include values that indicate that the pair of glyphs should be *grouped* together as a unit, without specifying whether merging is required. Whether or not merging is needed will get determined only as this group is compared with other glyphs. In the above figure 5.4, the accent is grouped together with the "e" glyph, but whether or not any merging is required will be determined by comparing that combination with the following "f" glyph.

Whenever a pair of glyphs is grouped or merged, then one or the other will be most relevant when the combination is evaluated in relation to the following glyph. In the example above, the required behavior for the "e"-plus-accent combination when it interacts with the following glyph can be determined by the "e". In a different example, it may be the second of a pair of glyphs that is most relevant for purposes of interaction with

subsequent glyphs. Thus, whenever a pair of glyphs is grouped or merged, the data indicates whether the merge class of the sequence takes on the class of the first or the second element of the pair. This is indicated by use of a flag: in the unmarked case (flag is not set), the sequence takes the merge class of the second glyph. But if a *second is subordinate* flag is set, then the sequence takes on the merge class of the first element of the pair.

See the following sections for complete details regarding the merge entry values and how they are processed.

5.7.5.2 Table formats

The 'MERG' table is comprised by a header, a set of class-definition tables, and an array of merge-entry data. The format of the header is as follows:

Merge header

Type	Name	Description
uint16	version	Version number of the merge table — set to 0.
uint16	mergeClassCount	The number of merge classes.
Offset16	mergeDataOffset	Offset to the array of merge-entry data.
uint16	classDefCount	The number of class definition tables.
Offset16	offsetToClassDefOffsets	Offset to an array of offsets to class definition tables — in bytes from the start of the 'MERG' table.

The offsetToClassDefOffsets field provides an offset to the start of an array of offsets. Each element in the array is an offset (unsigned 16-bit) from the start of the 'MERG' table to a class definition table. The classDefCount field gives the number of elements in the offsets array, and the number of class definition tables.

NOTE 1 A given class definition table can be used to assign different glyphs into multiple classes. The number of class definition tables does not determine the number of merge classes. Rather, the mergeClassCount field determines the number of classes that can be referenced by the merge-entry data. Specifically, merge entries are provided for merge classes 0 to mergeClassCount - 1. If any glyph is assigned to a class ID greater than or equal to mergeClassCount, there will be no merge entries for pairs involving that class, which effectively means that merging of that glyph with other glyphs is never required.

The class definition tables use the same formats as are used in OFF Layout tables. Both ClassDefFormat1 and ClassDefFormat2 may be used. For details on class definition table formats, see the Class definition table section of subclause 6.2.

NOTE 2 A class definition table gives an explicit assignment of glyphs to specific class IDs. Any glyph that is not assigned to a class are implicitly assigned to class zero.

Any given glyph must be assigned to at most one class. Moreover, as the class definition tables are read in order, glyph ID references must be in strictly increasing order. If glyph IDs are given out of order, the 'MERG' table is invalid and is ignored.

The merge-entry data array is a 2D table of entries for glyph-class pairs. Each entry is a uint8 value, and the total size of the data is mergeClassCount^2. The data are organized as mergeClassCount number of rows each having mergeClassCount number of column entries.

MergeEntry table

Type	Name	Description
MergeEntryRow	mergeEntryRows [mergeClassCount]	Array of merge-entry rows.

MergeEntryRow record:

Type	Name	Description
uint8	mergeEntries [mergeClassCount]	Array of merge entries.

Each merge entry specifies a behavior for a pair of merge classes: the row index represents the class of the first element, in logical order, and the column index represents the class of the second element.

Each merge entry is a bit field with six flags defined. These describe three different processing behaviors for both left-to-right and right-to-left visual orders. The flags are assigned as follows:

Merge Entry Flags

Mask	Name	Description
0x01	MERGE_LTR	Merge glyphs, for LTR visual order.
0x02	GROUP_LTR	Group glyphs, for LTR visual order.
0x04	SECOND_IS_SUBORDINATE_LTR	Second glyph is subordinate to the first glyph, for LTR visual order.
0x08	Reserved	Flag reserved for future use — set to 0.
0x10	MERGE_RTL	Merge glyphs, for RTL visual order.
0x20	GROUP_RTL	Group glyphs, for RTL visual order.
0x40	SECOND_IS_SUBORDINATE_RTL	Second glyph is subordinate to the first glyph, for RTL visual order.
0x80	Reserved	Flag reserved for future use — set to 0.

The Merge flags (MergeLTR, MergeRTL) indicate that the pair of items should be merged prior to antialiasing.

The Group flags, described in the previous section, indicate that the pair should be treated as a unit, without indicating whether or not merging is required — that will be determine be evaluating the combination in relation to other glyphs.

The SecondIsSubordinate flags, also described in the previous section, are used only if the Merge or Group flag for the same visual order was set. These indicate whether the class for the merged or grouped sequence should be that of the first or second item of the pair. If a SecondIsSubordinate flag is set but neither the Merge or Group flag for the same visual order was set, then it is ignored.

A detailed description of handling of the Merge, Group and SecondIsSubordinate flags is provided in the following subclause.

5.7.5.3 Processing

The merge entries are used while processing glyphs in a glyph run to determine which sequences of glyphs require merging before antialias filtering is performed. The following description is given in a way that is generic with regard to visual order. So, for instance “the Merge flag” refers to the MergeLTR flag if the visual order is LTR, or to the MergeRTL flag if the visual order is RTL.

In the following description, a *merge group* is a sequence of one or more glyphs that are processed as a unit. In addition to the glyph sequence, a merge group has a boolean *mergeRequired* property that is set by default to *false*. The group also has a *mergeClass* property, that is set as described below.

Merge processing proceeds as follows:

1. **Start:** The start state for the processing algorithm is one in which the current glyph did not need to be merged with a preceding glyph or glyph sequence. (This includes the start of a glyph run.) The current glyph is the start of a new merge group with group.mergeRequired = *false*.
2. Determine the merge class of the current glyph. Set the group.mergeClass to this class ID.
3. **Process next glyph:** Determine the merge class of the next glyph.
4. Using group.mergeClass as a row index and the merge class of the next glyph as a column index, retrieve the merge entry for the given row and column.
5. If the merge entry is zero, or if the merge class for either the current or next glyph was greater or equal to mergeClassCount, then the next glyph does not need to be merged into the current merge sequence. Do not add the next glyph into the merge group, but proceed to step 10.
6. Else, if the merge entry has the Merge flag set, then the next glyph is added to the current merge group, and group.mergeRequired is set to *true*. Proceed to step 8.
7. Else, if the merge entry has the Group flag set, then the next glyph is added to the current merge group. The group.mergeRequired property is not changed. Proceed to step 8.
8. Determine the new merge class for the group:
    a. If the merge entry has the SecondIsSubordinate flag set then the group.mergeClass property is not changed.
    b. Else (the SecondIsSubordinate flag is clear), then set the group.mergeClass property to be the merge class of the next glyph.
9. The merge group has been extended; proceed to the next glyph: next becomes current, and the group properties remain as set in steps 6 – 8. Return to step 3.
10. The merge group is terminated:
    a. If group.mergeRequired is *true*, then merge all of the glyphs in this merge group prior to antialias filtering.
    b. Else (group.mergeRequired is *false*), then merging is not required for any of the glyphs in the merge group.
    c. Proceed to next glyph (next becomes current) and return to the start state, step 1.

5.7.6 meta – Metadata table

The metadata table contains various metadata values for the font. Different categories of metadata are identified by four-character tags. Values for different categories can be either binary or text.

5.7.6.1 Table formats

The metadata table begins with a header, structured as follows:

Metadata header

Type	Name	Description
uint32	version	Version number of the metadata table – set to 1.
uint32	flags	Flags – currently unused; set to 0.
uint32	Reserved	Not used; set to 0.
uint32	dataMapsCount	The number of data maps in the table.
DataMap	dataMaps[dataMapsCount]	Array of data map records.

NOTE The reserved field was originally documented in Apple TrueType specification as a data offset. This was redundant, since DataMap records include offsets from the start of the 'meta' table, and therefore not used.

The data map record has the following format:

DataMap record

Type	Name	Description
Tag	tag	A tag indicating the type of metadata.
Offset32	dataOffset	Offset in bytes from the beginning of the metadata table to the data for this tag.
uint32	dataLength	Length of the data, in bytes. The data is not required to be padded to any byte boundary.

The data for a given record may be either textual or binary. The representation format is specified for each tag. Depending on the tag, multiple records for a given tag or multiple, delimited values in a record may or may not be permitted, as specified for each tag. If only one record or value is permitted for a tag, then any instances after the first may be ignored.

5.7.6.2 Metadata Tags

Metadata tags identify the category of information provided and representation format used for a given metadata value. A registry of commonly-used tags is maintained, but private, vendor-determined tags can also be used.

Like other OFF tags, metadata tags are four unsigned bytes that can equivalently be interpreted as a string of four ASCII characters. Metadata tags shall begin with a letter (0x41 to 0x5A, 0x61 to 0x7A) and must use only letters, digits (0x30 to 0x39) or space (0x20). Space characters must only occur as trailing characters in tags that have fewer than four letters or digits.

Privately-defined metadata tags shall begin with an uppercase letter (0x41 to 0x5A), and shall use only uppercase letters or digits. Registered axis tags must not use that pattern, but can be any other valid pattern.

Every registered tag defines the semantics of the associated metadata values, and the representation format of those values. Values for registered tags may be may be either textual or binary. If textual, it will be in UTF-8 encoding unless explicitly indicated otherwise.

The following registered tags are defined or reserved at this time:

Tag	Name		Description
appl	(reserved)		Reserved — used by Apple.
bild	(reserved)		Reserved — used by Apple.
dlng	Design languages	Text, using only Basic Latin (ASCII) characters.	Indicates languages and/or scripts for the user audiences that the font was primarily designed for. Only one instance is used. See below for additional details.
slng	Supported languages	Text, using only Basic Latin (ASCII) characters.	Indicates languages and/or scripts that the font is declared to be capable of supporting. Only one instance See below for additional details.

The values for 'dlng' and 'slng' are comprised of a series of comma-separated ScriptLangTags, which are described in detail below. Spaces may follow the comma delimiters and are ignored. Each ScriptLangTag identifies a language or script. A list of tags is interpreted to imply that all of the languages or scripts are included.

The 'dlng' value is used to indicate the languages or scripts of the primary user audiences for which the font was designed. This value may be useful for selecting default font formatting based on content language, for

presenting filtered font options based on user language preferences, or similar applications involving the language or script of content or user settings.

The 'slng' value is used to declare languages or scripts that the font is capable of supported. This value may be useful for font fallback mechanisms or other applications involving the language or script of content or user settings.

NOTE Implementations that use 'slng' values in a font may choose to ignore Unicode-range bits set in the OS/2 table.

Some examples will help to understand the distinction between design and supported languages:

— Consider the case of accented Latin letters: Although the accents are used in common by a number of languages, the precise shape of the accents can depend on the typographic traditions of a specific language. Polish, for example, prefers steeper accents than French. A font that was designed with accents specifically for Polish would declare Polish as a design language, but might declare support for any language using Latin script.

— Fonts designed for East Asian markets will generally include glyphs for Latin, Greek and Cyrillic because these characters are included in important East Asian character set standards, but using East Asian fonts for languages that are written with those scripts is generally unsatisfactory. Such fonts would therefore include these scripts in the 'slng' value, but not in their 'dlng' value.

— There are some systematic differences in glyph design for the characters shared by simplified and traditional Chinese, such as the way the “bone” radical is drawn in all characters using it. A font specifically designed for use with simplified Chinese can usually be used to display traditional Chinese, but any character with the “bone” radical will look wrong to readers of traditional Chinese. Such a font would include simplified Chinese 'dlng' value, but both simplified and traditional Chinese in its 'slng' value.

5.7.6.3 ScriptLangTag Values

The 'dlng' and 'slng' metadata use ScriptLangTag values, defined here.

A ScriptLangTag denotes a particular language or script associated with a font. These are adapted from the IETF BCP 47 specification [25].

BCP 47 language tags can include various subtags that provide different types of qualifiers, such as language, script or region. In a BCP 47 language tag, a language subtag element is mandatory and other subtags are optional. ScriptLangTag values used for 'dlng' and 'slng' metadata values use a modification of the BCP 47 syntax: a script subtag is mandatory and other subtags are optional. The following augmented BNF syntax, adapted from BCP 47, is used:

```
ScriptLangTag = [language "-"]
                script
                ["-" region]
                *("-" variant)
                *("-" extension)
                ["-" privateuse]
```

The expansion of the elements and the intended semantics associated with each are as defined in BCP 47. Script subtags are taken from ISO 15924. At present, no extensions are defined for use in ScriptLangTags, and any extension will be ignored. Private-use elements, which are prefixed with “-x”, are defined by private agreement between the source and recipient and may be ignored.

Subtags must be valid for use in BCP 47 and contained in the Language Subtag Registry [26] maintained by IANA. See also section 3 of BCP 47 for details.

NOTE OFF Layout script and language system tags are not the same as those used in BCP 47 and should not be referenced when creating or processing ScriptLangTags.

Any ScriptLangTag value not conforming to these specifications is ignored.

A ScriptLangTag can denote fairly specific information; for example, 'en-Latn-IN' would represent "Latin script as used for the English language in India". In most cases, however, generic tags should be used, and it is anticipated that most tags used in 'dlng' and 'slng' metadata declarations will consist only of a script subtag. Language or other subtags can be included, however, and may be appropriate in some cases. Implementations must allow for ScriptLangTags that include additional subtags, but they may also choose to interpret only the script subtag and ignore other subtags.

Examples:

- 'Latn' denotes Latin script (and any language or writing system using Latin script).
- 'Cyrl' denotes Cyrillic script.
- 'sr-Cyrl' denotes Cyrillic script as used for writing the Serbian language; a font that has this property value may not be suitable for displaying text in Russian or other languages written using Cyrillic script.
- 'en-Dsrt' denotes English written with the Deseret script.
- 'Hant' denotes Traditional Chinese.
- 'Hant-HK' denotes Traditional Chinese as used in China.
- 'Jpan' denotes Japanese writing – ISO 15924 defines 'Jpan' as an alias for Han + Hiragana + Katakana.
- 'Kore' denotes Korean writing – ISO 15924 defines 'Kore' as an alias for Hangul + Han.
- 'Hang' denotes Hangul script (exclusively – Hanja are not implied by 'Hang').

The Unicode Standard uses the ISO 15924 identifiers 'Zinh' ("inherited") and 'Zyyy' ("undetermined"). These should not be used in ScriptLangTags. Similarly, 'Zxxx' ("unwritten document") and 'Zzzz' ("unencoded script") should never be used.

On the other hand, 'Zmth' ("Mathematical notation") and 'Zsym' ("Symbols") are not used in the Unicode Standard, yet they may be very useful as declarations in font files. (They were, in fact, added to ISO 15924 for use in relation to fonts.)

In relation to East Asian scripts, a declaration of 'Jpan' can be used to cover hiragana, katakana and kanji. Similarly, 'Kore' can be used to cover Hangul and hanja, though a Korean font with only Hangul support should use 'Hang'. For Chinese fonts, 'Hans' and 'Hant' should normally be used to distinguish between Simplified and Traditional orthographies rather than the more generic declaration 'Hani'. Region-specific variations such as 'Hant-HK' can also be declared. In some cases, it may be appropriate to describe a font capability (but probably not design target) using the generic declaration 'Hani' (denoting 'Han / Hanzi / Kanji / Hanja').

The BCP 47 specification for region subtags allows for continental and sub-continental regions. For example, "039" can be used to denote Southern Europe. Use of such extended-region subtags in ScriptLangTag values is not recommended as software implementations may not have the logic to make appropriate correlations to more specific regions or languages associated with those regions.

5.7.7 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
uint16	majorVersion
uint16	minorVersion
uint32	FontNumber

uint16	Pitch
uint16	xHeight
uint16	Style
uint16	TypeFamily
uint16	CapHeight
uint16	SymbolSet
int8	Typeface[16]
int8	CharacterComplement[8]
int8	FileName[6]
int8	StrokeWeight
int8	WidthType
uint8	SerifStyle
uint8	Reserved (pad)

Major and Minor Version

The current PCLT table version is 1.0.

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 ensure 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 font design units (font design units 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 font design units. 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 font design units. 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 ensure 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 uint8 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.

5.7.8 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**
uint16	version	Version number (0 or 1).
uint16	numRecs	Number of VDMX groups present
uint16	numRatios	Number of aspect ratio groupings
RatioRange	ratRange[numRatios]	Ratio record array
Offset16	Offset[numRatios]	Offset from start of this table to the VDMX group for a corresponding RatioRange record.
Vdmx	groups	The actual VDMX groupings (documented below)

RatioRange Record		
Type	**Name**	**Description**
uint8	bCharSet	Character set (see below).
uint8	xRatio	Value to use for x-Ratio
uint8	yStartRatio	Starting y-Ratio value.
uint8	yEndRatio	Ending y-Ratio value.

Ratios are set up as follows:

For a 1:1 aspect ratio	Ratios.xRatio = 1; Ratios.yStartRatio = 1; Ratios.yEndRatio = 1;
For 1:1 through 2:1 ratio	Ratios.xRatio = 2; Ratios.yStartRatio = 1; Ratios.yEndRatio = 2;
For 1.33:1 ratio	Ratios.xRatio = 4; Ratios.yStartRatio = 3; Ratios.yEndRatio = 3;

For *all* aspect ratios	Ratio.xRatio = 0; Ratio.yStartRatio = 0; Ratio.yEndRatio = 0;

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

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

NOTE 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**

uint16	recs	Number of height records in this group
uint8	startsz	Starting yPelHeight
uint8	endsz	Ending yPelHeight
vTable	entry[recs]	The VDMX records

vTable Record		
Type	**Name**	**Description**
uint16	yPelHeight	yPelHeight to which values apply.
int16	yMax	Maximum value (in pels) for this yPelHeight.
int16	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 int16 and uint16 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.

5.7.9 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 the following fields:

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

Version 1.0 of the vertical header table format is as follows:

Vertical Header Table v1.0

Version 1.0 Type	Name	Description
Fixed	version	Version number of the vertical header table; 0x00010000 for version 1.0
int16	ascent	Distance in font design units from the centerline to the previous line's descent.
int16	descent	Distance in font design units from the centerline to the next line's ascent.
int16	lineGap	Reserved; set to 0
int16	advanceHeightMax	The maximum advance height measurement -in font design units found in the font. This value must be consistent with the entries in the vertical metrics table.
int16	minTop SideBearing	The minimum top sidebearing measurement found in the font, in font design units. This value must be consistent with the entries in the vertical metrics table.
int16	minBottom SideBearing	The minimum bottom sidebearing measurement found in the font, in font design units. This value must be consistent with the entries in the vertical metrics table.
int16	yMaxExtent	Defined as max(tsb + (yMax-yMin))
int16	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.
int16	caretSlopeRun	See the caretSlopeRise field. Value=1 for nonslanted vertical fonts.
int16	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.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	metricDataFormat	Set to 0.
uint16	numOf LongVerMetrics	Number of advance heights in the vertical metrics table.

Version 1.1 of the vertical header table format is as follows:

Vertical Header Table v1.1

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.
int16	vertTypoAscender	The vertical typographic ascender for this font. It is the distance in font design units 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 subclause 6.4.4. Baseline Tags of the OFF Layout Tag Registry for a description of the ideographic em-box.
int16	vertTypoDescender	The vertical typographic descender for this font. It is the distance in font design units 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.
int16	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
int16	advanceHeightMax	The maximum advance height measurement -in font design units found in the font. This value must be consistent with the entries in the vertical metrics table.
int16	minTop SideBearing	The minimum top sidebearing measurement found in the font, in font design units. This value must be consistent with the entries in the vertical metrics table.
int16	minBottom SideBearing	The minimum bottom sidebearing measurement found in the font, in font design units. This value must be consistent with the entries in the vertical metrics table.
int16	yMaxExtent	Defined as max(tsb + (yMax-yMin))
int16	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.
int16	caretSlopeRun	See the caretSlopeRise field. Value=1 for nonslanted vertical fonts.

int16	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.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	metricDataFormat	Set to 0.
uint16	numOf LongVerMetrics	Number of advance heights in the vertical metrics table.

'vhea' Table and OFF Font Variations

In a variable font, various font-metric values within the 'vhea' table may need to be adjusted for different variation instances. Variation data for 'vhea' entries can be provided in the metrics variations ('MVAR') table. Different 'post' entries are associated with particular variation data in the 'MVAR' table using value tags, as follows:

'vhea' entry	Tag
ascent	'vasc'
caretOffset	'vcof'
caretSlopeRun	'vcrn'
caretSlopeRise	'vcrs'
descent	'vdsc'
lineGap	'vlgp'

For general information on OFF Font Variations, see subclause 7.1.

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 font design units.
10/2	2079	advanceHeightMax	The maximum advance height measurement found in the font is 2079 font design units.

12/2	-342	minTopSideBearing	The minimum top sidebearing measurement found in the font is -342 font design units.
14/2	-333	minBottomSideBearing	The minimum bottom sidebearing measurement found in the font is -333 font design units.
16/2	2036	yMaxExtent	max (tsb + (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.

5.7.10 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.

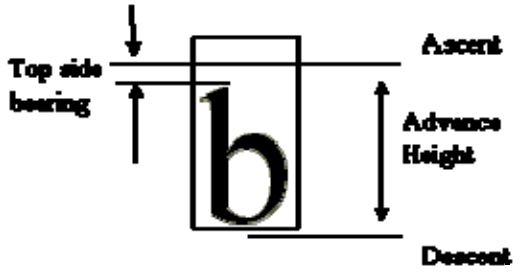


Figure 5.5 – Vertical Metrics

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 shall 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
uint16	advanceHeight	The advance height of the glyph. Unsigned integer in font design units
int16	topSideBearing	The top sidebearing of the glyph. Signed integer in font design units.

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 shall 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 shall 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
int16	topSideBearing[]	The top sidebearing of the glyph. Signed integer in font design units.

5.7.11 COLR – Color Table

The COLR table adds support for multi-colored glyphs in a manner that is compatible with existing text engines and easy to support with current OFF font files.

The COLR table defines a list of base glyphs — which are regular glyphs, typically associated with a cmap entry. Each base glyph is associated by the COLR table to a list of glyphs, each corresponding to layers that can be combined, creating a colored representation of the base glyph. The layered glyphs are explicitly defined in bottom-up z-order and each of their advance widths must match those of the base glyph. If the font has vertical metrics, the associated layer glyphs must also have the same advance height and vertical Y origin as the base glyph.

The COLR table works together with the CPAL table which holds the color palettes used by the color layers.

Fonts using 'COLR' and 'CPAL' tables shall implement glyph ID 1 as the .null glyph. If the COLR table is present in a font but no CPAL table exists, then the COLR table will not be supported for this font.

Header

The table starts with a fixed portion describing the overall setup for the color font records. All offsets, unless otherwise noted, will be from the beginning of the table

Type	Name	Description
uint16	version	Table version number (starts at 0).
uint16	numBaseGlyphRecords	Number of Base Glyph Records.
Offset32	baseGlyphRecordOffset	Offset (from beginning of COLR table) to Base Glyph records.
Offset32	layerRecordOffset	Offset (from beginning of COLR table) to Layer Records.
uint16	numLayerRecords	Number of Layer Records.

Base Glyph Record

The header of the COLR table points to the base glyph record. This record is used to match the base glyph to the layered glyphs. Each base glyph record contains a base glyph index. This is usually the glyph index that is referenced in the cmap table. The number of layers is used to indicate how many color layers will be used for this base glyph. Each record then has an index to a glyph layer record. There will be numLayers of layer records for each base glyph. The firstLayerIndex refers to the lowest z-order, or bottom, glyph id for the colored glyph. The next layer record will represent the next highest glyph in the z-order, and this continues bottom-up until it reaches the numLayers glyph at the top of the z-order. The index is relative to the start of the Layer Records. The index does not have to be unique for each base glyph ID. If two base glyphs need to share the color glyphs and palette indices, this is acceptable. Likewise a Base Glyph Record could point partway into a z-order of another base glyph.

The base glyph records are sorted by glyph id. It is assumed that a binary search can be used to efficiently access the glyph IDs that have color values. Any use scenario that attempts to map glyphs to character codes must be aware of the mapping done by the COLR table.

Type	Name	Description
uint16	gID	Glyph ID of reference glyph. This glyph is for reference only and is not rendered for color.
uint16	firstLayerIndex	Index (from beginning of the Layer Records) to the layer record. There will be numLayers consecutive entries for this base glyph.
uint16	numLayers	Number of color layers associated with this glyph.

Layer Record

Type	Name	Description
uint16	gID	Glyph ID of layer glyph (must be in z-order from bottom to top).
uint16	paletteIndex	Index value to use with a selected color palette. This value must be less than numPaletteEntries in the CPAL table. A palette entry index value of 0xFFFF is a special case indicating that the text foreground color (defined by a higher-level client) should be used and shall not be treated as actual index into CPAL ColorRecord array.

The selection of color palette to be used for a given layer record is the responsibility of a higher-level client. With CPAL version 0 – the palette selection needs to be made based on the information distributed with a font. CPAL version 1 offers user-selectable color palettes based on a textual descriptions of palette entries and palette labels.

5.7.12 CPAL – Palette Table

The palette table is a set of one or more palettes, each containing a predefined number of color records with BGRA values. It may also contain name table IDs describing the palettes and their entries.

Palettes are defined by a set of color records. All palettes have the same number of color records, specified by numColorRecords. All color records for all palettes are arranged in a single array, and the color records for any given palette are a contiguous sequence of color records within that array. The first color record of each palette is provided in the colorRecordIndices array.

Multiple colorRecordIndices may refer to the same color record, in which case multiple palettes would use the same color records; hence the number of functionally-distinct palettes may be fewer than the numPalettes entry. Also, the sequence of color records for different palettes may overlap, with certain color records shared between multiple palettes. Thus, the total number of color records in the CPAL table may be less than the number of palette entries multiplied by the number of palettes.

The first palette, number 0, is the default palette. A minimum of one palette must be provided in the CPAL table if the table is present. Palettes shall have a minimum of one color record. An empty CPAL table, with no palettes and no color records is not permitted.

Colors within a palette are referenced by base-zero index. The number of colors in each palette is given by numPaletteEntries. The number of color records in the color records array (numColorRecords) shall be greater than or equal to max(colorRecordIndices) + numPaletteEntries.

Palette Table Header

The CPAL table begins with a header that starts with a version number. Currently, only versions 0 and 1 are defined.

CPAL version 0

The CPAL header version 0 is organized as follows:

Type	Name	Description
uint16	version	Table version number (=0).
uint16	numPaletteEntries	Number of palette entries in each palette.
uint16	numPalettes	Number of palettes in the table.
uint16	numColorRecords	Total number of color records, combined for all palettes.
Offset32	offsetFirstColorRecord	Offset from the beginning of CPAL table to the first ColorRecord in the Color Records Array.
uint16	colorRecordIndices[numPalettes]	Index of each palette's first color record in the Color Records Array.

CPAL version 1

The CPAL header version 1 adds three additional fields to the end of the table header and is organized as follows:

Type	Name	Description
uint16	version	Table version number (=1).
uint16	numPaletteEntries	Number of palette entries in each palette.
uint16	numPalettes	Number of palettes in the table.
uint16	numColorRecords	Total number of color records, combined for all palettes.
Offset32	offsetFirstColorRecord	Offset from the beginning of CPAL table to the first ColorRecord in the Color Records Array.
uint16	colorRecordIndices[numPalettes]	Index of each palette's first color record in the Color Records Array.
Offset32	offsetPaletteTypeArray	Offset from the beginning of CPAL table to the Palette Type Array. Set to 0 if no array is provided.
Offset32	offsetPaletteLabelArray	Offset from the beginning of CPAL table to the Palette Label Array. Set to 0 if no array is provided.
Offset32	offsetPaletteEntryLabelArray	Offset from the beginning of CPAL table to the Palette Entry Label Array. Set to 0 if no array is provided.

Palette Entries and Color Records

Colors defined in the CPAL table are referenced by a palette index plus a palette-entry index. Indices are base zero. For a given palette index and palette-entry index, an entry within the color records array is derived: colorRecordIndex = colorRecordIndices[paletteIndex] + paletteEntryIndex.

The color records array is comprised of color records:

Type	Name	Description
ColorRecord	colorRecords[numColorRecords]	Color records for all palettes.

Each color record has BGRA values. The color space for these values is sRGB.

Type	Name	Description
uint8	blue	Blue value (B0).
uint8	green	Green value (B1).
uint8	red	Red value (B2).
uint8	alpha	Alpha value (B3).

The colors in the Color Record should not be pre-multiplied, and the alpha value should be explicitly set for each palette entry.

When placing and registering overlapping elements, there is the possibility of “seaming”, where the edge rendering of one element interferes with another element. This may be more or less visible based on the contrast of the colors used.

Palette Type Array

Type	Name	Description
uint32	paletteTypes [numPalettes]	Array of 32-bit flag fields that describe properties of each palette. See below for details.

The following flags are defined:

Mask	Name	Description
0x0001	USABLE_WITH_LIGHT_ BACKGROUND	Bit 0: palette is appropriate to use when displaying the font on a light background such as white.
0x0002	USABLE_WITH_DARK_ BACKGROUND	Bit 1: palette is appropriate to use when displaying the font on a dark background such as black.
0xFFFC	Reserved	Reserved for future use — set to 0.

Note that the usableWithLightBackground and usableWithDarkBackground flags are not mutually exclusive: they may both be set.

Palette Label Array

Type	Name	Description
uint16	paletteLabels [numPalettes]	Array of name table IDs (typically in the font-specific name ID range) that specify user interface strings associated with each palette. Use 0xFFFF if no name ID is provided for a particular palette.

Palette Entry Label Array

Type	Name	Description
uint16	paletteEntryLabels [numPaletteEntries]	Array of name table IDs (typically in the font-specific name ID range) that specify user interface strings associated with each palette entry, e.g. “Outline”, “Fill”. This set of palette entry labels applies to all palettes in the font. Use 0xFFFF if no name ID is provided for a particular palette entry.

Relationship to COLR and SVG Tables

Both the COLR and SVG tables can use CPAL to define their palettes.

COLR and CPAL

In fonts that have COLR table, the CPAL table is required, and contains all the font-specified colors used by multicolored glyphs.

As noted in the COLR table description, the palette entry index of 0xFFFF if specified in the COLR table represents the foreground color used in the system. This special value does not change across multiple palettes. The maximum palette entry index is 65535 – 1, as the 65536th position is used in the COLR table to indicate the foreground font color.

SVG and CPAL

In fonts that have an SVG table, the CPAL table can be used to contain the values of any color variables used by the SVG glyph descriptions in the SVG table. SVG glyph descriptions can also include color specifications directly, however. Thus, the CPAL table is optional for fonts with an SVG table.

Foreground color and foreground color opacity are expressed by the context-fill and context-fill-opacity attributes in the SVG glyph descriptions.

When used with an SVG table, the default palette’s colors must be set to the same values as the default values for the color variables in the SVG glyph descriptions; this is for text engines that support the SVG table but not color palettes. The SVG glyph descriptions are able to express their own explicit or “hard-coded” colors as well. These are not related to color variables and thus do not vary by palette selection. See subclause 5.5.1 for more details.

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.

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

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 6.1 – 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.

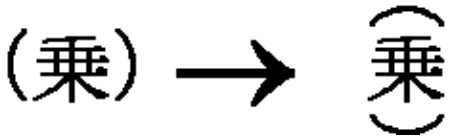


Figure 6.2 – Alternative forms of parentheses used when positioning Kanji vertically

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 6.3 – Two Latin glyphs and their associated ligature

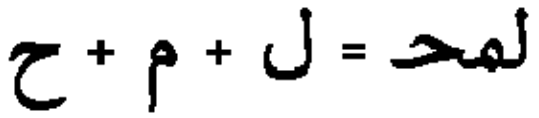


Figure 6.4 – Three Arabic glyphs and their associated ligature

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 6.5 – 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 6.6 – 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 6.7 – 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.

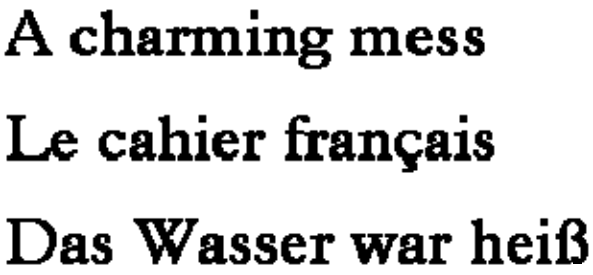


Figure 6.8 – Differences in the English, French, and German language system

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.

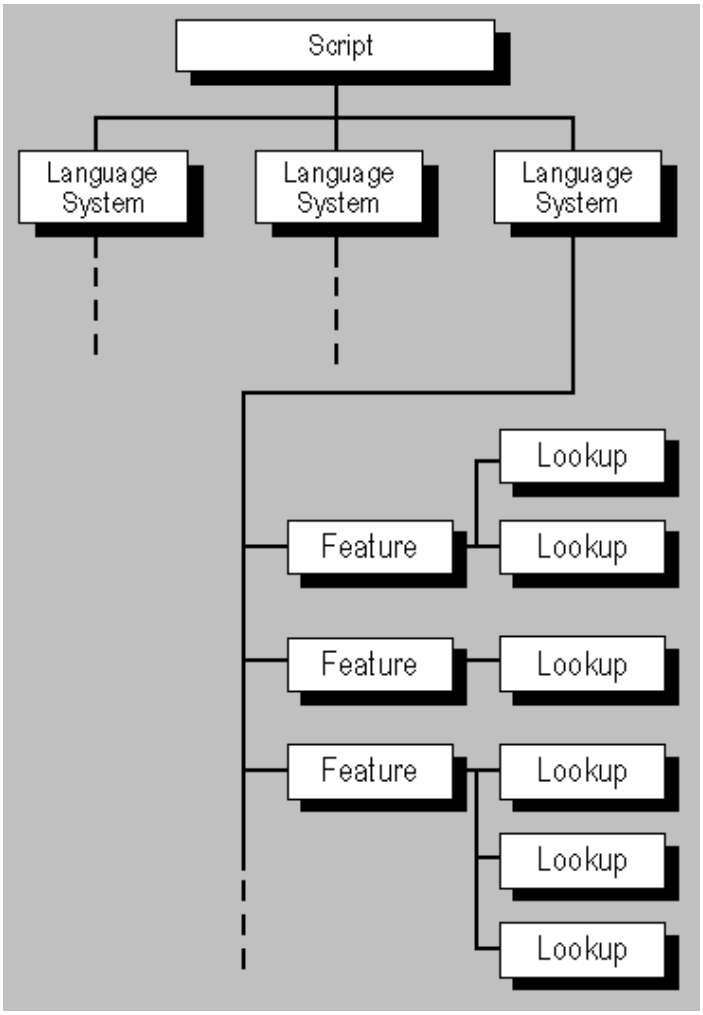


Figure 6.9 – The relationship of scripts, language systems, features, and lookups for substitution and positioning tables

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 C), and cmap(*j*) refers to the glyph mapped from 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.

6.1.5 OFF layout and Font variations

OFF Font variations allow a single font to support many design variations along one or more axes of design variation. For example, a font with weight and width variations might support weights from thin to black, and widths from ultra-condensed to ultra-expanded. For general information on OFF Font Variations, see subclause 7.1.

Mechanisms used to support font variations are integrated into the tables used for OFF layout. Variation of glyph outlines and metrics across a font's variation space can impact the design-grid distances that get used in OFF layout tables, such as anchor positions used in a GPOS attachment lookup. Enhancements to OFF layout formats allow the default values found in existing formats to be associated with variation data that describes how the given value is adjusted for different variation instances.

In some variable fonts, it may be desirable to have different glyph-substitution or glyph-positioning actions used for different regions within the font's variation space. For example, for narrow or heavy instances in which counters become small, it may be desirable to make certain glyph substitutions to use alternate glyphs with certain strokes removed or outlines simplified to allow for larger counters. Such effects can be achieved using a feature variations table within either the GSUB or GPOS table. The feature variations table is described in subclause 6.2. See also the "Required variation alternates ('rvrn')" feature in the OFF layout tag registry.

Different variation instances of a variable font have the same glyph IDs. For that reason, it might seem possible for lookups to be applied across a glyph sequence in which glyphs are formatted using different variation instances of a variable font. Doing so, however, could lead to unpredictable behaviors since font developers may not have sufficient control over how lookup tables are generated, and it would not be feasible to test the vast number of possible cross-instance interactions. For these reasons, layout processing implementations must treat different variation instances of a variable font as distinct style runs for purposes of OFF Layout processing.

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 6.10). 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 6.10 – 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 6.11). And the Arabic script contains different glyphs for writing the Farsi and Urdu languages. In OFF Layout, language systems are defined within scripts.

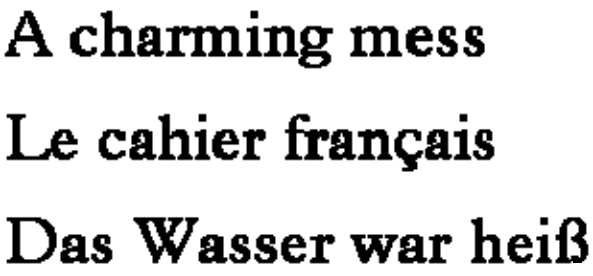


Figure 6.11 – Differences in the English, French, and German language systems

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 6.12). 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.

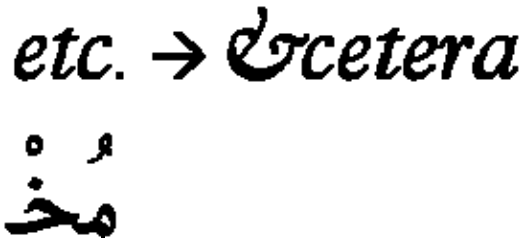


Figure 6.12 – A ligature glyph feature substitutes the <etc> ligature for individual glyphs, and a mark feature positions diacritical marks above an Arabic ligature glyph

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.

A font may also include FeatureVariations data within a GPOS or GSUB table that allows the default lookup data associated with a feature to be substituted by alternate lookup data when particular conditions apply. Currently, this mechanism is used only for variable fonts using OFF Font Variations.

6.2.2 OFF layout and Font variations

OFF Font variations allow a single font to support many design variations along one or more axes of design variation. For example, a font with weight and width variations might support weights from thin to black, and widths from ultra-condensed to ultra-expanded. For general information on OFF Font variations, see subclause 7.1.

When different variation instances are selected, the design and metrics of individual glyphs changes. This can impact font-unit values given in GPOS, BASE, JSTF or GDEF tables, such as the X and Y coordinates of an attachment anchor position. The font-unit values given in these tables apply to the default instance of a variable font. If adjustments are needed for different variation instances, this is done using variation data with processes similar to those used for glyph outlines and other font data, as described in subclause 7.1. The variation data for GPOS, JSTF or GDEF values is contained in an *ItemVariationStore* table which, in turn, is

contained within the GDEF table; variation data for BASE values is contained in an ItemVariationStore table within the BASE table itself. The format of the ItemVariationStore is described in detail in the subclause 7.2. For font-unit values within the GPOS, BASE, JSTF or GDEF tables that require variation, references to specific variation data within the ItemVariationStore are provided in *VariationIndex tables*, described below.

In some variable fonts, it may be desirable to have different glyph-substitution or glyph-positioning actions used for different regions within the font's variation space. For example, for narrow or heavy instances in which counters become small, it may be desirable to make certain glyph substitutions to use alternate glyphs with certain strokes removed or outlines simplified to allow for larger counters. Such effects can be achieved using a *FeatureVariations* table within either the GSUB or GPOS table. The FeatureVariations table is described below.

6.2.3 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, a LookupList table, and a FeatureVariations table. In GSUB, the tables define glyph substitution data. In GPOS, they define glyph positioning data. This subclause 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.

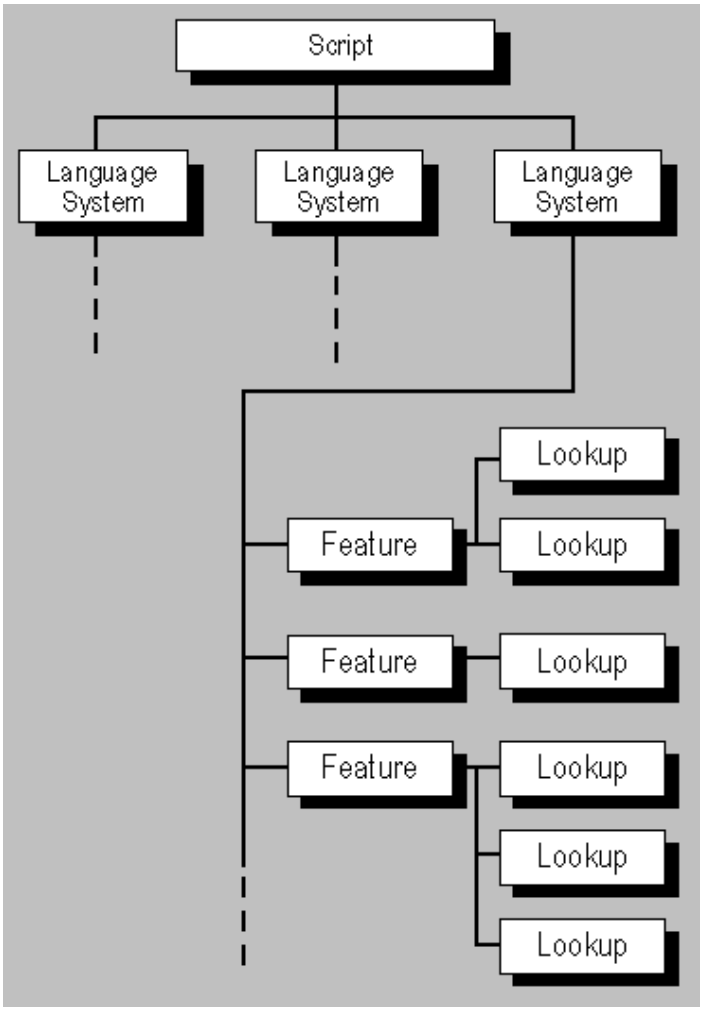


Figure 6.13 – The relationship of scripts, language systems, features, and lookups for substitution and positioning tables

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.

In non-variable fonts, GPOS lookup subtables may also contain Device tables to adjust scaled contour glyph coordinates for particular output sizes and resolutions. Device tables can also be used for similar adjustments to baseline metric or caret offset values in the BASE and GDEF tables. Similarly, in variable fonts, GPOS lookup subtables, BaseCoord tables and CaretValue tables may contain VariationIndex tables that reference variation data to adjust font-unit values as may be needed for different variation instances within a font's design variation space. Device and VariationIndex tables are described in this subclause.

As mentioned above, a feature table references a set of lookups in the lookup list. The FeatureVariations table allows the default set of lookups used for a given feature to be substituted by a different set of lookups under particular conditions. This can be used in variable fonts to provide different substitution or positioning actions for different variation instances. For example, for narrow or heavy instances in which counters become small, it may be desirable to make certain glyph substitutions to use alternate glyphs with certain strokes removed or outlines simplified to allow for larger counters.

6.2.4 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). Each ScriptRecord consists of a ScriptTag that identifies a script, and an offset to a Script table. The ScriptRecord array is stored in alphabetic order of the script tags.

A Script table with the script tag 'DFLT' (default) may be used in a font to define features that are not script-specific. An application should use a 'DFLT' script table if there is not a script table associated with the specific script of the text being formatted, or if the text does not have a specific script (for example, it contains only symbols or punctuation).

NOTE If symbols or punctuation have a Unicode script property "Common" but are used together with characters of a specific script, features that apply to those symbol or punctuation characters should not necessarily be organized

under the 'DFLT' script, but can be organized under the specific script. Applications may process script-neutral characters together with immediately-preceding or following script-specific characters for better processing efficiency. In that case, an application would look for features that operate on the neutral characters by using the Script table for the specific script. The 'DFLT' script would still be used if the text contained only the neutral characters, however.

If there is a 'DFLT' script table, it must have a non-NULL DefaultLangSys value, which provides the offset to a default Language System table (described below). As languages are written using particular scripts, it is normally expected that language-specific typographic effects will be associated with the particular script, not with the generic 'DFLT' script. For this reason, the LangSysCount value of a 'DFLT script table should normally be 0 (no non-default language system tables). However, a font is permitted to have a 'DFLT' script table with non-default language system tables, and an application may use features associated with one of these if the the 'DFLT' script table is applicable (no script table is present for the specific script, or there is no specific script in the text context), and if one of the particular language systems is specified. Applications should support use of a non-default language system table that is associated with 'DFLT' script, though some applications might not do so.

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	scriptRecords [scriptCount]	Array of ScriptRecords, listed alphabetically by script tag

ScriptRecord

Type	Name	Description
Tag	scriptTag	4-byte script tag identifier
Offset16	scriptOffset	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
Offset16	defaultLangSys	Offset to default LangSys table, from beginning of Script table – may be NULL
uint16	langSysCount	Number of langSysRecords for this script – excluding the DefaultLangSys
struct	langSysRecords [langSysCount]	Array of LangSysRecords, listed alphabetically by LangSys tag

LangSysRecord

Type	Name	Description
Tag	langSysTag	4-byte LangSysTag identifier
Offset16	langSysOffset	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
Offset16	lookupOrder	= NULL (reserved for an offset to a reordering table)
uint16	requiredFeatureIndex	Index of a feature required for this language system; if no required features = 0xFFFF
uint16	featureIndexCount	Number of feature index values for this language system – excludes the required feature
uint16	featureIndices[featureIndexCount]	Array of indices into the FeatureList, in arbitrary order

6.2.5 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 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 shall 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
FeatureRecord	featureRecords[featureCount]	Array of FeatureRecords – zero-based (first feature has FeatureIndex = 0), listed alphabetically by feature tag

FeatureRecord

Type	Name	Description
Tag	featureTag	4-byte feature identification tag
Offset16	featureOffset	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, see subclause 6.4.3 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
Offset16	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	lookupIndexCount	Number of LookupList indices for this feature
uint16	lookupListIndices [lookupIndexCount]	Array of indices into the LookupList – 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
Offset16	lookups[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 field 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
Offset16	subTableOffsets [subTableCount]	Array of offsets to lookup subtables, from beginning of Lookup table
uint16	markFilteringSet	Index (base 0) into GDEF mark glyph sets structure. This field is present only if bit useMarkFilteringSet of lookup flags 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

Value	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. NOTE 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.

6.2.6 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 glyph array
uint16	glyphArray[glyphCount]	Array of glyph IDs – 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 (startGlyphID), an end glyph index (endGlyphID), and the Coverage Index associated with the range's Start glyph. Ranges shall be in glyph ID 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 (endGlyphID - startGlyphID + 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 - startGlyphID.

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	rangeRecords [rangeCount]	Array of glyph ranges – ordered by startGlyphID

RangeRecord

Type	Name	Description
uint16	startGlyphID	First glyph ID in the range
uint16	endGlyphID	Last glyph ID in the range
uint16	startCoverageIndex	Coverage Index of first glyph ID in range

6.2.7 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 IDs covered by the table is identified by two values: the glyph ID of the first glyph (StartGlyphID), and the number of consecutive glyph IDs (including the first one) that will be assigned class values (GlyphCount). The ClassValueArray lists the class value assigned to each glyph ID, starting with the class value for StartGlyphID and following the same order as the glyph IDs. Any glyph not included in the range of covered glyph IDs 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
uint16	startGlyphID	First glyph ID of the classValueArray
uint16	glyphCount	Size of the classValueArray
uint16	classValueArray[glyphCount]	Array of Class Values – one per glyph ID

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 glyph ID 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
ClassRangeRecord	classRangeRecords [classRangeCount]	Array of ClassRangeRecords – ordered by startGlyphID

ClassRangeRecord

Type	Name	Description
uint16	startGlyphID	First glyph ID in the range
uint16	endGlyphID	Last glyph ID in the range
uint16	class	Applied to all glyphs in the range

6.2.8 Device and VariationIndex tables

Device tables and VariationIndex tables are used to provide adjustments to font-unit values in GPOS, JSTF, GDEF or BASE tables, such as the X and Y coordinates of an attachment anchor position. Device tables are used only in non-variable fonts. VariationIndex tables are used only in variable fonts and are a variant format of the Device table. When values require adjustment data, the table containing that value will also include an offset to a Device table or VariationIndex table.

NOTE Because the same fields are used to provide an offset to a Device table or an offset to a VariationIndex table, Device tables and VariationIndex tables cannot both be used for a given positioning value. Device tables should only be used in non-variable fonts; VariationIndex tables can only be used in variable fonts.

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, particularly at small PPEM sizes. 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 in non-variable fonts, 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 Device table 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}.

In variable fonts, X or Y font-unit values in GPOS, JSTF, or GDEF data may require adjustment for different variation instances within a font's variation space. The variation data for this is contained in an ItemVariationStore table contained within the GDEF table. Similarly, values in a BASE table may require adjustment, and the variation data for this is contained in an ItemVariationStore table within the BASE table. The format of the ItemVariationStore is described in detail in subclause 7.2. It contains a number of delta values organized into sets that are referenced using a *delta-set index*. Data stored outside the ItemVariationStore provides delta-set indices for each of target items requiring variation. Within the GPOS, JSTF, GDEF and BASE tables, delta-set indices are stored in VariationIndex tables.

The Device and VariationIndex tables contain a DeltaFormat field that identifies the format of data contained. Format values 0x0001 to 0x0003 are used for Device tables, and indicate the format of delta adjustment values contained directly within the device table: signed 2-, 4,- or 8-bit values. A format value of 0x8000 is used for the VariationIndex table, and indicates that a delta-set index is used to reference delta data in an ItemVariationStore table.

DeltaFormat values

Type	Name	Description
0x0001	LOCAL_2_BIT_DELTAS	Signed 2-bit value, 8 values per uint16
0x0002	LOCAL_4_BIT_DELTAS	Signed 4-bit value, 4 values per uint16
0x0003	LOCAL_8_BIT_DELTAS	Signed 8-bit value, 2 values per uint16
0x8000	VARIATION_INDEX	VariationIndex table, contains a delta-set index pairs.
0x7FFC	Reserved	For future use – set to 0.

The Device table includes an array of uint16 values (DeltaValue) that stores the adjustment delta values in a packed representation. The 2-, 4-, or 8-bit signed values are packed into uint16 values starting with the 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.

A single Device table provides delta information for one target value at a range of sizes. The DeltaValue array lists the number of pixels to adjust specified X or Y values 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.

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: 0x0001, 0x0002, or 0x0003
uint16	deltaValue[ ]	Array of compressed data

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

In a variable font, the ItemVariationStore table uses a two-level organization for variation data: a store can have multiple *Item Variation Data* subtables, and each subtable has multiple delta-set rows. A delta-set index is a two-part index: an outer index that selects a particular item variation data subtable, and an inner index that selects a particular delta-set row within that subtable. A VariationIndex table specifies both the outer and inner portions of the delta-set index.

VariationIndex table

Type	Name	Description
uint16	deltaSetOuterIndex	A delta-set outer index – used to select an item variation data subtable within the item variation store.
uint16	deltaSetInnerIndex	A delta-set inner index – used to select a delta-set row within an item variation data subtable.
uint16	deltaFormat	Format, = 0x8000

Note that the VariationIndex table is shorter than the Device table since it does not directly contain an array of delta data. Its format is similar to a Device table with an empty delta array. When applications get an offset to a Device or VariationIndex table, they should begin by reading the first three fields and then testing the DeltaFormat field to determine the interpretation of the first two fields and whether there is additional data to read.

6.2.9 Feature variations

FeatureVariations Table

A feature variations table describes variations on the effects of features based on various conditions. That is, it allows the default set of lookups for a given feature to be substituted with alternates of lookups under particular conditions.

The feature list provides an array of feature tables and associated feature tags, and a LangSys table identifies a particular set of the feature-table/tag pairs that will be supported for a given script and language system. The feature tables specified in a LangSys table are used by default when current conditions do not match any of the conditions for variation defined in the feature variations table. Those defaults will also be used under all conditions in implementations that do not support the feature variations table.

The feature variations table has an array of condition records, each of which references a set of conditions (a *condition set* table), and a set of alternate feature tables to use when a runtime context matches the condition set.

The substitutions given are replacements of one feature table for another. The alternate feature tables are appended at the end of the feature variations table, and are not included in the feature list table. Hence, there are no feature records in the feature list table that correspond to the alternate feature tables. An alternate feature table maintains the same feature tag association as the default feature table. Also, whereas the default feature tables in the feature list table are referenced using 16-bit offsets, the alternate feature tables are referenced using 32-bit offsets within the feature variations table.

When processing text, a default set of feature tables, each with an associated feature tag, is obtained from a LangSys table for a given script and language system. Condition sets are evaluated in order, testing for a condition set that matches the current runtime context. When the first match is found, the corresponding feature table substitution table is used to revise the set of feature tables obtained by default via the LangSys table, as described below (see FeatureTableSubstitution table).

The format of the feature variations table is as follows:

FeatureVariations Table

Type	Name	Description
uint16	majorVersion	Major version of the FeatureVariations table – set to 1.
uint16	minorVersion	Minor version of the FeatureVariations table – set to 0.
uint32	featureVariationRecordCount	Number of feature variation records.
FeatureVariationRecord	featureVariationRecords [featureVariationRecordCount]	Array of feature variation records.

A feature variation record has offsets to a condition set table and to a feature table substitution table.

If the ConditionSet offset is 0, there is no condition set table. This is treated as the universal condition: all contexts are matched.

If the FeatureTableSubstitution offset is 0, there is no feature table substitution table, and no substitutions are made.

Feature variation records shall be ordered in the order of precedence for the condition sets. During processing, the feature variation records will be read, and the corresponding condition sets tested, in the order in which they occur. If the condition set for a given record does not match the runtime context, then the next record is checked. The first feature variation record for which the condition set matches the runtime context will be considered as a candidate: if the version of the FeatureTableSubstitution table is supported, then this feature variation record will be used, and no additional feature variation records will be considered. If the version of the FeatureTableSubtitution table is not supported, then this feature variation record is rejected and processing will move to the next feature variation record.

FeatureVariationRecord

Type	Name	Description
Offset32	conditionSet	Offset to a condition set table, from beginning of FeatureVariations table.
Offset32	featureTableSubstitutionOffset	Offset to a feature table substitution table, from beginning of the FeatureVariations table.

ConditionSet Table

A condition set table specifies a set of conditions under which a feature table substitution is to be applied. A condition set may specify conditions related to various factors; currently, one type of factor is supported: the variation instance of a variable font. Individual conditions are represented in subtables, which may use different formats according to the nature of the factor defining the condition.

For a given condition set, conditions are conjunctively related (boolean AND): all of the specified conditions must be met in order for the associated feature table substitution to be applied. A condition set does not need to specify conditional values for all possible factors. If no values are specified for some factor, then the condition set matches all runtime values for that factor.

If a given condition set contains no conditions, then it matches all contexts, and the associated feature table substitution is always applied, unless there was a FeatureVariation record earlier in the array with a condition set matching the current context.

ConditionSet Table

Type	Name	Description
uint16	conditionCount	Number of conditions for this condition set.
uint32	conditions[conditionCount]	Array of offsets to condition tables, from beginning of the ConditionSet table.

Condition Table

The condition table describes a particular condition. Different formats for the condition table may be defined, with each format used for a particular kind of condition qualifier. Currently, one format is defined: ConditionTableFormat1, which is used to specify a value range for a variation axis value in a variable font.

New formats for other condition qualifiers may be added in the future, in which case the version of the ConditionSet table will be updated. If a layout engine supports a particular version of the condition set table and encounters a font with a later-version condition set table, it should fail to match the condition set whenever an unrecognized condition format is encountered. In this way, new condition formats can be defined and used in fonts that can work in a backward-compatible way in existing implementations. Therefore, introduction of new condition formats will typically result in minor version updates to the ConditionSet table.

Condition Table Format 1: Font Variation Axis Range

A font variation axis range condition refers to a range of values for a design variation axis in a variable font. The axes of variation are specified in the font variations ('fvar') table of a font. If a format 1 condition table is used, there must be an fvar table in the font, and the AxisIndex value (which is zero-based) must be less than the axisCount value in the fvar table. If the AxisIndex is invalid, the feature variation record containing this condition table is ignored.

A format 1 condition table specifies a matching range of variation instance values along a single axis. Absence of a format 1 condition for a given variation axis implies that that axis is not a factor in determining applicability of the condition set.

The fvar table defines a range of valid values for each variation axis. During processing for a particular variation instance, a normalization process is applied that maps user values in the range defined within the fvar table to a normalized scale with a range from -1 to 1. The values specified in a format 1 condition table are expressed in terms of the normalized scale, and so can be any value from -1 to 1.

A font variation axis range condition is met if the currently-selected variation instance has a value for the given axis that is greater than or equal to the FilterRangeMinValue, and that is less than or equal to the FilterRangeMaxValue.

ConditionTableFormat1

Type	Name	Description
uint16	Format	Format, = 1
uint16	AxisIndex	Index (zero-based) for the variation axis within the 'fvar' table.
F2DOT14	FilterRangeMinValue	Minimum value of the font variation instances that satisfy this condition.
F2DOT14	FilterRangeMaxValue	Maximum value of the font variation instances that satisfy this condition.

FeatureTableSubstitution Table

A feature table substitution table describes a set of feature table substitutions to be applied when the corresponding condition set matches the current runtime context. These substitutions are represented using an array of feature table substitution records. Each record gives a simple substitution of one feature table for another. When checking for a particular feature index, the first record having that index is matched, and searching ends if a record is encountered with a higher index value.

Note that the records shall be ordered in increasing order of the FeatureIndex values, and no two records may have the same FeatureIndex value.

FeatureTableSubstitution table

Type	Name	Description
uint16	majorVersion	Major version of the feature table substitution table – set to 1
uint16	minorVersion	Minor version of the feature table substitution table – set to 0.
uint16	substitutionCount	Number of feature table substitution records.
FeatureTableSubstitutionRecord	substitutions [substitutionCount]	Array of feature table substitution records.

FeatureTableSubstitutionRecord:

Type	Name	Description
uint16	featureIndex	The feature table index to match.
Offset32	alternateFeatureTable	Offset to an alternate feature table, from start of the FeatureTableSubstitution table.

As described above, condition sets are evaluated and may be selected for processing of the associated feature table substitution table to replace a default feature table obtained from a LangSys table with an alternate feature table. Given a default array of feature tables for selected features obtained from the LangSys table, substitution of alternate feature tables can be done as follows:

1. For each feature index, evaluate the FeatureTableSubstitutionRecords in order.
2. If a matching record is encountered (FeatureIndex = the current feature index), then replace the feature table for that feature index using the alternate feature table at the offset given in the record. Stop processing for that feature index.
3. If a record is encountered with a higher feature index value, stop searching for that feature index; no substitution is made.

6.2.10 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
	scriptRecords[0]	In alphabetical order by script tag.
68616E69	'hani'	scriptTag, Han Ideographic script
0014	HanIScriptTable	offset to Script table
	scriptRecords[1]	In alphabetical order by script tag.
6B616E61	'kana'	scriptTag, Hiragana and Katakana scripts
0018	KanaScriptTable	offset to Script table
	scriptRecords[2]	In alphabetical order by script tag.
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
	langSysRecords[0]	In alphabetical order by LangSys tag.
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	requiredFeatureIndex, no required features
0003	3	featureIndexCount
0000	0	featureIndices[0], in arbitrary order 'init' feature (initial glyph)
0001	1	featureIndices [1], 'fina' feature (final glyph)
0002	2	featureIndices [2], for 'medi' feature (medial glyph)
	LangSys UrduLangSys	LangSys table definition
0000	NULL	lookupOrder, reserved, null
0003	3	requiredFeatureIndex, numeral subsitution in Urdu
0003	3	featureIndexCount
0000	0	featureIndices [0], in arbitrary order 'init' feature (initial glyph)
0001	1	featureIndices [1], 'fina' feature (final glyph)
0002	2	featureIndices [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
	featureRecords[0]	
6C696761	'liga'	featureTag
0014	TurkishLigatures	offset to Feature table, FflFfFlLiga
	featureRecords[1]	
6C696761	'liga'	featureTag
001A	DefaultLigatures	offset to Feature table, FfiFiLiga, FflFfFlLiga
	featureRecords[2]	
6C696761	'liga'	featureTag
0022	GermanLigatures	Offset to Feature table, EszetLiga
	Feature TurkishLigatures	Feature table definition
0000	NULL	featureParams, reserved, null
0001	1	lookupIndexCount
0000	1	lookupListIndices[1], ffl, fl, ff ligature substitution Lookup
	Feature DefaultLigatures	Feature table definition
0000	NULL	featureParams - reserved, null
0002	2	lookupIndexCount
0000	0	lookupListIndices [0], in arbitrary order, ffi, fi ligatures
0001	1	lookupListIndices [1], ffl, fl, ff ligature substitution Lookup
	Feature GermanLigatures	Feature table definition
0000	NULL	featureParams - reserved, null
0001	3	lookupIndexCount
0000	0	lookupListIndices [0], in arbitrary order, ffi, fi ligatures

0001	1	lookupListIndices [1], ffl, fl, ff ligature substitution Lookup
0002	2	lookupListIndices [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 lookups[0] table, in design order
0010	FflFlFfLookup	offset to lookups[1] table
0018	EszetLookup	offset to lookups[2] table
	Lookup FfiFiLookup	lookups[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	lookups[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	lookups[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 (glyph ID 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: glyph ID list
0005	5	glyphCount
0038	gGlyphID	glyphArray[0], in glyph ID order
003B	jGlyphID	glyphArray[1]
0041	pGlyphID	glyphArray[2]
0042	qGlyphID	glyphArray[3]
004A	yGlyphID	glyphArray[4]

Example 6: CoverageFormat2 table (glyph ID 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 glyph IDs are ordered consecutively in the font. The StartCoverageIndex of zero (0) indicates that the first glyph ID, for the zero glyph, returns a Coverage Index of 0. The second glyph ID, 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: glyph ID ranges
0001	1	rangeCount
	rangeRecord[0]	
004E	0glyphID	startGlyphID
0057	9glyphID	endGlyphID
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: class array
0032	aGlyphID	startGlyph
001A	26	glyphCount
	classValueArray	
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 glyph IDs 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	classFormat: ranges
0003	3	classRangeCount
	classRangeRecords[0]	ordered by startGlyphID
0030	tahGlyphID	startGlyphID – first glyph ID in the range
0031	dhahGlyphID	endGlyphID – last glyph ID in the range
0002	2	class: high base glyphs
	classRangeRecords[1]	
0040	cafGlyphID	startGlyphID
0041	gafGlyphID	endGlyphID
0003	3	class: very high base glyphs
	classRangeRecords[2]	
00D2	fathatanDefaultGlyphID	startGlyphID
00D3	dammatanDefaultGlyphID	endGlyphID
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
MATH	Math layout 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 6.14). 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 6.14 – 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 6.15). Script min/max extents can be modified for particular language systems or features.

Figure 6.15 – 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.

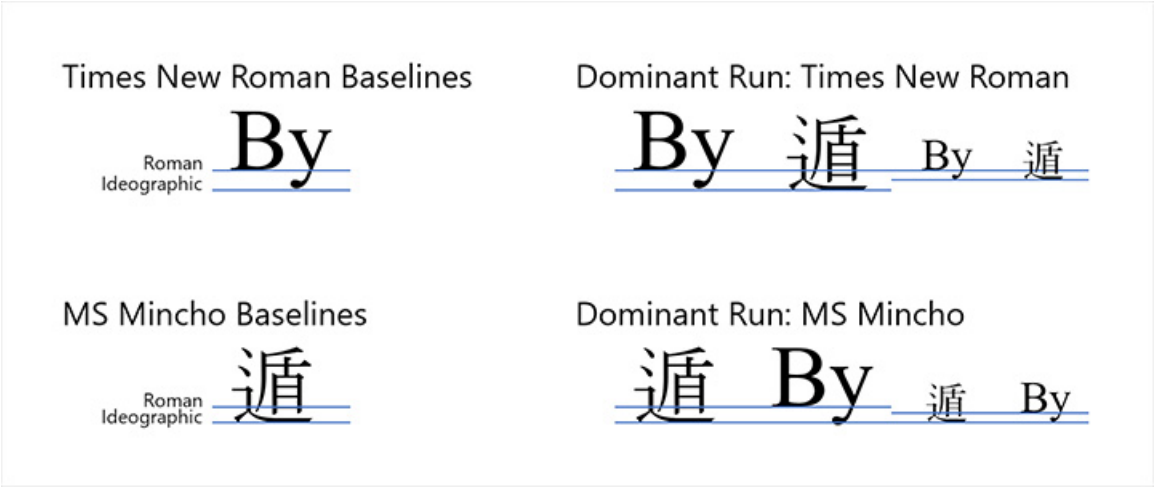


Figure 6.16 – Comparing Latin and Kanji baselines, with characters aligned according to the dominant run

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 6.16). 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 and OFF Font variations

OFF Font variations allow a single font to support many design variations along one or more axes of design variation. For example, a font with weight and width variations might support weights from thin to black, and widths from ultra-condensed to ultra-expanded. For general information on OFF Font variations, see subclause 7.1.

When different variation instances are selected, the design of individual glyphs changes, and the metric characteristics of the font as a whole may also change. As a result, corresponding changes may also be required for metric values in the BASE table.

Metrics in the BASE table are expressed directly in BaseCoord tables using explicit X or Y font-unit values. In a variable font, these X and Y values apply to the default instance and may need to be adjusted for the current variation instance. This is done using variation data with processes similar to those used for glyph outlines and other font data, as described in the OFF Font Variations Overview chapter.

NOTE Some BASE metrics can be expressed indirectly by reference to specific glyph outline points. In a variable font, use of glyph points to specify a metric value would require invoking the rasterizer to process the glyph-outline variation data in order to obtain the adjusted position of the point before the BASE metric value can be used. This may have a significant, negative impact on performance of text-layout processing. For this reason, it is recommended that, in a variable font, any BASE metric values that require adjustment for different variation instances should always be expressed directly as X and Y values.

Variation data for adjustment of BASE values is stored within an *item variation store* table within the BASE table. The item variation store and constituent formats are described in subclause7.2. The item variation store is also used in the 'GDEF' table, as well as in the 'MVAR' and other tables, but is different from the formats for variation data used in the 'cvar' or 'gvar' tables.

The variation data within an item variation store is comprised of a number of adjustment deltas that get applied to the default values of target items for variation instances within particular regions of the font's variation space. The item variation store format use *delta-set indices* to reference variation delta data for particular target, font-data items to which they are applied. Data external to the item variation store identifies the delta-set index to be used for each given target item. Within the BASE table, these indices are specified within *VariationIndex* tables, with one VariationIndex table referenced for each item that requires variation adjustment.

Note that the VariationIndex table is a variant of a Device table, with a distinct format value. (For full details on the Device and VariationIndex table formats, see subclause 6.2.) This is done so that the default instance of a variable font can be compatible with applications that do not support Font Variations. As a result, variable fonts cannot use device tables. A VariationIndex table will be ignored in applications that do not support Font Variations, or if the font is not a variable font.

The item variation store format uses a two-level organization for variation data: a store can have multiple *item variation data* subtables, and each subtable has multiple delta-set rows. A delta-set index is a two-part index: an outer index that selects a particular item variation data subtable, and an inner index that selects a particular delta-set row within that subtable. A VariationIndex table specifies both the outer and inner portions of the delta-set index.

6.3.1.2 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.

The figure below shows how the BASE table is organized.

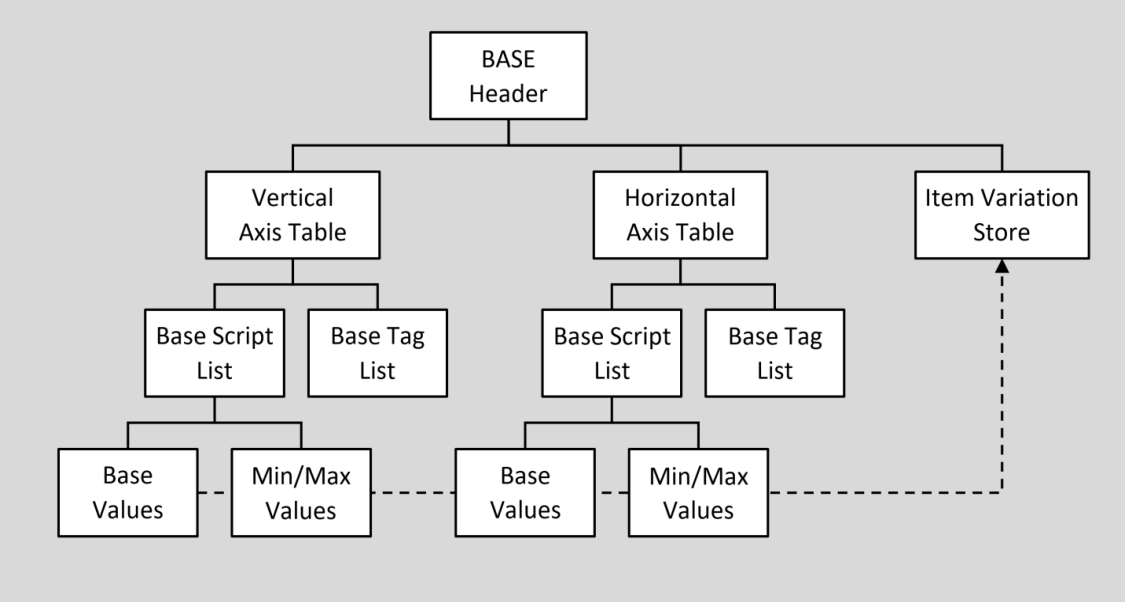


Figure 6.17 – High-level organization of BASE table

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.

6.3.1.3 BASE table structure

BASE header

The BASE table begins with a header that starts with a version number. Two versions are defined. Version1.0 contains offsets to horizontal and vertical Axis tables (HorizAxis and VertAxis). Version 1.1 also includes an offset to an Item Variation Store table.

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.

The optional Item Variation Store table is used in variable fonts to provide variation data for BASE metric values within the Axis tables.

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

BASE Header, Version 1.0

Type	Name	Description
uint16	majorVersion	Major version of the BASE table, =1
uint16	minorVersion	Minor version of the BASE table, =0

Offset16	horizAxisOffset	Offset to horizontal Axis table-from beginning of BASE table (may be NULL)
Offset16	vertAxisOffset	Offset to vertical Axis table-from beginning of BASE table (may be NULL)

BASE Header, Version 1.1

Type	Name	Description
uint16	majorVersion	Major version of the BASE table, = 1
uint16	minorVersion	Minor version of the BASE table, = 1
Offset16	horizAxisOffset	Offset to horizontal Axis table, from beginning of BASE table (may be NULL)
Offset16	vertAxisOffset	Offset to vertical Axis table, from beginning of BASE table (may be NULL)
Offset32	itemVarStoreOffset	Offset to Item Variation Store 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
Offset16	baseTagListOffset	Offset to BaseTagList table, from beginning of Axis table (may be NULL)
Offset16	baseScriptListOffset	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	baselineTags[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 baseScriptRecords 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
BaseScriptRecord	baseScriptRecords[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
Offset16	baseScriptOffset	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 default MinMax 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 default MinMax 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 (baseValuesOffset). 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 default MinMax table. If no default min/max extent data is defined for the script, this offset may be set to NULL.
- An array of BaseLangSysRecords (baseLangSysRecords). 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
Offset16	baseValuesOffset	Offset to BaseValues table-from beginning of BaseScript table-may be NULL
Offset16	defaultMinMaxOffset	Offset to MinMax table, from beginning of BaseScript table (may be NULL)
uint16	baseLangSysCount	Number of BaseLangSysRecords defined – may be zero (0)
BaseLangSysRecord	baseLangSysRecords [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
Offset16	minMaxOffset	Offset to MinMax table, from beginning of BaseScript table

BaseValues table

A BaseValues table lists the coordinate positions of all baselines named in the baselineTags 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 defaultBaselineIndex in the BaseValues table identifies the default baseline with a value that equals the array index position of the corresponding tag in the baselineTags 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 baselineTags[0] for the Han ideographic baseline, and 'romn' in baselineTags[1] for the Latin baseline. The BaseValues table for the Latin script will specify the roman baseline as the default, so the defaultBaselineIndex in the BaseValues table for Latin will be "1" to indicate the roman baseline tag. In the BaseValues table for the Han script, the defaultBaselineIndex 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 baselineTags array does not need a baseline tag for Cyrillic because Cyrillic and Latin share the same baseline. The defaultBaselineIndex 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 baselineTags array.

In addition to identifying the defaultBaselineIndex, the BaseValues table contains an offset to an array of BaseCoord tables (baseCoords) that list the coordinate positions for all baselines, including the default baseline, named in the associated baselineTags array. One BaseCoord table is defined for each baseline. The baseCoordCount field 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 the baseCoords array within the BaseValues table. The order of the stored offsets corresponds to the order of the tags listed in the baselineTags array of the BaseTagList. In other words, the first entry in the baseCoords array will define the offset to the BaseCoord table for the first baseline named in the baselineTags array, the second position will define the offset to the BaseCoord table for the second baseline named in the baselineTags 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	defaultBaselineIndex	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
Offset16	baseCoords[baseCoordCount]	Array of offsets to BaseCoord, from beginning of BaseValues table – order matches baselineTags 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 default MinMax table, referenced in a BaseScript table.
- Within the default MinMax 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 shall 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 (featMinMaxRecords) within the MinMax table. The featMinMaxCount field 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
Offset16	minCoord	Offset to BaseCoord table that defines the minimum extent value, from the beginning of MinMax table (may be NULL)
Offset16	maxCoord	Offset to BaseCoord table that defines the maximum extent value, from the beginning of MinMax table (may be NULL)
uint16	featMinMaxCount	Number of FeatMinMaxRecords – may be zero (0)
FeatMinMaxRecord	featMinMaxRecords [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
Offset16	minCoord	Offset to BaseCoord table that defines the minimum extent value, from beginning of MinMax table (may be NULL)
Offset16	maxCoord	Offset to BaseCoord table that defines the 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. In a variable font, the third format uses a VariationIndex table (a variant of a Device table), as needed, to reference variation data for adjustment of the X or Y values for the current variation instance.

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
uint16	referenceGlyph	Glyph ID of control glyph
uint16	baseCoordPoint	Index of contour point on the reference glyph

BaseCoord Format 3

The third BaseCoord format (BaseCoordFormat3) also specifies the BaseCoord value in design units, but, in a non-variable font, 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.)

In a variable font, BaseCoordFormat3 shall be used to reference variation data to adjust the X or Y value for different variation instances, if needed. In this case, BaseCoordFormat3 specifies an offset to a VariationIndex table, which is a variant of the Device table that is used for referencing variation data.

NOTE 1 While separate VariationIndex table references are required for each Coordinate value that requires variation, two or more values that require the same variation-data values can have offsets that point to the same VariationIndex table, and two or more VariationIndex tables can reference the same variation data entries.

NOTE 2 If no VariationIndex table is used for a particular X or Y value (the offset is zero, or a different BaseCoord format is used), then that value is used for all variation instances.

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

BaseCoordFormat3 table: Design units plus Device or VariationIndex table

Type	Name	Description
uint16	baseCoordFormat	Format identifier – format = 3
int16	coordinate	X or Y value, in design units
Offset16	deviceTable	Offset to Device table (non-variable font) / VariationIndex table (variable font) for X or Y value, from beginning of BaseCoord table (may be NULL).

Item Variation Store Table

The format and processing of the ItemVariationStore table and its constituent formats is described in subclause 7.2. Specification of the interpolation algorithm used to derive values for particular variation instances is given in subclause 7.1.

The ItemVariationStore contains adjustment-delta values arranged in one or more sets of deltas that are referenced using *delta-set indices*. For values that requires variation adjustment, a delta-set index is used to reference the particular variation data needed for that target value. Within the BASE table, delta-set indices are provided in *VariationIndex* tables contained within a BaseCoordFormat3 table. For a description of the VariationIndex table, see the OFF Layout Common Table Formats chapter. For details on use of VariationIndex tables within BaseCoord tables, see discussion earlier in this chapter.

6.3.1.4 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'	baselineTags[0], in alphabetical order
6964656F	'ideo'	baselineTags[1]
726F6D6E	'romn'	baselineTags[2]
	BaseScriptList HorizBaseScriptList	BaseScriptList table definition
0004	4	baseScriptCount
	baseScriptRecords[0]	Records in alphabetical order by baseScriptTag
6379726C	'cyrl'	baseScriptTag: Cyrillic script
001A	HorizCyrillicBaseScriptTable	Offset to BaseScript table for Cyrillic script
	baseScriptRecords[1]	
6465766E	'devn'	baseScriptTag: Devanagari script
0060	HorizDevanagariBaseScriptTable	Offset to BaseScript table for Devanagari script
	baseScriptRecords[2]	
68616E69	'hani'	baseScriptTag: Han script
008A	HorizHanBaseScriptTable	Offset to BaseScript table for Han script
	baseScriptRecords[3]	
6C61746E	'latn'	baseScriptTag: 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 default MinMax table – default script extents
0001	1	baseLangSysCount – BaseLangSysRecords for language-specific extents

	baseLangSysRecords[0]	Records in alphabetical order by baseLangSysTag.
52555320	'RUS '	BaseLangSysTag, Russian language system
0030	HorizRussianMinMaxTable	Offset to MinMax table – language-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	defaultBaselineIndex: roman baseline baselineTag index
0003	3	baseCoordCount, equals baseTagCount
000A	HorizHangingBaseCoordForCyrl	Offset to baseCoords[0] table: hanging baseline coordinate – order matches order of baselineTags array in BaseTagList
000E	HorizIdeographicBaseCoordForCyrl	Offset to baseCoords[1] table: ideographic baseline coordinate
0012	HorizRomanBaseCoordForCyrl	Offset to baseCoords[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	Default MinMax 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 featMinMaxRecords[]– 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
	featMinMaxRecords[0]	Records in alphabetical order by featureTableTab.

7469746C	'titl'	featureTableTag: Titling 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 default MinMax table for default script extents
0001	1	baseLangSysCount
	baseLangSysRecords[0]	For Russian feature-specific extents.
52555320	'RUS '	baseLangSysTag: Russian
001A	HorizRussian MinMaxTable	Offset to MinMax table for language-specific extents
	MinMax HorizCyrillicDefault MinMaxTable	Default MinMax 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 featMinMaxRecords[], 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 HorizRussian MinMaxTable	MinMax table definition for Russian feature – no extent differences for Russian language itself

0000	NULL	minCoord – offset to min BaseCoord table not defined, matches script default
0000	NULL	maxCoord – offset to max BaseCoord table not defined, matches script default
0001	1	featMinMaxCount
	featMinMaxRecords[0]	Records in alphabetical order by featureTableTag
7469746C	'titl'	featureTableTag: Titling
		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 'titl' 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 'titl' 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 six types of information in six independent subtables:

- 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 ItemVariationStore table is used in variable fonts to contain variation data used for adjustment of values in the GDEF, GPOS or JSTF tables.

The GSUB, GPOS or JSTF tables may reference certain GDEF table information used for processing of lookup tables. See, for example, the LookupFlag bit enumeration in "OFF layout common table formats".

In variable fonts, the GDEF, GPOS and JSTF tables may all reference variation data within the ItemVariationStore table contained within the GDEF table. See below for further discussion of variable fonts and the ItemVariationStore table.

6.3.2.1 Overview

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

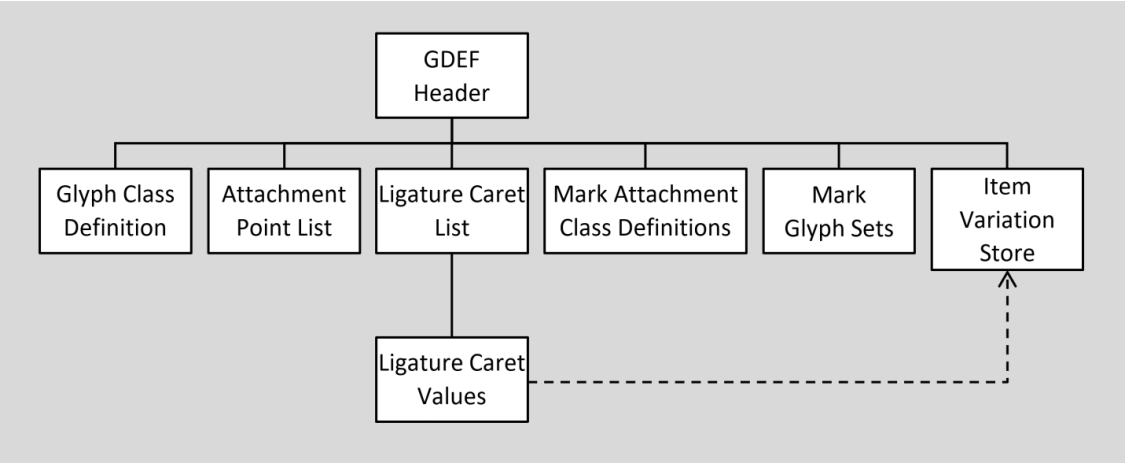


Figure 6.18 – High-level organization of GDEF table

6.3.2.2 Overview of GDEF subtables

Glyph Class Definition Table Overview

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 6.19). 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 6.19 – 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 Overview

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 Overview

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 6.20).

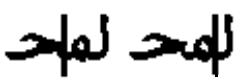


Figure 6.20 – Proper ligature caret postioning

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 (in non-variable fonts only) to adjust values at specific font sizes.

In a variable font, the caret positions may need to be adjusted for different variation instances. This is done using data in an ItemVariationStore table. See below for more regarding variable fonts and the ItemVariationStore table.

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.

Mark Attachment Class Definition Table Overview

A Mark Class Definition Table is used to assign mark glyphs into different classes that can be used in lookup tables within the GSUB or GPOS table to control how mark glyphs within a glyph sequence are treated by lookups. For more information on the use of mark attachment classes, see the description of lookup flags in the "Lookup Table" section of subclause 6.2.

Mark Glyph Sets Table Overview

A Mark Glyph Sets table is used to define sets of mark glyphs that can be used in lookup tables within the GSUB or GPOS table to control how mark glyphs within a glyph sequence are treated by lookups. For more information on the use of mark glyph sets, see the description of lookup flags in the "Lookup Table" section of subclause 6.2.

Item Variation Store Overview

An Item Variation Store table is used in variable fonts. OFF Font variations allow a single font to support many design variations along one or more axes of design variation. For example, a font with weight and width variations might support weights from thin to black, and widths from ultra-condensed to ultra-expanded. For general information on OFF Font Variations, see subclause 7.1.

When different variation instances are selected, the design of individual glyphs changes. The same contours and points are used, but the position in the design grid of each point can change, as can the overall glyph metrics. As a result, corresponding changes may also be required for caret X or Y positions.

Variation data for caret positions is stored in the Item Variation Store table. This same table, within the GDEF table, can also hold variation data used for X or Y values in the GPOS or JSTF tables. The Item Variation Store and constituent formats are described in subclause 7.2. The Item Variation Store is also used in the MVAR, HVAR and other tables, but is different from the formats for variation data used in the 'cvar' or 'gvar' tables.

The variation data within an Item Variation Store is comprised of a number of adjustment deltas that get applied to the default values of target items for variation instances within particular regions of the font's variation space. The Item Variation Store format uses *delta-set indices* to reference variation delta data for particular target, font-data items to which they are applied. Data external to the Item Variation Store identifies the delta-set index to be used for each given target item. Within the GDEF, GPOS or JSTF tables, these indices are specified within *VariationIndex* tables, with one VariationIndex table referenced for each item that requires variation adjustment.

Note that the VariationIndex table is a variant of a Device table, with a distinct format value. (For full details on the Device and VariationIndex table formats, see the OFF layout common table formats.) This is done so that the default instance of a variable font can be compatible with applications that do not support Font Variations. As a result, variable fonts cannot use device tables. A VariationIndex table will be ignored in applications that do not support Font Variations, or if the font is not a variable font.

The Item Variation Store format uses a two-level organization for variation data: a store can have multiple *Item Variation Data* subtables, and each subtable has multiple delta-set rows. A delta-set index is a two-part index: an outer index that selects a particular item variation data subtable, and an inner index that selects a particular delta-set row within that subtable. A VariationIndex table specifies both the outer and inner portions of the delta-set index.

6.3.2.3 GDEF table structure

GDEF header

The GDEF table begins with a header that starts with a version number. Three versions are defined. Version 1.0 contains an offset to a Glyph Class Definition table (GlyphClassDef), an offset to an Attachment List table (AttachList), an offset to a Ligature Caret List table (LigCaretList), and an offset to a Mark Attachment Class Definition table (MarkAttachClassDef). Version 1.2 also includes an offset to a Mark Glyph Sets Definition table (MarkAttachClassDef). Version 1.3 also includes an offset to an Item Variation Store table.

Example 1 in subclause 6.3.2.3 shows a GDEF Header table.

GDEF header, Version 1.0

Type	Name	Description
uint16	majorVersion	Major version of the GDEF table, =1
uint16	minorVersion	Minor version of the GDEF table, =0
Offset16	glyphClassDefOffset	Offset to class definition table for glyph type, from beginning of GDEF header (may be NULL)
Offset16	attachListOffset	Offset to attachment point list table, from beginning of GDEF header (may be NULL)
Offset16	ligCaretListOffset	Offset to ligature caret list table, from beginning of GDEF header (may be NULL)
Offset16	markAttachClassDefOffset	Offset to class definition table for mark attachment type, from beginning of GDEF header (may be NULL)

GDEF header, Version 1.2

Type	Name	Description
uint16	majorVersion	Major version of the GDEF table, =1
uint16	minorVersion	Minor version of the GDEF table, =2
Offset16	glyphClassDefOffset	Offset to class definition table for glyph type, from beginning of GDEF header (may be NULL)
Offset16	attachListOffset	Offset to attachment point list table, from beginning of GDEF header (may be NULL)
Offset16	ligCaretListOffset	Offset to ligature caret list table, from beginning of GDEF header (may be NULL)
Offset16	markAttachClassDefOffset	Offset to class definition table for mark attachment type, from beginning of GDEF header (may be NULL)
Offset16	markGlyphSetsDefOffset	Offset to the table of mark glyph set definitions, from beginning of GDEF header (may be NULL)

GDEF Header, Version 1.3

Type	Name	Description
uint16	majorVersion	Major version of the GDEF table, = 1
uint16	minorVersion	Minor version of the GDEF table, = 3
Offset16	glyphClassDefOffset	Offset to class definition table for glyph type, from beginning of GDEF header (may be NULL)
Offset16	attachListOffset	Offset to attachment point list table, from beginning of GDEF header (may be NULL)
Offset16	ligCaretListOffset	Offset to ligature caret list table, from beginning of GDEF header (may be NULL)
Offset16	markAttachClassDefOffset	Offset to class definition table for mark attachment type, from beginning of GDEF header (may be NULL)

Offset16	markGlyphSetsDefOffset	Offset to the table of mark glyph set definitions, from beginning of GDEF header (may be NULL)
Offset32	itemVarStoreOffset	Offset to the Item Variation Store table, 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). 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 Point List table

The Attachment Point 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
Offset16	coverageOffset	Offset to Coverage table, from beginning of AttachList table
uint16	glyphCount	Number of glyphs with attachment points
Offset16	attachPointOffsets[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	pointIndices[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
Offset16	coverageOffset	Offset to Coverage table, from beginning of LigCaretList table
uint16	ligGlyphCount	Number of ligature glyphs
Offset16	ligGlyphOffsets[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)
Offset16	caretValueOffsets [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 (in non-variable fonts) a Device table to fine-tune a caret's position at specific font sizes and device resolutions. In a variable font, the third format uses a VariationIndex table (a variant of a Device table) to reference variation data for adjustment of the caret position for the current variation instance, as needed. 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	caretValuePointIndex	Contour point index on glyph

CaretValue Format 3

The third format (CaretValueFormat3) also specifies the value in design units, but, in non-variable fonts, 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.)

In variable fonts, CaretValueFormat3 must be used to reference variation data to adjust caret positions for different variation instances, if needed. In this case, CaretValueFormat3 specifies an offset to a VariationIndex table, which is a variant of the Device table used for variations.

NOTE 1 While separate VariationIndex table references are required for each value that requires variation, two or more values that require the same variation-data values can have offsets that point to the same VariationIndex table, and two or more VariationIndex tables can reference the same variation data entries.

NOTE 2 If no VariationIndex table is used for a particular caret position value, then that value is used for all variation instances.

The format consists of a format identifier (CaretValueFormat), an X or Y value (Coordinate), and an offset to a Device or VariationIndex table.

Example 6 at the end of this clause shows a CaretValueFormat3 table.

CaretValueFormat3 table: Design units plus Device or VariationIndex table

Type	Name	Description
uint16	caretValueFormat	Format identifier: format = 3
int16	coordinate	X or Y value, in design units
Offset16	deviceOffset	Offset to Device table (non-variable font) / Variation Index table (variable font) 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).

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	markGlyphSetTableFormat	Format identifier = 1
uint16	markGlyphSetCount	Number of mark glyph sets defined
Offset32	coverageOffsets [markGlyphSetCount]	Array of offsets to mark glyph set Coverage tables

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 Offset32, not Offset16.

Item Variation Store Table

The format and processing of the Item Variation Store table and its constituent formats is described in the subcaluse "Font variations common table formats". Specification of the interpolation algorithm used to derive values for particular variation instances is given in subclause 7.1.

The Item Variation Store contains adjustment-delta values arranged in one or more sets of deltas that are referenced using *delta-set indices*. For values that require variation adjustment, a delta-set index is used to reference the particular variation data needed for that target value. Within the GDEF, GPOS or JSTF tables, delta-set indices are provided in VariationIndex tables associated with particular target items, such as caret positions in the GDEF table. For a description of the VariationIndex table, see subclause 6.2. For details on use of VariationIndex tables within the GDEF table, see discussion earlier in this chapter. For details on use of VariationIndex tables within the GPOS or JSTF tables, see the discussion of OFF Font variations in the chapters for each of those tables.

GDEF table examples

The rest of this subclause 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	GDEFHeader table definition
	TheGDEFHeader	
00010000	0x00010000	major/minor version
000C	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
	classRangeRecords[0]	
0024	iGlyphID	startGlyphID
0024	iGlyphID	endGlyphID
0001	1	class: base glyphs
	classRangeRecords[1]	
009F	ffiLigGlyphID	startGlyphID
009F	ffiLigGlyphID	endGlyphID
0002	2	class: ligature glyphs
	classRangeRecords[2]	
0058	umlautAccentGlyphID	startGlyphID
0058	umlautAccentGlyphID	endGlyphID
0003	3	class: mark glyphs
	classRangeRecords[3]	
018F	CurvedTailComponentGlyphID	startGlyphID
018F	CurvedTailComponentGlyphID	endGlyphID
0004	4	class: 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	attachPointOffsets[0]
000C	eAttachPoint	attachPointOffsets [1]
	AttachPoint aAttachPoint	AttachPoint table definition
0001	1	pointCount
0012	18	pointIndices[0]
	AttachPoint eAttachPoint	AttachPoint table definition
0002	2	pointCount
000E	14	pointIndices [0]
0017	23	pointIndices [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	ligGlyphOffsets [0]
0014	ffiLigGlyph	ligGlyphOffsets [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	caretValueOffsets[0]
	LigGlyph ffiLigGlyph	LigGlyph table definition
0002	2	caretCount, equals the number of components - 1
0006	CaretFFI1	caretValueOffsets [0]
000E	CaretFFI2	caretValueOffsets [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	caretValuePointIndex

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
	classRangeRecords[0]	
0268	graveAccentGlyphID	startGlyphID
026A	circumflexAccentGlyphID	endGlyphID
0001	1	Class: top marks
	classRangeRecords[1]	
0270	diaeresisAccentGlyphID	startGlyphID
0272	acuteAccentGlyphID	endGlyphID
0001	1	Class: top marks
	classRangeRecords[2]	
028C	diaeresisBelowGlyphID	startGlyphID
028F	cedillaGlyphID	endGlyphID
0002	2	Class: bottom marks
	classRangeRecords[3]	
0295	circumflexBelowGlyphID	startGlyphID
0295	circumflexBelowGlyphID	endGlyphID
0002	2	Class: 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.

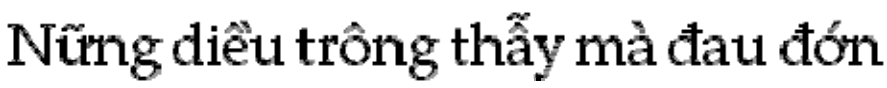


Figure 6.21 – Vietnamese words with marks.

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 6.22).

Figure 6.22 – 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 6.23). For horizontal text, placement corresponds to the left side bearing, and advance corresponds to the advance width.

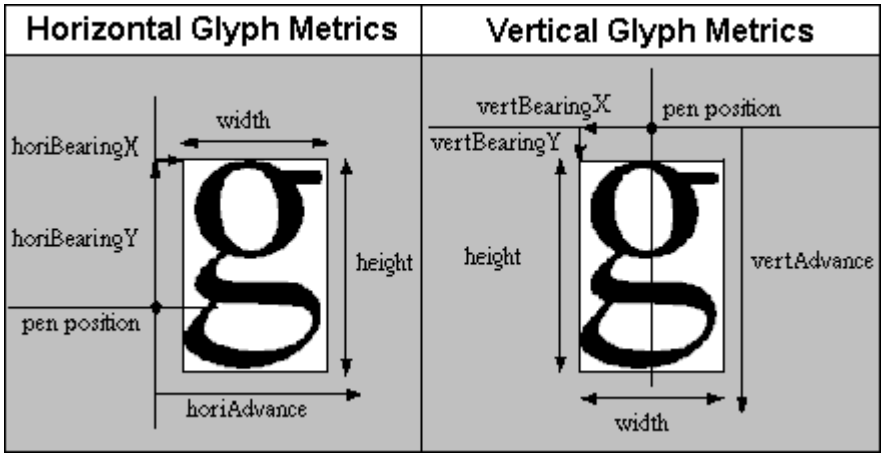


Figure 6.23 – Glyph positioning with TrueType

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.

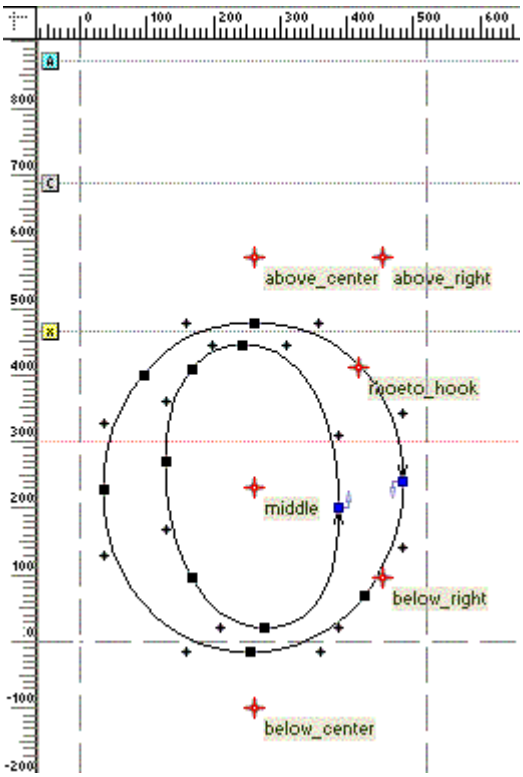


Figure 6.24 – Base glyph with multiple attachment points.

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 6.25).

Figure 6.25 – 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 m*ark-to-base attachment* positions combining marks with respect to base glyphs, as when positioning vowels, diacritical marks, or tone marks in Arabic, Hebrew, and Vietnamese.
- A m*ark-to-ligature 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 m*ark-to-mark 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 6.26 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 6.26 – Contextual positioning lowered the accent over a vowel glyph that followed an overhanging uppercase glyph

GPOS Table and OFF Font Variations

OFF Font variations allow a single font to support many design variations along one or more axes of design variation. For example, a font with weight and width variations might support weights from thin to black, and widths from ultra-condensed to ultra-expanded. For general information on OFF Font variations, see subclause 7.1.

When different variation instances are selected, the design of individual glyphs changes. The same contours and points are used, but the position in the design grid of each point can change, as can the default horizontal or vertical advance and side bearings. As a result, corresponding changes may also be required for positioning and advance adjustments in the GPOS table.

Positioning actions in the GPOS table are expressed directly using explicit X or Y font-unit values. In a variable font, these X and Y values apply to the default instance and may need to be adjusted for the current

variation instance. This is done using variation data with processes similar to those used for glyph outlines and other font data, as described in the OFF Font variations overview.

NOTE For certain GPOS actions, positions can be expressed indirectly by reference to specific glyph outline points. In a variable font, use of glyph points to specify a positioning action would require invoking the rasterizer to process the glyph-outline variation data in order to obtain the adjusted position of the point before the glyph positioning operation can be completed. This may have a significant, negative impact on performance of text-layout processing. For this reason, it is recommended that, in a variable font, positions that require adjustment for different variation instances should always be expressed directly as X and Y values.

Variation data for adjustment of GPOS X or Y values is stored within an *Item Variation Store* table located within the GDEF table. The same Item Variation Store is also used for adjustment of values in the GDEF and JSTF tables. The Item Variation Store and constituent formats are described in subclause 7.2. These formats are also used in the BASE table, as well as in the MVAR and other tables, but is different from the formats for variation data used in the 'cvar' or gvar tables.

The variation data within an Item Variation Store is comprised of a number of adjustment deltas that get applied to the default values of target items for variation instances within particular regions of the font's variation space. The Item Variation Store format uses *delta-set indices* to reference variation delta data for particular target, font-data items to which they are applied. Data external to the Item Variation Store identifies the delta-set index to be used for each given target item. Within the GPOS table, these indices are specified within *VariationIndex* tables, with one VariationIndex table referenced for each item that requires variation adjustment.

Note that the VariationIndex table is a variant of a Device table, with a distinct format value. (For full details on the Device and VariationIndex table formats, see subclause 6.2.) This is done so that the default instance of a variable font can be compatible with applications that do not support Font Variations. As a result, variable fonts cannot use device tables. A VariationIndex table will be ignored in applications that do not support Font Variations, or if the font is not a variable font.

The Item Variation Store format uses a two-level organization for variation data: a store can have multiple *Item Variation Data* subtables, and each subtable has multiple delta-set rows. A delta-set index is a two-part index: an outer index that selects a particular Item Variation Data subtable, and an inner index that selects a particular delta-set row within that subtable. A VariationIndex table specifies both the outer and inner portions of the delta-set index.

GPOS table organization

The GPOS table begins with a header that defines offsets to a ScriptList, a FeatureList, a LookupList, and an optional FeatureVariations table (see Figure 6.27):

- 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.
- •The FeatureVariations table can be used to substitute an alternate set of lookup tables to use for any given feature under specified conditions. This is currently used only in variable fonts.

For a detailed discussion of ScriptLists, FeatureLists, LookupLists, and FeatureVariations tables see the OFF layout common table formats. The following discussion summarizes how the GPOS table works.

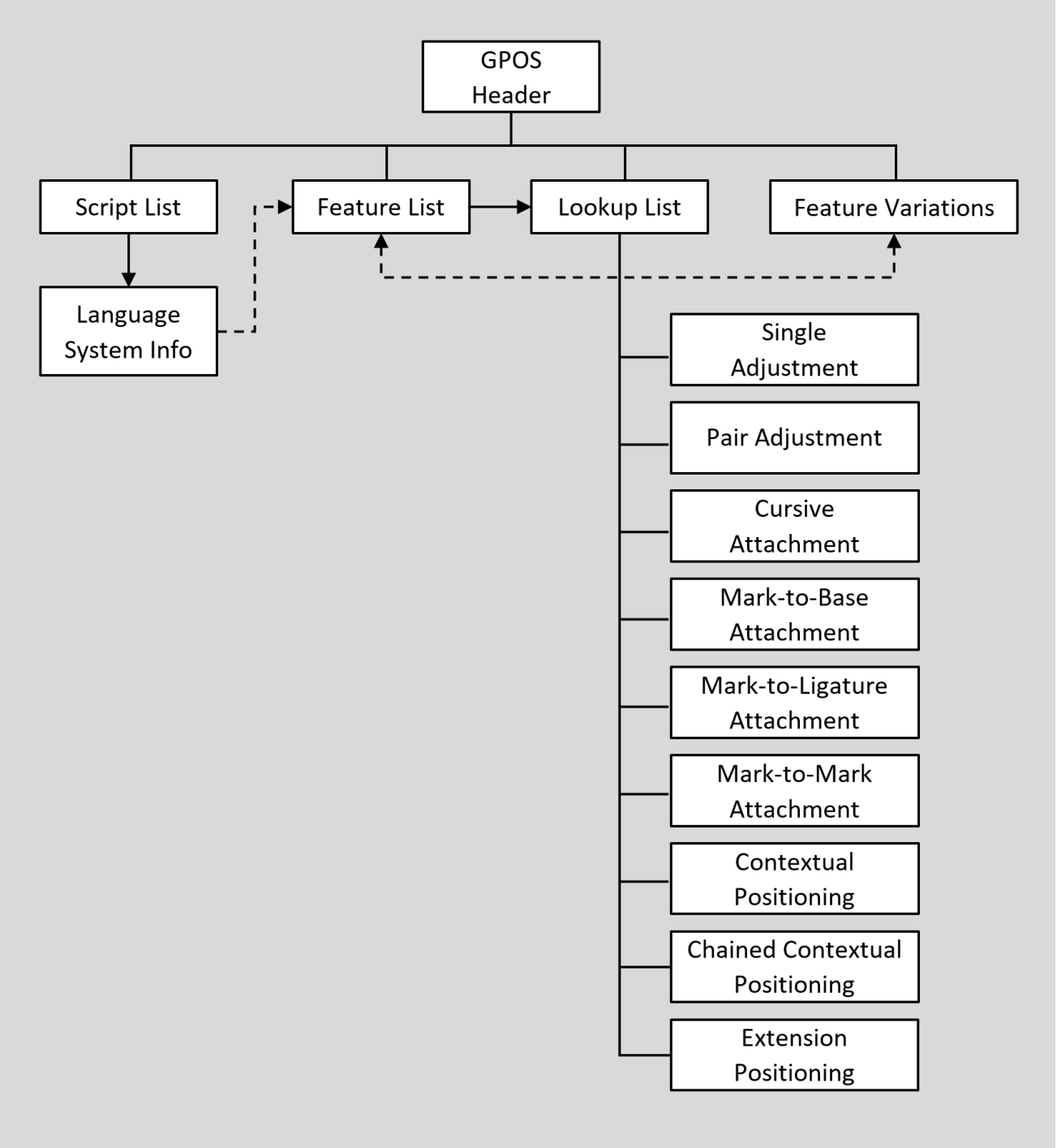


Figure 6.27 – High-level organization of GPOS table

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 LangSys 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 feature tables to apply to an input glyph string.

5. If a Feature Variation table is present, evaluate conditions in the Feature Variation table to determine if any of the initially-selected feature tables should be substituted by an alternate feature table.

6. Each feature provides an array of index numbers into the GPOS LookupList table. Assemble all lookups from the set of chosen feature tables, and apply the lookups in the order given in the LookupList table.

For a detailed description of the Feature Variation table and how it is processed, see subclause 6.2.

Lookup data is defined in Lookup tables, which are defined in subclause 6.2. A LookupTable contains one or more subtables that define the specific conditions, type, and results of a positioning action used to implement a feature. Specific Lookup subtable types are used for glyph positioning actions, and are defined in this subclause. All subtables within a Lookup table shall be of the same lookup type, as listed in the following table for the GPOS LookupType Enumeration:

GPOS LookupType Enumeration

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 has one or more subtable formats. The "best" 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.

Certain structures are used across multiple GPOS Lookup subtable types and formats. All Lookup subtables use the Coverage table, which is defined in subclause 6.2. Single and pair adjustments (LookupTypes 1 and 2) use a ValueRecord structure and associated ValueFormat enumeration, which are defined later in this subclause. Attachment subtables (LookupTypes 3, 4, 5 and 6) use Anchor and MarkArray tables, also defined later in this subclause.

A series of positioning operations on the same glyph or string requires multiple lookups, one for each separate action. Positioning adjustment values are accumulated in these cases. Each lookup has a different array index 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 action, 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 ValueRecord 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. Examples at the end of this clause illustrate the GPOS header and seven of the nine LookupTypes, as well as the ValueRecord and Anchor and MarkArray tables.

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. Example 1 at the end of this clause shows a GPOS Header version 1.0 table definition.

GPOS Header, Version 1.0

Type	Name	Description
uint16	majorVersion	Major version of the GPOS table, = 1
uint16	minorVersion	Minor version of the GPOS table, = 0
Offset16	scriptListOffset	Offset to ScriptList table, from beginning of GPOS table
Offset16	featureListOffset	Offset to FeatureList table, from beginning of GPOS table
Offset16	lookupListOffset	Offset to LookupList table, from beginning of GPOS table

GPOS Header, Version 1.1

Type	Name	Description
uint16	majorVersion	Major version of the GPOS table, = 1
uint16	minorVersion	Minor version of the GPOS table, = 1
Offset16	scriptListOffset	Offset to ScriptList table, from beginning of GPOS table
Offset16	featureListOffset	Offset to FeatureList table, from beginning of GPOS table
Offset16	lookupListOffset	Offset to LookupList table, from beginning of GPOS table
Offset32	featureVariationsOffset	Offset to FeatureVariations table, from beginning of GPOS table (may be NULL)

6.3.3.2 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 (Format 1), and one that applies a different adjustment for each unique glyph (Format 2).

Single Adjustment Positioning: Format 1: Single Positioning Value

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 (coverageOffset), 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 (valueRecord). 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:

Type	Name	Description
uint16	posFormat	Format identifier: format = 1
Offset16	coverageOffet	Offset to Coverage table, from beginning of SinglePos subtable.
uint16	valueFormat	Defines the types of data in the ValueRecord.
ValueRecord	valueRecord	Defines positioning value(s) – applied to all glyphs in the Coverage table.

Single Adjustment Positioning Format 2: Array of positioning values

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 A single ValueFormat applies to all ValueRecords defined in a SinglePos subtable. In this example, if xPlacement is the only value that a ValueRecord needs to optically align the glyphs, then the X_PLACEMENT flag will be the only flag set in the valueFormat field 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 (coverageOffset), and the Value Format flags field (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 (valueRecords). 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:

Type	Name	Description
uint16	posFormat	Format identifier: format = 2
Offset16	coverageOffset	Offset to Coverage table, from beginning of SinglePos subtable.
uint16	valueFormat	Defines the types of data in the ValueRecords.
uint16	valueCount	Number of ValueRecords – must equal glyphCount in the Coverage table.
ValueRecord	valueRecords [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), and one that identifies glyphs by class (Format 2).

Pair Adjustment Positioning Format 1:Adjustments for glyph pairs

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 ValueFormat fields:

— valueFormat1 applies to the ValueRecords for the first glyph in each pair. The single ValueFormat field applies to ValueRecords for all first glyphs. If valueFormat1 is set to zero (0), the corresponding glyph has no ValueRecord and, therefore, should not be repositioned.

— valueFormat2 applies to the ValueRecords for the second glyph in each pair. The single ValueFormat field applies to ValueRecords for all second glyphs. If valueFormat2 is set to 0, 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 (coverageOffet) 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 (pairSetOffsets) 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

Type	Name	Description
uint16	posFormat	Format identifier: format = 1
Offset16	coverageOffset	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).

uint16	valueFormat2	Defines the types of data in ValueRecord2 – for the second glyph in the pair (may be zero).
uint16	pairSetCount	Number of PairSet tables.
Offset16	pairSetOffsets [pairSetCount]	Array of offsets to PairSet tables. Offsets are 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 glyph ID of the second glyph in each pair. The pairValueCount field specifies the number of PairValueRecords in the set.

PairSet table

Type	Name	Description
uint16	pairValueCount	Number of PairValueRecords
PairValueRecord	pairValueRecords [pairValueCount]	Array of PairValueRecords, ordered by glyph ID of the second glyph.

A PairValueRecord specifies the second glyph in a pair (secondGlyph) and defines a ValueRecord for each glyph (valueRecord1 and valueRecord2). If valueFormat1 in the PairPos subtable is set to zero (0), valueRecord1 will be empty; similarly, if valueFormat2 is 0, valueRecord2 will be empty.

Example 4 at the end of this clause shows a PairPosFormat1 subtable that defines two cases of pair kerning.

PairValueRecord

Type	Name	Description
uint16	secondGlyph	Glyph ID of second glyph in the pair (first glyph is listed in the Coverage table).
ValueRecord	valueRecord1	Positioning data for the first glyph in the pair.
ValueRecord	valueRecord2	Positioning data for the second glyph in the pair.

Pair Adjustment Positioning Format 2: Class pair adjustment

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.

In a PairPosFormat2 subtable, glyph classes are defined using a Class Definition table, defined in subclause 6.2.

The PairPos Format2 subtable begins with a format identifier (posFormat) and an offset to a Coverage table (coverageOffset), 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 ValueFormat fields:

- valueFormat1 applies to the ValueRecords for the first glyph in each pair. The single ValueFormat field applies to ValueRecords for all first glyphs. If valueFormat1 is set to zero (0), then the ValueRecords for the first glyph will be empty and, therefore, the first glyph is not repositioned.

- valueFormat2 applies to the ValueRecords for the second glyph in each pair. The single ValueFormat field applies to ValueRecords for all second glyphs. If valueFormat2 is set to 0, then the ValueRecords for the second glyph of the pair will be empty, the second glyph is not repositioned, and it becomes the "next" glyph for which a lookup is performed.

PairPosFormat2 requires that each glyph in all pairs be assigned to a class, which is identified by an integer called a class value. 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 (classDef1Offset, classDef2Offset) 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 as for 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 Class 0.

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

Type	Name	Description
uint16	posFormat	Format identifier: format = 2
Offset16	coverageOffset	Offset to Coverage table, from beginning of PairPos subtable
uint16	valueFormat1	ValueRecord definition – for the first glyph of the pair (may be zero)
uint16	valueFormat2	ValueRecord definition – for the second glyph of the pair (may be zero)
Offset16	classDef1Offset	Offset to ClassDef table, from beginning of PairPos subtable – for the first glyph of the pair
Offset16	classDef2Offset	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
Class1Record	class1Record [class1Count]	Array of Class1 records, ordered by classes in classDef1

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

Type	Name	Description
Class2Record	class2Records[class2Count]	Array of Class2 records, ordered by classes in classDef 2

A Class2Record consists of two ValueRecords, one for the first glyph in a class pair (valueRecord1) and one for the second glyph (valueRecord2). If the PairPos subtable has a value of zero (0) for valueFormat1 or valueFormat2, then the corresponding record (valueRecord1 or valueRecord2) will be empty – that is, not present. For example, if valueFormat1 is zero, then the Class2Record will begin with and consist solely of valueRecord2. The text-processing client must be aware of the variable nature of the Class2Record and use the valueFormat1 and valueFormat2 fields to determine the size and content of the Class2Record.

Example 5 at the end of this clause demonstrates pair kerning with glyph classes in a PairPosFormat2 subtable.

Class2Record

Type	Name	Description
ValueRecord	valueRecord1	Positioning for first glyph – empty if ValueFormat1 = 0
ValueRecord	valueRecord2	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.

Cursive attachment positioning Format1: Cursive attachment

The CursivePosFormat1 subtable begins with a format identifier (posFormat) and an offset to a Coverage table (coverageOffset), 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 (entryExitRecords).

CursivePosFormat1 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of CursivePos subtable.
uint16	entryExitCount	Number of EntryExit records.
EntryExitRecord	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 (entryAnchorOffset), and an offset to an Anchor table that identifies the exit point on the glyph (exitAnchorOffset). (For a complete description of the Anchor table, see the end of this subclause.)

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

Type	Name	Description
Offset16	entryAnchorOffset	Offset to EntryAnchor table, from beginning of CursivePos subtable (may be NULL).
Offset16	exitAnchorOffset	Offset to ExitAnchor table, from beginning of CursivePos subtable (may be NULL).

Lookup Type 4: Mark-to-Base 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 also assigned to a mark class. Each base glyph then defines an anchor point for each class of marks it uses. When a mark is combined with a given base, the mark position is adjusted so that the mark anchor is aligned with the base anchor for the applicable mark class.

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.

A mark class is identified by a specific integer. Within the MarkBasePos subtable, the anchor definition of each mark, and the assignment of each mark to a mark class, is provided using a MarkArray table in combination with a mark Coverage table. First, a mark Coverage table specifies all of the mark glyphs covered by the subtable. Then, for every mark in the Coverage table, the MarkArray table has a corresponding MarkRecord that defines the anchor and class assignment for the mark. The MarkArray table and MarkRecord are defined later in this subclause.

The MarkToBase Attachment subtable has one format: MarkBasePosFormat1.

Mark-to-Base attachment positioning Format1: Mark-to-Base attachment point

The MarkBasePosFormat1 subtable begins with a format identifier (posFormat) and offsets (markCoverageOffset, baseCoverageOffset) 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).

The MarkBasePosFormat1 subtable also contains an offset (markArrayOffset) to a MarkArray table. For each mark glyph in the mark Coverage table, a MarkRecord in the MarkArray table specifies its class and an offset to the Anchor table that describes the mark's attachment point. The classCount field specifies the total number of distinct mark classes defined in all the MarkRecords.

The MarkBasePosFormat1 subtable also contains an offset to a BaseArray table (baseArrayOffset), which defines for each base glyph an array of anchors, one for each mark class.

MarkBasePosFormat1 subtable

Type	Name	Description
uint16	posFormat	Format identifier-format = 1
Offset16	markCoverageOffset	Offset to MarkCoverage table, from beginning of MarkBasePos subtable
Offset16	baseCoverageOffset	Offset to BaseCoverage table, from beginning of MarkBasePos subtable
uint16	markClassCount	Number of classes defined for marks
Offset16	markArrayOffset	Offset to MarkArray table, from beginning of MarkBasePos subtable
Offset16	baseArrayOffset	Offset to BaseArray table, from beginning of MarkBasePos subtable

The BaseArray table consists of an array (baseRecords) 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

Type	Name	Description
uint16	baseCount	Number of BaseRecords
BaseRecord	baseRecords[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

Type	Name	Description
Offset16	baseAnchorOffset[markClassCount]	Array of offsets (one per mark class) to Anchor tables. Offsets are from beginning of BaseArray table, ordered by class.

Lookup Type 5: Mark-to-Ligature 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, each base glyph has an attachment point defined for each class of marks. MarkToLigature attachment is similar, except that each ligature glyph is defined to have multiple components (in a virtual sense — not actual glyphs), and each component has a separate set of attachment points defined for the different mark classes.

As a result, a ligature glyph may have multiple base attachment points for one class of marks. For a given mark assigned to a particular class, the appropriate base attachment point is determined by which ligature component the mark is associated with. This is dependent on the original character string and subsequent character- or glyph-sequence processing, not the font data alone. While a text-layout client is performing any character-based preprocessing or any glyph-substitution operations using the GSUB table, the text-layout client must keep track of associations of marks to particular ligature-glyph components.

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. As with MarkToBase attachment, mark anchors and class assignments are defined using a MarkArray table in combination with a mark Coverage table. Every ligature glyph specifies a two-dimensional array of data: for each component in a ligature, an array of anchor points is defined, 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.

Mark-to-Ligature attachment positioning Format1: Mark-to-Ligature Attachment

The MarkLigPosFormat1 subtable begins with a format identifier (posFormat) and two offsets (markCoverageOffset, baseCoverageOffset) to Coverage tables that list all the mark glyphs (markCoverage) and Ligature glyphs (ligatureCoverage) referenced in the subtable.

The MarkLigPosFormat1 subtable also contains an offset to a MarkArray table (markArrayOffset). For each mark glyph in the mark Coverage table, a MarkRecord in the MarkArray table specifies its class and an offset to the Anchor table that describes the mark's attachment point. The markClassCount field specifies the total number of distinct mark classes defined in all the MarkRecords.

The MarkLigPosFormat1 subtable also contains an offset to a LigatureArray table (ligatureArrayOffset), which defines for each ligature glyph the two-dimensional array of anchor data: one anchor per ligature component per mark class.

MarkLigPosFormat1 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 1
Offset16	markCoverageOffset	Offset to markCoverage table, from beginning of MarkLigPos subtable
Offset16	ligatureCoverageOffset	Offset to ligatureCoverage table, from beginning of MarkLigPos subtable
uint16	markClassCount	Number of defined mark classes
Offset16	markArrayOffset	Offset to MarkArray table, from beginning of MarkLigPos subtable
Offset16	ligatureArrayOffset	Offset to LigatureArray table, from beginning of MarkLigPos subtable

The LigatureArray table contains a count (ligatureCount) and an array of offsets (ligatureAttachOffsets) to LigatureAttach tables. The ligatureAttachOffsets 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

Type	Name	Description
uint16	ligatureCount	Number of LigatureAttach table offsets
Offset16	ligatureAttachOffsets [ligatureCount]	Array of offsets to LigatureAttach tables. Offsets are from beginning of LigatureArray table, ordered by ligatureCoverage Index

Each LigatureAttach table consists of an array (componentRecords) 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 – that is, the logical 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

Type	Name	Description
uint16	componentCount	Number of ComponentRecords in this ligature.
ComponentRecord	componentRecords[componentCount]	Array of Component records, ordered in writing direction.

A ComponentRecord, one for each component in the ligature, contains an array of offsets (ligatureAnchorOffsets) to the Anchor tables that define all the attachment points used to attach marks to the component. 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 ligatureAnchorOffsets 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

Type	Name	Description
Offset16	ligatureAnchorOffsets [markClassCount]	Array of offsets (one per class) to Anchor tables. Offsets are from beginning of LigatureAttach table, ordered by class (may be NULL)

Lookup Type 6: Mark-to-Mark 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 assigned to a class of marks. As for mark glyphs in a MarkToBase attachment, the anchor point and class assignment of mark1 glyphs is defined using a MarkArray table in combination with a mark1 Coverage table. 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.

Mark-to-Mark attachment positioning Format1: Mark-to-Mark attachment

The MarkMarkPosFormat1 subtable begins with a format identifier (posFormat) and two offsets (mark1CoverageOffset, mark2CoverageOffset) to 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).

The subtable also has an offset to a MarkArray table for the mark1 glyphs (mark1ArrayOffset). For each mark glyph in the mark1Coverage table, a MarkRecord in the MarkArray table specifies its class and an offset to the Anchor table that describes the mark's attachment point. The markClassCount field specifies the total number of distinct mark classes defined in all the MarkRecords.

The MarkMarkPosFormat1 subtable also has an offset to a MarkArray table for mark2 glyph (mark2ArrayOffset), which defines for each mark2 glyph an array of anchors, one for each mark1 mark class.

MarkMarkPosFormat1 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 1
Offset16	mark1CoverageOffset	Offset to Combining Mark Coverage table, from beginning of MarkMarkPos subtable
Offset16	mark2CoverageOffset	Offset to Base Mark Coverage table, from beginning of MarkMarkPos subtable
uint16	markClassCount	Number of Combining Mark classes defined
Offset16	mark1ArrayOffset	Offset to MarkArray table for Mark1, from beginning of MarkMarkPos subtable
Offset16	mark2ArrayOffset	Offset to Mark2Array table for Mark2, from beginning of MarkMarkPos subtable

The Mark2Array table contains one Mark2Record for each mark2Coverage table. It stores the records in the same order as the mark2Coverage Index.

Mark2Array table

Type	Name	Description
uint16	mark2Count	Number of Mark2 records
Mark2Record	mark2Records [mark2Count]	Array of Mark2Records, in Coverage order.

Each Mark2Record contains an array of offsets to Anchor tables (mark2AnchorOffsets). The zero-based array of 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 referenced in the mark2AnchorOffsets 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 subclause shows a MarkMarkPosFormat1 subtable for attaching one mark to another in the Arabic script.

Mark2Record

Type	Name	Description
Offset16	mark2AnchorOffsets [markClassCount]	Array of offsets (one per class) to Anchor tables. Offsets are from beginning of Mark2Array table, in class order

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.

Position Lookup Record

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 field 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

Type	Name	Description
uint16	sequenceIndex	Index (zero-based) to input glyph sequence
uint16	lookupListIndex	Index (zero-based) into the LookupList for the Lookup table to apply to that position in the glyph sequence

Context Positioning Subtable Format 1: Simple Glyph Contexts

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-sequence 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 glyph-sequence 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 shall 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 begins processing a new glyph-sequence context. If the first 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 glyph sequence. If the context and glyph

sequence 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 (coverageOffset), a count of the number of PosRuleSets that are defined (posRuleSetCount), and an array of offsets to the PosRuleSet tables (posRuleSetOffsets). As mentioned, one PosRuleSet table shall be defined for each glyph listed in the Coverage table.

In the posRuleSetOffsets array, offsets for the PosRuleSet tables are ordered in the Coverage index order. The first PosRuleSet in the array applies to the first glyph ID listed in the Coverage table, the second PosRuleSet in the array applies to the second glyph ID listed in the Coverage table, and so on.

ContextPosFormat1 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of ContextPos subtable
uint16	posRuleSetCount	Number of PosRuleSet tables
Offset16	posRuleSetOffsets [posRuleSetCount]	Array of offsets to PosRuleSet tables. Offsets are from beginning of ContextPos subtable, ordered by Coverage Index

There is one PosRuleSet table for each glyph in the Coverage table. Each PosRuleSet table corresponds to a given glyph in the Coverage table, and describes all of the contexts that begin with that glyph.

A PosRuleSet table consists of an array of offsets to PosRule tables (posRuleOffsets), ordered by preference, and a count of the PosRule tables defined in the set (posRuleCount).

PosRuleSet Table

Type	Name	Description
uint16	posRuleCount	Number of PosRule tables
Offset16	posRuleOffsets [posRuleCount]	Array of offsets to PosRule tables. Offsets are 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 (inputSequence). The Coverage table specifies the index of the first glyph in the context, and the inputSequence array begins with the second glyph in the context sequence. As a result, the first element in the inputSequence array corresponds wtih glyph-sequence position index one (1), not zero (0). The inputSequence array lists the indices in the order the corresponding glyphs appear in the text, in writing direction (logical) order. 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 (posLookupRecords). 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 subclause demonstrates glyph kerning in context with a ContextPosFormat1 subtable.

PosRule Table

Type	Name	Description
uint16	glyphCount	Number of glyphs in the Input glyph sequence
uint16	posCount	Number of PosLookupRecords
uint16	inputSequence [glyphCount - 1]	Array of input glyph IDs – starting with the second glyph
PosLookupRecord	posLookupRecords[posCount]	Array of positioning lookups, in design order

Context positioning subtable Format 2: Class-based Glyph Contexts

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.

In a ContextPosFormat2 subtable, glyph classes are defined using a Class Definition table, defined in subclause 6.2.

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 (classDefOffset), 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 glyph ID.

The ContextPosFormat2 subtable also contains a format identifier (posFormat) and defines an offset to a Coverage table (coverageOffset). 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 (posClassSetOffsets), along with a count (including Class 0) of the PosClassSet tables (posClassSetCount). In the array, the PosClassSet tables are ordered by ascending class value (from 0 to posClassSetCount - 1).

The posClassSetOffsets 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. A PosClassSet enumerates all the PosClassRules that begin with a particular glyph class.

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

Type	Name	Description
uint16	posFormat	Format identifier: format = 2
Offset16	coverageOffset	Offset to Coverage table, from beginning of ContextPos subtable
Offset16	classDefOffset	Offset to ClassDef table, from beginning of ContextPos subtable
uint16	posClassSetCount	Number of PosClassSet tables
Offset16	posClassSetOffsets [posClassSetCount]	Array of offsets to PosClassSet tables. Offsets are 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 (posClassRuleCount) and an array of offsets to PosClassRule tables (posClassRuleOffsets). The PosClassRule tables are ordered by preference in the posClassRuleOffsets array of the PosClassSet.

PosClassSet Table

Type	Name	Description
uint16	posClassRuleCount	Number of PosClassRule tables
Offset16	posClassRuleOffsets[posClassRuleCount]	Array of offsets to PosClassRule tables. Offsets are 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 classes 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 classes array will read: classes[0] = 7, classes[1] = 5, and classes[2] = 0. The first class in the sequence, Class 2, is defined by the index into the posClassSetOffsets array. The total number and sequence of glyph classes listed in the classes 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

Type	Name	Description
uint16	glyphCount	Number of glyphs to be matched
uint16	posCount	Number of PosLookupRecords
uint16	classes [glyphCount - 1]	Array of classes to be matched to the input glyph sequence, beginning with the second glyph position
PosLookupRecord	posLookupRecords[posCount]	Array of PosLookupRecords-in design order

Context positioning subtable Format 3: Coverage-based glyph contexts

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 (coverageOffsets).

NOTE The Coverage tables referenced in the coverageOffsets 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 (posLookupRecords) in design order, or the order in which lookups are applied to the entire glyph sequence.

Example 12 at the end of this subclause changes the positions of math sign glyphs in math equations with a ContextPosFormat3 subtable.

ContextPosFormat3 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 3
uint16	glyphCount	Number of glyphs in the input sequence
uint16	posCount	Number of PosLookupRecords
Offset16	coverageOffsets [glyphCount]	Array of offsets to Coverage tables, from beginning of ContextPos subtable
PosLookupRecord	posLookupRecords [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 glyph contexts

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 tables instead of PosRule tables; and the ChainContextPosFormat1 subtable lists offsets to ChainPosRuleSet tables instead of PosRuleSet subtables.)

ChainContextPosFormat1 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of ContextPos subtable
uint16	chainPosRuleSetCount	Number of ChainPosRuleSet tables
Offset16	chainPosRuleSetOffsets [chainPosRuleSetCount]	Array of offsets to ChainPosRuleSet tables. Offsets are from beginning of ChainContextPos subtable, ordered by Coverage Index

There is one ChainPosRuleSet table for each glyph in the Coverage table. Each ChainPosRuleSet table corresponds to a given glyph in the Coverage table, and describes all of the contexts that begin with that glyph.

A ChainPosRuleSet table consists of an array of offsets to ChainPosRule tables (chainPosRuleOffsets), ordered by preference, and a count of the ChainPosRule tables defined in the set (chainPosRuleCount).

ChainPosRuleSet table

Type	Name	Description
uint16	chainPosRuleCount	Number of ChainPosRule tables
Offset16	chainPosRuleOffsets [chainPosRuleCount]	Array of offsets to ChainPosRule tables. Offsets are from beginning of ChainPosRuleSet, ordered by preference

ChainPosRule table

Type	Name	Description
uint16	backtrackGlyphCount	Total number of glyphs in the backtrack sequence
uint16	backtrackSequence [backtrackGlyphCount]	Array of backtracking glyph IDs
uint16	inputGlyphCount	Total number of glyphs in the input sequence - includes the first glyph
uint16	inputSequence [inputGlyphCount - 1]	Array of input glyph IDs - starts with second glyph)
uint16	lookaheadGlyphCount	Total number of glyphs in the look ahead sequence
uint16	lookAheadSequence [lookAheadGlyphCount]	Array of lookahead glyph IDs
uint16	posCount	Number of PosLookupRecords
PosLookupRecord	posLookupRecords [posCount]	Array of PosLookupRecords, in design order

Chaining context positioning Format 2: Class-based glyph contexts

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.

In a ChainContextPosFormat2 subtable, glyph classes are defined using a Class Definition table, defined in subclause 6.2.

To chain contexts, three classes are used in the glyph ClassDef table: backtrackClassDef, inputClassDef, and lookaheadClassDef.

ChainContextPosFormat2 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 2
Offset16	coverageOffset	Offset to Coverage table, from beginning of ChainContextPos subtable
Offset16	backtrackClassDefOffset	Offset to ClassDef table containing backtrack sequence context, from beginning of ChainContextPos subtable

Offset16	inputClassDefOffset	Offset to ClassDef table containing input sequence context, from beginning of ChainContextPos subtable
Offset16	lookaheadClassDefOffset	Offset to ClassDef table containing lookahead sequence context, from beginning of ChainContextPos subtable
uint16	chainPosClassSetCnt	Number of ChainPosClassSet tables
Offset16	chainPosClassSetOffsets [chainPosClassSetCnt]	Array of offsets to ChainPosClassSet tables. Offsets are 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

Type	Name	Description
uint16	chainPosClassRuleCount	Number of ChainPosClassRule tables
Offset16	chainPosClassRuleOffsets [chainPosClassRuleCount]	Array of offsets to ChainPosClassRule tables. Offsets are from beginning of ChainPosClassSet, ordered by preference

ChainPosClassRule table

Type	Name	Description
uint16	backtrackGlyphCount	Total number of glyphs in the backtrack sequence
uint16	backtrackSequence [backtrackGlyphCount]	Array of backtrack-sequence classes
uint16	inputGlyphCount	Total number of classes in the input sequence - includes the first class
uint16	inputSequence [inputGlyphCount - 1]	Array of input classes to be matched to the input glyph sequence, beginning with the second glyph position
uint16	lookaheadGlyphCount	Total number of classes in the look ahead sequence
uint16	lookAheadSequence [lookAheadGlyphCount]	Array of lookahead-sequence classes
uint16	posCount	Number of PosLookupRecords
PosLookupRecord	posLookupRecords [posCount]	Array of PosLookupRecords, in design order

Chaining context positioning Format 3: Coverage-based glyph contexts

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 (posLookupRecords) in design order: that is, the order in which lookups should be applied to the entire glyph sequence.

ChainContextPosFormat3 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 3
uint16	backtrackGlyphCount	Number of glyphs in the backtracking sequence
Offset16	backtrackCoverageOffsets [backtrackGlyphCount]	Array of offsets to coverage tables in backtracking sequence, in glyph sequence order
uint16	inputGlyphCount	Number of glyphs in input sequence
Offset16	inputCoverageOffsets [inputGlyphCount]	Array of offsets to coverage tables in input sequence, in glyph sequence order
uint16	lookaheadGlyphCount	Number of glyphs in lookahead sequence
Offset16	lookaheadCoverageOffsets [lookaheadGlyphCount]	Array of offsets to coverage tables in lookahead sequence, in glyph sequence order
uint16	posCount	Number of PosLookupRecords
PosLookupRecord	posLookupRecords [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 document, the subtable stored at the 32-bit offset location is termed the "extension" subtable.

This subtable type uses one format: ExtensionPosFormat1.

Extension Pospositioning Subtable Format1

ExtensionPosFormat1 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 1.
uint16	extensionLookupType	Lookup type of subtable referenced by ExtensionOffset (i.e. the extension subtable).
Offset32	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.

6.3.3.3 Shared tables: Value record, Anchor table and MarkArray table

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 table. These 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.

Value Record

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. In non-variable fonts, a ValueRecord may also contain an offset to a Device table for each of the specified values. In a variable font, it may also contain an offset to a VariationIndex table for each of the specified values.

Note that all fields of a ValueRecord are optional: to save space, only the fields that are required need be included in a given instance. Because the GPOS table uses ValueRecords for many purposes, the sizes and contents of ValueRecords may vary from subtable to subtable. A ValueRecord is always accompanied by a ValueFormat flags field that specifies which of the ValueRecord fields is present. If a ValueRecord specifies more than one value, the values should be listed in the order shown in the ValueRecord definition. If the associated ValueFormat flags indicate that a field is not present, then the next present field follows immediately after the last preceding, present field. The text-processing client must be aware of the flexible and variable nature of ValueRecords in the GPOS table.

ValueRecord

Type	Name	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 layout
int16	yAdvance	Vertical adjustment for advance, in design units - only used for vertical layout
Offset16	xPlaDeviceOffset	Offset to Device table (non-variable font) / VariationIndex table (variable font) for horizontal placement, from beginning of positioning subtable (SimplePos, PairPos – may be NULL)
Offset16	yPlaDeviceOffset	Offset to Device table (non-variable font) / VariationIndex table (variable font) for vertical placement, from beginning of positioning subtable (SimplePos, PairPos – may be NULL)
Offset16	xAdvDeviceOffset	Offset to Device table (non-variable font) / VariationIndex table (variable font) for horizontal advance, from beginning of positioning subtable (SimplePos, PairPos – may be NULL)

Offset16	yAdvDeviceOffset	Offset to Device table (non-variable font) / VariationIndex table (variable font) for vertical advance, from beginning of positioning subtable (SimplePos, PairPos – may be NULL)

NOTE 1 Device tables are used only in non-variable fonts, while VariationIndex tables are used only in variable fonts.

In variable fonts, VariationIndex tables shall be used to reference variation data for any placement or advance value that requires adjustment for different variation instances.

NOTE 2 While a separate reference to a VariationIndex table is required for each value that requires variation, two or more values that require the same variation data can have offsets that point to the same VariationIndex table, and two or more VariationIndex tables can reference the same variation data entries.

NOTE 3 If no VariationIndex table is used for a particular placement or advance value, then that value is used for all variation instances.

A ValueFormat flags field defines the types of positioning adjustment data that ValueRecords specify. SinglePos subtables will have ValueRecords for a single glyph position in a glyph sequence; PairPos subtables will have separate ValueRecords for two glyph positions. In a given subtable, the same ValueFormat applies to every ValueRecord for a given glyph poistion.

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 (in non-variable fonts) or VariationIndex tables (in variable fonts) for any of the specified values.

Each defined bit in the ValueFormat flags 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 flags shown below are used. 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 flags

Mask	Name	Description
0x0001	X_PLACEMENT	Includes horizontal adjustment for placement
0x0002	Y_PLACEMENT	Includes vertical adjustment for placement
0x0004	X_ADVANCE	Includes horizontal adjustment for advance
0x0008	Y_ADVANCE	Includes vertical adjustment for advance
0x0010	X_PLACEMENT_DEVICE	Includes Device table (non-variable font) / VariationIndex table (variable font) for horizontal placement
0x0020	Y_PLACEMENT_DEVICE	Includes Device table (non-variable font) / VariationIndex table (variable font) for vertical placement

0x0040	X_ADVANCE_DEVICE	Includes Device table (non-variable font) / VariationIndex table (variable font) for horizontal advance
0x0080	Y_ADVANCE_DEVICE	Includes Device table (non-variable font) / VariationIndex table (variable font) for vertical advance
0xFF000	Reserved	For future use (set to zero)

Anchor tables

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 X and Y coordinates, in design units, to specify a location for the anchor point in relation to the location of the outline for a given glyph. The other two formats refine the location of the anchor point using contour points (Format 2) or Device tables (Format 3). In a variable font, the third format uses a VariationIndex table (a variant of a Device table) to reference variation data for adjustment of the anchor position for the current variation instance, as needed.

Anchor table Format 1: Design Units

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

Type	Name	Description
uint16	anchorFormat	Format identifier, = 1
int16	xCoordinate	Horizontal value, in design units
int16	yCoordinate	Vertical value, in design units

Anchor table Format 2: Design Units Plus Contour Point

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. Hinting can be used to move the contour anchor point. In the rendered text, the anchor point will provide the final positioning data for a given ppem size.

Example 16 at the end of this clause uses AnchorFormat2.

AnchorFormat2 table

Type	Name	Description
uint16	anchorFormat	Format identifier, = 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: Design Units Plus Device or VariationIndex Tables

Like AnchorFormat1, AnchorFormat3 specifies a format identifier (anchorFormat) and locates an anchor point (xcoordinate and ycoordinate). And, like AnchorFormat 2, it permits fine adjustments in variable fonts 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 for the other coordinate may be set to NULL.

In variable fonts, AnchorFormat3 shall be used to reference variation data to adjust anchor points for different variation instances, if needed. In this case, AnchorFormat3 specifies an offset to a VariationIndex table, which is a variant of the Device table used for variations. If no VariationIndex table is used for a particular anchor point X or Y coordinate, then that value is used for all variation instances. While separate VariationIndex table references are required for each value that requires variation, two or more values that require the same variation-data values can have offsets that point to the same VariationIndex table, and two or more VariationIndex tables can reference the same variation data entries.

Example 17 at the end of the clause shows an AnchorFormat3 table.

AnchorFormat3 table

Type	Name	Description
uint16	anchorFormat	Format identifier, = 3
int16	xCoordinate	Horizontal value, in design units
int16	yCoordinate	Vertical value, in design units
Offset16	xDeviceOffset	Offset to Device table (non-variable font) / VariationIndex table (variable font) for X coordinate, from beginning of Anchor table (may be NULL)
Offset16	yDeviceOffset	Offset to Device table (non-variable font) / VariationIndex table (variable font) for Y coordinate, from beginning of Anchor table (may be NULL)

MarkArray table

The MarkArray table defines the class and the anchor point for a mark glyph. Three GPOS subtable types – MarkToBase attachment, MarkToLigature attachment, and MarkToMark attachment – use the MarkArray table to specify data for attaching marks.

The MarkArray table contains a count of the number of MarkRecords (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 zero (0), but the MarkRecord must explicitly assign that class value. (This differs from the Class Definition 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

Type	Name	Description
uint16	markCount	Number of MarkRecords
MarkRecord	markRecords [markCount]	Array of MarkRecords, ordered by corresponding glyphs in the associated mark Coverage order

MarkRecord

Type	Name	Description
uint16	markClass	Class defined for the associated mark
Offset16	markAnchorOffset	Offset to Anchor table, from beginning of MarkArray table

6.3.3.4 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	major / minor 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: Y_PLACEMENT
	ValueRecord	
FFB0	-80	move Y position down
	CoverageFormat2 LowerSubscriptsCoverage	Coverage table definition
0002	2	coverageFormat: ranges
0001	1	rangeCount
	rangeRecords[0]	
01B3	ZeroSubscriptGlyphID	Start, first glyph ID
01BC	NineSubscriptGlyphID	End, last glyph ID
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: X_PLACEMENT || X_ADVANCE
0003	3	valueCount
	valueRecords[0]	for dash glyph
0032	50	X_PLACEMENT
0032	50	X_ADVANCE
	valueRecords[1]	for en dash glyph
0019	25	X_PLACEMENT
0019	25	X_ADVANCE
	valueRecords[2]	for em dash glyph
000A	10	X_PLACEMENT

000A	10	X_ADVANCE
	CoverageFormat1 DashSpacingCoverage	Coverage table definition
0001	1	coverageFormat: lists
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: X_ADVANCE only
0001	0x0001	ValueFormat2: X_PLACEMENT only
0002	2	pairSetCount
000E	PPairSetTable	pairSetOffsets[0]
0016	TPairSetTable	pairSetOffsets [1]
	PairSetTable PPairSetTable	PairSet table definition
0001	1	pairValueCount
	pairValueRecords[0]	
0059	LowercaseOGlyphID	SecondGlyph
	valueRecord1	ValueRecord for first glyph
FFE2	-30	xAdvance
	valueRecord2	ValueRecord for second glyph
FFEC	-20	xPlacement

	PairSetTable PairSetTable	PairSet table definition
0001	1	pairValueCount
	pairValueRecords[0]	
0059	LowercaseOGlyphID	secondGlyph
	valueRecord1	ValueRecord for first glyph
FFD8	-40	xAdvance
	valueRecord2	ValueRecord for second glyph
FFE7	-25	xPlacement
	CoverageFormat1 PairKerningCoverage	Coverage table definition
0001	1	coverageFormat: lists
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: X_ADVANCE 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
	class1Records[0]	Class1Record, for contexts beginning with class 0
	class2Records[0]	First Class2Record for class1Records[0]; valueFormat2 is zero, so no valueRecord2
	valueRecord1	
0000	0	xAdvance: no change for first glyph
	class2Records[1]	no valueRecord2
	valueRecord1	
0000	0	xAdvance: no change for first glyph
	class1Records[1]	for contexts beginning with Class1
	class2Records[0]	no contexts with Class0 as second glyph; no valueRecord2
	valueRecord1	
0000	0	xAdvance: no change for first glyph
	class2Records[1]	contexts with Class1 as second glyph; no valueRecord2
	valueRecord1	
FFCE	-50	xAdvance: move punctuation glyph left
	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: ranges
0002	2	classRangeCount
	classRangeRecords[0]	
0046	LowercaseVGlyphID	startGlyphID
0047	LowercaseWGlyphID	endGlyphID
0001	1	class
	classRangeRecords[1]	

0049	LowercaseYGlyphID	startGlyphID
0049	LowercaseYGlyphID	endGlyphID
0001	1	class
	ClassDefFormat2 PunctuationClassDef	ClassDef table definition
0002	2	classFormat: ranges
0001	1	classRangeCount
	classRangeRecords[0]	
006A	PeriodPunctGlyphID	startGlyphID
006B	CommaPunctGlyphID	endGlyphID
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
	entryExitRecords[0]	EntryExitRecord for Kaf glyph
0016	KafEntryAnchor	offset to EntryAnchor table
001C	KafExitAnchor	offset to ExitAnchor table
	entryExitRecords[1]	EntryExitRecord for Ha glyph
0022	HaEntryAnchor	offset to EntryAnchor table
0028	HaExitAnchor	offset to ExitAnchor table
	CoverageFormat1 DiagonalWritingCoverage	Coverage table definition
0001	1	coverageFormat: lists
0002	2	glyphCount
0203	KafGlyphID	glyphArray[0]
027E	HaGlyphID	glyphArray[1]
	AnchorFormat1 KafEntryAnchor	Anchor table definition
0001	1	anchorFormat: design units only

05DC	1500	xCoordinate
002C	44	yCoordinate
	AnchorFormat1 KafExitAnchor	Anchor table definition
0001	1	anchorFormat: design units only
0000	0	xCoordinate
FFEC	-20	yCoordinate
	AnchorFormat1 HaEntryAnchor	Anchor table definition
0001	1	anchorFormat: design units only
05DC	1500	xCoordinate
002C	44	yCoordinate
	AnchorFormat1 HaExitAnchor	Anchor table definition
0001	1	anchorFormat: design units only
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	markClassCount
001A	MarkGlyphsArray	offset to MarkArray table
0030	BaseGlyphsArray	offset to BaseArray table
	CoverageFormat1 MarkGlyphsCoverage	Coverage table definition

0001	1	coverageFormat: lists
0002	2	glyphCount
0333	fathatanMarkGlyphID	glyphArray[0]
033F	kasraMarkGlyphID	glyphArray[1]
	CoverageFormat1 BaseGlyphsCoverage	Coverage table definition
0001	1	coverageFormat: lists
0001	1	glyphCount
0190	tahBaseGlyphID	glyphArray[0]
	MarkArray MarkGlyphsArray	MarkArray table definition
0002	2	markCount
	markRecords[0]	MarkRecords in Coverage index order
0000	0	markClass, for marks over base
000A	fathatanMarkAnchor	markAnchorOffset
	markRecords[1]	
0001	1	markClass, for marks under
0010	kasraMarkAnchor	markAnchorOffset
	AnchorFormat1 fathatanMarkAnchor	Anchor table definition
0001	1	anchorFormat: design units only
015A	346	xCoordinate
FF9E	-98	yCoordinate
	AnchorFormat1 kasraMarkAnchor	Anchor table definition
0001	1	anchorFormat: design units only
0105	261	xCoordinate
0058	88	yCoordinate
	BaseArray BaseGlyphsArray	BaseArray table definition
0001	1	baseCount
	baseRecords[0]	
0006	AboveBaseAnchor	baseAnchorOffset [0]
000C	BelowBaseAnchor	baseAnchorOffset [1]
	AnchorFormat1 AboveBaseAnchor	Anchor table definition

0001	1	anchorFormat: design units only
033E	830	xCoordinate
0640	1600	yCoordinate
	AnchorFormat1 BelowBaseAnchor	Anchor table definition
0001	1	anchorFormat: design units only
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	markClassCount
001A	MarkGlyphsArray	offset to MarkArray table
0030	LigGlyphsArray	offset to LigatureArray table
	CoverageFormat1 MarkGlyphsCoverage	Coverage table definition
0001	1	coverageFormat: lists
0002	2	glyphCount
033C	sukunMarkGlyphID	glyphArray[0]
033F	kasratanMarkGlyphID	glyphArray[1]
	CoverageFormat1 LigGlyphsCoverage	Coverage table definition

0001	1	coverageFormat: lists
0001	1	glyphCount
0234	LamWithMeemWithJeem LigatureGlyphID	glyphArray[0]
	MarkArray MarkGlyphsArray	MarkArray table definition
0002	2	markCount
	markRecords[0]	MarkRecords in Coverage index order
0000	0	markClass, for marks above components
000A	sukunMarkAnchor	markAnchorOffset
	markRecords[1]	
0001	1	markClass, for marks below components
0010	kasratanMarkAnchor	markAnchorOffset
	AnchorFormat1 sukunMarkAnchor	Anchor table definition
0001	1	anchorFormat: design units only
015A	346	xCoordinate
FF9E	-98	yCoordinate
	AnchorFormat1 kasratanMarkAnchor	Anchor table definition
0001	1	anchorFormat: design units only
0105	261	xCoordinate
01E8	488	yCoordinate
	LigatureArray LigGlyphsArray	LigatureArray table definition
0001	1	ligatureCount
0004	LamWithMeemWithJeemLig Attach	ligatureAttachOffsets[0]
	LigatureAttach LamWithMeemWithJeemLig Attach	LigatureAttach table definition
0003	3	componentCount
	componentRecords[0]	Right-to-left text; ComponentRecords in writing-direction (logical) order: right-most glyph first
000E	AboveLamAnchor	ligatureAnchorOffsets[0] – offsets ordered by mark class
0000	NULL	ligatureAnchorOffsets[1] – no attachment points for Class1 marks
	componentRecords[1]	
0000	NULL	ligatureAnchorOffsets[0] – no attachment points for Class0 marks
0014	BelowMeemAnchor	ligatureAnchorOffsets[1] – for Class1 marks (below)
	componentRecords[2]	

0000	NULL	ligatureAnchorOffsets[0] – no attachment points for Class0 marks
0000	NULL	ligatureAnchorOffsets[1] – no attachment points for Class1 marks
	AnchorFormat1 AboveLamAnchor	Anchor table definition
0001	1	anchorFormat: design units only
0271	625	xCoordinate
0708	1800	yCoordinate
	AnchorFormat1 BelowMeemAnchor	Anchor table definition
0001	1	anchorFormat: design units only
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	markClassCount
0018	Mark1GlyphsArray	offset to mark1Array table
0024	Mark2GlyphsArray	offset to mark2Array table
	CoverageFormat1 Mark1GlyphsCoverage	Coverage table definition
0001	1	coverageFormat: lists
0001	1	glyphCount
0296	dammaMarkGlyphID	glyphArray[0]

	CoverageFormat1 Mark2GlyphsCoverage	Coverage table definition
0001	1	coverageFormat: lists
0001	1	glyphCount
0289	hanzaMarkGlyphID	glyphArray[1]
	MarkArray Mark1GlyphsArray	MarkArray table definition
0001	1	markCount
	markRecords[0]	MarkRecords in Coverage index order
0000	0	markClass – for marks above base mark
0006	dammaMarkAnchor	markAnchorOffset
	AnchorFormat1 dammaMarkAnchor	Anchor table definition
0001	1	anchorFormat: design units only
00BD	189	xCoordinate
FF99	-103	yCoordinate
	Mark2Array Mark2GlyphsArray	Mark2Array table definition
0001	1	mark2Count
	mark2Records[0]	
0004	AboveMark2Anchor	mark2AnchorOffsets[0]
	AnchorFormat1 AboveMark2Anchor	Anchor table definition
0001	1	anchorFormat: design units only
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	posRuleSetOffsets[0]
		Coverage table offset
	CoverageFormat1 WordCoverage	
0001	1	coverageFormat: lists
0001	1	glyphCount
02A6	hehInitialGlyphID	glyphArray[0]
	PosRuleSet WordPosRuleSet	PosRuleSet table definition
0001	1	posRuleCount
0004	WordPosRule	posRuleOffsets[0]
	PosRule WordPosRule	PosRule table definition
0003	3	glyphCount
0001	1	posCount
02DD	thalFinalGlyphID	inputSequence [0]
02C6	hehIsolatedGlyphID	inputSequence [1]
	posLookupRecords[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	posClassSetCount
0000	NULL	posClassSetOffsets[0] – no contexts begin with Class0 glyphs
0060	UCWideOverhangPosClass1Set	posClassSetOffsets[1] – contexts beginning with Class1 glyphs
0070	UCNarrowOverhangPosClass2Set	posClassSetOffsets[2] – context beginning with Class2 glyphs
0000	NULL	posClassSetOffsets[3] – no contexts begin with Class3 glyphs
0000	NULL	posClassSetOffsets[4] – no contexts begin with Class4 glyphs
	CoverageFormat1 MoveAccentsCoverage	Coverage table definition
0001	1	coverageFormat: lists
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
	classRangeRecords[0]	
0029	UppercaseFGlyphID	startGlyphID
0029	UppercaseFGlyphID	endGlyphID

0002	2	class
	classRangeRecords[1]	
0033	UppercasePGlyphID	startGlyphID
0033	UppercasePGlyphID	endGlyphID
0002	2	class
	classRangeRecords[2]	
0037	UppercaseTGlyphID	startGlyphID
0037	UppercaseTGlyphID	endGlyphID
0001	1	class
	classRangeRecords[3]	
0039	UppercaseVGlyphID	startGlyphID
003A	UppercaseWGlyphID	endGlyphID
0001	1	class
	classRangeRecords[4]	
0042	LowercaseAGlyphID	startGlyphID
0042	LowercaseAGlyphID	endGlyphID
0003	3	class
	classRangeRecords[5]	
0046	LowercaseEGlyphID	startGlyphID
0046	LowercaseEGlyphID	endGlyphID
0003	3	class
	classRangeRecords[6]	
004A	LowercaseIGlyphID	startGlyphID
004A	LowercaseIGlyphID	endGlyphID
0003	3	class
	classRangeRecords[7]	
0051	LowercaseOGlyphID	startGlyphID
0051	LowercaseOGlyphID	endGlyphID
0003	3	class
	classRangeRecords[8]	
0056	LowercaseUGlyphID	startGlyphID
0056	LowercaseUGlyphID	endGlyphID
0003	3	class
	classRangeRecords[9]	
00F5	TildeAccentGlyphID	startGlyphID
00F6	UmlautAccentGlyphID	endGlyphID
0004	4	class
	PosClassSet UCWideOverhangPosClass1Set	PosClassSet table definition
0001	1	posClassRuleCount

0004	UCWideOverhangPosClassRule	posClassRuleOffsets[0]
	PosClassRule UCWideOverhangPosClassRule	PosClassRule table definition
0003	3	glyphCount
0001	1	posCount
0003	3	classes[0] – lowercase vowel
0004	4	classes[1], – accent
	posLookupRecords[0]	
0002	2	sequenceIndex
0001	1	lookupListIndex – lower the accent
	PosClassSet UCNarrowOverhangPosClass2Set	PosClassSet table definition
0001	1	posClassRuleCount
0004	UCNarrowOverhangPosClassRule	posClassRuleOffsets[0]
	PosClassRule UCNarrowOverhangPosClassRule	PosClassRule table definition
0003	3	glyphCount
0001	1	posCount
0003	3	classes[0], – lowercase vowel
0004	4	classes[1], – accent
	posLookupRecords[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	posCount

0010	XhtDescLCCoverage	coverageOffsets[0]
003C	MathSignCoverage	coverageOffsets[1]
0044	LCCoverage	coverageOffsets[2]
	posLookupRecords[0]	
0001	1	sequenceIndex
0001	1	lookupListIndex
	CoverageFormat1 XhtDescLCCoverage	Coverage table definition
0001	1	coverageFormat: lists
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: lists
0002	2	glyphCount
011E	EqualsSignGlyphID	glyphArray[0]

012D	PlusSignGlyphID	glyphArray[1]
	CoverageFormat2 LCCoverage	Coverage table definition
0002	2	coverageFormat: ranges
0001	1	rangeCount
	rangeRecords[0]	
0033	LCaGlyphID	startGlyphID
004C	LCzGlyphID	endGlyphID
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: X_PLACEMENT + Y_ADVANCE + X_PLACEMENT_DEVICE, Y_ADVANCE_DEVICE
0050	80	xPlacement
00D2	210	yAdvance
0018	XPlaDeviceTable	xPlaDeviceOffset
0020	YAdvDeviceTable	yAdvDeviceOffset

	CoverageFormat2 Cov	Coverage table definition
0002	2	coverageFormat: ranges
0001	1	rangeCount
	rangeRecords[0]	
00C8	200	startGlyphID
00D1	209	endGlyphID
0000	0	startCoverageIndex
	DeviceTableFormat1 XPlaDeviceTable	Device Table definition
000B	11	startSize
000F	15	endSize – five delta values (sizes 11 to 15)
0001	1	deltaFormat: LOCAL_2_BIT_DELTAS
	1	deltaValue[0]: increase 11ppem by 1 pixel
	1	deltaValue[1]: increase 12ppem by 1 pixel
	1	deltaValue[2]: increase 13ppem by 1 pixel
	1	deltaValue[3]: increase 14ppem by 1 pixel
5540	1	deltaValue[4]: increase 15ppem by 1 pixel
	DeviceTableFormat1 YAdvDeviceTable	Device Table definition
000B	11	startSize
000F	15	endSize
0001	1	deltaFormat: LOCAL_2_BIT_DELTAS
	1	deltaValue[0]: increase 11ppem by 1 pixel
	1	deltaValue[1]: increase 12ppem by 1 pixel
	1	deltaValue[2]: increase 13ppem by 1 pixel
	1	deltaValue[3]: increase 14ppem by 1 pixel
5540	1	deltaValue[4]: 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: design units only
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: design units plus contour points
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: design units plus device table
0117	279	xCoordinate
0515	1301	yCoordinate
000A	XDevice	xDeviceOffset (may be NULL)
0014	YDevice	yDeviceOffset (may be NULL)
	DeviceTableFormat2 XDevice	Device Table definition
000C	12	startSize
0011	17	endSize

0002	2	deltaFormat: LOCAL_4_BIT_DELTAS
	1	deltaValue[0]: increase 12ppem by 1 pixel
	1	deltaValue[1]: increase 13ppem by 1 pixel
	1	deltaValue[2]: increase 14ppem by 1 pixel
1111	1	deltaValue[3]: increase 15ppem by 1 pixel
	2	deltaValue[4]: increase 16ppem by 1 pixel
2200	2	deltaValue[5]: increase 17ppem by 1 pixel
	DeviceTableFormat2 YDevice	Device Table definition
000C	12	startSize
0011	17	endSize
0002	2	deltaFormat: LOCAL_4_BIT_DELTAS
	1	deltaValue[0]: increase 12ppem by 1 pixel
	1	deltaValue[1]: increase 13ppem by 1 pixel
	1	deltaValue[2]: increase 14ppem by 1 pixel
1111	1	deltaValue[3]: increase 15ppem by 1 pixel
	2	deltaValue[4]: increase 16ppem by 1 pixel
2200	2	deltaValue[5]: 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
	markRecords[0]	for first mark in MarkCoverage table: grave
0000	0	markClass – for marks placed above base glyphs
000A	graveMarkAnchor	markAnchorOffset
	markRecords[1]	for second mark in MarkCoverage table: cedilla
0001	1	markClass – for marks placed below base glyphs
0010	cedillaMarkAnchor	markAnchorOffset

6.3.4 GSUB – The glyph substitution table

6.3.4.1 GSUB – Table overview

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 6.28). 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 6.29).

Figure 6.28 – Isolated, initial, medial, and final forms of the Arabic character HAH

Figure 6.29 – Two Latin glyphs and their associated ligature

OFF fonts use character encoding standards, such as the Unicode Standard, that assumes a distinction between characters and glyphs: text is encoded as sequences of characters, and the 'cmap' table provides a mapping from that character to a single default glyph. Multiple characters are not directly mapped to a single glyph, as needed for ligatures; and a single character is not mapped directly to multiple glyphs, as may be needed for some complex-script. The GSUB table provides a way to describe such substitutions, enabling applications to apply such substitions during text layout and rendering to achieve desired results.

To access substitute glyphs, GSUB maps from the glyph index or indices defined in a 'cmap' subtable to the glyph index or indices of the substitute glyphs. 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 from this one default 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 seven 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 6.30).

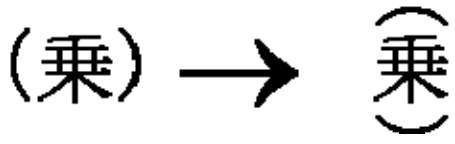


Figure 6.30 – Alternative forms of parentheses used when positioning Kanji vertically

— A *multiple substitution* replaces a single glyph with more than one glyph. This is used to specify actions such as ligature decomposition (see Figure 6.31).

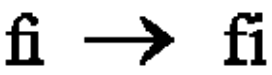


Figure 6.31 – Decomposing a Latin ligature glyph into its individual glyph components

— 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 6.32).

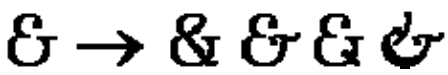


Figure 6.32 – Alternative ampersand glyphs in a font

— 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 6.33). 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 does not include those glyphs that are skipped as a result of lookup flags.

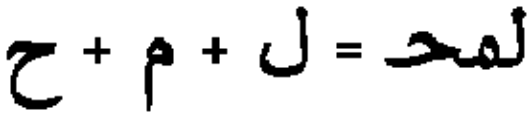


Figure 6.33 – Three Arabic glyphs and their associated ligature glyph

— *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.

GSUB table and OFF font variations

OFF Font variations allow a single font to support many design variations along one or more axes of design variation. For example, a font with weight and width variations might support weights from thin to black, and widths from ultra-condensed to ultra-expanded. For general information on OFF Font variations, see subclause 7.1.

In a variable font, it may be desirable to have different glyph-substitution actions used for different regions within the font's variation space. For example, for narrow or heavy instances in which counters become small, it may be desirable to make certain glyph substitutions to use alternate glyphs with certain strokes removed or outlines simplified to allow for larger counters. Such effects can be achieved using a FeatureVariations table

within the GSUB table. The FeatureVariations table is described in subclause 6.2. See also the Required Variation Alternates ('rvrn') feature in the OFF Layout tag registry.

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, a LookupList, and an optional FeatureVariations table (see Figure 6.34):

- 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.
- •The FeatureVariations table can be used to substitute alternate sets of lookup tables to use for any given feature under specified conditions. This currently used only in variable fonts.

For a detailed discussion of ScriptLists, FeatureLists, LookupLists, and FeatureVariations tables see subclause 6.2.

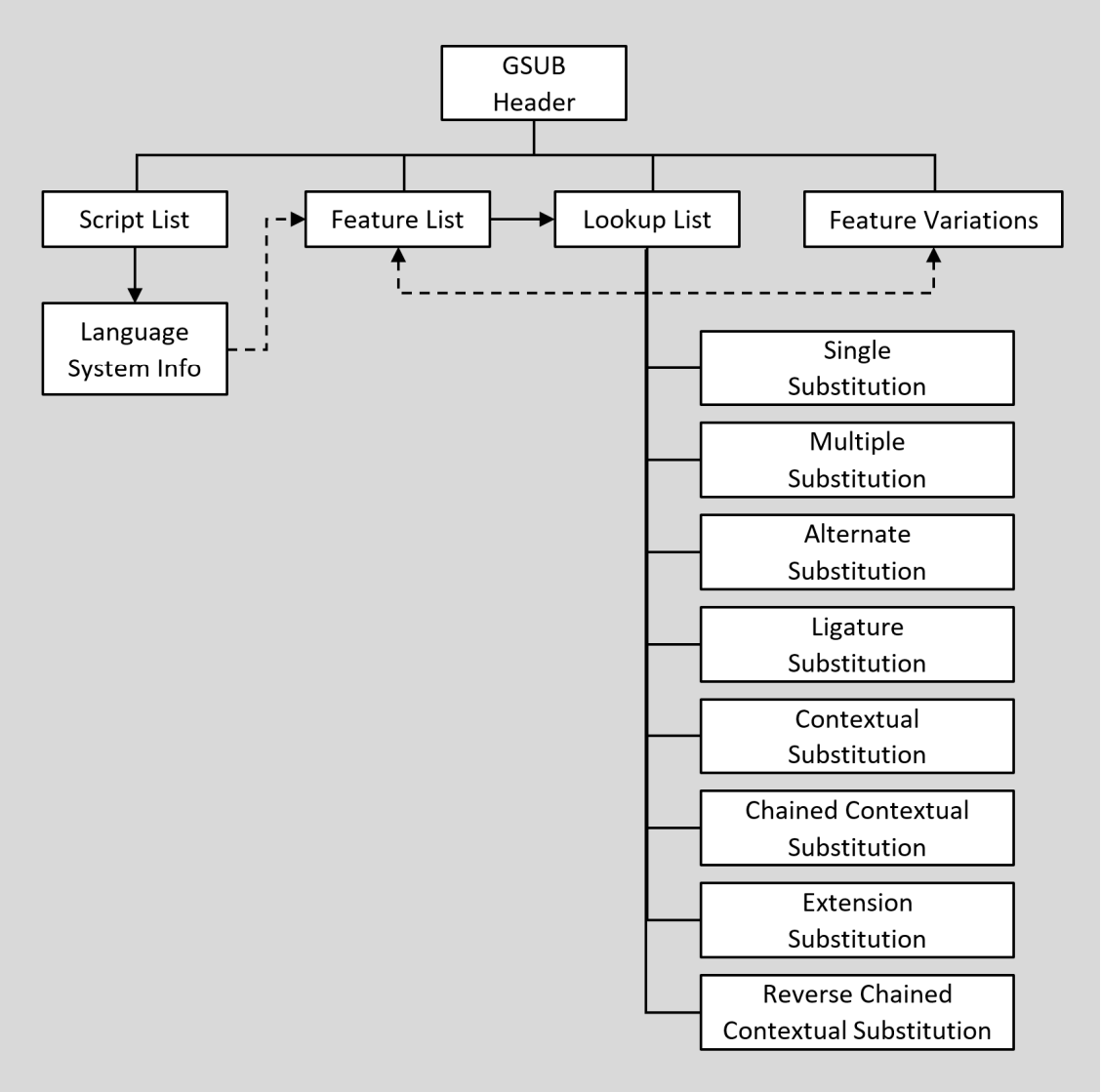


Figure 6.34 – High-level organization of GSUB table

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 table, and select the feature tables to apply to an input glyph string.
5. If a Feature Variations table is present, evaluate conditions in the Feature Variation table to determine if any of the initially-selected feature tables should be substituted by an alternate feature table.
6. Each feature provides an array of index numbers into the GSUB LookupList table. Assemble all lookups from the set of chosen features, and apply the lookups in the order given in the LookupList table.

For a detailed description of the FeatureVariations table and how it is processed, see the "FeatureVariations Table" in subclause 6.2.

Lookup data is defined in Lookup tables, which are defined in subclause 6.2. A Lookup table contains one or more Lookup subtables that define the specific conditions, type, and results of a substitution action used to implement a feature. Specific Lookup subtable types are used for glyph substitution actions, and are defined in this chapter. All subtables within a Lookup table shall be of the same lookup type, as listed in the following table for the GSUB LookupType Enumeration:

GSUB LookupType Enumeration

Value	Lookup 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 has one or more subtable 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 has a different array index 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 (LookupType 6), glyphs comprising backtrack and lookahead sequences may participate in more than one context.

The rest of this subclause describes the GSUB header and the subtables defined for each GSUB LookupType. Examples at the end of this subclause illustrate the GSUB header and six of the eight LookupTypes, including the three formats available for contextual substitutions (LookupType 5).

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 subclause shows a GSUB Header table definition.

GSUB Header, Version 1.0

Type	Name	Description
uint16	majorVersion	Major version of the GSUB table, = 1
uint16	minorVersion	Minor version of the GSUB table, = 0
Offset16	scriptList	Offset to ScriptList table, from beginning of GSUB table
Offset16	featureList	Offset to FeatureList table, from beginning of GSUB table
Offset16	lookupList	Offset to LookupList table, from beginning of GSUB table

GSUB Header, Version 1.1

Type	Name	Description
uint16	majorVersion	Major version of the GSUB table, = 1
uint16	minorVersion	Minor version of the GSUB table, = 1
Offset16	scriptList	Offset to ScriptList table, from beginning of GSUB table
Offset16	featureList	Offset to FeatureList table, from beginning of GSUB table
Offset16	lookupList	Offset to LookupList table, from beginning of GSUB table
Offset32	featureVariations	Offset to FeatureVariations table, from beginning of GSUB table (may be NULL)

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 shall 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. The deltaGlyphID is a constant value added to each input glyph index to calculate the index of the corresponding output glyph. Addition of deltaGlyphID is modulo 65536.

Example 2 at the end of this clause uses Format 1 to replace standard numerals with lining numerals.

SingleSubstFormat1 subtable:

Type	Name	Description
uint16	substFormat	Format identifier: format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of substitution subtable
int16	deltaGlyphID	Add to original glyph ID to get substitute glyph ID

Single substitution Format 2

Format 2 is more flexible than Format 1, but requires more space. It provides an array of output glyph indices (substituteGlyphIDs) 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 substituteGlyphIDs array (glyphCount), as well as the list of the output glyph indices in the substitute array (substituteGlyphIDs).

The substituteGlyphIDs array shall contain the same number of glyph indices as the Coverage table. To locate the corresponding output glyph index in the substituteGlyphIDs 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
Offset16	coverageOffset	Offset to Coverage table, from beginning of Substitution table
uint16	glyphCount	Number of glyph IDs in the substituteGlyphIDs array
uint16	substituteGlyphIDs [glyphCount]	Array of substitute glyph IDs – 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.

Multiple Substitution Format1: Multiple output glyphs

The Multiple Substitution Format1 subtable specifies a format identifier (substFormat), an offset to a Coverage table that defines the input glyph indices, a count of offsets in the sequenceOffsets array (sequenceCount), and an array of offsets to Sequence tables that define the output glyph indices (sequenceOffsets). 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 (substituteGlyphIDs).

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. The glyphCount value 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:

Type	Name	Description
uint16	substFormat	Format identifier: format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint16	sequenceCount	Number of Sequence table offsets in the sequenceOffsets array
Offset16	sequenceOffsets [sequenceCount]	Array of offsets to Sequence tables. Offsets are from beginning of substitution subtable, ordered by Coverage index

Sequence table

Type	Name	Description
uint16	glyphCount	Number of glyph IDs in the substituteGlyphIDs array. This shall always be greater than 0.
uint16	substituteGlyphIDs [glyphCount]	String of glyph IDs 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.

Alternate Substitution Format1: Alternative output glyphs

The Alternate Substitution Format1 subtable contains a format identifier (substFormat), an offset to a Coverage table containing the indices of glyphs with alternative forms (coverageOffset), a count of offsets to AlternateSet tables (alternateSetCount), and an array of offsets to AlternateSet tables (alternateSetOffets).

For each glyph, an AlternateSet subtable contains a count of the alternative glyphs (glyphCount) and an array of their glyph indices (alternateGlyphIDs). 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:

Type	Name	Description
uint16	substFormat	Format identifier: format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint16	alternateSetCount	Number of AlternateSet tables
Offset16	alternateSetOffsets [alternateSetCount]	Array of offsets to AlternateSet tables. Offsets are from beginning of substitution table, ordered by Coverage index

AlternateSet table

Type	Name	Description
uint16	glyphCount	Number of glyph IDs in the alternateGlyphIDs array
uint16	alternateGlyphIDs[glyphCount]	Array of alternate glyph IDs, 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 has one format: LigatureSubstFormat1.

Ligature Substitution Format1: All ligature substitutions in a script

It contains a format identifier (substFormat), a Coverage table offset (coverageOffset), a count of the ligature sets defined in this table (ligatureSetCount), and an array of offsets to LigatureSet tables (ligatureSetOffsets). The Coverage table specifies only the index of the first glyph component of each ligature set.

LigatureSubstFormat1 subtable:

Type	Name	Description
uint16	substFormat	Format identifier: format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of Substitution table
uint16	ligatureSetCount	Number of LigatureSet tables
Offset16	ligatureSetOffsets [ligatureSetCount]	Array of offsets to LigatureSet tables. Offsets are from beginning of substitution subtable, 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 (ligatureSetOffsets) to Ligature tables, which define the glyphs in each 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
Offset16	ligatureSetOffsets [ligatureCount]	Array of offsets to Ligature tables. Offsets are from beginning of LigatureSet table, ordered by preference

For each ligature in the set, a Ligature table specifies the glyph ID of the output ligature glyph (ligatureGlyph); a count of the total number of component glyphs in the ligature, including the first component (componentCount); and an array of glyph IDs for the components (componentGlyphIDs). The array starts with the second component glyph in the ligature (glyph sequnce index = 1, componentGlyphIDs array index = 0) because the first component glyph is specified in the Coverage table.

NOTE The componentGlyphIDs array lists glyph IDs according to the writing direction – that is, the logical order – 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
uint16	ligatureGlyph	Glyph ID of ligature to substitute
uint16	componentCount	Number of components in the ligature
uint16	componentGlyphIDs [componentCount - 1]	Array of component glyph IDs – start with the second component, ordered in writing direction

Substitution Lookup Record

Substitution subtable types 1 to 4 allow for describing glyph substitution actions without providing a way to describe glyph-sequence contexts — that is, contexts in which glyphs must occur in order for substitutions to be applied. Subtable types 5, 6 and 8, on the other hand, allow for describing glyph substitutions that occur in particular contexts. Contextual substitution subtables of types 5 and 6 specify the substitution data in a Substitution Lookup Record (SubstLookupRecord), which identifies which glyphs within a context sequence are to be acted on, and which substitution action (given in a separate Lookup table) is to be applied.

Each record contains a glyphSequenceIndex, which indicates the position within a context glyph sequence where the substitution will occur. In addition, a lookupListIndex identifies the lookup to be applied at the glyph position specified by the glyphSequenceIndex.

The contextual substitution subtables defined in Examples 7, 8, and 9 at the end of this chapter show SubstLookupRecords.

SubstLookupRecord

Type	Name	Description
uint16	glyphSequenceIndex	Index into current glyph sequence – first glyph = 0
uint16	lookupListIndex	Lookup to apply to that position – zero-based index.

The glyphSequenceIndex in a SubstLookupRecord must take into consideration the order in which lookups are applied to the entire glyph sequence. Because multiple substitutions may occur in a given context, the glyphSequenceIndex refer to the glyph sequence after the text-processing client has applied any previous lookups. In other words, the glyphSequenceIndex 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. In this example, the ligature substitution and single substition are both to be expressed as contextual substitutions, with a context that begins at the first 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 glyphSequenceIndex 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 lookup subtable specifies a context that consists of three glyphs, not four; and the SubstLookupRecord defines the glyphSequenceIndex as position 2, not position 3. This position and that context reflect 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.

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 specify substitution data in a SubstLookupRecord,described above.

Context substitution Format 1: Simple Glyph Contexts

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 (coverageOffset), a count of defined SubRuleSets (subRuleSetCount), and an array of offsets to the SubRuleSet tables (subRuleSetOffsets). 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 glyph ID listed in the Coverage table, the second SubRuleSet in the array applies to the second glyph ID listed in the Coverage table, and so on.

Example 7 at the end of this subclause shows how to use the ContextSubstFormat1 subtable to replace a sequence of three glyphs with a sequence preferred for the French language system.

ContextSubstFormat1 subtable:

Type	Name	Description
uint16	substFormat	Format identifier: format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint16	subRuleSetCount	Number of SubRuleSet tables – must equal glyphCount in Coverage table
Offset16	subRuleSetOffsets [subRuleSetCount]	Array of offsets to SubRuleSet tables. Offsets are from beginning of Substitution table, ordered by Coverage index

A SubRuleSet table consists of an array of offsets to SubRule tables (subRuleOffsets), 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
Offset16	subRuleOffsets [subRuleCount]	Array of offsets to SubRule tables. Offsets are 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 (inputSequence). 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 (glyph sequence index = 1, inputSequence array index = 0).

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 (substitutionCount) and an array of SubstLookupRecords (substLookupRecords). Each record specifies a position in the input glyph sequence and a LookupList index 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	substitutionCount	Number of SubstLookupRecords
uint16	inputSequence [glyphCount - 1]	Array of input glyph IDs – start with second glyph
SubstLookupRecord	substLookupRecords [substitutionCount]	Array of SubstLookupRecords, in design order

Context substitution Format 2: Class-based Glyph Contexts

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 (classDefOffset), 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 glyph ID.

The ContextSubstFormat2 subtable also contains a format identifier (substFormat) and defines an offset to a Coverage table (coverageOffset). 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 (subClassSetOffsets) and a count of the SubClassSet tables (subClassSetCount). 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 subClassSetCount - 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.

Example 8 at the end of this subclause uses Format 2 to substitute Arabic mark glyphs for base glyphs of different heights.

ContextSubstFormat2 subtable:

Type	Name	Description
uint16	substFormat	Format identifier: format = 2
Offset16	coverageOffset	Offset to Coverage table, from beginning of substitution table
Offset16	classDefOffset	Offset to glyph ClassDef table, from beginning of substitution table
uint16	subClassSetCount	Number of SubClassSet tables
Offset16	subClassSetOffsets [subClassSetCount]	Array of offsets to SubClassSet tables. Offsets are from beginning of substitution subtable, 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 (subClassRuleCount) and an array of offsets to SubClassRule tables (subClassRuleOffsets). The SubClassRule tables are ordered by preference in the SubClassRule array of the SubClassSet.

SubClassSet subtable

Type	Name	Description
uint16	subClassRuleCount	Number of SubClassRule tables
Offset16	subClassRuleOffsets [subClassRuleCount]	Array of offsets to SubClassRule tables. Offsets are 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 class for the second context position (glyph sequence index = 1, inputSequence array index = 0).

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 (substitutionCount) and an array of SubstLookupRecords (substLookupRecords) 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 substLookupRecords 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	substitutionCount	Number of SubstLookupRecords

uint16	inputSequence [glyphCount - 1]	Array of classes to be matched to the input glyph sequence, beginning with the second glyph position
SubstLookupRecord	substLookupRecords [substitutionCount]	Array of Substitution lookups, in design order

Context substitution Format 3: Coverage-based Glyph Contexts

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 (coverageOffsets).

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 (substitutionCount) and an array of SubstLookupRecords (substLookupRecords) 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 subclause uses ContextSubstFormat3 to substitute a swash glyph for two out of three glyphs in a sequence.

ContextSubstFormat3 subtable:

Type	Name	Description
uint16	substFormat	Format identifier: format = 3
uint16	glyphCount	Number of glyphs in the input glyph sequence
uint16	substitutionCount	Number of SubstLookupRecords
Offset16	coverageOffsets[glyphCount]	Array of offsets to Coverage table. Offsets are -from beginning of substitution subtable, in glyph sequence order
SubstLookupRecord	substLookupRecords [substitutionCount]	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. Note that LookupFlags 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 (coverageOffset), a count of defined ChainSubRuleSets (chainSubRuleSetCount), and an array of offsets to the ChainSubRuleSet tables (chainSubRuleSetOffsets). 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 glyph ID listed in the Coverage table, the second ChainSubRuleSet in the array applies to the second glyph ID listed in the Coverage table, and so on.

ChainContextSubstFormat1 subtable:

Type	Name	Description
uint16	substFormat	Format identifier: format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint16	chainSubRuleSetCount	Number of ChainSubRuleSet tables – must equal glyphCount in Coverage table
Offset16	chainSubRuleSetOffsets [chainSubRuleSetCount]	Array of offsets to ChainSubRuleSet tables. Offsets are from beginning of substitution table, ordered by Coverage index

A ChainSubRuleSet table consists of an array of offsets to ChainSubRule tables (chainSubRuleOffsets), 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
Offset16	chainSubRuleOffsets [chainSubRuleCount]	Array of offsets to ChainSubRule tables. Offsets are 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 in the context sequence (glyph sequence index = 1, inputSequence array index = 0).

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 (substitutionCount) and an array of SubstitutionLookupRecords (substLookupRecord). Each record specifies a position in the input glyph sequence and a LookupList index 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 of the input sequence)
uint16	backtrackSequence [backtrackGlyphCount]	Array of backtracking glyph IDs – to be matched before the input sequence.
uint16	inputGlyphCount	Total number of glyphs in the input sequence – includes the first glyph.
uint16	inputSequence [inputGlyphCount - 1]	Array of input glyph IDs – start with second glyph.
uint16	lookaheadGlyphCount	Total number of glyphs in the lookahead sequence (number of glyphs to be matched after the input sequence)
uint16	lookAheadSequence [lookAheadGlyphCount]	Array of lookahead glyph IDs – to be matched after the input sequence.
uint16	substitutionCount	Number of SubstLookupRecords
SubstLookupRecord	substLookupRecords [substitutionCount]	Array of SubstLookupRecords, in design order

Chaining context substitution Format 2: Class-based glyph contexts

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 (coverageOffset). 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 (chainSubClassSetOffsets) and a count of the ChainSubClassSet tables (chainSubClassSetCount). 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 chainSubClassSetCount - 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:

Type	Name	Description
uint16	substFormat	Format identifier: format = 2
Offset16	coverageOffset	Offset to Coverage table, from beginning of substitution table
Offset16	backtrackClassDef	Offset to glyph ClassDef table containing backtrack sequence data, from beginning of substitution table
Offset16	inputClassDef	Offset to glyph ClassDef table containing input

		sequence data, from beginning of substitution table
Offset16	lookaheadClassDef	Offset to glyph ClassDef table containing lookahead sequence data, from beginning of substitution table
uint16	chainSubClassSetCount	Number of ChainSubClassSet tables
Offset16	chainSubClassSetOffsets [chainSubClassSetCount]	Array of offsets to ChainSubClassSet tables. Offsets are 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 (chainSubClassRuleCount) and an array of offsets to ChainSubClassRule tables (chainSubClassRuleOffsets). The ChainSubClassRule tables are ordered by preference in the ChainSubClassRule array of the ChainSubClassSet.

ChainSubClassSet subtable

Type	Name	Description
uint16	chainSubClassRuleCount	Number of ChainSubClassRule tables
Offset16	chainSubClassRuleOffsets [chainSubClassRuleCount]	Array of offsets to ChainSubClassRule tables. Offsets are from beginning of ChainSubClassSet, ordered by preference

For each context (backtrack, input, lookahead), a ChainSubClassRule table contains a count of the glyph classes for each portion of the context sequence (backtrackGlyphCount, inputGlyphCount, lookaheadGlyphCount), including the first class. A Class array lists the classes, beginning with the class for the second context position (input glyph sequence index = 1, inputSequence array index = 0), 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 (substitutionCount) and an array of SubstLookupRecords (substLookupRecords) 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 of the input sequence).
uint16	backtrackSequence [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	inputSequence iInputGlyphCount - 1]	Array of classes to be matched with the input glyph sequence – beginning with

		second glyph position.
uint16	lookaheadGlyphCount	Total number of glyphs in the lookahead sequence (number of glyphs to be matched after the input sequence).
uint16	lookAheadSequence [lookAheadGlyphCount]	Array of lookahead classes – to be matched with glyph sequence after the input sequence.
uint16	substitutionCount	Number of SubstLookupRecords
SubstLookupRecord	substLookupRecords [substitutionCount]	Array of SubstLookupRecords, in design order.

Chaining context substitution Format 3: Coverage-based glyph contexts

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 (substitutionCount) and an array of SubstLookupRecords (substLookupRecords) in design order – that is, the order in which lookups should be applied to the entire glyph sequence.

ChainContextSubstFormat3 subtable:

Type	Name	Description
uint16	substFormat	Format identifier: format = 3
uint16	backtrackGlyphCount	Number of glyphs in the backtracking sequence
Offset16	backtrackCoverageOffsets [backtrackGlyphCount]	Array of offsets to coverage tables in backtracking sequence. Offsets are from the beginning of substitution subtable, in glyph sequence order.
uint16	inputGlyphCount	Number of glyphs in input sequence
Offset16	inputCoverageOffsets [inputGlyphCount]	Array of offsets to coverage tables in input sequence. Offsets are from the beginning of substitution subtable, in glyph sequence order
uint16	lookaheadGlyphCount	Number of glyphs in lookahead sequence
Offset16	lookaheadCoverageOffsets [lookaheadGlyphCount]	Array of offsets to coverage tables in lookahead sequence. Offsets are from the beginning of substitution subtable, in glyph sequence order.
uint16	substitutionCount	Number of SubstLookupRecords
SubstLookupRecord	substLookupRecords [substitutionCount]	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 document, the subtable stored at the 32-bit offset location is termed the "extension" subtable.

This subtable type uses one format: ExtensionSubstFormat1.

ExtensionSubstFormat1 subtable:

Type	Name	Description
uint16	substFormat	Format identifier. Set to 1.
uint16	extensionLookupType	Lookup type of subtable referenced by extensionOffset (that is, the extension subtable).
Offset32	extensionOffset	Offset to the extension subtable, of lookup type extensionLookupType, relative to the start of the ExtensionSubstFormat1 subtable.

The extensionLookupType field must be set to any lookup type other than 7. All subtables in a LookupType 7 lookup shall have the same extensionLookupType. All offsets within 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.

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.

Compared to Chaining Contextual Sustitution (lookup subtable type 6), this format is restricted to only coverage-based subtable format, input sequence can contain only a single glyph, and only single substitutions are allowed on this glyph. This constraint is integrated into the 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.

Reverse chaining contextual single substitution Format 1: Coverage-based contexts

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 processing a reverse chaining substitution, *i* began at the logical end of the string and moves to the beginning.

The subtable contains a Coverage table for the input glyph and Coverage table arrays for backtrack and lookahead sequences. It also contains an array of substitute glyphs indices (substituteGlyphIDs), which are substititions for glyphs in the Coverage table, and a count of glyphs in the substituteGlyphIDs array. The substituteGlyphIDs array shall contain the same number of glyph indices as the Coverage table. To locate the corresponding output glyph index in the substituteGlyphIDs array, this format uses the Coverage index returned from the Coverage table.

Example 10 at the end of this subclause uses ReverseChainSingleSubstFormat1 to substitute Arabic glyphs with a correct stroke thickness on the left (exit) to match the stroke thickness on the right (entry) of the following glyph (in logical order).

ReverseChainSingleSubstFormat1 subtable:

Type	Name	Description
uint16	substFormat	Format identifier; format = 1
Offset16	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint16	backtrackGlyphCount	Number of glyphs in the backtracking sequence.
Offset16	backtrackCoverageOffsets [backtrackGlyphCount]	Array of offsets to coverage tables in backtracking sequence, in glyph sequence order.
uint16	lookaheadGlyphCount	Number of glyphs in lookahead sequence.
Offset16	lookaheadCoverageOffsets [lookaheadGlyphCount]	Array of offsets to coverage tables in lookahead sequence, in glyph sequence order.
uint16	glyphCount	Number of glyph IDs in the substituteGlyphIDs array.
uint16	substituteGlyphIDs[glyphCount]	Array of substitute glyph IDs – 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	major/minor 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: calculated output glyph indices
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	substituteGlyphIDs[0], ordered by

		Coverage index
0135	VerticalClosedBracketGlyph	substituteGlyphIDs [1]
013E	VerticalOpenParenthesisGlyph	substituteGlyphIDs [2]
0143	VerticalClosedParenthesisGlyph	substituteGlyphIDs [3]
	CoverageFormat1 VerticalPunctuationCoverage	Coverage table definition
0001	1	coverageFormat: lists
0004	4	glyphCount
003C	HorizontalOpenBracketGlyph	glyphArray[0], ordered by glyph ID
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	sequenceOffsets[0] (offset to Sequence table 0)
	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	alternateSetCount – equals glyphCount in Coverage table
000E	AltAmpersandSet	alternateSetOffsets[0] (offset to AlternateSet table 0)
	CoverageFormat1 AltAmpersandCoverage	Coverage table definition
0001	1	coverageFormat: lists
0001	1	glyphCount
003A	DefaultAmpersandGlyphID	glyphArray[0]
	AlternateSet AltAmpersandSet	AlternateSet table definition
0002	2	glyphCount
00C9	AltAmpersand1GlyphID	alternateGlyphIDs[0] – glyphs in arbitrary order
00CA	AltAmpersand2GlyphID	alternateGlyphIDs [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, ligatureSetCount specifies two LigatureSet tables, one for each covered glyph, and the ligatureSetOffsets 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 ligatureOffsets 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	ligatureSetCount
0014	ELigaturesSet	ligatureSetOffsets[0] (offset to LigatureSet table 0) – LigatureSet tables in Coverage index order
0020	FLigaturesSet	ligatureSetOffsets[1]
	CoverageFormat2 LigaturesCoverage	Coverage table definition
0002	2	coverageFormat: ranges
0001	1	rangeCount
	rangeRecord[0]	
0019	eGlyphID	Start, first glyph ID
001A	fGlyphID	End, last glyph ID in range
0000	0	startCoverageIndex: coverage index of start glyphID = 0
	LigatureSet ELigaturesSet	LigatureSet table definition – all ligatures that start with e
0001	1	ligatureCount
0004	etcLigature	ligatureOffsets[0] (offset to Ligature table 0)
	Ligature etcLigature	Ligature table definition
015B	etcGlyphID	ligatureGlyph – output glyph ID
0003	3	componentCount
0028	tGlyphID	componentGlyphIDs[0] – second component in ligature
0017	cGlyphID	componentGlyphIDs[1] – third component in ligature
	LigatureSet FLigaturesSet	LigatureSet table definition all ligatures start with f
0002	2	ligatureCount
0006	ffiLigature	ligatureOffsets[0] – listed first because ffi ligature is preferred to fi ligature

000E	fiLigature	ligatureOffsets [1]
	Ligature ffiLigature	Ligature table definition
00F1	ffiGlyphID	ligatureGlyph – output glyph ID
0003	3	componentCount
001A	fGlyphID	componentGlyphIDs[0] – second component in ligature
001D	iGlyphID	componentGlyphIDs[1] – third component in ligature
	Ligature fiLigature	Ligature table definition
00F0	fiGlyphID	ligatureGlyph – output glyph ID
0002	2	componentCount
001D	iGlyphID	componentGlyphIDs[0] – 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 glyph IDs 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	subRuleSetOffsets[0] (offset to SubRuleSet table

		0) – SubRuleSets ordered by Coverage index
0020	DashAndSpaceSubRuleSet	subRuleSetOffsets[1]
	CoverageFormat1 DashCoverage	Coverage table definition
0001	1	coverageFormat: lists
0002	2	glyphCount
0028	SpaceGlyph	glyphArray[0] – glyphs in numeric order
005D	DashGlyph	glyphArray[1], dash glyph ID
	SubRuleSet SpaceAndDashSubRuleSet	SubRuleSet[0] table definition
0001	1	subRuleCount
0004	SpaceAndDashSubRule	subRuleSetOffsets[0] (offset to SubRule table 0) – SubRule tables ordered by preference
	SubRule SpaceAndDashSubRule	SubRule[0] table definition
0002	2	glyphCount – number in input sequence
0001	1	substitutionCount
005D	DashGlyph	inputSequence[0], starting with second glyph – SpaceGlyph, in Coverage table, is first glyph
	substLookupRecords[0]	
0000	0	sequenceIndex – substitution at first glyph position (0)
0001	1	lookupListIndex – index for SpaceToThinSpaceLookup in LookupList
	SubRuleSet DashAndSpaceSubRuleSet	SubRuleSet[1] table definition
0001	1	subRuleCount
0004	DashAndSpaceSubRule	subRuleOffsets[0] (offset to SubRule table 0) – SubRule tables ordered by preference
	SubRule DashAndSpaceSubRule	SubRule[0] table definition
0002	2	glyphCount – number in the input glyph sequence
0001	1	substitutionCount
0028	SpaceGlyph	inputSequence[0] – starting with second glyph
	substLookupRecords[0]	
0001	1	sequenceIndex – substitution at second glyph position (glyph sequence index = 1)
0001	1	lookupListIndex – index 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	subClassSetCount
0000	NULL	subClassSetOffsets[0] – NULL: no contexts that begin with Class 0 glyphs are defined
0000	NULL	subClassSetOffsets[1] – NULL: no contexts that begin with Class 1 glyphs are defined
0032	SetMarksHighSubClassSet2	subClassSetOffsets[2] – offset to SubClassSet table for contexts that begin with Class 2 glyphs (high base glyphs)
0040	SetMarksVeryHighSubClassSet3	subClassSetOffsets[2] – offset to SubClassSet 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
	classRangeRecords[0]	ClassRangeRecords ordered by startGlyphID; record 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: 2
	classRangeRecords[1]	ClassRangeRecord 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: 3
	classRangeRecords[2]	ClassRangeRecord for Class 1, mark glyphs
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: 1
	SubClassSet SetMarksHighSubClassSet2	SubClassSet[2] table definition all contexts that begin with Class 2 glyphs
0001	1	subClassRuleCount
0004	SetMarksHighSubClassRule2	subClassRuleOffsets[0] (offset to SubClassRule table 0) – SubClassRule tables 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	substitutionCount
0001	1	input Sequence[0] – input sequence beginning with the second Class in the input context sequence; Class 1, mark glyphs
	substLookupRecords[0]	substLookupRecords array in design order
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	subClassRuleCount
0004	SetMarksVeryHighSubClassRule3	subClassRuleOffsets[0]
	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	substitutionCount
0001	1	inputSequence[0] – input sequence beginning with the second Class in the input context sequence; Class 1 , mark glyphs
	substLookupRecords[0]	substLookupRecords array in design order
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 – number in input glyph sequence
0002	2	substitutionCount
0030	AscenderDescenderCoverage	coverageOffsets[0] – offsets to Coverage tables, in context sequence order
004C	XheightCoverage	coverageOffsets [1]

006E	DescenderCoverage	coverageOffsets [2]
	substLookupRecords[0]	SubstLookupRecords in glyph position order
0000	0	sequenceIndex
0001	1	lookupListIndex – single substitution to output ascender or descender swash
	substLookupRecords[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] – glyphs in glyph ID 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]
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]
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 glyph IDs 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	backtrackCoverageOffsets[0]
0001	1	lookaheadGlyphCount
0026	ThickEntryCoverage	lookaheadCoverageOffsets[0]
000C	12	glyphCount
00A7	BEm2	substituteGlyphIDs[0] – substitute glyphs ordered by Coverage index

00B9	BEi3	substituteGlyphIDs [1]
00C5	JIMm3	substituteGlyphIDs [2]
00D4	JIMi2	substituteGlyphIDs [3]
00EA	SINm2	substituteGlyphIDs [4]
00F2	SINi2	substituteGlyphIDs [5]
00FD	SADm2	substituteGlyphIDs [6]
010D	SADi2	substituteGlyphIDs [7]
011B	TOEm3	substituteGlyphIDs [8]
012B	TOEi3	substituteGlyphIDs [9]
013B	AINm2	substituteGlyphIDs [10]
0141	AINi2	substituteGlyphIDs [11]
	CoverageFormat1 ThickEntryCoverage	Coverage table definition
0001	1	coverageFormat: lists
001F	31	glyphCount
00A5	ALEFf1	glyphArray[0] – glyphs in glyph ID 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]
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:

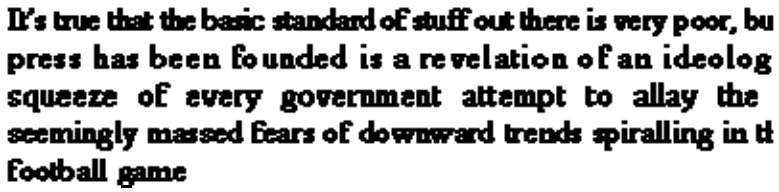


Figure 6.35 – Poorly justified text

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 6.36). 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 6.36 – 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 6.37). 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 (JstfLangSys) defines script-specific data that applies to the entire script in the absence of any language-specific information.
- A justification language system table stores the justification data for each language system.

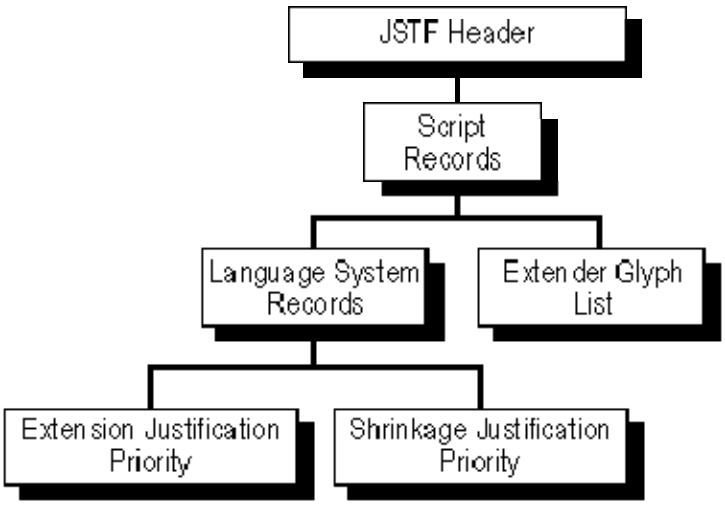


Figure 6.37 – High-level organization of JSTF table

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 1 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.

NOTE 2 A FeatureVariations table may be used in either the GSUB or GPOS table to substitute the lookups triggered by a given feature with an alternate set of lookups based on certain conditions. (Currently, these conditions can pertain only to the use of variable fonts, which are discussed further below.) The actual lookups that were applied for a given feature may be different from the default set of lookups for that feature. When processing a justification suggestion, the list of lookups to check for previous application should be the actual lookups that applied, with any feature variations in effect, not the default lookups. Also, when adding data in the JSTF table to disable GSUB or GPOS lookups, the font developer should consider possible interactions with feature variation tables, such as a need to include such alternate lookups in the set of lookups to disable. Note that this applies only to features and lookups in the GSUB and GPOS table: for lookups contained directly within the JSTF table, there is no analogous feature-variation mechanism.

Later in this subclause, the following tables and records used by the JSTF table for scripts and language systems will be described:

— Script information, including the JstfScript table (plus its associated JstfLangSysRecords) and the ExtenderGlyph table.

— Language system information, including the JstfLangSys table, JstfPriority table (and its associated JstfDataRecord), the JstfModList table, and the JstfMax table.

JSTF table and OFF font variations

OFF font variations allow a single font to support many design variations along one or more axes of design variation. For example, a font with weight and width variations might support weights from thin to black, and widths from ultra-condensed to ultra-expanded. For general information on OFF font variations, see subclause 7.1.

When different variation instances are selected, the design and metrics of individual glyphs change, and the metric characteristics of the font as a whole may also change. As metrics are relevant for justification, the interaction between justification data and variations requires consideration.

As noted above, justification is assumed to be an iterative process in which the application tests different suggestions defined in the font in priority order to find a suggestion that results in a line length that meets application-determined justification requirements. When an instance of a variable font is selected, the line layout will use glyph outlines and metrics that are adjusted for that instance. Thus, the metrics for one variation instance may be different from another, and the text content of a given line after justification may be different if formatted with different variation instances, but the justification processing proceeds in the same manner.

As also noted above, justification suggestions are applied to the results after selected GSUB and GPOS features have been processed. If the GSUB or GPOS table includes a FeatureVariations table, there may be interactions between the effects of the FeatureVariations table and the justification suggestions. See above for additional discussion.

As noted, justification suggestions can make use of GPOS lookups contained within the GPOS table or directly within the JSTF table. GPOS lookup subtables contain X or Y font-unit values that specify modifications to individual glyph positions or metrics. In a variable font, these X and Y values apply to the default instance, and may require adjustment for different variation instances. This is done using variation data with processes similar to those used for glyph outlines and other font data, as described in subclause 7.1 (Font variations overview). Variation data for adjustment of X or Y values in GPOS lookups is stored within an item variation store table located within the GDEF table. This is true for lookups in either the GPOS or the JSTF table. The same item variation store is also used for adjustment of values in the GDEF table. See the GPOS chapter for additional details regarding variation of GPOS lookup values in variable fonts.

JSTF header

The JSTF table begins with a header that contains a version number for the table, 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 (jstfScriptOffset).

Note that the jstfScriptTag values shall correspond with the script tags 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
uint16	majorVersion	Major version of the JSTF table, = 1
uint16	minorVersion	Minor version of the JSTF table, = 0
uint16	jstfScriptCount	Number of JstfScriptRecords in this table
JstfScriptRecord	jstfScriptRecords[jstfScriptCount]	Array of JstfScriptRecords, in alphabetical order by jstfScriptTag

JstfScriptRecord

Type	Name	Description
Tag	jstfScriptTag	4-byte JstfScript identification
Offset16	jstfScriptOffset	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 (extenderGlyphOffset), an offset to a default justification table for the script (defJstfLangSysOffset), 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 default JstfLangSys table and sets the jstfLangSysCount value 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 (jstfLangSysRecords), one for each language system. Each JstfLangSysRecord contains a language system tag (jstfLangSysTag) and an offset to a justification language system table (jstfLangSysOffset). In the jstfLangSysRecords 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
Offset16	extenderGlyphOffset	Offset to ExtenderGlyph table, from beginning of JstfScript table (may be NULL)
Offset16	defJstfLangSysOffset	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)
JstfLangSysRecord	jstfLangSysRecords [jstfLangSysCount]	Array of JstfLangSysRecords, in alphabetical order by jstfLangSysTag

JstfLangSysRecord

Type	Name	Description
Tag	jstfLangSysTag	4-byte JstfLangSys identifier
Offset16	jstfLangSysOffset	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 (extenderGlyphs), 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
uint16	extenderGlyphs[glyphCount]	Extender glyph IDs – 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 (jstfPriorityCount) and an array of offsets to Justification Priority tables (jstfPriorityOffsets), 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	jstfPriorityCount	Number of JstfPriority tables
Offset16	jstfPriorityOffsets[jstfPriorityCount]	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
Offset16	shrinkageEnableGSUB	Offset to shrinkage-enable JstfGSUBModList table, from beginning of JstfPriority table (may be NULL)
Offset16	shrinkageDisableGSUB	Offset to shrinkage-disable JstfGSUBModList table, from beginning of JstfPriority table (may be NULL)
Offset16	shrinkageEnableGPOS	Offset to shrinkage-enable JstfGPOSModList table, from beginning of JstfPriority table (may be NULL)
Offset16	shrinkageDisableGPOS	Offset to shrinkage-disable JstfGPOSModList table, from beginning of JstfPriority table (may be NULL)
Offset16	shrinkageJstfMax	Offset to shrinkage JstfMax table, from beginning of JstfPriority table (may be NULL)
Offset16	extensionEnableGSUB	Offset to extension-enable JstfGSUBModList table, from beginning of JstfPriority table (may be NULL)
Offset16	extensionDisableGSUB	Offset to extension-disable JstfGSUBModList table, from beginning of JstfPriority table (may be NULL)
Offset16	extensionEnableGPOS	Offset to extension-enable JstfGPOSModList table, from beginning of JstfPriority table (may be NULL)
Offset16	extensionDisableGPOS	Offset to extension-disable JstfGPOSModList table, from beginning of JstfPriority table (may be NULL)
Offset16	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	gsubLookupIndices[lookupCount]	Array of Lookup indices into the GSUB Lookup List, in increasing numerical order

JstfGPOSModList table

Type	Name	Description
uint16	lookupCount	Number of lookups for this modification
uint16	gposLookupIndices[lookupCount]	Array of Lookup indices into the GPOS Lookup List, 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
Offset16	lookupOffsets[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	major/minor version
0001	1	jstfScriptCount
	jstfScriptRecords[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 default JstfLangSys 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	extenderGlyphOffset
0012	ArabicDefJstfLangSys	Offset to default JstfLangSys table
0001	1	jstfLangSysCount
	jstfLangSysRecords[0]	
50455220	'FAR '	jstfLangSysTag
0018	FarsiJstfLangSys	jstfLangSysOffset
	ExtenderGlyph ArabicExtenders	ExtenderGlyph table definition
0002	2	glyphCount

01D3	TatweelGlyphID	extenderGlyphs[0]
01D4	LongTatweelGlyphID	extenderGlyphs[1]
	JstfLangSys ArabicDefJstfLangSys	JstfLangSys table definition
0002	2	jstfPriorityCount
000A	ArabicScriptJstfPriority1	jstfPriorityOffsets[0]
001E	ArabicScriptJstfPriority2	jstfPriorityOffsets[1]
	JstfLangSys FarsiJstfLangSys	JstfLangSys table definition
0001	1	jstfPriorityCount
002C	FarsiLangJstfPriority1	jstfPriorityOffsets[0]

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	shrinkageEnableGSUB (offset to shrinkage-enable JstfGSUBModList table)
0000	NULL	shrinkageDisableGSUB
0000	NULL	shrinkageEnableGPOS
0000	NULL	shrinkageDisableGPOS
0000	NULL	shrinkageJstfMax
0000	NULL	eExtensionEnableGSUB
0038	DisableGSUBLookupsToExtend	extensionDisableGSUB
0000	NULL	extensionEnableGPOS
0000	NULL	extensionDisableGPOS
0000	NULL	extensionJstfMax
	JstfPriority USEnglishSecondJstfPriority	JstfPriority table definition
0000	NULL	shrinkageEnableGSUB
0000	NULL	shrinkageDisableGSUB

0000	NULL	shrinkageEnableGPOS
001C	DisableGPOSLookupsToShrink	shrinkageDisableGPOS
0000	NULL	shrinkageJstfMax
0000	NULL	extensionEnableGSUB
0000	NULL	extensionDisableGSUB
002C	EnableGPOSLookupsToExtend	extensionEnableGPOS
0000	NULL	extensionDisableGPOS
0000	NULL	extensionJstfMax
	JstfGSUBModList EnableGSUBLookupsToShrink	JstfGSUBModList table definition, enable three ligature substitution lookups
0003	3	lookupCount
002E	46	gsubLookupIndices[0]
0035	53	gsubLookupIndices [1]
0063	99	gsubLookupIndices [2]
	JstfGPOSModList DisableGPOSLookupsToShrink	JstfGPOSModList table definition, disable three tight kerning lookups
0003	3	lookupCount
006C	108	gposLookupIndices[0]
006E	110	gposLookupIndices [1]
0070	112	gposLookupIndices [2]
	JstfGSUBModList DisableGSUBLookupsToExtend	JstfGSUBModList table definition, disable three ligature substitution lookups
0003	3	lookupCount
002E	46	gsubLookupIndices [0]
0035	53	gsubLookupIndices [1]
0063	99	gsubLookupIndices [2]
	JstfGPOSModList EnableGPOSLookupsToExtend	JstfGPOSModList table definition enable three tight kerning lookups
0003	3	lookupCount
006C	108	gposLookupIndices [0]
006E	110	gposLookupIndices [1]
0070	112	gposLookupIndices [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	lookupOffsets[0] (offset to JSTF Lookup table)
	Lookup WordSpaceExpandLookup	Jstf Lookup table definition
0001	1	lookupType: SinglePos Lookup
0000	0x0000	lookupFlag
0001	1	subTableCount
0008	WordSpaceExpandSubtable	subtableOffsets[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]

6.3.6 MATH – The mathematical typesetting table

6.3.6.1 MATH table overview

Mathematical formulas are complex text objects in which multiple elements with various metric, style or positioning attributes are combined. In order for a math layout engine to support layout of mathematical formulas, several types of font-specific information particular to the layout of formulas are required. The MATH table provides this font-specific information necessary for math formula layout.

Note that this is not a complete specification of math layout. The MATH table provides font data required for math layout, but detailed algorithms for use of the data are not specified. Different math-layout engine implementations can use this data to produce different layout results in accordance with different purposes or goals.

Layout of math formulas is quite different from regular text layout that is done using tables such as GSUB and GPOS. Regular text layout mainly deals with a line of text, often formatted with a single font. In this situation, actions such as contextual substitution or kerning can be done with access to the complete context of the line of text, and the rules can be expressed in terms of known glyph sequences. Math layout is quite different from this.

The general structure of math formulas is hierarchical, with formulas composed of smaller sub-formula expressions, where each expression may be composed of even simpler expressions, and so on down to individual strings – operator symbols, variable names and numbers.

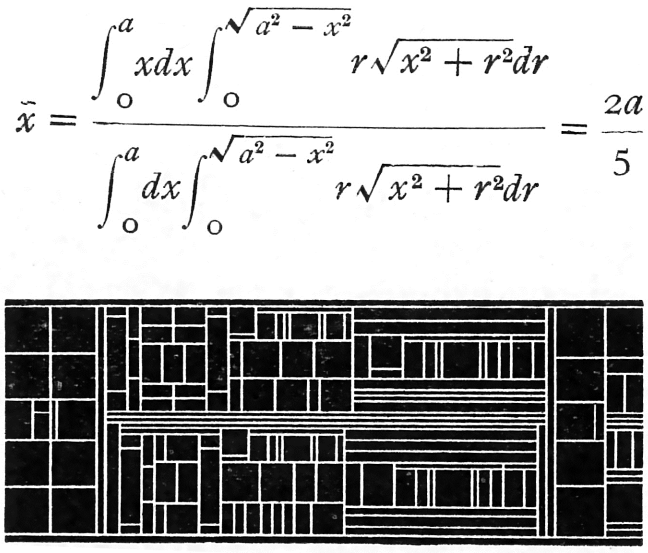


Figure 6.38 – Example of a complex hierarchical math formula and the diagram showing the pieces of metal type and spacing materials used in a traditional printing application

Likewise, the process of math formula layout is also recursive. Child components are formatted first and then arranged to form their parent's layout with this process repeated on every level starting from simplest blocks up to the whole formula. Each sub-formula has its own component structure and rules for how to perform layout. For example, a fraction expression consists of numerator and denominator sub-expressions, which will be placed on top one another with a fraction rule separating them. An integral expression contains an integral sign, optional upper- and lower- limit expressions, and a following sub-expression.

The simplest blocks within a formula typically contain strings. In isolation, each could be layed out by regular text-layout processing using other tables such as GSUB and GPOS. Math layout goes beyond this. It deals mainly with layout processing involved in composing these simple blocks together into a complex, hierarchical formula. The data provided in the MATH table allows math-layout processing to be typographically aware, so that presentation with high typographic quality can be achieved.

A math layout engine works with boxes representing individual formula components as units of layout. During the layout process, individual boxes can be arranged relative to each other- they can be stretched depending on the sizes of other boxes they interact with; or they can use different glyph variants, based on the box size or position in the formula. The MATH table provides data that informs how these operations are done. Each box has an associated font that supplies information comprising typographic requirements for that box. These may be requirements for that box alone, based on the results of layout operations internal to the box; or they may be requirements on how other boxes interact with that box in layout.

By providing information about a font generally or about specific glyphs in the font, the MATH table can enable math-layout engines to produce layout that is appropriate for the font design, and that realizes the font's full typographic potential by using stylistic and stretching glyph variants provided by the font.

6.3.6.2 MATH – Table organization and structure

6.3.6.2.1 Shared Formats: MathValueRecord, Coverage

MathValueRecord

MATH subtables use math value records to describe valuesthat place or adjust elements of math formulas. These values are expressed in font design units.

A MathValueRecord may also specify an offset to a device table which provides corrections to a given value at particular display resolutions. The device table format is described in subclause 6.2. The device table can provide multiple correction values to be used for several different PPEM sizes. Different formats for representing the correction values are provided to allow for efficient representation, according to the size requirements of the correction values. When used in MathValueRecords, format 1 is recommended.

When designing a MATH table, device tables may be specified for many values used for positioning elements of a formula. This creates the potential that many device corrections may accumulate in a given layout. However, such accumulation would result in metrics for formula elements that are significantly different from scaled-adjusted dimensions of the same elements rendered on a high-resolution device. It would produce inconsistencies between screen and print renditions, and could also lead to clipping. For these reasons, accumulation of many corrections is undesirable.

While a font may specify device corrections, use of these corrections is under the control of a given layout engine implementtion. To maintain consistency across devices with different resolutions, an engine may limit the number of device corrections that are accumulated, or may ignore them altogether. A layout engine can also supply its own corrections where none are indicated in the font. Since the accumulated size of corrections should be kept to a minimum it is recommended that device tables referenced by a MathValueRecord use format 1 for representation of correction values. This format allows corrections of at most -2to +1 pixels. The recommended values to use in MathValueRecords are -1, 0, or 1.

MathValueRecord

Type	Name	Description
int16	value	The X or Y value in design units
Offset16	deviceTableOffset	Offset to the device table – from the beginning of parent table. May be NULL. Suggested format for device table is 1.

Coverage tables

MATH subtables make use of Coverage tables, defined in subclause 6.2, to specify sets of glyphs. All Coverage table formats may be used in the MATH table.

6.3.6.2.2 MATH Header

The MATH table begins with a header that consists of a version number for the table (majorVersion/minorVersion), which is currently set to 1.0, and specified offsets to the following tables that store information on positioning of math formula elements:

- MathConstants table stores font parameters to be used in typesetting of each supported mathematical function, such as a fraction or a radical.
- MathGlyphInfo table supplies positioning information that is defined on a per-glyph basis.
- MathVariants table contains information to be used in constructing glyph variants of different height or width, either by finding a pre-defined glyph with desired measurements in the font, or by assembling the required shape from pieces found in the glyph set.

Math Header

Type	Name	Description
uint16	majorVersion	Major version of the MATH table, = 1
uint16	minorVersion	Minor version of the MATH table, = 0
Offset16	mathConstantsOffset	Offset to MathConstants table, from the beginning of MATH table.
Offset16	mathGlyphInfoOffset	Offset to MathGlyphInfo table, from the beginning of MATH table.
Offset16	mathVariantsOffset	Offset to MathVariants table, from the beginning of MATH table.

6.3.6.2.3 MathConstants Table

The MathConstants table defines a number of constants required to properly position elements of mathematical formulas. These constants belong to several groups of semantically-related values, such as values for positioning of accents, positioning of superscripts and subscripts, and positioning of elements of fractions. The table also contains general-use constants that may affect all parts of the formula, such as axis height and math leading. Note that most of the constants deal with aspects of vertical positioning.

In most cases, values in the MathConstants table are assumed to be positive. For example, for descenders and shift-down values a positive constant signifies movement in a downwards direction. Most values in the MathConstants table are represented by a MathValueRecord, which allows the font designer to supply device corrections to those values when necessary.

For values that pertain to layout interactions between a base and dependent elements (e.g. superscripts or limits), the specific value used is taken from the font associated with the base, and the size of the value is relative to the size of the base.

The following naming convention are used for fields in the MathConstants table:

- **Height** – Specifies a distance from the main baseline.
- **Kern** – Represents a fixed amount of empty space to be introduced.
- **Gap** – Represents an amount of empty space that may need to be increased to meet certain criteria.
- **Drop and Rise** – Specifies the relationship between measurements of two elements to be positioned relative to each other (but not necessarily in a stack-like manner) that must meet certain criteria. For a Drop, one of the positioned elements has to be moved down to satisfy those criteria; for a Rise, the movement is upwards.
- **Shift** – Defines a vertical shift applied to an element sitting on a baseline.
- **Dist** – Defines a distance between baselines of two elements.

MathConstants Table

Type	Name	Description
int16	scriptPercentScaleDown	Percentage of scaling down for level 1 superscripts and subscripts. Suggested value: 80%.
int16	scriptScriptPercentScaleDown	Percentage of scaling down for level 2 (scriptScript) superscripts and subscripts. Suggested value: 60%.
uint16	delimitedSubFormulaMinHeight	Minimum height required for a delimited expression (contained within parentheses, etc.) to be treated as a sub-formula. Suggested value: normal line height × 1.5.
uint16	displayOperatorMinHeight	Minimum height of n-ary operators (such as integral and summation) for formulas in display mode (that is, appearing as standalone page elements, not embedded inline within text).
MathValueRecord	mathLeading	White space to be left between math formulas to ensure proper line spacing. For example, for applications that treat line gap as a part of line ascender, formulas with ink going above (os2.sTypoAscender + os2.sTypoLineGap – MathLeading) or with ink going below

		os2.sTypoDescender will result in increasing line height.
MathValueRecord	axisHeight	Axis height of the font. In math typesetting, the term axis refers to a horizontal reference line used for positioning elements in a formula. The math axis is similar to but distinct from the baseline for regular text layout. For example, in a simple equation, a minus symbol or fraction rule would be on the axis, but a string for a variable name would be set on a baseline that is offset from the axis. The axisHeight value determines the amount of that offset.
MathValueRecord	accentBaseHeight	Maximum (ink) height of accent base that does not require raising the accents. Suggested: x-height of the font (os2.sxHeight) plus any possible overshots.
MathValueRecord	flattenedAccentBaseHeight	Maximum (ink) height of accent base that does not require flattening the accents. Suggested: cap height of the font (os2.sCapHeight).
MathValueRecord	subscriptShiftDown	The standard shift down applied to subscript elements. Positive for moving in the downward direction. Suggested: os2.ySubscriptYOffset.
MathValueRecord	subscriptTopMax	Maximum allowed height of the (ink) top of subscripts that does not require moving subscripts further down. Suggested: 4/5 x- height.
MathValueRecord	subscriptBaselineDropMin	Minimum allowed drop of the baseline of subscripts relative to the (ink) bottom of the base. Checked for bases that are treated as a box or extended shape. Positive for subscript baseline dropped below the base bottom.
MathValueRecord	superscriptShiftUp	Standard shift up applied to superscript elements. Suggested: os2.ySuperscriptYOffset.
MathValueRecord	superscriptShiftUpCramped	Standard shift of superscripts relative to the base, in cramped style.
MathValueRecord	superscriptBottomMin	Minimum allowed height of the (ink) bottom of superscripts that does not require moving subscripts further up. Suggested: ¼ x-height.
MathValueRecord	superscriptBaselineDropMax	Maximum allowed drop of the baseline of superscripts relative to the (ink) top of the base. Checked for bases that are treated as a box or extended shape. Positive for superscript baseline below the base top.

MathValueRecord	subSuperscriptGapMin	Minimum gap between the superscript and subscript ink. Suggested: 4 × default rule thickness.
MathValueRecord	superscriptBottomMaxWith Subscript	The maximum level to which the (ink) bottom of superscript can be pushed to increase the gap between superscript and subscript, before subscript starts being moved down. Suggested: 4/5 x-height.
MathValueRecord	spaceAfterScript	Extra white space to be added after each subscript and superscript. Suggested: 0.5 pt for a 12 pt font.
MathValueRecord	upperLimitGapMin	Minimum gap between the (ink) bottom of the upper limit, and the (ink) top of the base operator.
MathValueRecord	upperLimitBaselineRiseMin	Minimum distance between baseline of upper limit and (ink) top of the base operator.
MathValueRecord	lowerLimitGapMin	Minimum gap between (ink) top of the lower limit, and (ink) bottom of the base operator.
MathValueRecord	lowerLimitBaselineDropMin	Minimum distance between baseline of the lower limit and (ink) bottom of the base operator.
MathValueRecord	stackTopShiftUp	Standard shift up applied to the top element of a stack.
MathValueRecord	stackTopDisplayStyleShiftUp	Standard shift up applied to the top element of a stack in display style.
MathValueRecord	stackBottomShiftDown	Standard shift down applied to the bottom element of a stack. Positive for moving in the downward direction.
MathValueRecord	stackBottomDisplayStyleShiftDown	Standard shift down applied to the bottom element of a stack in display style. Positive for moving in the downward direction.
MathValueRecord	stackGapMin	Minimum gap between (ink) bottom of the top element of a stack, and the (ink) top of the bottom element. Suggested: 3 × default rule thickness.
MathValueRecord	stackDisplayStyleGapMin	Minimum gap between (ink) bottom of the top element of a stack, and the (ink) top of the bottom element in display style. Suggested: 7 × default rule thickness.
MathValueRecord	stretchStackTopShiftUp	Standard shift up applied to the top element of the stretch stack.

MathValueRecord	stretchStackBottomShiftDown	Standard shift down applied to the bottom element of the stretch stack. Positive for moving in the downward direction.
MathValueRecord	stretchStackGapAboveMin	Minimum gap between the ink of the stretched element, and the (ink) bottom of the element above. Suggested: same value as upperLimitGapMin.
MathValueRecord	stretchStackGapBelowMin	Minimum gap between the ink of the stretched element, and the (ink) top of the element below. Suggested: same value as lowerLimitGapMin.
MathValueRecord	fractionNumeratorShiftUp	Standard shift up applied to the numerator.
MathValueRecord	fractionNumeratorDisplayStyle ShiftUp	Standard shift up applied to the numerator in display style. Suggested: same value as stackTopDisplayStyleShiftUp.
MathValueRecord	fractionDenominatorShiftDown	Standard shift down applied to the denominator. Positive for moving in the downward direction.
MathValueRecord	fractionDenominatorDisplay StyleShiftDown	Standard shift down applied to the denominator in display style. Positive for moving in the downward direction. Suggested: same value as stackBottomDisplayStyleShiftDown.
MathValueRecord	fractionNumeratorGapMin	Minimum tolerated gap between the (ink) bottom of the numerator and the ink of the fraction bar. Suggested: default rule thickness.
MathValueRecord	fractionNumDisplayStyle GapMin	Minimum tolerated gap between the (ink) bottom of the numerator and the ink of the fraction bar in display style. Suggested: 3 × default rule thickness.
MathValueRecord	fractionRuleThickness	Thickness of the fraction bar. Suggested: default rule thickness.
MathValueRecord	fractionDenominatorGapMin	Minimum tolerated gap between the (ink) top of the denominator and the ink of the fraction bar. Suggested: default rule thickness.
MathValueRecord	fractionDenomDisplayStyleGap Min	Minimum tolerated gap between the (ink) top of the denominator and the ink of the fraction bar in display style. Suggested: 3 × default rule thickness.
MathValueRecord	skewedFractionHorizontalGap	Horizontal distance between the top and bottom elements of a skewed fraction.
MathValueRecord	skewedFractionVerticalGap	Vertical distance between the ink of the top and bottom elements of a skewed fraction.

MathValueRecord	overbarVerticalGap	Distance between the overbar and the (ink) top of he base. Suggested: 3 × default rule thickness.
MathValueRecord	overbarRuleThickness	Thickness of overbar. Suggested: default rule thickness.
MathValueRecord	overbarExtraAscender	Extra white space reserved above the overbar. Suggested: default rule thickness.
MathValueRecord	underbarVerticalGap	Distance between underbar and (ink) bottom of the base. Suggested: 3 × default rule thickness.
MathValueRecord	underbarRuleThickness	Thickness of underbar. Suggested: default rule thickness.
MathValueRecord	underbarExtraDescender	Extra white space reserved below the underbar. Always positive. Suggested: default rule thickness.
MathValueRecord	radicalVerticalGap	Space between the (ink) top of the expression and the bar over it. Suggested: 1¼ default rule thickness.
MathValueRecord	radicalDisplayStyleVerticalGap	Space between the (ink) top of the expression and the bar over it. Suggested: default rule thickness + ¼ x-height.
MathValueRecord	radicalRuleThickness	Thickness of the radical rule. This is the thickness of the rule in designed or constructed radical signs. Suggested: default rule thickness.
MathValueRecord	radicalExtraAscender	Extra white space reserved above the radical. Suggested: same value as radicalRuleThickness.
MathValueRecord	radicalKernBeforeDegree	Extra horizontal kern before the degree of a radical, if such is present.
MathValueRecord	radicalKernAfterDegree	Negative kern after the degree of a radical, if such is present. Suggested: −10/18 of em.
int16	radicalDegreeBottomRaise Percent	Height of the bottom of the radical degree, if such is present, in proportion to the ascender of the radical sign. Suggested: 60%.

6.3.6.2.4 MathGlyphInfo Table

The MathGlyphInfo table contains positioning information that is defined on per-glyph basis. The table consists of the following parts:

- A MathItalicsCorrectionInfo table that contains information on italics correction values.
- A MathTopAccentAttachment table that contains horizontal positions for attaching mathematical accents.
- A Extended Shape coverage table. The glyphs covered by this table are to be considered extended shapes.

— A MathKernInfo table that provides per-glyph information for mathematical kerning.

MathGlyphInfo Table

Type	Name	Description
Offset16	mathItalicsCorrectionInfoOffset	Offset to MathItalicsCorrectionInfo table, from the beginning of the MathGlyphInfo table.
Offset16	mathTopAccentAttachmentOffset	Offset to MathTopAccentAttachment table, from the beginning of the MathGlyphInfo table.
Offset16	extendedShapeCoverageOffset	Offset to ExtendedShapes coverage table, from the beginning of the MathGlyphInfo table. When the glyph to the left or right of a box is an extended shape variant, the (ink) box should be used for vertical positioning purposes, not the default position defined by values in MathConstants table. May be NULL.
Offset16	mathKernInfoOffset	Offset to MathKernInfo table, from the beginning of MathGlyphInfo table.

6.3.6.2.5 MathItalicsCorrectonInfo Table

The MathItalicsCorrectionInfo table contains italics correction values for slanted glyphs used in math layout. The top portion of slanted glyphs may protrude beyond the glyph's advance width. This can result in collision with other interacting elements, or an appearance in the placement of other interacting elements that is unpleasing unless some accommodation is made for the protrustion. The MathItalicsCorrectionInfo table provides correction values to accommodate for such protrusion.

The table consists of the following parts:

— Coverage of glyphs for which the italics correction values are provided. It is assumed to be zero for all other glyphs.

— Count of covered glyphs.

— Array of italic correction values for each covered glyph, in order of coverage. The italics correction value can be used as an adjustment for positioning of interacting elements to make allowance for protrusion to the right of the top part of the glyph. For example, taller letters tend to have larger italics correction, and a *V* will probably have larger italics correction than an *L*.

Italics correction can be used in the following situations:

— When a run of slanted characters is followed by a straight character (such as an operator or a delimiter), the italics correction of the last glyph is added to its advance width.

— When positioning limits on an N-ary operator (e.g., integral sign), the horizontal position of the upper limit is moved to the right by ½ of the italics correction, while the position of the lower limit is moved to the left by the same distance.

— When positioning superscripts and subscripts, their default horizontal positions are also different by the amount of the italics correction of the preceding glyph.

MathItalicsCorrectionInfo Table

Type	Name	Description
Offset16	italicCorrectionCoverageOffset	Offset to Coverage table, from the beginning of MathItalicsCorrectionInfo table.

uint16	italicsCorrectionCount	Number of italics correction values. Should coincide with the number of covered glyphs.
MathValueRecord	italicsCorrection [italicsCorrectionCount]	Array of MathValueRecords defining italics correction values for each covered glyph.

6.3.6.2.6 MathTopAccentAttachment Table

The MathTopAccentAttachment table contains information on horizontal positioning of top math accents. The table consists of the following parts:

- Coverage of glyphs for which information on horizontal positioning of math accents is provided. To position accents over any other glyph, its geometrical center (with respect to advance width) can be used.
- Count of covered glyphs.
- Array of top accent attachment points for each covered glyph, in order of coverage. These attachment points are to be used for finding horizontal positions of accents over characters. It is done by aligning the attachment point of the base glyph with the attachment point of the accent. Note that this is very similar to mark-to-base attachment, but here alignment only happens in the horizontal direction, and the vertical positions of accents are determined by different means.

MathTopAccentAttachment Table

Type	Name	Description
Offset16	topAccentCoverageOffset	Offset to Coverage table, from the beginning of MathTopAccentAttachment table.
uint16	topAccentAttachmentCount	Number of top accent attachment point values. Must be the same as the number of glyph IDs referenced in the Coverage table.
MathValueRecord	topAccentAttachment [topAccentAttachmentCount]	Array of MathValueRecords defining top accent attachment points for each covered glyph.

6.3.6.2.7 ExtendedShapeCoverage Table

The glyphs covered by this table are to be considered extended shapes. These glyphs are variants extended in the vertical direction, e.g., to match height of another part of the formula. Because their dimensions may be very large in comparison with normal glyphs in the glyph set, the standard positioning algorithms will not produce the best results when applied to them. In the vertical direction, other formula elements will be positioned not relative to those glyphs, but instead to the ink box of the subexpression containing them.

For example, consider a fraction enclosed in parentheses with a superscript. Notice how the superscripts on '*z*' and '*Z*' are aligned vertically, although they have different heights. If the right parenthesis was not considered an extended shape, the superscript would be put in position aligned with any other superscript on the line, like this:

$$\left(\frac{x}{y}\right)^{2}+z^{2}+Z^{2}$$

Because this is undesirable, the right parenthesis in this case should be considered an extended shape, resulting in superscript positioned relative to the whole subexpression:

$$\left(\frac{x}{y}\right)^2 + z^2 + Z^2$$

6.3.6.2.8 MathKernInfo Table

The MathKernInfo table provides mathematical kerning values used for kerning of subscript and superscript glyphs relative to a base glyph. Its purpose is to improve spacing in situations such as ω^f or V_A.

Mathematical kerning is height dependent; that is, different kerning amounts can be specified for different heights within a glyph's vertical extent. For any given glyph, different values can be specified for four corner positions — top-right, to-left, etc. — allowing for different kerning adjuments according to whether the glyph occurs as a subscript, a superscript, a base being kerned with a subscript, or a base being kerned with a superscript.

The kerning algorithm for subscripts and superscripts is as follows:

- Calculate the vertical positions of subscripts and superscripts using the MathConstants table.
- Set the default horizontal position for the subscript immediately after the base glyph.
- Set the default horizontal position for a superscript after the base, applying a shift for italics correction if indicated for the base glyph in the MathItalicsCorrectionInfo table.
- Calculate a superscript kerning value as follows:
  - Evaluate two correction heights (illustrated in the figure below):
    - At the bottom of the superscript-glyph bounding box. (The corresponding height for the base glyph is the distance from the base-glyph baseline to the bottom of the superscript bounding box.)
    - At the top of the base-glyph bounding box. (The corresponding height for the superscript glyph is the distance from the superscript baseline to the top of the base-glyph bounding box.)
  - For each correction height, add the top-right kerning value for the base glyph to the bottom-left kerning value for the superscript glyph.
  - Take the minimum of these two sums: kern the base and superscript by that amount.
- Calculate a subscript kerning value as follows:
  - Evaluate two correction heights:
    - At the top of the subscript-glyph bounding box. (The corresponding height for the base glyph is the distance from the base-glyph baseline to the top of the subscript bounding box.)
    - At the bottom of the base-glyph bounding box. (The corresponding height for the subscript glyph is the distance from the subscript baseline to the bottom of the base-glyph bounding box.)
  - For each correction height, add the bottom-right kerning value for the base glyph to the top-left kerning value for the subscript glyph.
  - Take the minimum of these two sums: kern the base and subscript by that amount.

NOTE If a base expression, subscript expression or superscript expression is a box, a math-layout engine may use kerning values of zero for each corner of the box, or may calculate height-dependent kerning amounts by some means

The following figure illustrates the correction heights for a base and superscript:

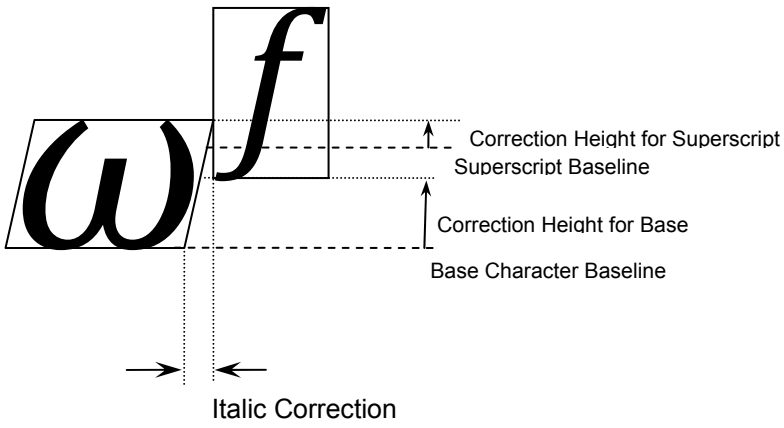


Fig. 6.39 Example of horizontal and vertical kerning adjustments for superscript positioning

The MathKernInfo table consists of the following fields:

- A Coverage table of glyphs for which mathematical kerning information is provided. Mathematical kerning amounts are assumed to be zero for all other glyphs.
- Count of MathKernInfoRecords.
- Array of MathKernInfoRecords for each covered glyph, the order of glyphs in the Coverage table.

MathKernInfo Table

Type	Name	Description
Offset16	mathKernCoverageOffset	Offset to Coverage table, from the beginning of the MathKernInfo table.
uint16	mathKernCount	Number of mathKernInfoRecords. Must be the same as the number of glyph IDs referenced in the Coverage table.
MathKernInfoRecord	mathKernInfoRecords [mathKernCount]	Array of MathKernInfoRecords, one for each covered glyph.

MathKernInfoRecord

Each MathKernInfoRecord points to up to four kern tables for each of the corners around the glyph. If no kern table is provided for a corner, a kerning amount of zero is assumed.

MathKernInfoRecord

Type	Name	Description
Offset16	topRightMathKernOffset	Offset to MathKern table for top right corner, from the beginning of MathKernInfo table. May be NULL.
Offset16	topLeftMathKernOffset	Offset to MathKern table for the top left corner, from the beginning of MathKernInfo table. May be NULL.
Offset16	bottomRightMathKernOffset	Offset to MathKern table for bottom right corner, from the beginning of MathKernInfo table. May be NULL.

Offset16	bottomLeftMathKernOffset	Offset to MathKern table for bottom left corner, from the beginning of MathKernInfo table. May be NULL.

6.3.6.2.9 MathKern Table

The MathKern table provides kerning amounts for different heights in a glyph's vertical extent. An array of kerning values is provided, each of which applies to a height range. A corresponding array of heights indicate the transition points between consecutive ranges.

Correction heights for each glyph are relative to the glyph baseline, with positive height values above the baseline, and negative height values below the baseline. The correctionHeights array is sorted in increasing order, from lowest to highest.

The kerning value corresponding to a particular height is determined by finding two consecutive entries in the correctionHeight array such that the given height is greater than or equal to the first entry and less than the second entry. The index of the second entry is used to look up a kerning value in the kernValues array. If the given height is less than the first entry in the correctionHeights array, the first kerning value (index 0) is used. For a height that is greater than or equal to the last entry in the correctionHeights array, the last entry is used.

MathKern Table

Type	Name	Description
uint16	heightCount	Number of heights at which the kern value changes.
MathValueRecord	correctionHeight[heightCount]	Array of correction heights, in design units, sorted from lowest to highest.
MathValueRecord	kernValue[heightCount+1]	Array of kern values for height ranges. Negative values are used to move glyphs closer to each other.

6.3.6.2.10 MathVariants Table

The MathVariants table solves the following problem: given a particular default glyph shape and a certain width or height, find a variant shape glyph (or construct created by putting several glyph together) that has the required measurement. This functionality is needed for growing the parentheses to match the height of the expression within, growing the radical sign to match the height of the expression under the radical, stretching accents like tilde when they are put over several characters, for stretching arrows, horizontal curly braces, and so forth.

The MathVariants table consists of the following fields:

- Count and coverage of glyphs that can grow in the vertical direction.
- Count and coverage of glyphs that can grow in the horizontal direction.
- The minConnectorOverlap defines by how much two glyphs need to overlap with each other when used to construct a larger shape. Each glyph to be used as a building block in constructing extended shapes will have a straight part at either or both ends. This connector part is used to connect that glyph to other glyphs in the assembly. These connectors need to overlap to compensate for rounding errors and hinting corrections at a lower resolution. The minConnectorOverlap value tells how much overlap is necessary for this particular font.
- Two arrays of offsets to MathGlyphConstruction tables: one array for glyphs that grow in the vertical direction, and the other array for glyphs that grow in the horizontal direction. The arrays must be arranged in coverage order and have specified sizes.

MathVariants Table

Type	Name	Description
uint16	minConnectorOverlap	Minimum overlap of connecting glyphs during glyph construction, in design units.
Offset16	vertGlyphCoverageOffset	Offset to Coverage table, from the beginning of MathVariants table.
Offset16	horizGlyphCoverageOffset	Offset to Coverage table, from the beginning of MathVariants table.
uint16	vertGlyphCount	Number of glyphs for which information is provided for vertically growing variants. Must be the same as the number of glyph IDs referenced in the vertical Coverage table.
uint16	horizGlyphCount	Number of glyphs for which information is provided for horizontally growing variants. Must be the same as the number of glyph IDs referenced in the horizontal Coverage table.
Offset16	vertGlyphConstructionOffsets [vertGlyphCount]	Array of offsets to MathGlyphConstruction tables, from the beginning of the MathVariants table, for shapes growing in the vertical direction.
Offset16	horizGlyphConstructionOffsets [horizGlyphCount]	Array of offsets to MathGlyphConstruction tables, from the beginning of the MathVariants table, for shapes growing in the horizontal direction.

6.3.6.2.11 MathGlyphConstruction Table

The MathGlyphConstruction table provides information on finding or assembling extended variants for one particular glyph. It can be used for shapes that grow in either horizontal or vertical directions.

The first entry is the offset to the GlyphAssembly table that specifies how the shape for this glyph can be assembled from parts found in the glyph set of the font. If no such assembly exists, this offset will be set to NULL.

The MathGlyphConstruction table also contains the count and array of ready-made glyph variants for the specified glyph. Each variant consists of the glyph index and this glyph's measurement in the direction of extension (vertical or horizontal).

Note that it is quite possible that both the GlyphAssembly table and some variants are defined for a particular glyph. For example, the font may provide several variants of curly braces with different sizes, and also a general mechanism for constructing larger versions of curly braces by stacking parts found in the glyph set. First, an attempt is made to find glyph among provided variants. If the required size is larger than any of the glyph variants provided, however, then the general mechanism can be employed to typeset the curly braces as a glyph assembly.

MathGlyphConstruction Table

Type	Name	Description
Offset16	glyphAssemblyOffset	Offset to the GlyphAssembly table for this shape, from the beginning of the MathGlyphConstruction table. May be NULL.

uint16	variantCount	Count of glyph growing variants for this glyph.
MathGlyphVariantRecord	mathGlyphVariantRecords [variantCount]	MathGlyphVariantRecords for alternative variants of the glyphs.

MathGlyphVariantRecord

Type	Name	Description
uint16	variantGlyph	Glyph ID for the variant.
uint16	advanceMeasurement	Advance width/height, in design units, of the variant, in the direction of requested glyph extension.

6.3.6.2.12 GlyphAssembly Table

The GlyphAssembly table specifies how the shape for a particular glyph can be constructed from parts found in the glyph set. The table defines the italics correction of the resulting assembly, and a number of parts that have to be put together to form the required shape. Some glyph parts can be designated as extenders, which can be repeated as needed to obtain a target size.

GlyphAssembly Table

Type	Name	Description
MathValueRecord	italicsCorrection	Italics correction of this GlyphAssembly. Should not depend on the assembly size.
uint16	partCount	Number of parts in this assembly.
GlyphPartRecord	partRecords[partCount]	Array of GlyphPartRecords, from left to right (for assemblies that extend horizontally) or bottom to top (for assemblies that extend vertically).

The result of the assembly process is an array of glyphs with an offset specified for each of those glyphs. When drawn consecutively at those offsets, the glyphs should combine correctly and produce the required shape.

The offsets in the direction of growth (advance offsets), as well as the number of extender parts, are determined based on the size requirement for the resulting assembly.

Note that the glyphs comprising the assembly should be designed so that they align properly in the direction that is orthogonal to the direction of growth.

Thus, a GlyphPartRecord consists of the following fields:

- Glyph ID for the part.
- Lengths of the connectors on each end of the glyph. The connectors are straight parts of the glyph that can be used to link it with the next or previous part. The connectors of neighboring parts can overlap by varying amounts, providing flexibility in how these glyphs can be put together. However, the overlap should not be less than the minConnectorOverlap value defined in the MathVariants tables, and it should not exceed the specified connector length for that end of the glyph. If the part does not have a connector on one of its ends, the corresponding connector length should be set to zero.
- The full advance of the part. It is also used to determine the measurement of the result by using the following formula:

Size of Assembly = Offset of the Last Part + Full Advance of the Last Part

— PartFlags is the last field. It identifies certain parts as *extenders*: those parts that can be repeated (that is, multiple instances of them can be used in place of one) or skipped altogether. Usually the extenders are vertical or horizontal bars of the appropriate thickness, aligned with the rest of the assembly.

To ensure that the width/height is distributed equally and the symmetry of the shape is preserved, following steps can be used by the math-layout engine.

1. Assemble all parts with all extenders removed and with connectionsoverlapping by the maximum amount. This gives the smallest possible result.
2. Determine how much extra width/height can be obtained from all existing connections between neighboring parts by using minimal overlaps. If that is enough to achieve the size goal, extend each connection equally by changing overlaps of connectors to finish the job.
3. If all connections have been extended to the minimum overlap and further growth is needed, add one of each extender, and repeat the process from the first step.

Note that for assemblies growing in the vertical direction, the distribution of height between ascent and descent is not defined. The math-layout engine is responsible for positioning the resulting assembly relative to the baseline.

GlyphPartRecord Table

Type	Name	Description
uint16	glyphID	Glyph ID for the part.
uint16	startConnectorLength	Advance width / height, in design units, of the straight bar connector material at the start of the glyph in the direction of the extension (the left end for horizontal extension, the bottom end for vertical extension).
uint16	endConnectorLength	Advance width / height, in design units, of the straight bar connector material at the end of the glyph in the direction of the extension (the right end for horizontal extension, the top end for vertical extension).
uint16	fullAdvance	Full advance width/height for this part, in the direction of the extension, in design units.
uint16	partFlags	Part qualifiers. PartFlags enumeration currently uses only one bit: 0x0001 fExtender If set, the part can be skipped or repeated. 0xFFFE Reserved.

6.3.6.3 OFF layout tags used with the MATH Table

The following OFF layout tags can be used by math-layout engine to access a particular set of glyph variants. For detailed descriptions of the feature tags see subclause 6.4.3.2.

OFF layout tags for math processing

Tag	Description
math	Script tag to be used for features in math layout. The only language system supported with this tag is the default language system.

ssty	Script Style This feature provides glyph variants adjusted to be more suitable for use in subscripts and superscripts. These script style forms should *not* be scaled or moved in the font; scaling and moving them is done by the math-layout engine. Instead, the ssty feature should provide glyph forms that result in shapes that look good as superscripts and subscripts when scaled and positioned by the math engine. When designing the script forms, the font developer may assume that the scriptPercentScaleDown and scriptScriptPercentScaleDown values in the MathConstants table will be scaling factors applied to the size of the alternate glyphs by the math engine. This feature can have a parameter indicating the script level: 1 for simple subscripts and superscripts, 2 for second level subscripts and superscripts (that is, scripts on scripts), and so on. (Currently, only the first two alternates are used). For glyphs that are not covered by this feature, the original glyph is used in subscripts and superscripts. Recommended format: Alternate Substitution table (Single Substitution if there are no second level forms). There should be no context.
flac	Flattened Accents over Capitals This feature provides flattened forms of accents to be used over high-rise bases such as capitals. This feature should only change the shape of the accent and should *not* move it in the vertical or horizontal direction. Moving of the accents is done by the math-layout engine. Accents are flattened by the math engine if their base is higher than theflattenedAccentBaseHeight value in the MathConstants table. Recommended format: Single Substitution table. There should be no context.
dtls	Dotless Forms This feature provides dotless forms for Math Alphanumeric characters, such as U+1D422 MATHEMATICAL BOLD SMALL I, U+1D423 MATHEMATICAL BOLD SMALL J, U+1D456 U+MATHEMATICAL ITALIC SMALL I, U+1D457 MATHEMATICAL ITALIC SMALL J, and so on. The dotless forms are to be used as base forms for placing mathematical accents over them. Recommended format: Single Substitution table. There should be no context.

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 generally correspond to a Unicode script. However, the associations between them may not always be one-to-one, and the OFF tags are not guaranteed to be the same as Unicode Script property-value aliases or ISO 15924 script IDs. Since the development of OFF (and the prior history of OpenType) script tags predates the ISO 15924 or Unicode Script property, the rules for script tags defined in this document may not always be the same as rules for ISO 15924 script IDs. The OFF script tags can also correlate with a particular OFF layout implementation, with the result that more than one script tag may be registered for a given Unicode script (e.g. ‘deva’ and ‘dev2’).

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.

Use and processing of script tags in Script Records is described in the "Script list table and Script record" section of subclause 6.2.4 (Scripts and Languages).

Script	Script Tag
Adlam	'adlm'
Ahom	'ahom'
Anatolian Hieroglyphs	'hluw'
Arabic	'arab'
Armenian	'armn'
Avestan	'avst'
Balinese	'bali'
Bamum	'bamu'
Bassa Vah	'bass'
Batak	'batk'
Bengali	'beng'
Bengali v.2	'bng2'
Bhaiksuki	'bhks'
Bopomofo	'bopo'
Brahmi	'brah'
Braille	'brai'
Buginese	'bugi'
Buhid	'buhd'
Byzantine Music	'byzm'
Canadian Syllabics	'cans'
Carian	'cari'
Caucasian Albanian	'aghb'
Chakma	'cakm'
Cham	'cham'

Cherokee	'cher'
CJK Ideographic	'hani'
Coptic	'copt'
Cypriot Syllabary	'cprt'
Cyrillic	'cyrl'
Default	'DFLT'
Deseret	'dsrt'
Devanagari	'deva'
Devanagari v.2	'dev2'
Dogra	'dogr'
Duployan	'dupl'
Egyptian hieroglyphs	'egyp'
Elbasan	'elba'
Ethiopic	'ethi'
Georgian	'geor'
Glagolitic	'glag'
Gothic	'goth'
Grantha	'gran'
Greek	'grek'
Gujarati	'gujr'
Gujarati v.2	'gjr2'
Gunjala Gondi	'gong'
Gurmukhi	'guru'
Gurmukhi v.2	'gur2'
Hangul	'hang'
Hangul Jamo	'jamo'
Hanifi Rohingya	'rohg'
Hanunoo	'hano'
Hatran	'hatr'
Hebrew	'hebr'
Hiragana	'kana'
Imperial Aramaic	'armi'
Inscriptional Pahlavi	'phli'
Inscriptional Parthian	'prti'
Javanese	'java'
Kaithi	'kthi'
Kannada	'knda'
Kannada v.2	'knd2'

Katakana	'kana'
Kayah Li	'kali'
Kharosthi	'khar'
Khmer	'khmr'
Khojki	'khoj'
Khudawadi	'sind'
Lao	'lao '
Latin	'latn'
Lepcha	'lepc'
Limbu	'limb'
Linear A	'lina'
Linear B	'linb'
Lisu (Fraser)	'lisu'
Lycian	'lyci'
Lydian	'lydi'
Mahajani	'mahj'
Makasar	'maka'
Malayalam	'mlym'
Malayalam v.2	'mlm2'
Mandaic, Mandaean	'mand'
Manichaean	'mani'
Masaram Gondi	'gonm'
Marchen	'marc'
Mathematical Alphanumeric Symbols	'math'
Medefaidrin (Oberi Okaime, Oberi Ɔkaimɛ)	'medf'
Meitei Mayek (Meithei, Meetei)	'mtei'
Mende Kikakui	'mend'
Meroitic Cursive	'merc'
Meroitic Hieroglyphs	'mero'
Miao	'plrd'
Modi	'modi'
Mongolian	'mong'
Mro	'mroo'
Multani	'mult'
Musical Symbols	'musc'
Myanmar	'mymr'
Myanmar v.2	'mym2'
Nabataean	'nbat'

Newa	'newa'
New Tai Lue	'talu'
N'Ko	'nko '
Nüshu	'nshu'
Odia (formerly Oriya)	'orya'
Odia (formerly Oriya) v.2	'ory2'
Ogham	'ogam'
Ol Chiki	'olck'
Old Italic	'ital'
Old Hungarian	'hung'
Old North Arabian	'narb'
Old Permic	'perm'
Old Persian Cuneiform	'xpeo'
Old Sogdian	'sogo'
Old South Arabian	'sarb'
Old Turkic, Orkhon Runic	'orkh'
Osage	'osge'
Osmanya	'osma'
Pahawh Hmong	'hmng'
Palmyrene	'palm'
Pau Cin Hau	'pauc'
Phags-pa	'phag'
Phoenician	'phnx'
Psalter Pahlavi	'phlp'
Rejang	'rjng'
Runic	'runr'
Samaritan	'samr'
Saurashtra	'saur'
Sharada	'shrd'
Shavian	'shaw'
Siddham	'sidd'
Sign Writing	'sgnw'
Sinhala	'sinh'
Sogdian	'sogd'
Sora Sompeng	'sora'
Soyombo	'soyo'
Sumero-Akkadian Cuneiform	'xsux'
Sundanese	'sund'

Syloti Nagri	'sylo'
Syriac	'syrc'
Tagalog	'tglg'
Tagbanwa	'tagb'
Tai Le	'tale'
Tai Tham (Lanna)	'lana'
Tai Viet	'tavt'
Takri	'takr'
Tamil	'taml'
Tamil v.2	'tml2'
Tangut	'tang'
Telugu	'telu'
Telugu v.2	'tel2'
Thaana	'thaa'
Thai	'thai'
Tibetan	'tibt'
Tifinagh	'tfng'
Tirhuta	'tirh'
Ugaritic Cuneiform	'ugar'
Vai	'vai '
Warang Citi	'wara'
Yi	'yi '
Zanabazar Square (Zanabazarin Dörböljin Useg, Xewtee Dörböljin Bicig, Horizontal Square Script)	'zanb'

6.4.2 Language tags

Language system tags identify the language systems supported in an OFF Layout font data. 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. Spaces (0x20) may only occur as a trailing sequence within the tag. As a general convention, capital letters (0x41 – 0x5A) are used. If a language system tag consists of three or less visible letters, the letters are followed by the requisite number of spaces, each consisting of a single byte, to complete a 4-byte tag.

Language System	Language System Tag	Corresponding ISO 639 ID (if applicable)
Abaza	'ABA '	abq
Abkhazian	'ABK '	abk
Acholi	'ACH '	ach
Achi	'ACR '	acr
Adyghe	'ADY '	ady
Afrikaans	'AFK '	afr
Afar	'AFR '	aar
Agaw	'AGW '	ahg
Aiton	'AIO '	aio
Akan	'AKA '	aka
Alsatian	'ALS '	gsw
Altai	'ALT '	atv, alt
Amharic	'AMH '	amh
Anglo-Saxon	'ANG '	ang
Phonetic transcription – Americanist conventions	'APPH'	
Arabic	'ARA '	ara
Aragonese	'ARG '	arg
Aari	'ARI '	aiw
Rakhine	'ARK '	mhv, rmz, rki
Assamese	'ASM '	asm
Asturian	'AST '	ast
Athapaskan	'ATH '	apk, apj, apl, apm, apw, nav, bea, sek, bcr, caf, crx, clc, gwi, haa, chp, dgr, scs, xsl, srs, ing, hoi, koy, hup, ktw, mvb, wlk, coq, ctc, gce, tol, tuu, kkz, tgx, tht, aht, tfn, taa,

		tau, tcb, kuu, tce, ttm, txc
Avar	'AVR '	ava
Awadhi	'AWA '	awa
Aymara	'AYM '	aym
Torki	'AZB '	azb
Azerbaijani	'AZE '	aze
Badaga	'BAD '	bfq
Banda	'BAD0'	bad
Baghelkhandi	'BAG '	bfy
Balkar	'BAL '	krc
Balinese	'BAN '	ban
Bavarian	'BAR '	bar
Baoulé	'BAU '	bci
Batak Toba	'BBC '	bbc
Berber	'BBR '	
Bench	'BCH '	bcq
Bible Cree	'BCR '	
Bandjalang	'BDY '	bdy
Belarusian	'BEL '	bel
Bemba	'BEM '	bem
Bengali	'BEN '	ben
Haryanvi	'BGC '	bgc
Bagri	'BGQ '	bgq
Bulgarian	'BGR '	bul
Bhili	'BHI '	bhi, bhb
Bhojpuri	'BHO '	bho
Bikol	'BIK '	bik, bhk, bcl, bto, cts, bln
Bilen	'BIL '	byn
Bislama	'BIS '	bis
Kanauji	'BJJ '	bjj
Blackfoot	'BKF '	bla
Baluchi	'BLI '	bal
Pa'o Karen	'BLK '	blk
Balante	'BLN '	bjt, ble
Balti	'BLT '	bft
Bambara (Bamanankan)	'BMB '	bam
Bamileke	'BML '	

Bosnian	'BOS '	bos
Bishnupriya Manipuri	'BPY '	bpy
Breton	'BRE '	bre
Brahui	'BRH '	brh
Braj Bhasha	'BRI '	bra
Burmese	'BRM '	mya
Bodo	'BRX '	brx
Bashkir	'BSH '	bak
Burushaski	'BSK '	bsk
Beti	'BTI '	btb
Batak Simalungun	'BTS '	bts
Bugis	'BUG '	bug
Medumba	'BYV '	byv
Kaqchikel	'CAK '	cak
Catalan	'CAT '	cat
Zamboanga Chavacano	'CBK '	cbk
Chinantec	'CCHN'	cco, chj, chq, chz, cle, cnl, cnt, cpa, csa, cso, cte, ctl, cuc, cvn
Cebuano	'CEB '	ceb
Chechen	'CHE '	che
Chaha Gurage	'CHG '	sgw
Chattisgarhi	'CHH '	hne
Chichewa (Chewa, Nyanja)	'CHI '	nya
Chukchi	'CHK '	ckt
Chuukese	'CHK0'	chk
Choctaw	'CHO '	cho
Chipewyan	'CHP '	chp
Cherokee	'CHR '	chr
Chamorro	'CHA '	cha
Chuvash	'CHU '	chv
Cheyenne	'CHY '	chy
Chiga	'CGG '	cgg
Western Cham	'CJA '	cja
Eastern Cham	'CJM '	cjm
Comorian	'CMR '	swb, wlc, wni, zdj
Coptic	'COP '	cop
Cornish	'COR '	cor
Corsican	'COS '	cos

Creoles	'CPP '	cpp
Cree	'CRE '	cre
Carrier	'CRR '	crx, caf
Crimean Tatar	'CRT '	crh
Kashubian	'CSB '	csb
Church Slavonic	'CSL '	chu
Czech	'CSY '	ces
Chittagonian	'CTG '	ctg
San Blas Kuna	'CUK '	cuk
Danish	'DAN '	dan
Dargwa	'DAR '	dar
Dayi	'DAX '	dax
Woods Cree	'DCR '	cwd
German	'DEU '	deu
Dogri	'DGO '	dgo
Dogri	'DGR '	doi
Dhangu	'DHG '	dhg
Divehi (Dhivehi, Maldivian)	'DHV ' (deprecated)	div
Dimli	'DIQ '	diq
Divehi (Dhivehi, Maldivian)	'DIV '	div
Zarma	'DJR '	dje
Djambarrpuyngu	'DJR0'	djr
Dangme	'DNG '	ada
Dan	'DNJ '	dnj
Dinka	'DNK '	din
Dari	'DRI '	prs
Dhuwal	'DUJ '	duj
Dungan	'DUN '	dng
Dzongkha	'DZN '	dzo
Ebira	'EBI '	igb
Eastern Cree	'ECR '	crj, crl
Edo	'EDO '	bin
Efik	'EFI '	efi
Greek	'ELL '	ell
Eastern Maninkakan	'EMK '	emk
English	'ENG '	eng
Erzya	'ERZ '	myv
Spanish	'ESP '	spa

Central Yupik	'ESU '	esu
Estonian	'ETI '	est
Basque	'EUQ '	eus
Evenki	'EVK '	evn
Even	'EVN '	eve
Ewe	'EWE '	ewe
French Antillean	'FAN '	acf
Fang	'FAN0'	fan
Persian	'FAR '	fas
Fanti	'FAT '	fat
Finnish	'FIN '	fin
Fijian	'FJI '	fij
Dutch (Flemish)	'FLE '	vls
Fe'fe'	'FMP '	fmp
Forest Nenets	'FNE '	enf
Fon	'FON '	fon
Faroese	'FOS '	fao
French	'FRA '	fra
Cajun French	'FRC '	frc
Frisian	'FRI '	fry
Friulian	'FRL '	fur
Arpitan	'FRP '	frp
Futa	'FTA '	fuf
Fulah	'FUL '	ful
Nigerian Fulfulde	'FUV '	fuv
Ga	'GAD '	gaa
Scottish Gaelic (Gaelic)	'GAE '	gla
Gagauz	'GAG '	gag
Galician	'GAL '	glg
Garshuni	'GAR '	
Garhwali	'GAW '	gbm
Ge'ez	'GEZ '	gez
Githabul	'GIH '	gih
Gilyak	'GIL '	niv
Kiribati (Gilbertese)	'GIL0'	gil
Kpelle (Guinea)	'GKP '	gkp
Gilaki	'GLK '	glk
Gumuz	'GMZ '	guk

Gumatj	'GNN '	gnn
Gogo	'GOG '	gog
Gondi	'GON '	gon, gno, ggo
Greenlandic	'GRN '	kal
Garo	'GRO '	grt
Guarani	'GUA '	grn
Wayuu	'GUC '	guc
Gupapuyngu	'GUF '	guf
Gujarati	'GUJ '	guj
Gusii	'GUZ '	guz
Haitian (Haitian Creole)	'HAI '	hat
Halam	'HAL '	flm
Harauti	'HAR '	hoj
Hausa	'HAU '	hau
Hawaiian	'HAW '	haw
Haya	'HAY '	hay
Hazaragi	'HAZ '	haz
Hammer-Banna	'HBN '	amf
Herero	'HER '	her
Hiligaynon	'HIL '	hil
Hindi	'HIN '	hin
High Mari	'HMA '	mrj
Hmong	'HMN '	hmn
Hiri Motu	'HMO '	hmo
Hindko	'HND '	hno, hnd
Ho	'HO '	hoc
Harari	'HRI '	har
Croatian	'HRV '	hrv
Hungarian	'HUN '	hun
Armenian	'HYE '	hye
Armenian East	'HYE0'	hye
Iban	'IBA '	iba
Ibibio	'IBB '	ibb
Igbo	'IBO '	ibo
Ijo languages	'IJO '	ijc
Ido	'IDO '	ido
Interlingue	'ILE '	ile
Ilokano	'ILO '	ilo

Interlingua	'INA '	ina
Indonesian	'IND '	ind
Ingush	'ING '	inh
Inuktitut	'INU '	iku
Inupiat	'IPK '	ipk
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
Jamaican Creole	'JAM '	jam
Japanese	'JAN '	jpn
Lojban	'JBO '	jbo
Krymchak	'JCT '	jct
Ladino	'JUD '	lad
Jula	'JUL '	dyu
Kabardian	'KAB '	kbd
Kabyle	'KAB0'	kab
Kachchi	'KAC '	kfr
Kalenjin	'KAL '	kln
Kannada	'KAN '	kan
Karachay	'KAR '	krc
Georgian	'KAT '	kat
Kazakh	'KAZ '	kaz
Makonde	'KDE '	kde
Kabuverdianu (Crioulo)	'KEA '	kea
Kebena	'KEB '	ktb
Kekchi	'KEK '	kek
Khutsuri Georgian	'KGE '	kat
Khakass	'KHA '	kjh
Khanty-Kazim	'KHK '	kca
Khmer	'KHM '	khm
Khanty-Shurishkar	'KHS '	kca

Khamti Shan	'KHT '	kht
Khanty-Vakhi	'KHV '	kca
Khowar	'KHW '	khw
Kikuyu (Gikuyu)	'KIK '	kik
Kirghiz (Kyrgyz)	'KIR '	kir
Kisii	'KIS '	kqs, kss
Kirmanjki	'KIU '	kiu
Southern Kiwai	'KJD '	kjd
Eastern Pwo Karen	'KJP '	kjp
Kokni	'KKN '	kex
Kalmyk	'KLM '	xal
Kamba	'KMB '	kam
Kumaoni	'KMN '	kfy
Komo	'KMO '	kmw
Komso	'KMS '	kxc
Khorasani Turkic	'KMZ '	kmz
Kanuri	'KNR '	kau
Kodagu	'KOD '	kfa
Korean Old Hangul	'KOH '	okm
Konkani	'KOK '	kok
Kikongo	'KON '	ktu
Kongo	'KON0'	kon
Komi	'KOM '	kom
Komi-Permyak	'KOP '	koi
Korean	'KOR '	kor
Kosraean	'KOS '	kos
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
Ripuarian	'KSH0'	ksh
Khasi	'KSI '	kha
Kildin Sami	'KSM '	sjd

S’gaw Karen	'KSW '	ksw
Kuanyama	'KUA '	kua
Kui	'KUI '	kxu
Kulvi	'KUL '	kfx
Kumyk	'KUM '	kum
Kurdish	'KUR '	kur
Kurukh	'KUU '	kru
Kuy	'KUY '	kdt
Koryak	'KYK '	kpy
Western Kayah	'KYU '	kyu
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
Ligurian	'LIJ '	lij
Limburgish	'LIM '	lim
Lingala	'LIN '	lin
Lisu	'LIS '	lis
Lampung	'LJP '	ljp
Laki	'LKI '	lki
Low Mari	'LMA '	mhr
Limbu	'LMB '	lif
Lombard	'LMO '	lmo
Lomwe	'LMW '	ngl
Loma	'LOM '	lom
Luri	'LRC '	lrc, luz, bqi, zum
Lower Sorbian	'LSB '	dsb
Lule Sami	'LSM '	smj
Lithuanian	'LTH '	lit
Luxembourgish	'LTZ '	ltz
Luba-Lulua	'LUA '	lua
Luba-Katanga	'LUB '	lub

Ganda	'LUG '	lug
Luyia	'LUH '	luy
Luo	'LUO '	luo
Latvian	'LVI '	lav
Madura	'MAD '	mad
Magahi	'MAG '	mag
Marshallese	'MAH '	mah
Majang	'MAJ '	mpe
Makhuwa	'MAK '	vmw
Malayalam Traditional	'MAL '	mal
Mam	'MAM '	mam
Mansi	'MAN '	mns
Mapudungun	'MAP '	arn
Marathi	'MAR '	mar
Marwari	'MAW '	mwr, dhd, rwr, mve, wry, mtr, swv
Mbundu	'MBN '	kmb
Mbo	'MBO '	mbo
Manchu	'MCH '	mnc
Moose Cree	'MCR '	crm
Mende	'MDE '	men
Mandar	'MDR '	mdr
Me'en	'MEN '	mym
Meru	'MER '	mer
Morisyen	'MFE '	mfe
Minangkabau	'MIN '	min
Mizo	'MIZ '	lus
Macedonian	'MKD '	mkd
Makasar	'MKR '	mak
Kituba	'MKW '	mkw
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, mnk, myq, mku, msc, emk, mwk, mlq
Manx	'MNX '	glv
Mohawk	'MOH '	mho
Moksha	'MOK '	mdf
Moldavian	'MOL '	mol
Mon	'MON '	mnw
Moroccan	'MOR '	
Mossi	'MOS '	mos
Maori	'MRI '	mri
Maithili	'MTH '	mai
Maltese	'MTS '	mlt
Mundari	'MUN '	unr
Muscogee	'MUS '	mus
Mirandese	'MWL '	mwl
Hmong Daw	'MWW '	mww
Mayan	'MYN '	myn
Mazanderani	'MZN '	mzn
Naga-Assamese	'NAG '	nag
Nahuatl	'NAH '	nah
Nanai	'NAN '	gld
Neapolitan	'NAP '	nap
Naskapi	'NAS '	nsk
Nauruan	'NAU '	nau
Navajo	'NAV '	nav
N-Cree	'NCR '	csw
Ndebele	'NDB '	nbl, nde
Ndau	'NDC '	ndc
Ndonga	'NDG '	ndo
Low Saxon	'NDS '	nds
Nepali	'NEP '	nep
Newari	'NEW '	new
Ngbaka	'NGA '	nga
Nagari	'NGR '	
Norway House Cree	'NHC '	csw
Nisi	'NIS '	dap
Niuean	'NIU '	niu
Nyankole	'NKL '	nyn

N'Ko	'NKO '	ngo
Dutch	'NLD '	nld
Nimadi	'NOE '	noe
Nogai	'NOG '	nog
Norwegian	'NOR '	nob
Novial	'NOV '	nov
Northern Sami	'NSM '	sme
Sotho, Northern	'NSO '	nso
Northern Thai	'NTA '	nod
Esperanto	'NTO '	epo
Nyamwezi	'NYM '	nym
Norwegian Nynorsk (Nynorsk, Norwegian)	'NYN '	nno
Mbembe Tigon	'NZA '	nza
Occitan	'OCI '	oci
Oji-Cree	'OCR '	ojs
Ojibway	'OJB '	oji
Odia (formerly Oriya)	'ORI '	ori
Oromo	'ORO '	orm
Ossetian	'OSS '	oss
Palestinian Aramaic	'PAA '	sam
Pangasinan	'PAG '	pag
Pali	'PAL '	pli
Pampangan	'PAM '	pam
Punjabi	'PAN '	pan
Palpa	'PAP '	plp
Papiamentu	'PAP0'	pap
Pashto	'PAS '	pus
Palauan	'PAU '	pau
Bouyei	'PCC '	pcc
Picard	'PCD '	pcd
Pennsylvania German	'PDC '	pdc
Polytonic Greek	'PGR '	ell
Phake	'PHK '	phk
Norfolk	'PIH '	pih
Filipino	'PIL '	fil
Palaung	'PLG '	pce, rbb, pll
Polish	'PLK '	pol

Piemontese	'PMS '	pms
Western Panjabi	'PNB '	pnb
Pocomchi	'POH '	poh
Pohnpeian	'PON '	pon
Provencal	'PRO '	pro
Portuguese	'PTG '	por
Western Pwo Karen	'PWO '	pwo
Chin	'QIN '	bgr, cnh, cnw, czt, sez, tcp, csy, ctd, flm, pck, tcz, zom, cmr, dao, hlt, cka, cnk, mrh, cbl, cnb, csh
K'iche'	'QUC '	quc
Quechua (Bolivia)	'QUH '	quh
Quechua	'QUZ '	quz
Quechua (Ecuador)	'QVI '	qvi
Quechua (Peru)	'QWH '	qwh
Rajasthani	'RAJ '	raj
Rarotongan	'RAR '	rar
R-Cree	'RCR '	atj
Russian Buriat	'RBU '	bxr
Rejang	'REJ '	rej
Riang	'RIA '	ria
Tarifit	'RIF '	rif
Ritarungo	'RIT '	rit
Arakwal	'RKW '	rkw
Romansh	'RMS '	roh
Vlax Romani	'RMY '	rmy
Romanian	'ROM '	ron
Romany	'ROY '	rom
Rusyn	'RSY '	rue
Rotuman	'RTM '	rtm
Kinyarwanda	'RUA '	kin
Rundi	'RUN '	run
Aromanian	'RUP '	rup
Russian	'RUS '	rus
Sadri	'SAD '	sck
Sanskrit	'SAN '	san
Sasak	'SAS '	sas
Santali	'SAT '	sat

Sayisi	'SAY '	chp
Sicilian	'SCN '	scn
Scots	'SCO '	sco
North Slavey	'SCS '	scs
Sekota	'SEK '	xan
Selkup	'SEL '	sel
Old Irish	'SGA '	sga
Sango	'SGO '	sag
Samogitian	'SGS '	sgs
Tachelhit	'SHI '	shi
Shan	'SHN '	shn
Sibe	'SIB '	sjo
Sidamo	'SID '	sid
Silte Gurage	'SIG '	xst
Skolt Sami	'SKS '	sms
Slovak	'SKY '	slk
Slavey	'SLA '	scs, xsl
Slovenian	'SLV '	slv
Somali	'SML '	som
Samoan	'SMO '	smo
Sena	'SNA '	she
Shona	'SNA0'	sna
Sindhi	'SND '	snd
Sinhala (Sinhalese)	'SNH '	sin
Soninke	'SNK '	snk
Sodo Gurage	'SOG '	gru
Songe	'SOP '	sop
Sotho, Southern	'SOT '	sot
Albanian	'SQI '	gsw
Serbian	'SRB '	srp
Sardinian	'SRD '	srd
Seraiki	'SRK '	skr
Serer	'SRR '	srr
South Slavey	'SSL '	xsl
Southern Sami	'SSM '	sma
Saterland Frisian	'STQ '	stq
Sukuma	'SUK '	suk
Sundanese	'SUN '	sun

Suri	'SUR '	suq
Svan	'SVA '	sva
Swedish	'SVE '	swe
Swadaya Aramaic	'SWA '	aii
Swahili	'SWK '	swa
Swati	'SWZ '	ssw
Sutu	'SXT '	ngo
Upper Saxon	'SXU '	sxu
Sylheti	'SYL '	syl
Syriac	'SYR '	aii, amw, cld, syc, syr, tru
Syriac, Estrangela script-variant (equivalent to ISO 15924 'Syre')	'SYRE'	syc, syr
Syriac, Western script-variant (equivalent to ISO 15924 'Syrj')	'SYRJ'	syc, syr
Syriac, Eastern script-variant (equivalent to ISO 15924 'Syrn')	'SYRN'	syc, syr
Silesian	'SZL '	szl
Tabasaran	'TAB '	tab
Tajik	'TAJ '	tgk
Tamil	'TAM '	tam
Tatar	'TAT '	tat
TH-Cree	'TCR '	cwd
Dehong Dai	'TDD '	tdd
Telugu	'TEL '	tel
Tetum	'TET '	tet
Tagalog	'TGL '	tgl
Tongan	'TGN '	ton
Tigre	'TGR '	tig
Tigrinya	'TGY '	tir
Thai	'THA '	tha
Tahitian	'THT '	tah
Tibetan	'TIB '	bod
Tiv	'TIV '	tiv
Turkmen	'TKM '	tuk
Tamashek	'TMH '	tmh
Temne	'TMN '	tem
Tswana	'TNA '	tsn
Tundra Nenets	'TNE '	enh
Tonga	'TNG '	toi

Todo	'TOD '	xal
Toma	'TOD0'	tod
Tok Pisin	'TPI '	tpi
Turkish	'TRK '	tur
Tsonga	'TSG '	tso
Turoyo Aramaic	'TUA '	tru
Tulu	'TUL '	tcy
Tumbuka	'TUM '	tum
Tuvin	'TUV '	tyv
Tuvalu	'TVL '	tvl
Twi	'TWI '	twi
Tày	'TYZ '	tyz
Tamazight	'TZM '	tzm
Tzotzil	'TZO '	tzo
Udmurt	'UDM '	udm
Ukrainian	'UKR '	ukr
Umbundu	'UMB '	umb
Urdu	'URD '	urd
Upper Sorbian	'USB '	hsb
Uyghur	'UYG '	uig
Uzbek	'UZB '	uzb, uzn, uzs
Venetian	'VEC '	vec
Venda	'VEN '	ven
Vietnamese	'VIT '	vie
Volapük	'VOL '	vol
Võro	'VRO '	vro
Wa	'WA '	wbm
Wagdi	'WAG '	wbr
Waray-Waray	'WAR '	war
West-Cree	'WCR '	crk
Welsh	'WEL '	cym
Walloon	'WLN '	wln
Wolof	'WLF '	wol
Mewati	'WTM '	wtm
Lü	'XBD '	khb
Xhosa	'XHS '	xho
Minjangbal	'XJB '	xjb
Soga	'XOG '	xog

Kpelle (Liberia)	'XPE '	xpe
Sakha	'YAK '	sah
Yao	'YAO '	yao
Yapese	'YAP '	yap
Yoruba	'YBA '	yor
Y-Cree	'YCR '	cre
Yi Classic	'YIC '	
Yi Modern	'YIM '	iii
Zealandic	'ZEA '	zea
Standard Morrocan Tamazigh	'ZGH '	zgh
Zhuang	'ZHA '	zha
Chinese, Hong Kong SAR	'ZHH '	zho
Chinese Phonetic	'ZHP '	zho
Chinese Simplified	'ZHS '	zho
Chinese Traditional	'ZHT '	zho
Zande	'ZND '	zne
Zulu	'ZUL '	zul
Zazaki	'ZZA '	zza

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, in which initial glyph forms appear at the beginning of connected letter groups determined by character-joining properties, text-processing clients must be able to identify the glyph to which the feature should be applied, based on character context and joining properties. 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
'dtls'	Dotless Forms
'expt'	Expert Forms
'falt'	Final Glyph on Line Alternates
'fin2'	Terminal Forms #2
'fin3'	Terminal Forms #3
'fina'	Terminal Forms

'flac'	Flattened ascent 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
'rclt'	Required Contextual Alternates
'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
'rvrn'	Required Variation Alternates
'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
'ssty'	Math script style alternates
'stch'	Stretching Glyph Decomposition
'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
'vrtr'	Vertical Alternates for 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 desirable to decompose the default glyph for a character into two or more glyphs. Additionally, it may be preferable to compose default glyphs for two or more 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
uint16	format	Format number is set to 0.
uint16	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.)
uint16	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.)
uint16	sampleTextNameId	The ‘name’ table name ID that specifies sample text that illustrates the effect of this feature. (May be NULL.)
uint16	numNamedParameters	Number of named parameters. (May be zero.)
uint16	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.)
uint16	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, sampleTextNameId 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. For each respective[/distinct] 'cvXX' feature, the FeatureParams in the FeatureList must point to the same set of values.

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: 'dtls'

Friendly name: Dotless forms

Function: This feature provides dotless forms for Math Alphanumeric characters, such as U+1D422 MATHEMATICAL BOLD SMALL I, U+1D423 MATHEMATICAL BOLD SMALL J, U+1D456 U+MATHEMATICAL ITALIC SMALL I, U+1D457 MATHEMATICAL ITALIC SMALL J, and so on.

The dotless forms are to be used as base forms for placing mathematical accents over them.

Example: In  formula dotted I is substituted with dotless form before attaching tilde accent on top of it.

Recommended implementation: Single substitution, for all dotted characters.

Application interface: Feature is invoked automatically by math layout handler depending on height of the base formula box.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of math formula. It should be applied in the appropriate contexts, as determined by math layout handler. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Applied to math formula layout.

Feature interaction: This feature is applied to individual glyphs during layout of math formula.

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'

NOTE This feature description was significantly revised in 2016.

Friendly name: Terminal Forms

Registered by: Microsoft/Adobe

Function: Replaces glyphs for characters that have applicable joining properties with an alternate form when occurring in a final context. This applies to characters that have one of the following Unicode Joining_Type property values:

— Right_Joining, if the characters are from a right-to-left script.

— Left_Joining, if the characters are from a left-to-right script.

— Dual_Joining.

Unicode Joining_Type property values are obtained from the Unicode Character Database (UCD) [22]. Specifically, Joining_Type property values are documented in the Unicode Character Database file for joining-script properties [20].

Example: In an Arabic-script font, the application would apply the 'fina' feature to the letter ARABIC LETTER WAW (U+0648 "و") when it follows a left-joining character, thereby replacing the default "و" glyph with its right-joining, final form.

Recommended implementation: The 'fina' feature is used to map default forms to corresponding single-joining, final forms. This will usually be implemented using a single substitution (type 1) GSUB lookup, though contextual substitution GSUB lookups (types 5, 6 or 8) may also be appropriate.

Application interface: The application is responsible for parsing character strings and identifying which of the joining-related features — initial forms ('init'), medial forms ('medi'), terminal forms ('fina'), and isolated forms ('isol') — to apply to which GIDs, based on character Joining_Type properties. Additional factors, such as the presence of control characters, may also be considered. For GIDs to which the 'fina' feature is applied and that are found in the 'fina' coverage table, the application passes a GID to the lookup tables associate with the feature and gets back a new GID.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied by default 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: Can be used in any script with joining behavior — that is, the scripts for which Joining_Type properties are given explicitly in Unicode Character Database file for joining-script properties [20].

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also 'init', 'isol', and 'medi'.

Tag: 'flac'

Friendly name: Flattened ascent forms

Function: This feature provides flattened forms of accents to be used over high-rise bases such as capitals. This feature should only change the shape of the accent and should *not* move it in the vertical or horizontal direction. Moving of the accents is done by the math handling client. Accents are flattened by the Math engine if their base is higher than MATH.MathConstants. FlattenedAccentBaseHeight.

Example: Depending on the font parameters, in  formula tilde may used in default form and in it may use

flattened form

Recommended implementation: Single substitution, replacing ascent glyph with its flattened form. See MATH table specification for details.

Application interface: Feature is invoked automatically by math layout handler depending on height of the base formula box.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of math formula. It should be applied in the appropriate contexts, as determined by math layout handler. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Applied to math formula layout.

Feature interaction: This feature is applied to individual glyphs during layout of math formula.

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: In general, this feature should be off by default. Different behavior should be used, however, in applications that conform to Requirements for Japanese Text Layout (JLREQ [21]) or similar CJK text-layout specifications that expect half-width forms of characters whose default glyphs are full-width. Such implementations should turn this feature on by default, or should selectively apply this feature to particular characters that require special treatment for CJK text-layout purposes, such as brackets, punctuation, and quotation marks.

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' (DEPRECATED in 2016)

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'

NOTE This feature description was significantly revised in 2016.

Friendly name: Initial Forms

Registered by: Microsoft/Adobe

Function: Replaces glyphs for characters that have applicable joining properties with an alternate form when occurring in an initial context. This applies to characters that have one of the following Unicode Joining_Type property values:

— Right_Joining, if the characters are from a left-to-right script.

— Left_Joining, if the characters are from a right-to-left script.

— Dual_Joining.

Unicode Joining_Type property values are obtained from the Unicode Character Database (UCD) [22]. Specifically, Joining_Type property values are documented in the Unicode Character Database file for joining-script properties [20].

Example: In an Arabic-script font, the application would apply the 'init' feature to the letter ARABIC LETTER SEEN (U+0633 "س") when it precedes a right-joining character, thereby replacing the default "س" glyph with its left-joining, initial form.

Recommended implementation: The 'init' feature is used to map default forms to corresponding single-joining, iniital forms. This will usually be implemented using a single substitution (type 1) GSUB lookup, though contextual substitution GSUB lookups (types 5, 6 or 8) may also be appropriate.

Application interface: The application is responsible for parsing character strings and identifying which of the joining-related features — initial forms ('init'), medial forms ('medi'), terminal forms ('fina'), and isolated forms ('isol') — to apply to which GIDs, based on character Joining_Type properties. Additional factors, such as the presence of control characters, may also be considered. For GIDs to which the 'init' feature is applied and that are found in the 'init' coverage table, the application passes a GID to the lookup tables associate with 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 script with joining behavior — that is, the scripts for which Joining_Type properties are given explicitly in Unicode Character Database file for joining-script properties [20].

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also 'fina', 'isol', and 'medi'.

Tag: 'isol'

NOTE This feature description was significantly revised in 2016.

Friendly name: Isolated Forms

Registered by: Microsoft

Function: Replaces glyphs for characters that have applicable joining properties with an alternate form when occurring in a isolate (non-joining) context. This applies to characters that have one of the following Unicode Joining_Type property values:

— Right_Joining.

— Left_Joining.

— Dual_Joining.

— Non_Joining, if the characters are from a script with joining behavior.

Unicode Joining_Type property values are obtained from the Unicode Character Database (UCD) [22]. Specifically, Joining_Type property values are documented in the Unicode Character Database file for joining-script properties [20]. Scripts that have joining behavior are those scripts with character properties given explicitly in [20].

Note that, in many fonts that support the relevant scripts, this feature may not be implemented since the default forms of the relevant characters are the isolated forms. In some fonts, this feature may involve contextual substitution based on the specific, isolated context.

Example: In an Arabic-script font, the application would apply the 'isol' feature to the letter ARABIC LETTER HEH (U+0647 "ه") when not adjacent to any joining character, thereby potentially replacing the default "ه" glyph with a special, isolated form (likely, a contextual and language-specific substitution, substituting one isolated form for another).

Recommended implementation: The 'isol' feature is used to map default forms to alternate non-joining, isolate forms. This will usually be implemented using a single substitution (type 1) GSUB lookup or, often, a contextual substitution GSUB lookup (types 5, 6 or 8).

Application interface: The application is responsible for parsing character strings and identifying which of the joining-related features — initial forms ('init'), medial forms ('medi'), terminal forms ('fina'), and isolated forms ('isol') — to apply to which GIDs, based on character Joining_Type properties. Additional factors, such as the presence of control characters, may also be considered. For GIDs to which the 'isol' feature is applied and that are found in the 'isol' coverage table, the application passes a GID to the lookup tables associated with the feature and gets back a new GID.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied by default 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: Can be used in any script with joining behavior — that is, the scripts for which Joining_Type properties are given explicitly in Unicode Character Database file for joining-script properties [20].

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also 'fina', 'init', and 'medi'.

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 non-lining figures to lining figures.

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). If the default figures are non-lining, they too are mapped to the corresponding lining form.

Application interface: For GIDs found in the lnum coverage table, the application passes a GID to the lnum 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 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 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 Forms #2

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'

NOTE This feature description was significantly revised in 2016.

Friendly name: Medial Forms

Registered by: Microsoft/Adobe

Function: Replaces glyphs for characters that have applicable joining properties with an alternate form when occurring in a medial context. This applies to characters that have the Unicode Joining_Type property value Dual_Joining.

Unicode Joining_Type property values are obtained from the Unicode Character Database (UCD) [22]. Specifically, Joining_Type property values are documented in the Unicode Character Database file for joining-script properties [20].

Example: In an Arabic-script font, the application would apply the 'medi' feature to the letter ARABIC LETTER QAF (U+0642 "ق") when it follows a left-joining character and precedes a right-joining character, thereby replacing the default "ق" glyph with its dual-joining, medial form.

Recommended implementation: The 'medi' feature is used to map default forms to corresponding dual-joining, medial forms. This will usually be implemented using a single substitution (type 1) GSUB lookup, though contextual substitution GSUB lookups (types 5, 6 or 8) may also be appropriate.

Application interface: The application is responsible for parsing character strings and identifying which of the joining-related features — initial forms ('init'), medial forms ('medi'), terminal forms ('fina'), and isolated forms ('isol') — to apply to which GIDs, based on character Joining_Type properties. Additional factors, such as the presence of control characters, may also be considered. For GIDs to which the 'medi' feature is applied and that are found in the 'medi' coverage table, the application passes a GID to the lookup tables associate with the feature and gets back a new GID.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of the given script. It should be applied by default 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: Can be used in any script with joining behavior — that is, the scripts for which Joining_Type properties are given explicitly in Unicode Character Database file for joining-script properties [20].

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. See also 'fina', 'init', and 'isol'.

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 or 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). If the default figures are non-lining, they too are mapped to the corresponding oldstyle form.

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: This feature should be inactive by default. Users can switch between the default and oldstyle figure 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.◦ (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.

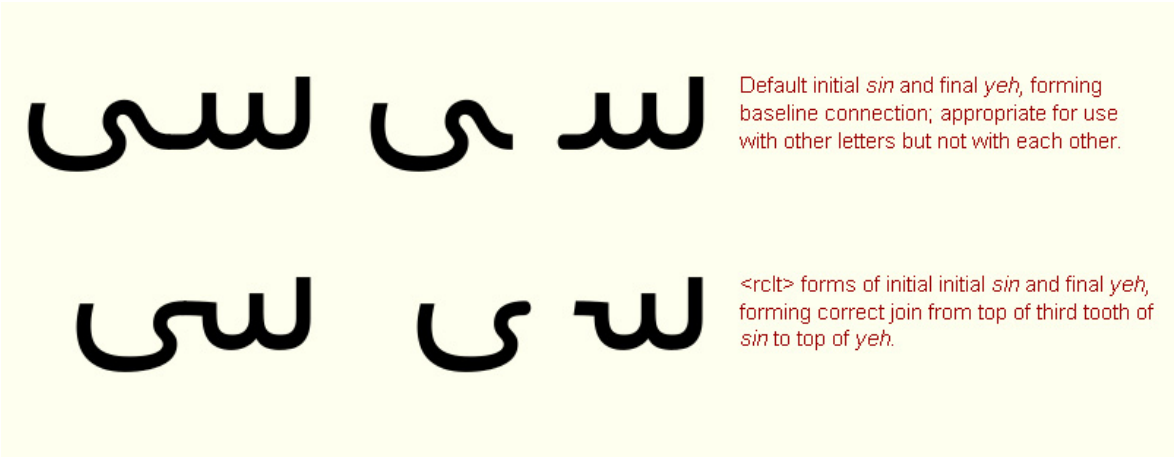
Script/language sensitivity: None.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override.

Tag: 'rclt'

Friendly name: Required Contextual Alternates

Function: In specified situations, replaces default glyphs with alternate forms which provide for better joining behavior or other glyph relationships. Especially important in 'script' typefaces which are designed to have some or all of their glyphs join, but applicable also to e.g. variants to improve spacing. This feature is similar to 'calt', but with the difference that it should not be possible to turn off 'rclt' substitutions: they are considered essential to correct layout of the font.



Example: In an Arabic calligraphic font the 'rclt' feature is used to contextually substitute variant forms of letters sin and yeh providing for a correct join between these two letters that differs from the default join of either to other letters.

Recommended implementation: The rclt 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. Note that full sequences must be passed.

UI suggestion: This feature should be active by default. It is recommended that this feature not be turned off, to avoid breaking obligatory shaping.

Script/language sensitivity: May apply to any script, but is especially important for many styles of Arabic.

Feature interaction: This feature may be used in combination with other substitution (GSUB) features, whose results it may override. For complex scripts, lookups for this feature should be ordered and processed after basic script and language shaping features.

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 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 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: 'rvrn'

Friendly name: Required Variation Alternates

Registered by: Microsoft

Function: This feature is used in fonts that support OFF Font Variations in order to select alternate glyphs for particular variation instances. (For background on OFF font variations, see "Font variations overview".)

When a variable font is used, all of the interpolated variants of a given glyph ID have exactly the same contours and points. It is possible to use glyph variation mechanisms to make significant outline changes, such as reducing strokes in heavy-weight or narrow-width variants, but this approach may be difficult to implement and may not produce desired results for all variation instances. Instead, better results for these scenarios might be achieved by substitution to a different glyph ID. The specific substitutions applied would be conditioned by the particular variation instance that is selected by the user. This conditional behavior is implemented using the required variation alternates feature in conjunction with a FeatureVariations table within the GSUB table.

Example: A variable font supports weight variations ranging from thin to black. The default glyph for the dollar sign has two vertical strokes running through the full extent of the glyph. In the bold variation instance, the default glyph is substituted to an alternate glyph that has only one vertical stroke. In the black variation instance, the default glyph is substituted to an alternate glyph have has only single vertical bars at the top and bottom extremities, with no vertical bars in the two counters in between.

Recommended implementation: The feature is used to activate single substitution (GSUB type 1) lookups, and is always used in conjunction with a FeatureVariations table. Typically, a FeatureTable referenced in a FeatureRecord with the 'rvrn' tag will have LookupCount set to 0; in this way, the default variation instance does not have any glyph substitution applied but, rather, uses default glyphs. Alternate glyphs for particular variation instances are obtained by adding a substitution of the feature table to an alternate feature table

within a FeatureVariations table. Different alternate feature tables may be selected using condition sets that specify particular variation-axis value ranges.

One or more Condition tables is used to determine variation-axis value ranges for which an alternate feature table (and associated lookups) is selected. The axis values used to trigger a condition should normally be midway between values used for named instances. This will avoid any possibility of inconsistent behavior in different applications when using named instances that might arise due to small discrepancies in processing the numeric values.

The default language system for the 'DFLT' script can reference a feature record for this feature with a feature table that will be substituted for particular variation instances to use lookups that apply default, language-independent glyph substitutions; this feature record should be the first feature record for this feature. Some applications may choose to process this feature without processing other features or the script/language system hierarchy; for this purpose, they should choose the first feature record for this feature to obtain the most suitable substitutions for language-independent results.

Application interface: Application of the 'rvrn' feature is mandatory in implementations that support OFF Font Variations whenever a variable font is in use. The feature should be processed in any layout process that supports use of variations, even if other OFF Layout processing is not supported.

The feature is applied only during the process of deriving final glyph IDs (GSUB); it is not used for glyph positioning (GPOS). It should be processed early in GSUB processing, before application of the localized forms feature or features related to shaping of complex scripts or discretionary typographic effects.

Processing of the 'rvrn' feature also requires processing of the FeatureVariations table. ConditionSet tables are scanned for a ConditionSet matching the current variation instance, and then a corresponding FeatureTableSubstitution table is used to locate an alternate feature table. For complete details on processing a FeatureVariations table, see "OFF layout common table formats".

When an applicable feature table is located, only single substitution (GSUB type 1) lookups are processed; any other lookup types are ignored. For any glyph IDs in the coverage table, the application passes the glyph ID to the lookup, and gets back the new glyph ID.

UI suggestion: The 'rvrn' feature is mandatory: it should be active by default and not directly exposed to user control.

Script/language sensitivity: Used for all languages and scripts.

Feature interaction: The feature should be processed early after initial character-to-glyph mapping, before application of the localized forms ('locl') feature, any features related to shaping of complex scripts, or any discretionary features.

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'

NOTE Use of this feature has been superseded by the 'STAT' table. See subclause 9.9 in the "General recommendations" for more information.

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 TypographicFamily name (name ID 16), or Font Family name (name ID 1) if the TypographicFamily 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 Typographic 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 TypographicFamily 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 uint16 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 value of uiLabelNameId is 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. For each respective[/distinct] 'ssXX' feature, the FeatureParams in the FeatureList must point to the same set of values.

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: 'ssty'

Friendly name: Math script style alternates

Function: This feature provides glyph variants adjusted to be more suitable for use in subscripts and superscripts. The script style forms should *not* be scaled or moved in the font; scaling and moving them is done by the math handling client. Instead, the ssty feature should provide glyph forms that result in shapes that look good as superscripts and subscripts when scaled and positioned by the Math engine. When designing the script forms, the font developer may assume that MATH.MathConstants.ScriptPercentScaleDown and MATH.MathConstants.ScriptScriptPercentScaleDown will be the scaling factors used by the Math engine.

This feature can have a parameter indicating the script level: 1 for simple subscripts and superscripts, 2 for second level subscripts and superscripts (that is, scripts on scripts), and so on. (Currently, only the first two alternates are used). For glyphs that are not covered by this feature, the original glyph is used in subscripts and superscripts..

Example: Depending In  formula letter b will be substituted with script level 1 variant and letter c will be

substituted with level 2 variant

Recommended implementation: Alternate substitution, with parameter 1 or 2 corresponding to sub- or super-script level alternate glyphs. If there are no second-level alternates defined in the font, single substitution may

also be used. Glyphs that don't have script alternates can be omitted from this table. See MATH table specification for details.

Application interface: Feature is invoked automatically by math layout handler depending on nesting level inside the math formula.

UI suggestion: In recommended usage, this feature triggers substitutions that are required for correct display of math formula. It should be applied in the appropriate contexts, as determined by math layout handler. Control of the feature should not generally be exposed to the user.

Script/language sensitivity: Applied to math formula layout.

Feature interaction: This feature is applied to individual glyphs during layout of math formula.

Tag: 'stch'

Friendly name: Stretching Glyph Decomposition

Function: Unicode characters, such as the Syriac Abbreviation Mark (U+070F), that enclose other characters need to be able to stretch in order to dynamically adapt to the width of the enclosed text. This feature defines a decomposition set consisting of an odd number of glyphs which describe the stretching glyph. The odd numbered glyphs in the decomposition are fixed reference points which are distributed evenly from the start to the end of the enclosed text. The even numbered glyphs may be repeated as necessary to fill the space between the fixed glyphs. The first and last glyphs may either be simple glyphs with width at the baseline, or mark glyphs. All other decomposition glyphs should have width, but must be defined as mark glyphs.

Example: In Syriac, the character 0x070F is a control character that is rendered as a line above an abbreviation in Syriac script. The line should have a circle at each end and at the mid point. The decomposition sequence for this character should consist of a circle at the start of a line, a connecting line, a circle on a line for the mid point, a second connecting line, and a circle at the end of the line. The connecting lines will repeat in order to fill the space between the circle glyphs.

Recommended implementation: The stch table maps the character to a set containing an odd number of corresponding glyphs (GSUB lookup type 2). The rendering engine reorders the last glyph from the substituted set to the end of the set of characters being enclosed. The remaining glyphs from the substituted set are positioned at the start of the set of characters being enclosed. Odd-numbered glyphs in the decomposition set are positioned so that they are distributed evenly over the width of the text being enclosed. Even-numbered glyphs in the decomposition set are repeated by the rendering engine so the width of the space between fixed, odd-numbered glyphs is filled by the spacing, even-numbered glyphs.

Application interface: For GIDs found in the stch coverage table, the application passes the sequence of GIDs to the table, and gets back the GIDs for the multiple substitution.

UI suggestion: This feature should be on by default.

Script/language sensitivity: None.

Feature interaction: None.

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

Registerd by: Adobe/Microsoft

Function: Transforms default glyphs into glyphs that are appropriate for upright presentation in vertical writing mode. While the glyphs for most characters in East Asian writing systems remain upright when set in vertical writing mode, some must be transformed – usually by rotation, shifting, or different component ordering – for vertical writing mode.

Example: In vertical writing mode, the opening parenthesis (U+FF08) is replaced by the rotated form (U+FE35).
In vertical writing mode, the glyph for HIRAGANA LETTER SMALL A (U+3041; "ぁ") is transformed into a glyph that is shifted up and to the right, which is properly positioned for upright presentation in vertical writing mode.
In vertical writing mode, the glyph for SQUARE MAIKURO (U+3343; "㍁"), whose component katakana characters are ordered from left to right then top to bottom (like horizontal writing mode), is transformed into a glyph whose component katakana characters are ordered from top to bottom then right to left (like vertical writing mode line progression).

Recommended implementation: The font includes versions of the glyphs covered by this feature that differ in some visual way from the default glyphs, such as by rotation, shifting, or different component ordering. The 'vert' feature maps the default glyphs to the corresponding, alternate glyphs for vertical writing mode using a type 1 (single substitution) GSUB lookup.

Application interface: For GIDs found in the 'vert' coverage table, the layout engine passes GIDs to the feature, then gets back new GIDs.

UI suggestion: This feature should be active by default in vertical writing mode.

Script/language sensitivity: Applies only to scripts with vertical writing capability.

Feature interaction: The 'vert' and 'vrtr' features are intended to be used in conjunction: 'vert' for glyphs to be presented upright in vertical writing, and 'vrtr' for glyphs intended to be presented sideways. Since they must never be activated simultaneously for a given glyph, there should be no interaction between the two features. These features are intended for layout engines that graphically rotate glyphs for sideways runs in vertical writing mode, such as those conforming to Unicode Technical Report #50: Unicode vertical text Layout.
Note that layout engines that instead depend on the font to supply pre-rotated glyphs for all sideways glyphs should use the 'vrt2' feature in lieu of 'vert' and 'vrtr'. Because 'vrt2' supplies pre-rotated glyphs, the 'vert' feature should never be used with 'vrt2', but 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.

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: In general, this feature should be off by default. Different behavior should be used, however, in applications that conform to Requirements for Japanese Text Layout (JLREQ [21]) or similar CJK text-layout specifications that expect half-width forms of characters whose default glyphs are full-width. Such implementations should turn this feature on by default, or should selectively apply this feature to particular characters that require special treatment for CJK text-layout purposes, such as brackets, punctuation, and quotation marks.

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.

Adobe Type Manager /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: 'vrtr'

Friendly name: Vertical Alternates for Rotation

Registered by: Adobe/Microsoft/W3C

Function: Transforms default glyphs into glyphs that are appropriate for sideways presentation in vertical writing mode. While the glyphs for most characters in East Asian writing systems remain upright when set in vertical writing mode, glyphs for other characters (such as those of other scripts or for particular Western-style punctuation) are expected to be presented sideways in vertical writing.

Example: As a first example, the glyphs for FULLWIDTH LESS-THAN SIGN (U+FF1C; "<") and FULLWIDTH GREATER-THAN SIGN (U+FF1E; ">") in a font with a non-square em-box are transformed into glyphs whose aspect ratio differs from the default glyphs, which are properly sized for sideways presentation in vertical writing mode. As a second example, the glyph for LEFT SQUARE BRACKET (U+005B, "[") in a brush-script font that exhibits slightly rising horizontal strokes may use an obtuse angle for its upper-left corner when in horizontal writing mode, but an alternate glyph with an acute angle for that corner is supplied for vertical writing mode.

Recommended implementation: The font includes versions of the glyphs covered by this feature that, when rotated 90° clockwise by the layout engine for sideways presentation in vertical writing, differ in some visual way from rotated versions of the default glyphs, such as by shifting or shape. The 'vrtr' feature maps the default glyphs to the corresponding to-be-rotated glyphs (GSUB lookup type 1).

Application interface: For GIDs found in the 'vrtr' coverage table, the application passes GIDs to the lookup tables associated with the feature, then gets back new GIDs.

UI suggestion: This feature should be active by default for sideways runs in vertical writing mode.

Script/language sensitivity: Applies to any script when set in vertical writing mode.

Feature interaction: The 'vrtr' and 'vert' features are intended to be used in conjunction: 'vrtr' for glyphs intended to be presented sideways in vertical writing, and 'vert' for glyphs to be presented upright. Since they must never be activated simultaneously for a given glyph, there should be no interaction between the two features. These features are intended for layout engines that graphically rotate glyphs for sideways runs in vertical writing mode, such as those conforming to Unicode Technical Report #50: Unicode Vertical Text Layout [23].
Note that layout engines that instead depend on the font to supply pre-rotated glyphs for all sideways glyphs should use the 'vrt2' feature in lieu of 'vrtr' and vert. Because 'vrt2' supplies pre-rotated glyphs, the 'vrtr' feature should never be used with 'vrt2', but it 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	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

	usage.)	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 9.8 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 9.8 for more information about constructing CJK fonts.

7 OFF font variations

7.1 Font variations overview

This chapter provides an overview of OFF Font Variations, including an introduction to essential concepts, a glossary of terminology, and a specification of key algorithms: coordinate normalization, and interpolation of instance values.

7.1.1 General

OFF Font Variations allow a font designer to incorporate multiple font faces within a font family into a single font resource. Variable fonts – fonts that use OFF Font Variations mechanisms – provide great flexibility for content authors and designers while also allowing the font data to be represented in an efficient format.

A variable font allows for continuous variation along some given design axis, such as weight:

Figure 7.1: Continuous variation along a design axis

Conceptually, variable fonts define one or more axes over which design characteristics can vary. Weight is one possible axis of variation, but many different kinds of variation are possible. Variable fonts can combine two or more different axes of variation. For example, the following illustrates a combination of weight and width variation:

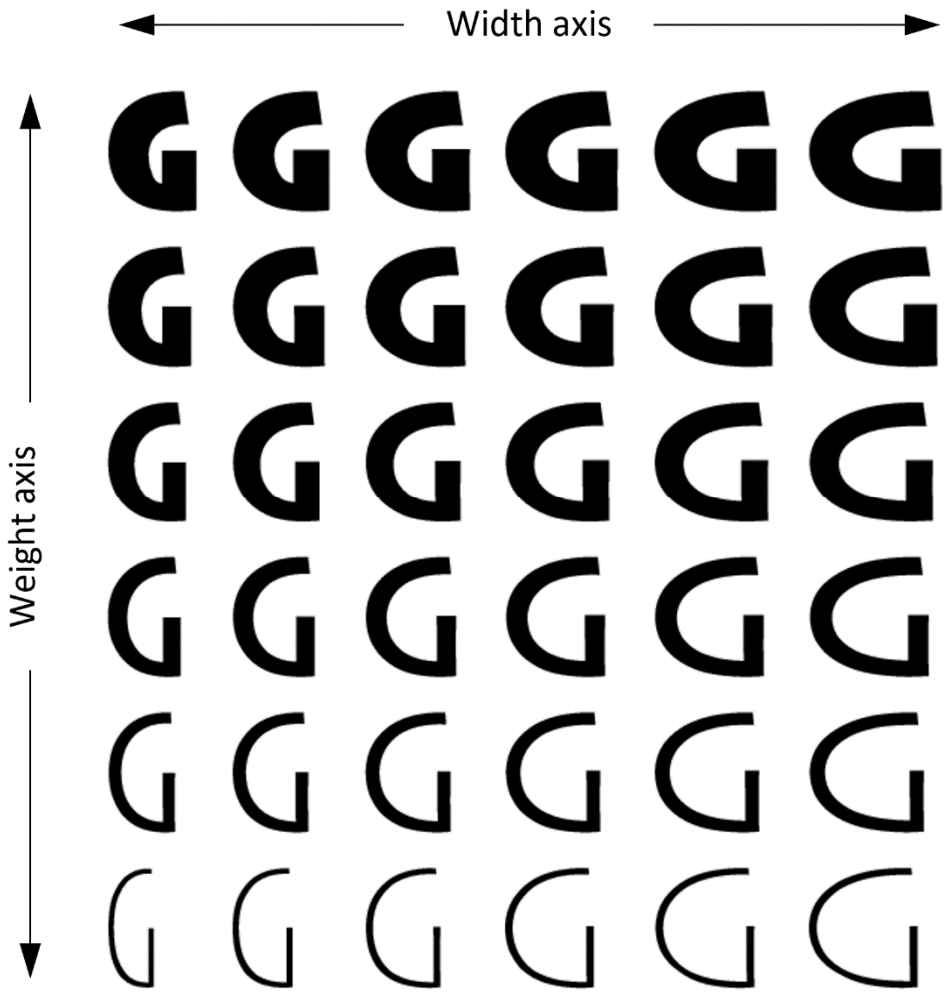


Figure 7.2: Continuous variation along multiple design axes

A variable font includes a table, the font variations ('fvar') table, that describes the axes of variation used by that font. This table determines how a variable font and its variation parameters will be presented to users and applications. Each axis is defined by a numeric range, using 16.16 floating values. Conceptually, this provides a continuous gradient of variation, a allowing for a large number of design-variation instances to be selected. Each instance would be designated by a coordinates array within the design-variation space – a specific value along each of the design axes. So, for instance, if a user or application requires a very-slightly narrower width or slightly more pronounced serifs, fine control over such axes of variation is available.

A font designer can also pre-define certain of those instances to have particular names. For example, a font can have continuous variation on a weight axis, but the designer may identify particular variation instances as “Light” or “Semibold”. Named instances can be used for any instance in the supported design-variation space. For example, in a font with weight and width axes, named instances might include “Light”, “Extended”, or “Semibold Condensed”. Details regarding named instances are also included in the font variations table.

Weight and width are commonly-used axes of design variation, but a variable font may use a wide range of other, possible axes of variation. For more information regarding supported axes, see the font variations ('fvar') table description.

In addition to a feature variations table, a variable font also includes a style attributes ('STAT') table that describes additional details about each axis of variation and about particular values (chosen by the designer) along each axis. These details include descriptor strings for those values, such as “Bold”, “Extended” or “Semi-sans”. For example, a weight/width variable font might support a “Bold Extended” variation, and the 'STAT' table would provide strings for “Bold” and “Extended” corresponding to the particular values along the weight and width axes, respectively. These strings can be used in creation of font-picker user interfaces. They

can also be used for projecting members of a multi-axis font family into different models for font families that assume a limited number of axes of sub-family variation, such as a weight/width/slant model. For example, a named instance “Semi-sans Light Condensed” might be projected into a “Light Condensed” member of a separate “Semi-sans” family. Because the 'STAT' table identifies values on each axis, software never needs to parse subfamily strings and guess that string tokens such as “Halbfett” refer to a particular value on some axis.

NOTE The style attributes table makes it possible for fonts with many design axes to be defined as a single, multi-axis family, yet still have instances across all of those axes supported in older applications that may only recognize a limited set of axes of variation, or a limited number of values on an axis. The host platform, which must support the style attributes table, can translate instances in a multi-axis family into fewer instances in multiple families that older applications will recognize.

As different variation instances of a font are selected, various items of data within a font must be adjusted accordingly. For example, a 'glyf' table can provide the default outline of a given glyph, but the outline would need to be adjusted in some manner to reflect different design variations. Several other data items besides glyph outlines may also need similar adjustments, including font-wide metrics, CVT values, or anchor positions within glyph-positioning lookup tables. A variable font includes required and optional tables that describe how such items within the font change from default values to different values as needed for different design-variation instances. For example, while a 'glyf' table can provide default outlines for glyphs, a glyph variations ('gvar') table would provide corresponding data that describes how each glyph outline changes for different variation instances.

A variable font has a default instance, with axis parameter values set to the defaults defined for each axis in the 'fvar' table. Several tables in the font provide default values for many different data items – such as positions of glyph outline points in the 'glyf' table, or a font-wide ascender distance in the OS/2 table. The default instance of a font uses the default values for such items without any adjustments, and the variation-specific tables are not needed. If the variation-specific tables – 'fvar', 'gvar', 'MVAR', etc. – were to be removed from the font or ignored, the remaining data would comprise a complete font for the default instance.

Font variation mechanisms for fonts using TrueType outlines were first introduced by Apple in “TrueType GX”. Some of the tables used for OFF Font Variations have been adapted from Apple's earlier specifications with some enhancements and revisions. (In particular, there are significant changes in the 'fvar' table specification in regard to both format and data values used, and the 'fmtx' table is not used.) Other extensions have also been created in order to integrate variation mechanisms into OFF. Implementers may wish to refer to Apple's specifications for historical insights, but should refer to the OFF specification as the reference for implementation of OFF Font Variations.

7.1.2 Terminology

Several terms are useful in discussing font variations and will be used in this document.

OFF Font Variations: The name of the technology described in this chapter.

Font face: A logical collection of glyph data sharing specific design parameters, along with associated metric data, and names or other metadata.

Font resource: OFF data that includes (at least) the minimal set of tables needed to comprise a functional font face.

NOTE Within OFF font files, each offset table and the tables it references comprise a font resource. A well-formed .OTF or .TTF file includes a single font resource; a well-formed .OTC or .TTC file includes one or more font resources. A font resource without variation-related tables provides data for a single font face. A single font resource that includes variation-related tables can provide data for multiple font faces.

Font family: A set of font resources that have a common family name – the same string values for name ID 16 (Typographic Family Name) or name ID 1.

NOTE It is assumed that all fonts within a family will share certain design characteristics, but differ in others. The design characteristics that are different potentially might be supported using variations.

Axis of variation: A designer-determined variable in a font face design that can be used to derive multiple, variant designs within a family.

Variable font: A font resource that supports multiple font faces in a family along designer-defined axes of variation using OFF Font Variations mechanisms – that is, by means of variation tables and other variation data in tables generally.

Glyph design grid: The visual, two-dimensional space in which a font's glyph outlines are designed.

Design-variation space: An abstract, multi-dimensional space defined by the axes of variation used by a font designer when designing a font family. In the context of a variable font, the variation space refers to the n-dimensional space defined by the axes of variation specified in the font's 'fvar' table.

NOTE A variation space can have one or more axes. In a variable font, the variation space is bounded by minimum and maximum values specified in the 'fvar' table. The zero origin has no special significance within a design-variation space. Within a variable font, however, the zero origin (using normalized coordinate scales – defined below) is a marked position since it corresponds to the font face represented directly by the font resource's name, glyph and metric tables without reference to any variation tables or other variation data.

Variation data: Data used in a variable font to describe the way that values for data items in the font are adjusted from default values to alternate values needed for different instances within the variation space.

Variation tables: OFF tables specifically related to Font Variations, including the following:

- Axis variations ('avar') table
- CVT variations ('cvar') table
- Font variations ('fvar') table
- Glyph variations ('gvar') table
- Horizontal metrics variations ('HVAR') table
- Metrics variations ('MVAR') table
- Vertical metrics variations ('VVAR') table

NOTE The 'fvar' table describes a font's variation space, and other variation tables provide variation data to describe how different data items are varied across the font's variation space. Note that not all of these tables are required in a variable font. Also note that variation data for certain font data items may be contained in other tables not specifically related to Font Variations. In addition, certain tables not specifically related to Font Variations are required in variable fonts. See subclause 7.1.6 (Variation data tables and miscellaneous requirements) for more details.

Point: In order to avoid ambiguity, *point* will be used only to refer to (X, Y) positions within the glyph design grid. When discussing the design variation space, *position* will be used to refer to positions within that space.

Variation instance: A font face corresponding to a particular position within the variation space of a variable font.

Named instance: A variation instance that is specifically defined and assigned a name within the 'fvar' table.

User coordinate scale: The numeric scale used to characterize a given axis of variation, and the scale used by applications when selecting instances of a variable font.

NOTE Some axes of variation have a prescribed, limited range, expressed in terms of the user scale. When using a particular variable font, the user scale for a given axis is bounded by minimum and maximum specified within the 'fvar' table, and may be a sub-range of the valid range for that axis generally.

Normalized coordinate scale: When processing variation data in a variable font to derive values for particular instances, a normalization process is applied to map user-scale values on each axis to a normalized scale applicable within that font that ranges from -1 to 1.

NOTE The 'fvar' table specifies user-scale minimum, default and maximum values for each axis. In the normalization process, these get mapped to -1, 0 and 1 respectively, with other values along each axis mapping to intervening points. Mapping of other values is modulated by the 'avar' table, if present. All of the variation data within the font makes reference to axis values or positions within the font's variation space in terms of normalized-scale values.

Tuple / N-tuple: An ordered set of coordinate values used to designate a position within the variation space of a font.

NOTE "Tuple" is used here with a meaning that is consistent with conventional usage in computer science and mathematics. In Apple TrueType specifications, "tuple" has been used with a different meaning to refer to sets of variation data associated with a particular region of the font's design variation space. In this document, "tuple variation data" is used for that meaning, and "n-tuple" is used in many cases so as to avoid confusion with usage in Apple specifications.

Region: A sub-space (that is, some portion or subset) of the design variation space over which a variation adjustment is described.

NOTE A region involves all of the axes of the font's variation space; it is not a "sub-space" in the sense of involving only a subset of axes. In normalized coordinates, regions are always rectilinear: they have straight edges and right-angled corners. Variation data may be defined for up to 65,535 regions in a font's variation space.

Master: A set of source font data that includes complete outline data for a particular font face, used in a font-development workflow.

NOTE Some font-development workflows utilize several masters as source data for creating font resources for different faces within a family. Multiple source masters might also be used to create a variable font. Each source master would correspond to a single instance in the variation space, and possibly might correspond to variation data for a particular region in the variable font. Whereas each master includes complete outline data, however, the variable font includes only a single set of complete outline data (in the 'glyf' or 'CFF2' table), which is complemented with variation data for different regions to represent the full range of instances supported by the font.

***Deltas / Adjustment deltas*:** Numeric values in variation data that specify adjustments to default values of data items for particular regions within the variation space or for sub-ranges within a particular axis.

Delta set: A set of adjustment deltas associated with a particular region of the variation space.

Scalars: Co-efficient values applied to deltas to derive adjustment values needed for a particular variation instance.

Interpolation: The process of deriving adjusted values for some font data items, such as the X and Y coordinates of glyph outline points, for a particular variation instance.

7.1.3 Variation space, default instances and adjustment deltas

A variable font supports one or more axes of variation. Commonly-used axes of variation should be registered, though custom, designer-defined axes can also be used. Each axis has a distinct tag that is used to identify it in the 'fvar' table. See the 'fvar' table description for more details about axis tags.

The specification of axes used for a variable font is given in the 'fvar' table, along with minimum, default and maximum values for each axis. This defines a variation space for the font. It is entirely up to the designer what range of design variation is supported for each axis, and how the designs align with the scale for each axis.

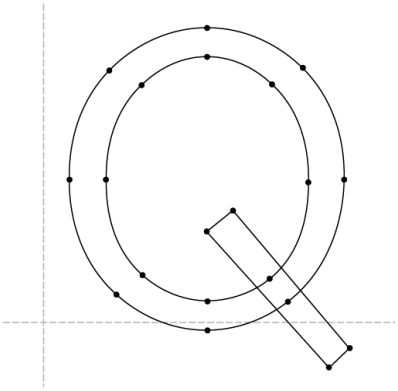
For example, a variable font may support a full range of weights from thin to black:

Font A: Thin to black weight variation

But a designer might also choose to support only a limited weight range:

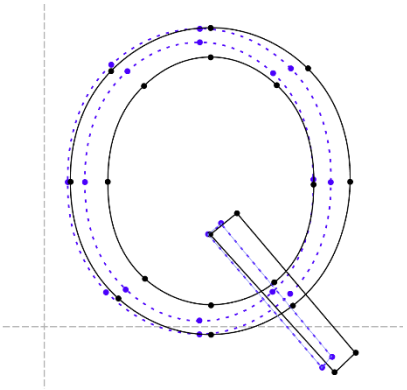
Font B: Regular to black weight variation

The variable font has a default instance, which corresponds to the position in the variation space with coordinates set to the default values for each axis specified in the 'fvar' table. The default instance uses default values for various data items that are provided directly in non-variations-specific font tables, such as the grid coordinates of outlines points for a glyph in the 'glyf' table.

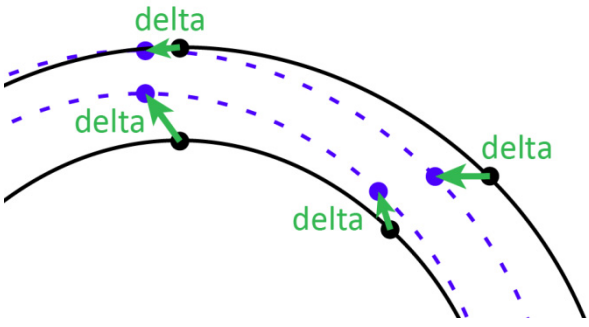


Default glyph outline data in a ‘glyf’ table entry

All other instances have non-default coordinate values for one or more axes. These other instances are supported by variation data that provide adjustment deltas for various font data items that produce an adjustment from their default values.



Default glyph outline and adjusted point positions for a non-default instance



Detail inset showing contour point adjustments

Typically, deltas are provided for the extremes on each variation axis, though deltas can be provided for other positions in the variation space as well. (See below for more details.) For axis positions between the default and minimum or maximum extreme, other values are interpolated.

The font designer can determine which design is considered the default, and what deltas are provided. For example, a font with thin-to-black weight variation might be implemented with Regular (400) as the default, and Thin (100) and Black (900) as minimum/maximum values. In this case, variation data would include deltas for the Thin extreme and also deltas for the Black extreme.

But a different font with thin-to-black weight variation might be implemented with Thin as the *default and minimum* value and Black as the maximum. In this case, variation data might include deltas for only the Black extreme.

Note that a consideration in the choice of default is desired behavior in legacy applications or platforms that do not support Font Variations: in such software, only the default instance of a variable font will be supported.

A common process for developing a variable font involves the use of multiple, master source fonts. Each master provides complete glyph outline data for designs for a different position within a variation space. For example, a font designer might create fonts for thin and heavy extremes along a weight axis.

From these two source masters, font tools can derive a variable font that has complete glyph outlines for a default weight plus deltas for one or more non-default weights, including the minimum or maximum weights.

Note that each of the source, master fonts has complete outline data for a particular design variant. In contrast, the variable font has complete outlines for only one variation instance, with all other instances derived using the default outlines plus deltas. Each source master may correspond to a region with associated variation data in the variable font, though the relationship between source masters and the sets of variation data within the font will depend on the nature of the designs and on the tools used to produce the variable font.

Also note that a requirement for using multiple, master, font sources to derive a variable font is that corresponding glyph outlines must be point-compatible: they shall have the same number of contours and the same number of points in each contour.

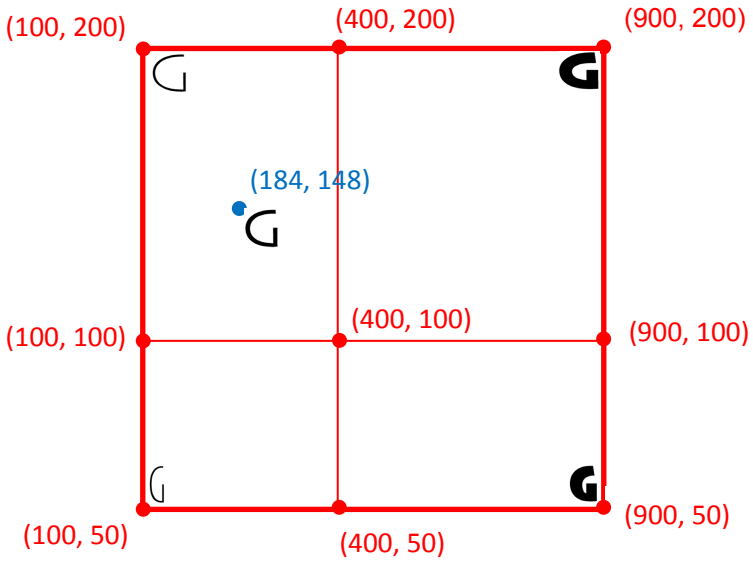
7.1.4 Coordinate scales and normalization

Positions within the variation space can be represent as an n-tuple – an ordered list of coordinate values. Examples will be seen below. The coordinate values of an n-tuple may use user-axis scales, or may use normalized scales. The precise relationship between these scales will be described.

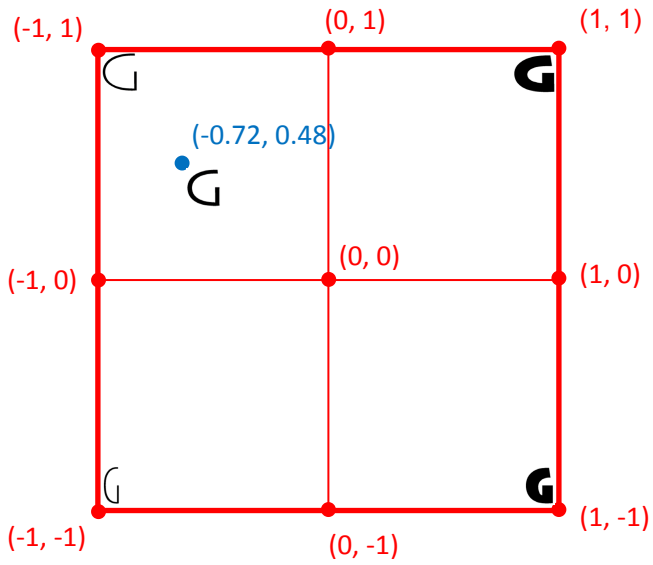
User coordinates refers to an n-tuple of coordinate values expressed using *user axis scales* . User scales refer to the numeric scales used to describe a variation axis within the 'fvar' table. Each variation axis uses its own numeric scale, as appropriate to the nature of that axis of variation. The scales for registered axis tags are defined as part of the axis tag registration, though different fonts may support different sub-ranges of an axis scale. In this way, the 'fvar' table of a given font defines a particular coordinate system for the variation space of that font that may be unlike that of other fonts.

Whereas the definitions in the 'fvar' table are expressed in user coordinates, the variation data formats used within a variable font use a normalized coordinate system – *normalized coordinates* – in which the minimum, default and maximum values specified for each axis in the 'fvar' table are mapped to -1, 0 and 1, respectively.

For example, the following figure illustrates the user coordinate system of the variation space for a possible font with weight and width axes of variation:



The following figure illustrates the normalized coordinate system for the same font:



The normalization transformation uses a default transformation followed by a secondary modification of the transformation defined in the 'avar' table, if present. An 'avar' table does not affect the mapping of minimum, default, and maximum values to -1, 0 and 1; it can only affect mapping of intervening values. For more details on this modification, see the 'avar' table description.

The default normalization mapping divides the variation range for each axis into two segments: minimum value to default value, and default value to maximum value. The minimum, default and maximum values are mapped into -1, 0 and 1 respectively. Within each segment, all other values are interpolated linearly, as follows:

Let userValue be the user-scale coordinate value for a user-selected instance value for a given axis, let normalizedValue be the normalized instance value, let axisMin be the minimum value for the axis specified in the ‘fvar’ table, etc.

Force the user-scale coordinate value to be in range by clamping to the minimum and maximum values:

```
if userValue < axisMin
    userValue = axisMin;
if userValue > axisMax
    userValue = axisMax;
```

Interpolate values linearly within the different segments:

```
if (userValue < axisDefault)
{
    normalizedValue = -(axisDefault - userValue) / (axisDefault – axisMin);
}
else if (userValue > axisDefault)
{
    normalizedValue = (userValue – axisDefault) / (axisMax – axisDefault);
}
else
{
    normalizedValue = 0;
}
```

When processing variation instance coordinates and variation data, the amount of precision used and the handling of rounding can potentially have noticeable impacts on visual results. In order to ensure consistent behavior for a given font across implementations, implementations must observe the following requirements in relation to precision and rounding:

1. The input to normalization must be in 16.16 format. If an application provides an input value represented as either a *float* or *double* data type, the method described below must be used for conversion to 16.16.
2. The math calculations for normalization, specified above, are done in 16.16.
3. After the default normalization calculation is performed, some results may be slightly outside the range [-1, +1]. Values must be clamped to this range:

```
if result < -1
    result = -1;
if result > 1
    result = 1;
```

4. If an 'avar' table is present, math calculations are done in 16.16, and results are clamped to the range [-1, +1] as above.
5. Convert the final, normalized 16.16 coordinate value to 2.14 by this method: add 0x00000002, and sign-extend shift to the right by 2.
6. The 2.14 result must be stored and returned in certain operations, as described below.
7. For subsequent calculations – calculation of interpolation scalars or accumulation of scaled delta values – the 2.14 representation may be converted to float, 16.16 or other implementation-specific representations. It is recommended that at least 16 fractional bits of precision be maintained, and that any rounding be done at the last point before a value is used.

When converting from float or double data types to 16.16, the following method shall be used:

1. Multiply the fractional component by 65536, and round the result to the nearest integer (for fractional values of 0.5 and higher, take the next higher integer; for other fractional values, truncate). Store the result in the low-order word.
2. Move the two's-complement representation of the integer component into the high-order word.

A normalized value in 2.14 representation must be obtained exactly as specified in steps 1 to 5 above. In fonts with TrueType instructions, this exact value must be returned by the GETVARIATION instruction. (See TrueType InstructionSet [[# Get Variation]].) If a font has OFF Layout tables in which FeatureVariation tables are used, this exact value must be used when comparing with axis range values specified in a condition table.

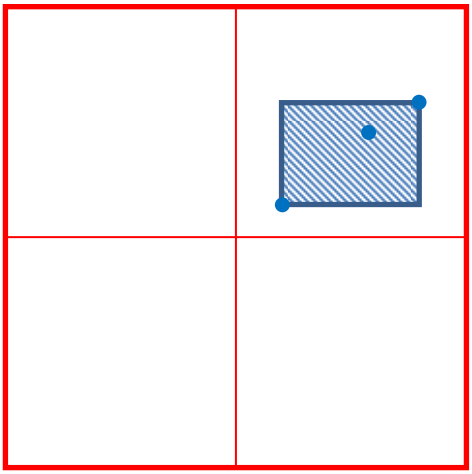
7.1.5 Variation data

Variation data provide data that describes variation of particular font values over the variation space. For example, variation data in the 'gvar' table describes how glyph outlines in the 'glyf' table are transformed by specifying how individual points in a glyph outline get moved for different variation instances.

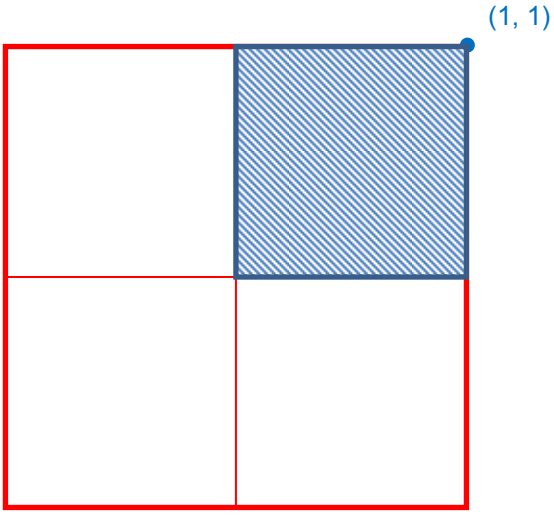
Variation for a given font value is expressed as combinations of deltas that apply to different regions of the variation space, and that are combined in a weighted manner to derive adjustments for instances at different positions in the variation space. Each delta in the variation data is associated with a specific region of the variation space over which it has an effect. The aggregate combination of deltas and their associated regions comprise the variation data. The variation data for different items in a font are stored in different locations. For example, variation data for entries in a 'glyf' table are stored in a 'gvar' table; variation data for certain entries in the 'OS/2' table is stored in an 'MVAR' table. In the case of outline data in a 'CFF2' table, variation data is stored within the 'CFF2' table itself. See the following section below for more details.

As mentioned, each delta value is associated with a particular region of the variation space over which it applies. The effective region for a delta is always rectilinear (in normalized coordinates). Therefore, this region can always be specified by a pair of n-tuples designating positions at diagonal-opposite corners of the region. Within the specified region, variation effects will vary from zero change to some peak change at a particular position within the region. Thus, in the general case, there are three positions that matter: the diagonal-opposite corners that define the extent of the region, and a position at which the peak change occurs.

NOTE The figures shown below will use two axes of variation. The concepts and the statements made, however, apply to fonts with any number of axes of variation: regions are always rectilinear, and the diagonal-opposite corners plus a peak are the positions that describe a region.



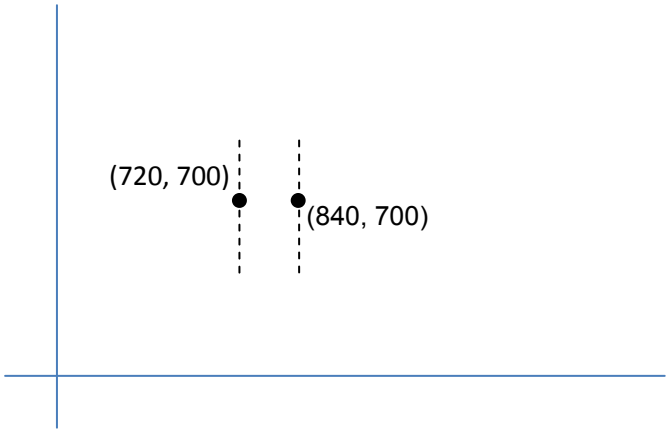
This general case is not the most common in practice. In most cases, there is a need to describe a maximal variation at an outer position of the variation space that diminishes to zero change at the zero origin — the default instance. In this case, then, the zero origin is one of the corner positions for the applicable region, and the peak change occurs at the diagonal-opposite position. For this common case, then, the effective region and peak position can be described using a single n-tuple.



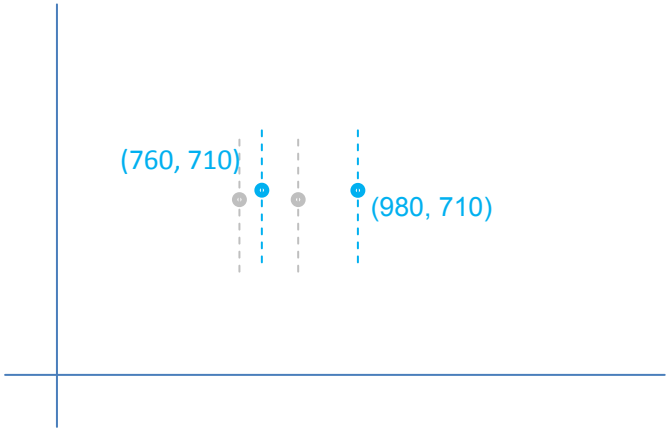
The more general, but less common, case involves arbitrary regions, illustrated earlier; these are referred to as *intermediate* regions. In these cases, the variation data requires three n-tuples: one for the peak-change position, and two for *start* and *end* positions at diagonal-opposite corners.

Delta values in the variation data specify a maximal adjustment for an instance at the peak position. The effects taper off for other instances, falling to zero adjustment for instances outside the region of applicability. When a given variation instance is selected, a scalar value is calculated and applied to a given delta to derive a net adjustment associated with that delta for that instance. These scalars will always be in the range 0 (zero adjustment) to 1 (maximal adjustment). Specific details on this scalar calculation are provided below.

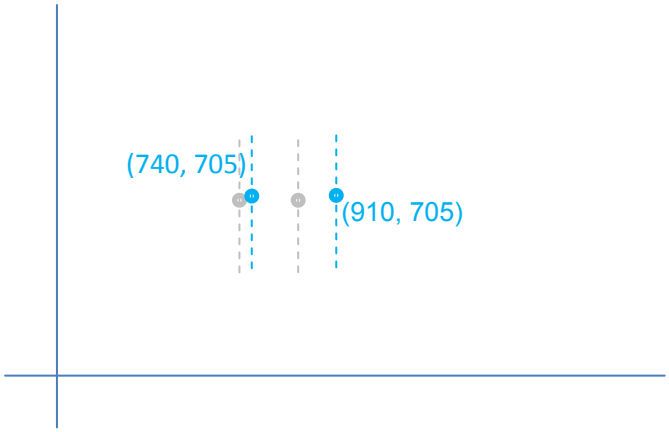
An example will help to explain these concepts. Consider a single-axis font with weight variation. A particular glyph outline defined in the ‘glyf’ table might have a pair of points (among others) that are on-curve points on opposite sides of a stem. The entry in the ‘glyf’ table would specify glyph-design-grid coordinates for these points for the default instance of the font, perhaps corresponding to the regular weight:



Variation data would be defined for the maximum value on the weight axis, corresponding to 1.0 in the normalized weight scale. This data would provide X and Y deltas for the two contour points to shift their positions as needed for the heaviest-supported weight instance:



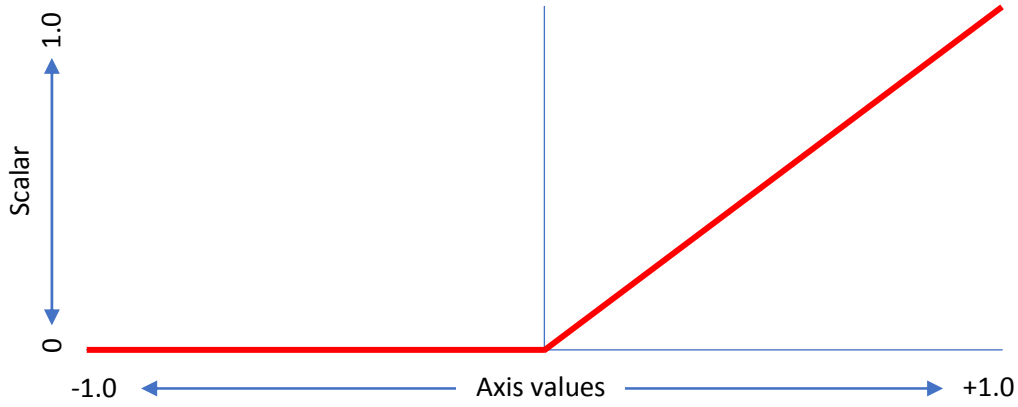
In this example, the first point would have X and Y deltas of +40 and +10, respectively; the second point would have deltas of +140 and +10. These provide a maximal adjustment of the outline points, applied when the user-selected instance is at the maximum weight. For weights between the default and the maximum, such as a normalized weight value of 0.5, the effect is scaled back.



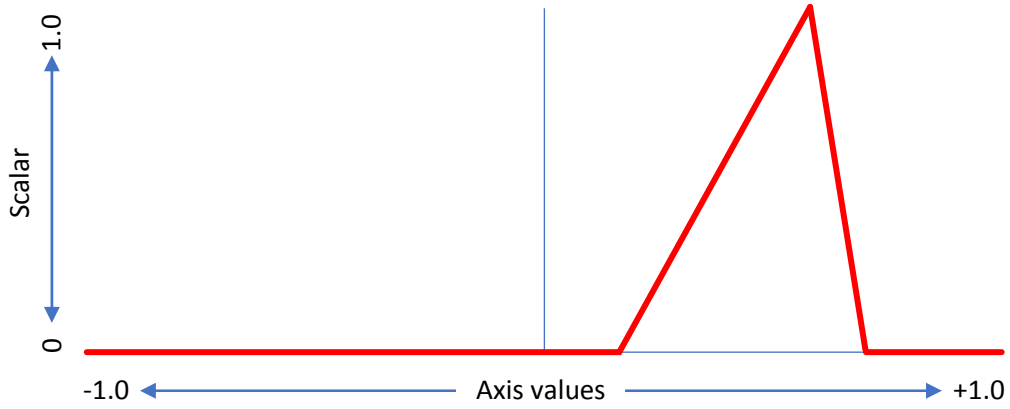
In this case, a scalar co-efficient of 0.5 is applied to the delta values.

The scalar calculation can be thought of as a function that maps each normalized axis value from -1 to 1 onto a scalar range of 0 to 1. Each region that has associated variation data has its own scalar function, and the scalar function is defined precisely by the region description.

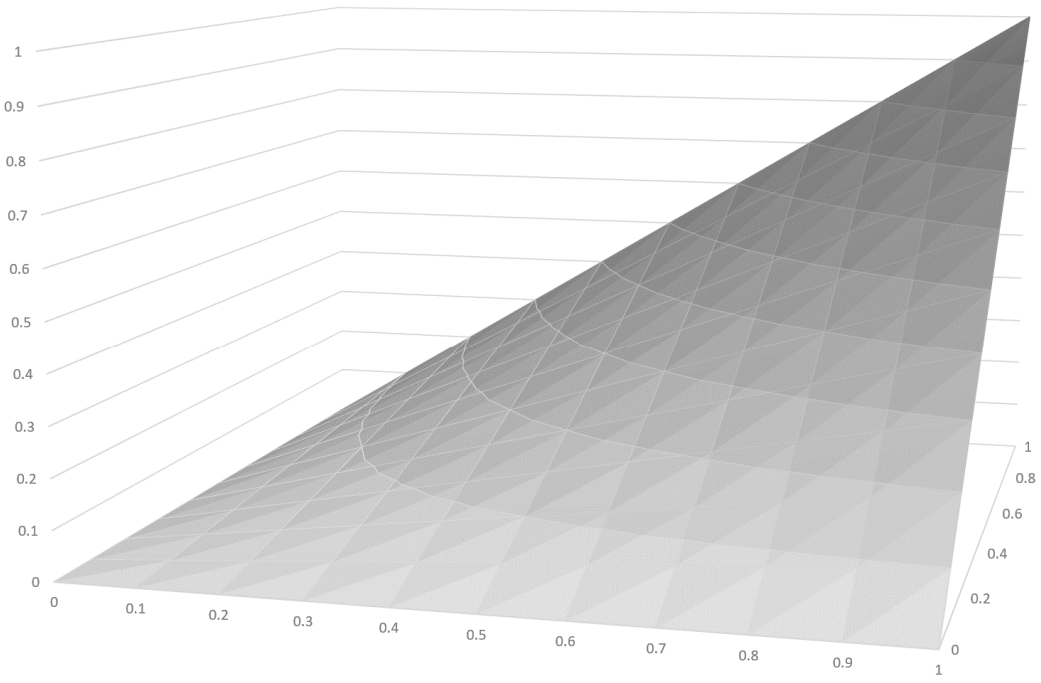
For example, in a single-axis font, if a delta is provided for the region from 0 to 1 with the peak effect at 1, the scalar function would be as follows:



This example considers a non-intermediate region. The same concepts can be generalized to intermediate regions. An intermediate region has start and end axis values between which there is some adjustment effect, and a peak axis value at which the full adjustment effect is applied. The scalar function has a triangular shape within the applicable range, with a value of 1.0 at the peak axis value, 0 at or below the start axis value, and 0 above the end axis value.



When generalizing to two or more axes, similar concepts apply, but contributions for each axis are combined into an overall effect. Scalars are calculated for each axis, and the per-axis scalars are multiplied together to produce an overall scalar for the given delta and the given instance. For example, the following graph illustrates an approximation of the scalar function for a region in a two-axis font with peak at (1, 1):

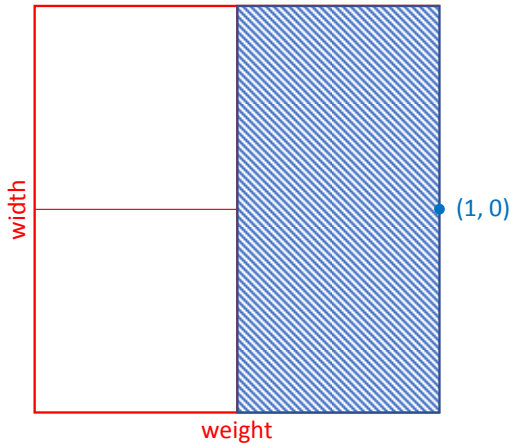


Since the scalar value calculated for each axis is between 0 and 1, the product when scalars for each axis are multiplied together is also between 0 and 1. The maximal adjustment effect for a given delta is obtained only when the instance axis values for all axes align with the peak coordinate values for the region associated with that delta.

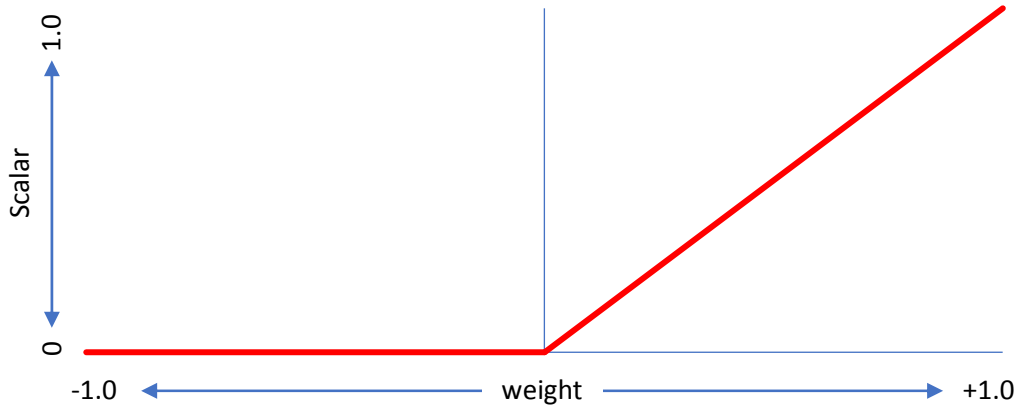
The minimum and maximum values specified for an axis in the ‘fvar’ table determine limits on instances that can be selected by a user. If a user requests an instance with an axis value below the minimum, the minimum value is used; or if an axis value above the maximum is requested, the maximum is used. Thus, when processing variation data for a selected instance, the normalized axis values will always be between -1 and 1.

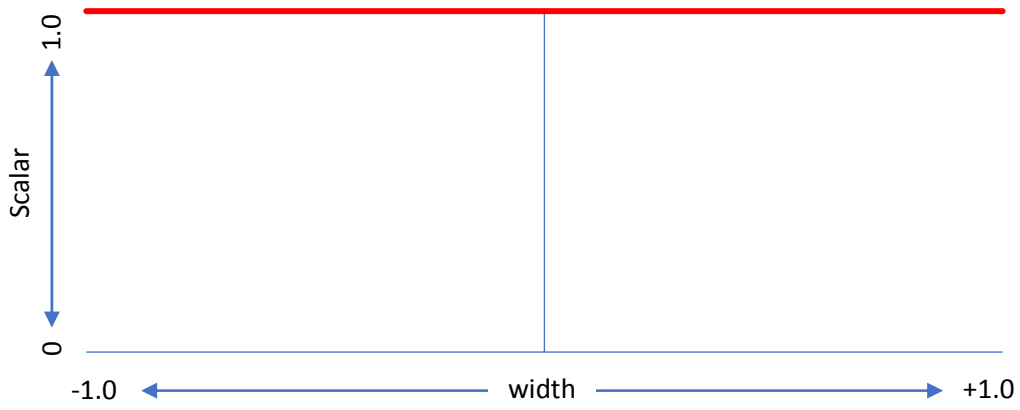
With that constraint assumed, let us consider the scalar value for a given delta when instance axis values are outside the region of applicability. If the selected instance is out of range on any axis, then the scalar value pertaining to that axis will be 0. As mentioned, per-axis scalars are multiplied together to produce an overall scalar. Thus, if the selected instance is out of range on any axis, then the overall scalar for that delta will be 0, and no adjustment from that delta will be applied.

When a delta is provided for a region defined by n-tuples that have a peak value of 0 for some axis, then that axis does not factor into scalar calculations. This means that the adjustment effect is the same for any value in that axis, if other axis values remain constant. In effect, the region of applicability spans the full range for the zeroed axis. For example, suppose a font has two axes, weight and width, and that deltas are provided for a region from (0, 0) to (1, 0). In this case, the deltas are applicable for any instance value on the second axis (width), so long as the instance value in the first axis (weight) is in range:



In this case, the scalar function *for the second axis* (width) is, effectively, a constant value of 1, with no effect on the net scalar calculation. The following graphs illustrate the scalar functions for each of the two axes, weight and width, in this example:

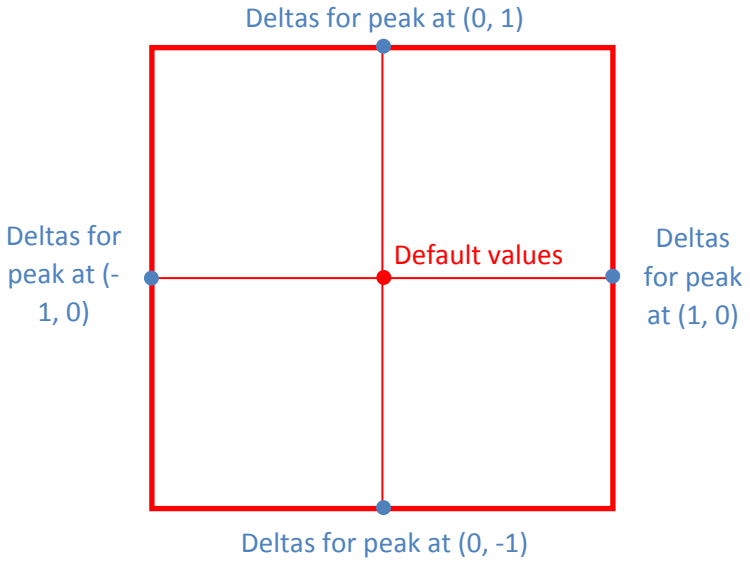




For a given font value, deltas may be provided for several different regions in the variation space. When a particular variation instance is selected, zero, one or many of those deltas may have an effect, according to whether the position of the instance falls within the region of applicability for each delta. Different scalars are

calculated for each applicable delta, and the scaled values for applicable deltas are combined to derive a net adjustment.

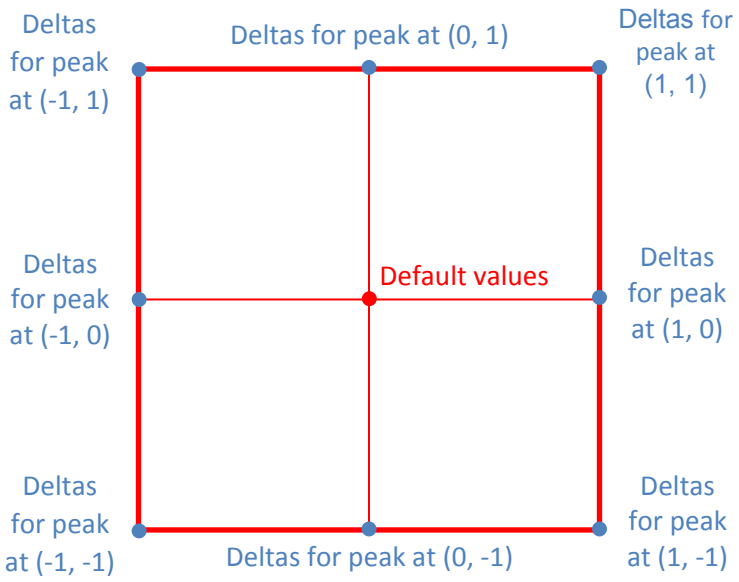
When creating a single-axis font, deltas will be required for both the minimum and maximum extremes of that axis. (Both extremes, that is, unless one is also the default.) Additional intermediate-region deltas may also be provided. When creating a multi-axis font, deltas would typically be provided for the minimum and maximum extremes on each axis. The following figure illustrates this for a two-axis font:



Two-axis font with deltas at minimum/maximum extremes on each axis

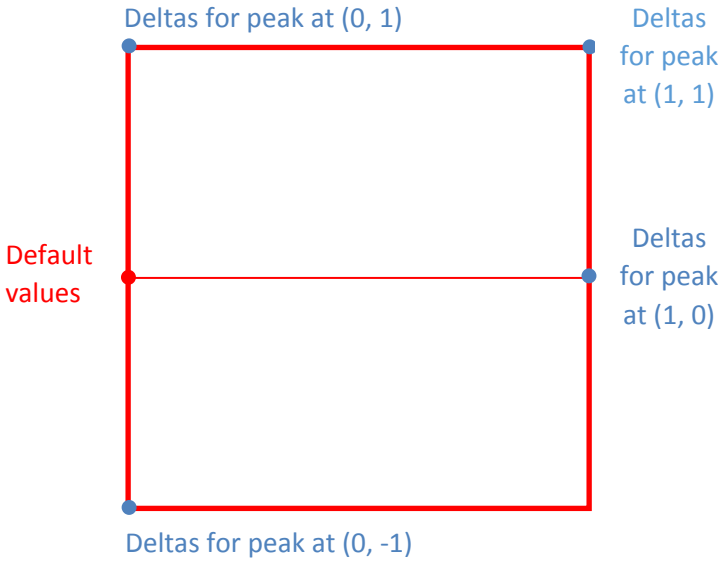
As noted above, when deltas are specified for a region with some axis value being zero, then the deltas apply to all values on that axis. Therefore, for the instance at position (1, 1), deltas for (1, 0) and (0, 1) will both apply. This means that the adjustments for the (1, 0) deltas and the adjustments for the (0, 1) deltas will both be applied to produce a combined effect. If the adjustments made for each axis are entirely independent of the adjustments for the other axis, then the two sets of deltas may be sufficient to provide the intended values for the (1, 1) instance.

Often, however, these two sets of deltas alone will not be sufficient to provide the desired results for all instances, and that additional deltas are required for the (1, 1) position in addition. Generalizing, in a multi-axis font, it will often be the case that at least some deltas are needed for the corner extremes as well as for the axis end points.

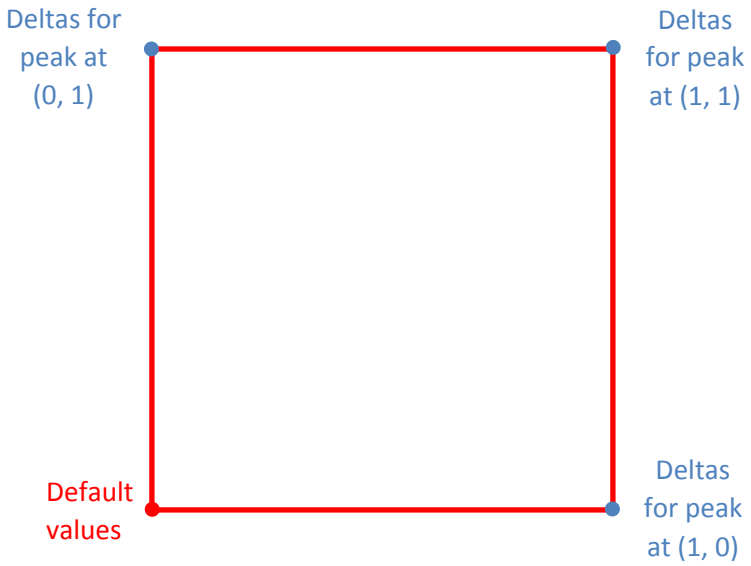


Deltas at minimum/maximum extreme plus extrema-intersection corners

As noted in the Variation Space, Default Instances and Adjustment Deltas section above, the default instance can correspond to the minimum or maximum value on one or more axes. This can allow variations across a variation space to be implemented using fewer regions and associated delta data. The following figures illustrate some additional possibilities for a two-axis font.

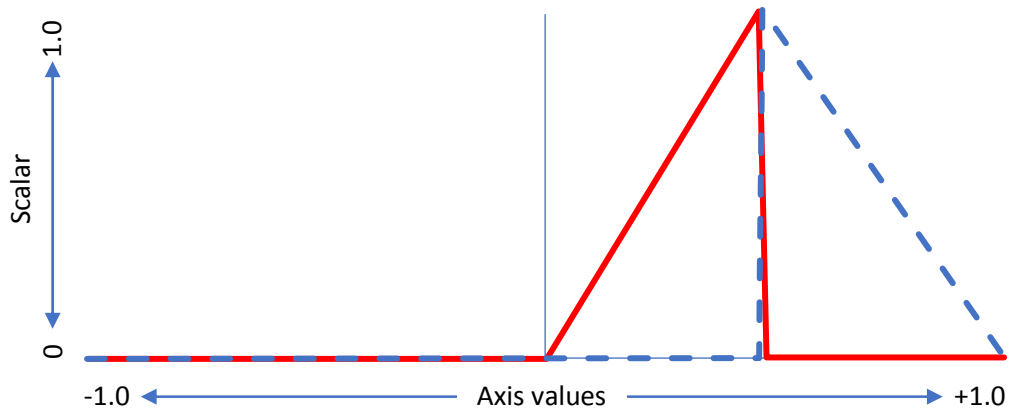


Two-axis font with default at minimum for one axis



Two-axis font with default at minimum for both axes

As noted above, an intermediate region provides an axis scalar function with a triangular, or “tooth”, shape. A pair of intermediate regions that barely overlap and that have a sharp incline at the overlap can be used to provide an inflection point along an axis in regard to some variation behavior.



Note that each intermediate region has its own associated delta values, and deltas can be used to give some sharp transition at the overlap point. For example, contour points could suddenly shift to make some element of a glyph's structure appear or disappear, as illustrated in the following figure:

Glyph structure simplified at heavier weights

NOTE When using such techniques, placement of such a transition point along an axis and placement of named instances should be considered together so that sharp transitions do not occur close to a named instance. This will avoid any possibility of inconsistent behavior in different applications when using named instances that might arise due to small discrepancies in processing the numeric values.

NOTE When using such techniques, it is important to bear in mind that some applications will support selection of arbitrary instances, including those with axis values in the overlapping range, and that, in the overlapping range, scaled deltas for both of the intermediate regions will apply with cumulative effect. Some design iteration may be needed, with small adjustments to delta values or the way that the regions overlap, in order to avoid unexpected or undesired results in the transitional range.

NOTE The above figure illustrates the use of intermediate regions to implement a “stroke-reduction” effect. Another implementation technique that can be used to change the structure of a glyph for particular variation-axis value ranges is glyph substitution. The OFF Layout [[Require Variation Alternates]] feature in combination with a [[FeatureVariations]] table within the 'GSUB' table can be used to perform glyph substitutions when a variation instance is selected in some range along one or more axes. This may be an easier and more-easily maintained technique, and is generally recommended for achieving such effects.

The above has provided an overview of the basic concepts involved in variation data: regions of applicability, per-axis and overall scalars, and combined effects of multiple, applicable deltas. A detailed specification of the interpolation process is provided below.

7.1.6 Variation data tables and miscellaneous requirements

The previous subclause identified X and Y coordinates of glyph outline points as data items requiring adjustment for different variation instances. Many other data items in a font may also require similar adjustment, including the following:

- Font-wide metric values in the ‘OS/2’, ‘hhea’, vhea’ or ‘post’ tables.
- Glyph metric values in the ‘hmtx’, ‘vmtx’ or VORG tables.
- PPEM ranges in the ‘gasp’ table.
- Anchor positions, and adjustments to glyph positions or advance in the ‘GPOS’ or ‘JSTF’ tables.
- X or Y coordinates for ligature caret positions in the ‘GDEF’ table.
- X or Y coordinates for baseline metrics in the ‘BASE’ table.
- CVT values.

A variable font may contain variation data for any or all of these. The variation data for different items is provided in various tables within a font.

NOTE While several data items in a font may require adjustment for different instances, there will be other items that do not change across instances. For example, the font family and unitsPerEm are not impacted by variation. It should be noted in particular, however, that certain values that can potentially be impacted by variations are not supported with variation data. In particular, the xMin, yMin, xMax, yMax, macStyle and lowestRecPPEM fields in the font header (‘head’) table are not supported by variation data and should only be used in relation to the default instance for the font. Also, variations for values in the kerning (‘kern’) table are not supported; variable fonts should handle kerning using the ‘GPOS’ table.

Two tables are required in all variable fonts:

- A font variations (‘fvar’) table is required to describe the variations supported by the font.
- A style attributes (‘STAT’) table is required and is used to establish relationships between different fonts belonging to a family and to provide some degree of compatibility with legacy implementations.

If a variable font has TrueType outlines in a ‘glyf’ table, the outline variation data will be provided in the glyph variations (‘gvar’) table, which is required. Variation data for CVT values can be provided in the optional CVT variations (‘cvar’) table.

If a variable font has PostScript-style outlines in a Compact Font Format 2.0 (‘CFF2’) table, the ‘CFF2’ table itself also contains the associated variation data.

NOTE A ‘CFF2’ table can be used in non-variable fonts as well as in variable fonts. Also note that variations for outlines using the Compact Font Format version 1.0 (‘CFF ’) table are not supported.

The metrics variations (‘MVAR’) table is used to provide variation data for various font-wide metrics or other numeric values in the ‘gasp’, ‘hhea’, ‘OS/2’, ‘post’ and ‘vhea’ tables. An ‘MVAR’ table should be added if adjustments to any of these values are required. Note that it is not required to provide variation data for all of the data items covered by the ‘MVAR’ table: variation data is optional for all items. If there is no variation data for a given item, the default value applies to all instances.

NOTE Apple platforms allow for use of a *font metrics* ('fmtx') table to specify various font-wide metric values by reference to the X or Y coordinates of contour points for a specified glyph. OFF font variations does not use the font metrics table.

The 'hmtx' and 'vmtx' tables provide horizontal and vertical glyph metrics. Variation data for horizontal and vertical glyph metrics can be provided using the horizontal metrics variations ('HVAR') and vertical metrics ('VVAR') tables.

In a font with TrueType outlines, the rasterizer combines 'hmtx' and 'vmtx' values with glyph xMin, xMax, yMin and yMax values in the 'glyf' table to generate four "phantom" points that correspond to the glyph horizontal and vertical metric values. (See "Instructing TrueType Glyphs" [24] for more background on phantom points.) In a variable font, the variation data for a glyph in the 'gvar' table will include adjustment deltas for the glyph's phantom points. As a result, interpolated glyph metrics for a given instance can be obtained by interpolating the phantom point positions for the instance. This may be costly for some text-layout operations, however. In order to provide the best performance on all platforms, it is recommended that all variable fonts with TrueType outlines include an 'HVAR' table. If the font supports vertical layout and includes 'vhea' and 'vmtx' tables, it is recommended that the font include a 'VVAR' table.

The CFF2 rasterizer does not generate phantom points, and CFF2 variation data will not include adjustment deltas for phantom points. For this reason, in a variable font with CFF2 outlines, an 'HVAR' table is required. If the font supports vertical layout, then a 'VVAR' table is required.

NOTE The 'hdmx' and 'VDMX' tables are not used in variable fonts.

If a font has OFF Layout tables, variation data for values from the 'GDEF', 'GPOS' or 'JSTF' table will be included, as needed, within the 'GDEF' table. Variation data for the 'BASE' table will be included, as needed, within the 'BASE' table itself.

In some variable fonts, it may be desirable to have different glyph-substitution or glyph-positioning actions used for different regions within the font's variation space. For example, for narrow-width or heavy-weight instances in which counters become small, it may be desirable to make certain glyph substitutions to use alternate glyphs with certain strokes removed or outlines simplified to allow for larger counters. Such effects can be achieved using a feature variations subtable within either the 'GSUB' or 'GPOS' table. See "OFF Layout Common Table Formats" for more information.

In a variable font with TrueType outlines, the left side bearing for each glyph must equal xMin, and bit 1 in the flags field of the 'head' table must be set.

In all variable fonts, bit 5 in the flags field of the 'head' table must be cleared. (On certain platforms, bit 5 affects metrics in vertical layout. Bit 5 must be clear to ensure compatible behavior on all platforms.)

7.1.7 Algorithm for interpolation of instance values

The process of interpolating adjusted values for different variation instances is used for all font data items that require variation — positions of outline glyph points, ascender or other font-wide metrics, etc. The interpolation process involves the following:

- Determining the deltas that are applicable for that instance.
- For each applicable delta, calculating per-axis scalars for that instance, then multiplying the per-axis scalars together to produce an overall scalar for that delta.
- Scaling each applicable delta by the calculated scalar for that delta.
- Combining all of the scaled deltas to produce an overall adjustment.

When processing the 'gvar' table, there is an additional step in calculations, which is to infer delta adjustments for points when deltas are not given explicitly. This applies only to the 'gvar' table, and is described in the 'gvar' table subclause.

As described earlier, an instance axis value that is outside the region of applicability for a given delta is equivalent to having a per-axis scalar value of zero. Also, having an axis that has no effect in relation to a given delta (the n-tuples have a peak value of zero for that axis) is equivalent to having a per-axis scalar value of one. Thus, determination of applicability and axis interactions can all be combined into a step of deriving an overall scalar.

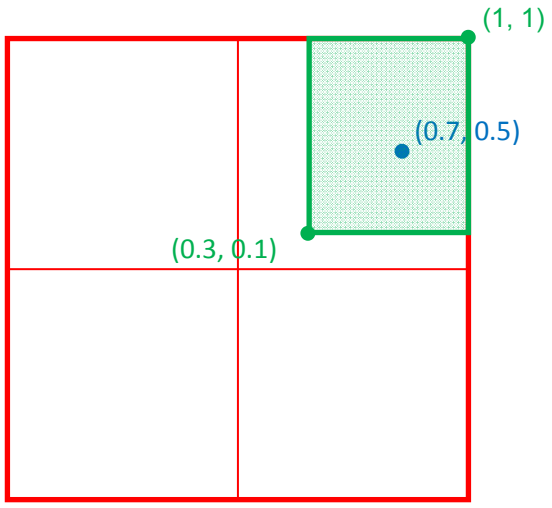
The description of the interpolation process below will refer to start, peak, and end coordinate values. As described earlier, an intermediate region is described using three n-tuples, two for diagonal-opposite corners (start and end) that specify the extent of the region, and a peak. Non-intermediate regions have one of the corners at the peak and the other corner at the zero origin. In some variation data structures, a non-intermediate region is specified using a single n-tuple, that of the peak. In this case, the start and end coordinates are implicit: one is the same as the peak, and the other is the zero origin.

In order for the definition of a region within variation data to be valid, start, peak and end values must be well ordered. That is, for each axis, the start axis coordinate must be less than or equal to the peak coordinate, and the peak coordinate must be less than or equal to the end. Also, the start and end coordinates must both be non-negative or non-positive — they cannot cross zero.

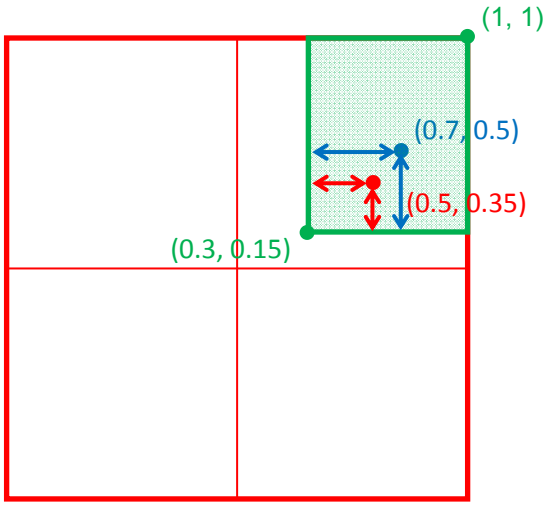
In the discussion to this point, individual deltas have been described as having an associated region of applicability. Variation data can be organized in different ways. In some cases, as in the ‘gvar’ table, several deltas corresponding to many target items (all the outline points of a glyph) and a single region of the variation space are organized together. In some other cases, as in the ‘MVAR’ or ‘CFF2’ tables, deltas covering multiple regions are organized together by individual target items. In either case, each individual delta is associated with a particular region of the variation space. The following description of the interpolation process will refer to interpolating a value for an individual item, but when applied to particular contexts such as the ‘gvar’ table, it should be understood that the same calculations are applied to many different items in parallel.

As described above, the effect of a given delta is modulated by a scalar function ranging from 0 to 1, with a value of 1 for an instance at the peak position of the region associated with that delta. The overall scalar is the product of per-axis scalars, and each per-axis scalar is calculated as a proportion of the proximity of the instance coordinate value to the peak coordinate value relative to the distance of the peak from the edges of the region.

For example, consider an intermediate region (green in the figure below) in a two-axis variation space, with corners at (0.3, 0.15) and (1, 1), and a peak (blue in the figure below) at (0.7, 0.5):



Then consider the instance (red in the figure below) at (0.5, 0.35). The per-axis scalars will be the ratio of the distance of the instance coordinate value from the nearest edge of the region divided by the distance of the peak from that edge:



$$\textit{Axis 1 scalar} = \frac{0.5 - 0.3}{0.7 - 0.3} = 0.5$$

$$\textit{Axis 2 scalar} = \frac{0.35 - 0.15}{0.5 - 0.15} = 0.571429$$

The overall scalar for the instance in relation to this region will be the product of the two axis scalars: 0.5 × 0.571429 = 0.285714.

The detailed algorithm for calculating the interpolated value for a given target item and for a given instance is as follows.

- Let instanceCoords be the normalized instance coordinate n-tuple for the instance, with axis elements instanceCoords[i].
- Let Regions be the set of regions for which associated deltas are provided for the given item, and let R be a region within that set.
- Let startCoords, peakCoords and endCoords be the start, peak and end n-tuples for some specified region, R. Let startCoords[i], etc. be coordinate values for a given axis.
- Let AS be a per-axis scalar, and let S be an overall scalar for a given region.
- Let d be the delta value in the variation data associated with a given region, and let scaledDelta be the scaled delta for a given region and instance.
- Let netAdjustment be the accumulated adjustment for the given item.
- Let defaultValue be the default value of the item specified in the font, and let interpolatedValue be the interpolated value of the item for a given instance.

The following pseudo-code provides a specification of the interpolation algorithm:

```
netAdjustment = 0; /* initialize the accumulated adjustment to zero */

(for each R in Regions) /* For each region, calculate a scalar S */
{
    S = 1; /* initialize the overall scalar for the region to one */

    /* for each axis, calculate a per-axis scalar AS */
    (for i = 0; i < axisCount; i++)
    {
        /* If a region definition is not valid in relation to some axis,
           then ignore the axis. For a region to be valid in relation to a
           given axis, it must have a peak that is between the start and
           end values, and the start and end values cannot have different
           signs if the peak is non-zero. (Start and end can have different
           signs if the peak is zero, however: this can be used if an axis is
           to be ignored in the scalar calculation.) */
        if (startCoords[i] > peakCoords[i] || peakCoords[i] > endCoords[i])
            AS = 1;
        else if (startCoords[i] < 0 && endCoords[i] > 0 && peakCoords[i] != 0)
            AS = 1;

        /* Note: for remaining cases, start, peak and end will all be <= 0 or
           will all be >= 0, or else peak will be == 0. */

        /* If the peak is zero for some axis, then ignore the axis. */
        else if (peakCoords[i] == 0)
            AS = 1;
```

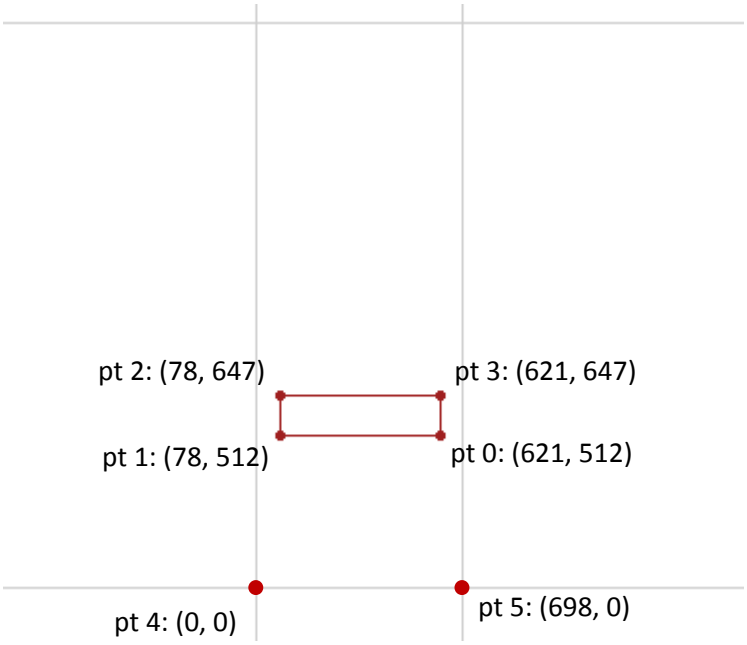
```
            /* If the instance coordinate is out of range for some axis, then the
               region and its associated deltas are not applicable. */
            else if (instanceCoords[i] < startCoords[i]
                     || instanceCoords[i] > endCoords[i])
                AS = 0;

            /* The region is applicable: calculate a per-axis scalar as a proportion
               of the proximity of the instance to the peak within the region. */
            else
            {
                if (instanceCoords[i] == peakCoords[i])
                    AS = 1;
                else if (instanceCoords[i] < peakCoords[i])
                {
                    AS = (instanceCoords[i] - startCoords[i])
                         / (peakCoords[i] - startCoords[i]);
                }
                else /* instanceCoords[i] > peakCoords[i] */
                {
                    AS = (endCoords[i] - instanceCoords[i])
                         / (endCoords[i] - peakCoords[i]);
                }
            }

            /* The overall scalar is the product of all per-axis scalars.
               Note: the axis scalar and the overall scalar will always be
               >= 0 and <= 1. */
            S = S * AS;

        } /* per-axis loop */

        /* get the scaled delta for this region */
        scaledDelta = S * delta;

        /* accumulate the adjustments from each region */
        netAdjustment = netAdjustment + scaledDelta;

    } /* per-region loop */

    /* apply the accumulated adjustment to the default to derive the interpolated value */
    interpolatedValue = defaultValue + netAdjustment;
```

7.1.8 Interpolation example

The following example illustrates the interpolation process for a particular instance. This example is based on glyph 45 of the Skia font, which is the glyph for the hyphen-minus character.

NOTE The Skia font is included in Apple's OSX platform. At the time of publication of the ISO/IEC 14496-22 OFF (4th edition) specification, existing versions of the Skia font do not conform to the OFF specification as a whole, but the implementation of variation data in the 'gvar' table, which is what is illustrated here, does conform.

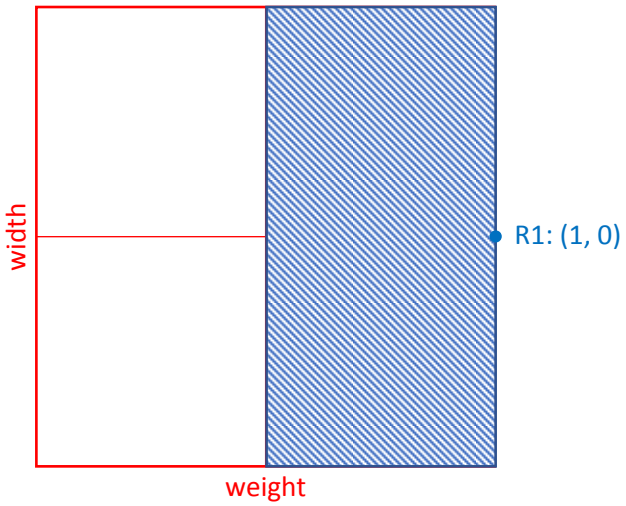
The glyph entry in the ‘glyf’ table has one contour with four points. Based on values for glyph 45 in the ‘hmtx’ table, “phantom” points are inferred in the rasterizer to represent left and right side-bearings. (For this example, horizontal layout is assumed, and so top and bottom phantom points are ignored.) These phantom points are at (0, 0) and (698, 0). Thus, there are six points requiring interpolation.



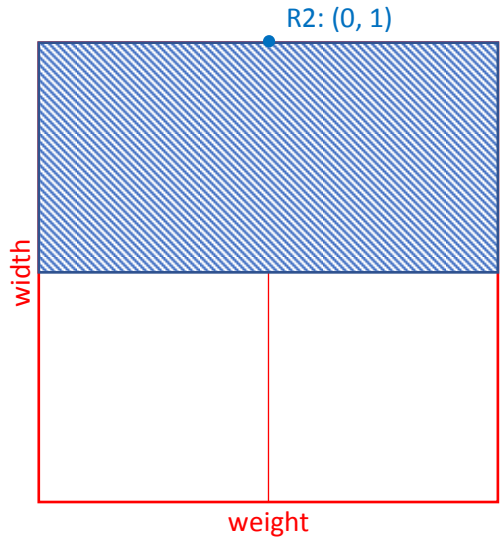
The Skia font has weight and width axes. The variation data for glyph 45 in the 'gvar' table has deltas associated with 8 regions within the weight-width variation space. Three of these will be considered, and will be referred to as R1, R2 and R3. Each of these is a non-intermediate region, and so is defined using a single n-tuple. The n-tuples for each are as follows:

Region	(weight, width)
R1	(1, 0)
R2	(0, 1)
R3	(1, 1)

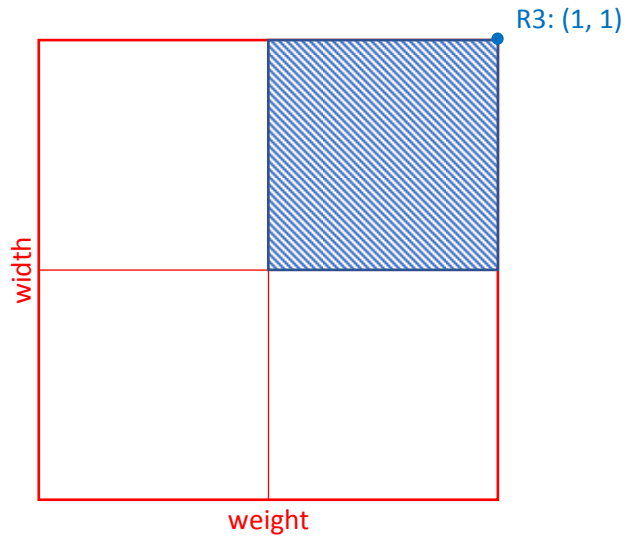
The following figures illustrate the range of applicability over the variation space for each of these regions:



R1 has a zero coordinate value for the width axis, which means that changes in width for the variation instance have no effect on the scalar calculations for this region.



R2 has a zero coordinate value for the weight axis, which means that changes in weight for the variation instance have no effect on the scalar calculations for this region.



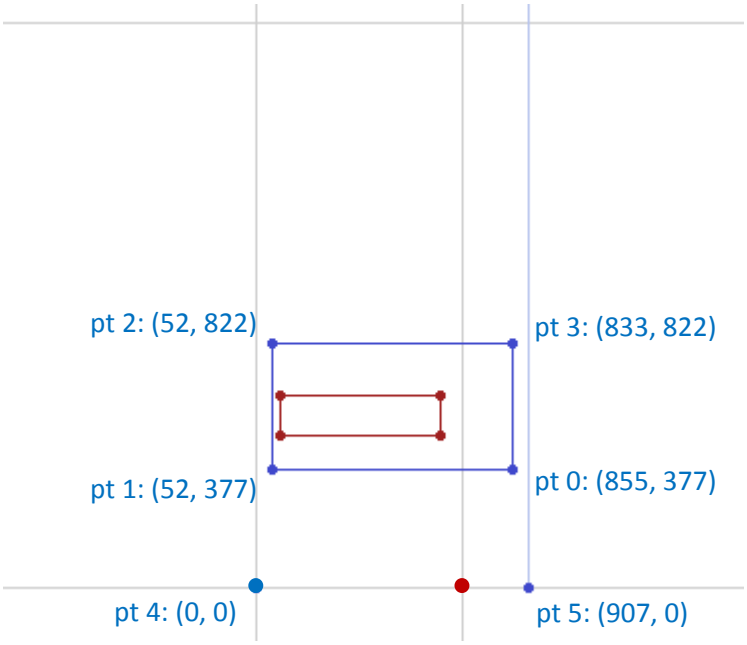
R3 has non-zero coordinate values for both weight and width axes, which means that changes for the variation instance in either weight or width will affect the scalar calculations for this region.

Now consider the delta values specified in the font for each point in association with these three regions. X and Y deltas are specified for each point.

R1 has the following associated deltas:

	pt 0	pt 1	pt 2	pt 3	pt 4	pt 5
X	234	-26	-26	234	0	209
Y	-135	-135	175	175	0	0

Applying these deltas to the original point positions, the maximal effect of deltas associated with R1 would be to modify the outline as follows:

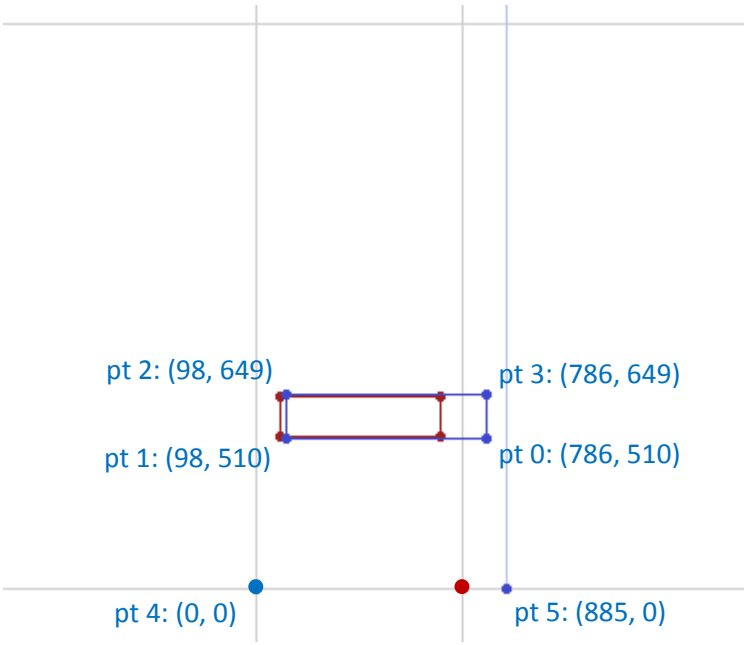


For a variation instance of (1, 0) (heaviest weight, default width), the scalars for other regions would be zero, and so this would be the resulting glyph outline for that instance. Reducing the weight value of the instance would attenuate the extent of the change, with the outline interpolated in between the original outline and this maximal modification of the outline.

Now consider R2: it has the following deltas associated with it:

	pt 0	pt 1	pt 2	pt 3	pt 4	pt 5
X	165	20	20	165	0	187
Y	-2	-2	2	2	0	0

Applying these deltas to the original point positions, the maximal effect of deltas associated with R2 would be to modify the outline as follows:



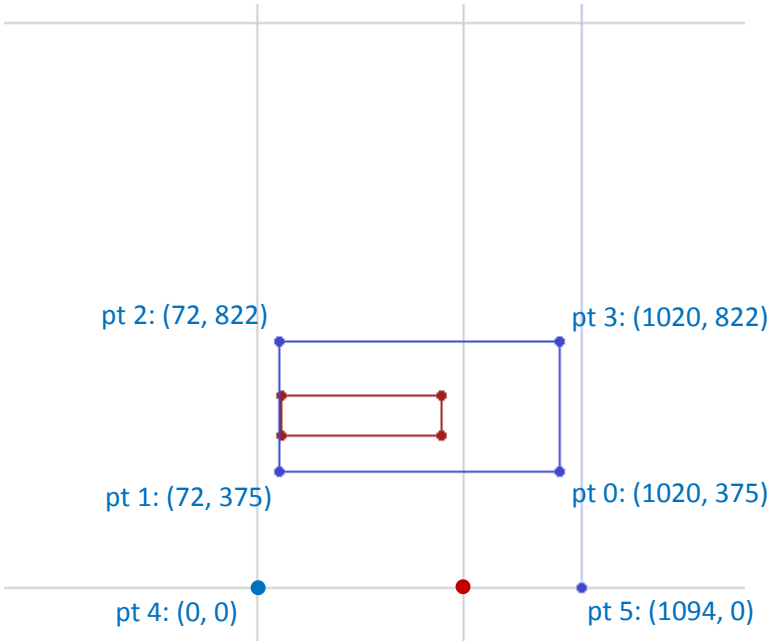
For a variation instance of (0, 1) (regular weight, widest width), the scalars for other regions would be zero, and so this would be the resulting, interpolated glyph outline for that instance.

Now consider R3: it has the following associated deltas:

	pt 0	pt 1	pt 2	pt 3	pt 4	pt 5
X	0	0	0	0	0	0
Y	0	0	0	0	0	0

Since all of the delta values are zero, the data associated with this region has no effect at all on the glyph outline. (In fact, this data is superfluous.)

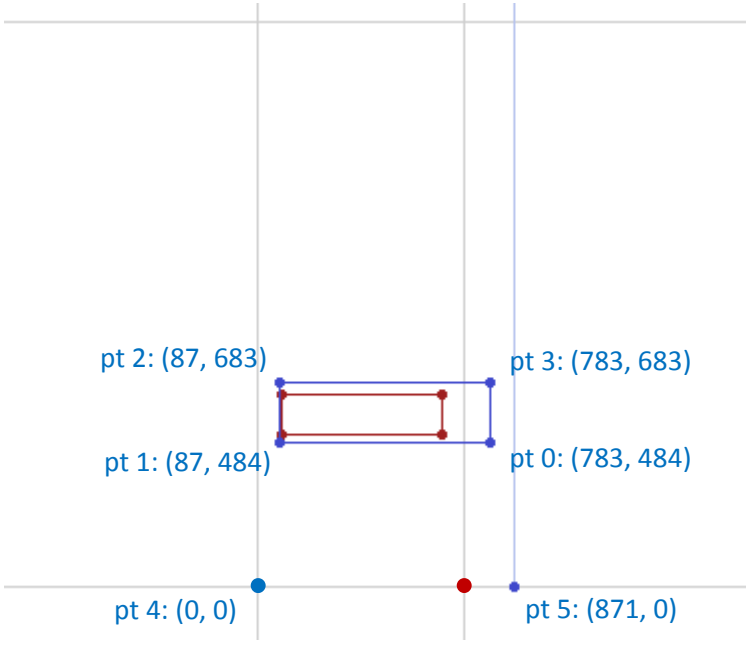
Now, consider a variation instance of (1, 1) (heaviest weight, widest width). All three regions, R1, R2 and R3, are applicable for this instance. As noted, the variation data associated with R3 will have no effect on the glyph. But the data for regions R1 and R2 would also be applicable for this instance, and their maximal effects would be combined. That is, the X and Y deltas for each point from data associated with both R1 and R2 would be applied to the point X and Y coordinates. This would result in the glyph outline being modified as follows:



For other variation instances with weight > 0 and < 1 and with width > 0 and < 1, the data for regions R1 and R2 would both be applied, but scalars for the two regions would vary, resulting in different proportional effects on the outline of the data for each region. For example, consider a variation instance with coordinates (0.2, 0.7) — a slight weight increase and a large width increase. The region scalars for R1 and R2 would be 0.2 and 0.7. Each of these would be applied to the deltas for each region, and the scaled delta values for a given point combined:

	pt 0	pt 1	pt 2	pt 3	pt 4	pt 5
X	0.2 × 234 + 0.7 × 165 = 162.3	0.2 × -26 + 0.7 × 20 = 8.8	0.2 × -26 + 0.7 × 20 = 8.8	0.2 × 234 + 0.7 × 165 = 162.3	0.2 × 0 + 0.7 × 0 = 0	0.2 × 209 + 0.7 × 187 = 172.7
Y	0.2 × -135 + 0.7 × -2 = -134.8	0.2 × -135 + 0.7 × -2 = -134.8	0.2 × 175 + 0.7 × 2 = 36.4	0.2 × 175 + 0.7 × 2 = 36.4	0.2 × 0 + 0.7 × 0 = 0	0.2 × 0 + 0.7 × = 0

This would result in the glyph outline being modified as follows:



7.1.9 Dynamic generation of static instance fonts

In certain application workflows, it may be necessary to dynamically generate a static font resource for a particular instance – that is, conventional, non-variation font tables that use interpolated values for a particular instance. This may be needed in order to provide font data to legacy software or data formats that do not understand or support variable fonts, such as legacy printer drivers, or PDF or XPS files with embedded font data.

For example, it may be necessary to process ‘glyf’ and ‘gvar’ tables in a variable font to generate a new ‘glyf’ table that has interpolated outline data for a particular instance; or to process ‘hhea’ and ‘MVAR’ tables to generate a new ‘hhea’ table with data for a particular instance.

Different application scenarios may require more- or less-complete font data, entailing different sets of non-variations-specific font tables that need to be generated. No minimal requirements are specified here. The following points should be noted, however:

- Some scenarios may require use of a PostScript name (name ID 6) in instance font data, with distinct names for each instance that is used. An Adobe technical note provides a specification for Postscript name generation that can be used for instance fonts derived from variable fonts. See Adobe Technical Note #5902: “PostScript Name Generation for Variation Fonts” [27].
- For a variable font with CFF2 outlines, some workflows – for example, printing – may require an instance font to be generated with a ‘CFF ’ table. In such cases, if the variable font has more than one Font DICT in the FDArray, then a CID-keyed CFF font should be generated, with an ROS of “Adobe-Identity-0”. If the variable font has one Font DICT in the FDArray, then a name-keyed CFF font can be generated if glyph names are supplied in the ‘post’ table (some legacy workflows look to a glyph name for semantics); otherwise, a CID-keyed CFF can be generated as above. Converting CFF2 CharStrings to Type2 CharStrings would involve re-optimizing the CharString arguments and operators to avoid exceeding the maximum permitted stack depth. Most of the CFF fields removed from the CFF2 specification can be omitted, so that they will inherit the CFF default values. The source information for filling the rest of the fields is documented in "CFF2 changes from CFF 1.0" [28].
- A variable font that has a ‘glyf’ table may utilize the GET VARIATION instruction to provide current variation axis coordinates to the glyph program. In scenarios that require dynamic generation of instance font data, it should be assumed that this instruction will not be supported. In the process for generating an interpolated ‘glyf’ table, special treatment of the GETVARIATION instruction will be needed to ensure that the program gets appropriate axis coordinate values for the given instance. For details, see TrueType InstructionSet [[# Get Variation]].

7.2 Font variations common table formats

OFF font variations allow a font designer to incorporate multiple faces within a font family into a single font resource. Variable fonts allow for continuous variation along one or more design axes, such as weight or width. Applications can select arbitrary variation instances in a font's design variation space to format text. The font has default values for various data items, such as the X and Y coordinates of glyph outline points. Layout and rendering processes combine these default values with variation data to interpolate new values appropriate to the instance.

An overview of font variations and a specification of the algorithm for interpolating variation instance values is provided in subclause 7.1, which should be read first. This subclause documents the formats for variation data that are used in various font tables, such as the 'gvar' or 'MVAR' tables.

7.2.1 Overview

A font has many different data items found in several different font tables that provide values that are specific to a particular font face. Examples include glyph-specific values, such as the positions of glyph outline points and glyph advance widths, and face-wide values, such as a sub-family name, a weight class, or ascender and descender values. In a variable font, most or all of these values may need to vary for different variation instances. When an application selects a particular variation instance within the font's variation space, new values for such items appropriate to that instance need to be derived. This is done using delta adjustment values that are specified for a given font data item and a particular region within the variation space.

For example, the OS/2 table of a font may provide a default sxHeight value of 970. The 'MVAR' table might provide a delta value of +50 that is used for weight-axis values from the default to the heaviest-supported weight. For a particular instance, the interpolation process might scale that delta with a scalar co-efficient of 0.4, deriving an instance sxHeight value of 990.

These concepts and the interpolation algorithm for deriving instance values are described in detail in the Font variations overview.

The variation data for a font consists of a number of delta adjustment values. Each individual delta applies to a particular, target data item – the X coordinate of a particular point of a particular glyph, or the font's sTypoAscender – and is also associated with a specific region within the font's design variation space over which it is applicable. Thus, a given delta is logically keyed by the target data item and the applicable region.

A variable font includes a large number of deltas. At the highest level, deltas are organized into collections for different target item sets:

- Deltas for positions of points of a 'glyf' table are stored in a 'gvar' table.
- Deltas for positions of points of a 'CFF2' table are stored within the 'CFF2' table.
- Deltas for CVT values are stored in a 'cvar' table.
- Deltas for glyph metrics in an 'hmtx' table are stored in an 'HVAR' table; and deltas for glyph metrics in a 'vmtx' or 'VORG' table are stored in a 'VVAR' table.
- Deltas for anchor positions in 'GPOS' lookups and other items used in 'GDEF', 'GPOS' or 'JSTF' tables are stored within variation data contained in the 'GDEF' table.
- Deltas for baseline metrics in a 'BASE' table are stored within the 'BASE' table.
- Deltas for font-wide metrics and other items from the 'OS/2', 'hhea', 'gasp' or other tables are stored in an 'MVAR' table.

In a variable font, the largest group of deltas are for the positions of glyph outline points. For TrueType outlines in a 'glyf' table, the deltas are stored within the 'gvar' table, with a second level of organization is to group deltas by glyph ID. See the [['gvar' table specification]] for details.

Below these higher levels of organization, most variation data is organized in one of two ways. (Variation data for CFF2 outlines is a partial exception – see below.)

- Organize sets of deltas for several target items into groupings by the variation-space region over which they apply. Since regions are defined using n-tuples (or "tuples"), such data sets will be referred to as *tuple variation stores*.
- Organize sets of deltas associated with different regions into groupings by the target item to which they apply. Such data sets will be referred to as *item variation stores*.

The two formats have different ways of representing n-tuples that define regions of applicability, and different ways of associating deltas with target font-data items. The tuple variation store format is optimized for compact representation of glyph outline variation data that is all processed for a given variation instance. The item variation store format, on the other hand, is designed to allow direct access to variation data for arbitrary target items, allowing more efficient processing in contexts that do not require interpolated values for all items to be computed. (Additional details are provided below.) The 'gvar' and 'cvar' table use the tuple variation store format, while variation data in most other tables, including the 'MVAR', 'HVAR' and 'GDEF' tables, use item variation store formats.

Variation data for CFF2 outlines are handled slightly differently than other cases. The deltas for glyph outline descriptions are interleaved directly within the outline descriptions in the Compact Font Format 2 (‘CFF2’) table. The sets of regions which are associated with the delta sets are defined in an item variation store, contained as a subtable within the CFF2 table.

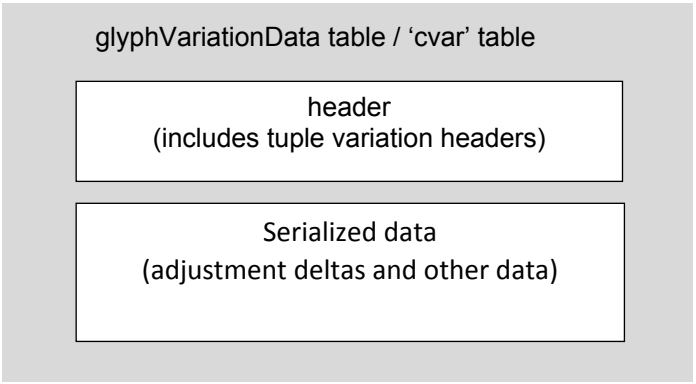
7.2.2 Tuple variation store

Tuple variation stores are used in the ‘gvar’ and ‘cvar’ tables, and organize sets of variation data into groupings, each of which is associated with a particular region of applicability within the variation space. Within the ‘gvar’ table, there is a separate variation store for each glyph. Within the ‘cvar’ table, there is one variation store providing variations for all CVT values.

There is a minor difference in the top-level structure of the store in these two contexts. Within the ‘cvar’ table, it is the entire ‘cvar’ table that comprises the specific variation store format, with a header that begins with major/minor version fields. The specific variation store format for glyph-specific data within the ‘gvar’ table is the *GlyphVariationData* table (one per glyph ID), which does not include any version fields. In other respects, the ‘cvar’ table and GlyphVariationData table formats are the same. There is also a minor difference in certain data that can occur in a GlyphVariationData table versus a ‘cvar’ table. Differences between the ‘gvar’ and ‘cvar’ tables will be summarized later in this subclause.

In terms of logical information content, the GlyphVariationData and ‘cvar’ tables consist of a set of logical, tuple variation data tables, each for a particular region of the variation space. In physical layout, however, the logical tuple variation tables are divided into separate parts that get stored separately: a header portion, and a serialized-data portion.

In terms of overall structure, the GlyphVariationData table and the ‘cvar’ table each begin with a header, which is followed by serialized data. The header includes an array with all of the tuple variation headers. The serialized data include deltas and other data that will be explained below.



7.2.2.1 Tuple records

The tuple variation store formats make reference to regions within the font’s variation space using tuple records. These references identify positions in terms of normalized coordinates, which use F2DOT14 values.

Tuple Record (F2DOT14)

Type	Name	Description
F2DOT14	coordinates[axisCount]	Coordinate array specifying a position within the font's variation space. The number of elements must match the axisCount specified in the 'fvar' table.

7.2.2.2 Tuple variation store header

The two variants of a tuple variation store header, the GlyphVariationData table header and the 'cvar' header, are only slightly different. The formats of each are as follows:

GlyphVariationData Header:

Type	Name	Description
uint16	tupleVariationCount	A packed field. The high 4 bits are flags (see below), and the low 12 bits are the number of tuple variation tables for this glyph. The count can be any number between 1 and 4095.
Offset16	dataOffset	Offset from the start of the GlyphVariationData table to the serialized data
TupleVariationHeader	tupleVariationHeaders [tupleVariationCount]	Array of tuple variation headers.

'cvar' table header

Type	Name	Description
uint16	majorVersion	Major version number of the CVT variations table – set to 1.
uint16	minorVersion	Minor version number of the CVT variations table – set to 0.
uint16	tupleVariationCount	A packed field. The high 4 bits are flags (see below), and the low 12 bits are the number of tuple variation tables for this glyph. The count can be any number between 1 and 4095.
Offset16	dataOffset	Offset from the start of the 'cvar' table to the serialized data.
TupleVariationHeader	tupleVariationHeaders [tupleVariationCount]	Array of tuple variation headers.

The tupleVariationCount field contains a packed value that includes flags and the number of logical tuple variation tables — which is also the number of physical tuple variation headers. The format of the tupleVariationCount value is as follows:

Value	Name	Description
0x8000	SHARED_POINT_NUMBERS	Flag indicating that some or all tuple variation tables reference a shared set of "point" numbers. These shared numbers are represented as packed point number data at the start of the serialized data.
0x7000	Reserved	Reserved for future use.
0x0FFF	COUNT_MASK	Mask for the low bits to give the number of tuple variation tables.

If the sharedPointNumbers flag is set, then the serialized data following the header begins with packed “point” number data. In the context of a GlyphVariationData table within the ‘gvar’ table, these identify outline point numbers for which deltas are explicitly provided. In the context of the ‘cvar’ table, these are interpreted as CVT indices rather than point indices. The format of packed point number data is described below.

7.2.2.3 TupleVariationHeader

The GlyphVariationData and ‘cvar’ header formats include an array of tuple variation headers. The TupleVariationHeader format is as follows:

TupleVariationHeader

Type	Name	Description
uint16	variationDataSize	The size in bytes of the serialized data for this tuple variation table.
uint16	tupleIndex	A packed field. The high 4 bits are flags (see below). The low 12 bits are an index into a shared tuple records array.
Tuple	peakTuple	Peak tuple record for this tuple variation table — optional, determined by flags in the tupleIndex value. NOTE this must always be included in the ‘cvar’ table.
Tuple	intermediateStartTuple	Intermediate start tuple record for this tuple variation table — optional, determined by flags in the tupleIndex value.
Tuple	intermediateEndTuple	Intermediate end tuple record for this tuple variation table — optional, determined by flags in the tupleIndex value.

Note that the size of the TupleVariationHeader is variable, depending on whether peak or intermediate tuple records are included. (See below for more information.)

The variationDataSize value indicates the size of serialized data for the given tuple variation table that is contained in the serialized data. It does not include the size of the TupleVariationHeader.

Every tuple variation table has an associated peak tuple record. Most tuple variation tables use non-intermediate regions, and so require only the peak tuple record to define the region. In the ‘cvar’ table, there is only one variation store, and so any given region will only need to be referenced once. Within the ‘gvar’ table, however, there is a GlyphVariationData table for each glyph ID, and so any region may be referenced numerous times; in fact, most regions will be referenced within the GlyphVariationData tables for most glyphs. To provide a more efficient representation, the tuple variation store formats allow for an array of tuple records, stored outside the tuple variation store structures that can be shared across many tuple variation stores. This is used only within the ‘gvar’ table; it is not needed or supported in the ‘cvar’ table. The formats alternately allow for a peak tuple record that is non-shared, specific to the given tuple variation table, to be embedded directly within a TupleVariationHeader. This is optional within the ‘gvar’ table, but required in the ‘cvar’ table, which does not use shared peak tuple records.

See the glyph variations (‘gvar’) table description for details on the representation of shared tuple records within that table.

The tupleIndex field contains a packed value that includes flags and an index into a shared tuple records array (not used in the ‘cvar’ table). The format of the tupleIndex field is as follows:

tupleIndex format

Mask	Name	Description
0x8000	EMBEDDED_PEAK_TUPLE	Flag indicating that this tuple variation header includes an embedded peak tuple record, immediately after the tupleIndex field. If set, the low 12 bits of the tupleIndex value are ignored. NOTE this must always be set within the 'cvar' table.
0x4000	INTERMEDIATE_REGION	Flag indicating that this tuple variation table applies to an intermediate region within the variation space. If set, the header includes the two intermediate-region, start and end tuple records, immediately after the peak tuple record (if present).
0x2000	PRIVATE_POINT_NUMBERS	Flag indicating that the serialized data for this tuple variation table includes packed "point" number data. If set, this tuple variation table uses that number data; if clear, this tuple variation table uses shared number data found at the start of the serialized data for this glyph variation data or 'cvar' table.
0x1000	Reserved	Flag reserved for future use.
0x0FFF	TUPLE_INDEX_MASK	Mask for the low 12 bits to give the shared tuple records index.

Note that the intermediateRegion flag is independent of the embeddedPeakTuple flag or the shared tuple records index. Every tuple variation table has a peak n-tuple indicated either by an embedded tuple record (always true in the 'cvar' table) or by an index into a shared tuple records array (only in the 'gvar' table). An intermediate-region tuple variation table additionally has start and end n-tuples that also get used in the interpolation process; these are always represented using embedded tuple records.

Also note that the privatePointNumbers flag is independent of the sharedPointNumbers flag in the tupleVariationCount field of the GlyphVariationData or 'cvar' header. A GlyphVariationData or 'cvar' table may have shared point number data used by multiple tuple variation tables, but any given tuple variation table may have private point number data that it uses instead.

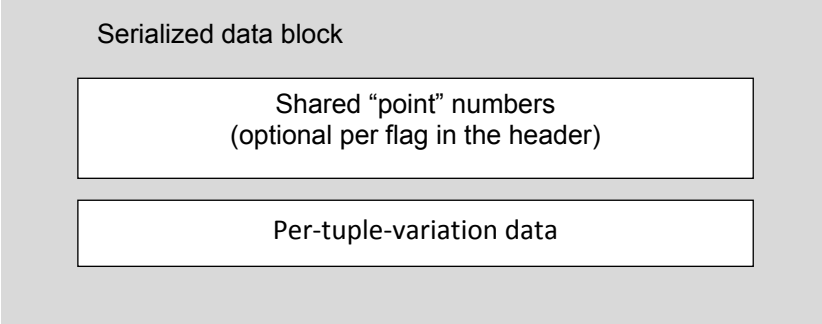
As noted, the size of tuple variation headers is variable. The next TupleVariationHeader can be calculated as follows:

```
const TupleVariationHeader*
NextHeader( const TupleVariationHeader* currentHeader, int axisCount )
{
    int bump = 2 * sizeof( uint16 );
    int tupleIndex = currentHeader->tupleIndex;
    if ( tupleIndex & embeddedPeakTuple )
        bump += axisCount * sizeof( F2DOT14 );
    if ( tupleIndex & intermediateRegion )
        bump += 2 * axisCount * sizeof( F2DOT14 );
    return (const TupleVariationHeader*)((char*)currentHeader + bump);
}
```

7.2.2.4 Serialized data

After the GlyphVariationData or 'cvar' header (including the TupleVariationHeader array) is a block of serialized data. The offset to this block of data is provided in the header.

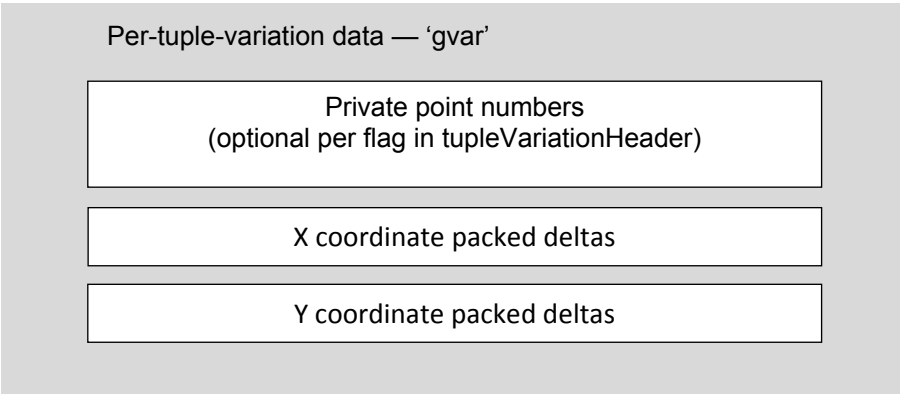
The serialized data block begins with shared "point" number data, followed by the variation data for the tuple variation tables. The shared point number data is optional: it is present if the corresponding flag is set in the tupleVariationCount field of the header. If present, the shared number data is represented as packed point numbers, described below.



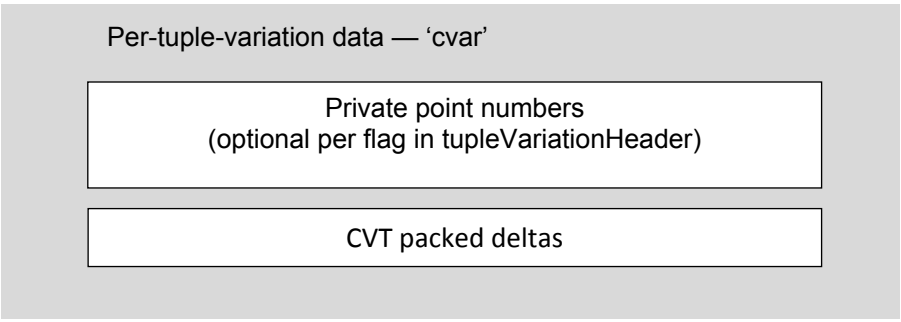
The remaining data contains runs of data specific to individual tuple variation tables, in order of the tuple variation headers. Each TupleVariationHeader indicates the data size for the corresponding run of data for that tuple variation table.

The per-tuple-variation-table data optionally begins with private “point” numbers, present if the privatePointNumbers flag is set in the tupleIndex field of the TupleVariationHeader. Private point numbers are represented as packed point numbers, described below.

After the private point number data (if present), the tuple variation data will include packed delta data. The format for packed deltas is given below. Within the ‘gvar’ table, there are packed deltas for X coordinates, followed by packed deltas for Y coordinates.



Within the ‘cvar’ table, there is one set of packed deltas.



The data size indicated in the TupleVariationHeader includes the size of the private point number data, if present, plus the size of the packed deltas.

7.2.2.5 Packed “point” numbers

Tuple variation data specify deltas to be applied to specific items: X and Y coordinates for glyph outline points within the ‘gvar’ table, and CVT values in the ‘cvar’ table. For a given glyph, deltas may be provided for any or all of a glyph’s points, including “phantom” points generated within the rasterizer that represent glyph side bearing points. (See "Instructing TrueType Glyphs" [24] for more background on phantom points.) Similarly,

within the ‘cvar’ table, deltas may be provided for any or all CVTs. The set of glyph points or CVTs for which deltas are provided is specified by packed point numbers.

NOTE If a glyph is a composite glyph, then “point” numbers are interpreted as indices for the components that make up the composite glyph. See the ‘gvar’ table chapter for complete details. Likewise, in the context of the ‘cvar’ table, “point” numbers are indices for CVT entries.

NOTE Within the ‘gvar’ table, if deltas are not provided explicitly for some points, then inferred delta values may need to be calculated – see the ‘gvar’ table chapter for details. This does not apply to the ‘cvar’ table, however: if deltas are not provided for some CVT values, then no adjustments are made to those CVTs in connection with the particular tuple variation table.

Packed point numbers are stored as a count followed by one or more runs of point number data.

The count may be stored in one or two bytes. After reading the first byte, the need for a second byte can be determined. The count bytes are processed as follows:

- If the first byte is 0, then a second count byte is not used. This value has a special meaning: the tuple variation data provides deltas for all glyph points (including the “phantom” points), or for all CVTs.
- If the first byte is non-zero and the high bit is clear (value is 1 to 127), then a second count byte is not used. The point count is equal to the value of the first byte.
- If the high bit of the first byte is set, then a second byte is used. The count is read from interpreting the two bytes as a big-endian word, with the high-order bit masked out.

Thus, if the count fits in 7 bits, it is stored in a single byte, with the value 0 having a special interpretation. If the count does not fit in 7 bits, then the count is stored in the first two bytes with the high bit of the first byte set as a flag that is not part of the count – the count uses 15 bits.

For example, a count of 0x00 indicates that deltas are provided for all point numbers / all CVTs, with no additional point number data required; a count of 0x32 indicates that there are a total of 50 point numbers specified; a count of 0x81 0x22 indicates that there are a total of 290 (= 0x0122) point numbers specified.

Point number data runs follow after the count. Each data run begins with a control byte that specifies the number of point numbers defined in the run, and a flag bit indicating the format of the run data. The control byte’s high bit specifies whether the run is represented in bytes or words. The low 7 bits specify the number of elements in the run minus 1. The format of the control byte is as follows:

Mask	Name	Description
0x80	POINTS_ARE_WORDS	Flag indicating the data type used for point numbers in this run. If set, the point numbers are stored as unsigned words (uint16); if clear, the point numbers are stored as unsigned bytes (uint8).
0x7F	POINT_RUN_COUNT_MASK	Mask for the low 7 bits of the control byte to give the number of point number elements, minus 1.

For example, a control byte of 0x02 indicates that the run has three elements represented as uint8s; a control byte of 0xD4 indicates that the run has 0x54 + 1 = 85 elements represented as uint16s.

In the first point run, the first point number is represented directly (that is, as a difference from zero). Each subsequent point number in that run is stored as the difference between it and the previous point number. In subsequent runs, all elements, including the first, represent a difference from the last point number.

Since the values in the packed data are all unsigned, point numbers will be given in increasing order. Since the packed representation can include zero values, it is possible for a given point number to be repeated in the derived point number list. If that case, there will be multiple delta values in the deltas data associated with that point number. All of these deltas must be applied cumulatively to the given point.

7.2.2.6 Packed deltas

Tuple variation data specify deltas to be applied to glyph point coordinates or to CVT values. As in the case of point number data, deltas are stored in a packed format.

Packed delta data does not include the total number of delta values within the data. Logically, there are deltas for every point number or CVT index specified in the point-number data. Thus, the count of logical deltas is equal to the count of point numbers specified for that tuple variation table. But since the deltas are represented in a packed format, the actual count of stored values is typically less than the logical count. The data is read until the expected logic count of deltas is obtained.

NOTE In the 'gvar' table, there will be two logical deltas for each point number: one that applies to the X coordinate, and one that applies to the Y coordinate. Therefore, the total logical delta count is two times the point number count. The packed deltas are arranged with all of the deltas for X coordinates first, followed by the deltas for Y coordinates.

Packed deltas are stored as a series of runs. Each delta run consists of a control byte followed by the actual delta values of that run. The control byte is a packed value with flags in the high two bits and a count in the low six bits. The flags specify the data size of the delta values in the run. The format of the control byte is as follows:

Mask	Name	Description
0x80	DELTAS_ARE_ZERO	Flag indicating that this run contains no data (no explicit delta values are stored), and that all of the deltas for this run are zero.
0x40	DELTAS_ARE_WORDS	Flag indicating the data type for delta values in the run. If set, the run contains 16-bit signed deltas (int16); if clear, the run contains 8-bit signed deltas (int8).
0x3F	DELTA_RUN_COUNT_MASK	Mask for the low 6 bits to provide the number of delta values in the run, minus one.

For example, a control byte of 0x03 indicates that there are four 8-bit signed delta values following the control byte; a control byte of 0x40 indicates that there is one 16-bit signed delta value following the control byte; a control byte of 0x94 indicates that there is no additional data for this run, and that the run represents a sequence of 0x14 + 1 = 21 deltas equal to zero.

The following is an example of a block of packed delta data:

03 0A 97 00 C6 87 41 10 22 FB 34

This data has three runs: a run of four 8-bit values, a run interpreted as eight zeroes, and a run of two 16-bit values:

Run 1: 03 0A 97 00 C6

Run 2: 87

Run 3: 41 10 22 FB 34

This packed data would represent the following logical sequence of delta values:

10, -105, 0, -58, 0, 0, 0, 0, 0, 0, 0, 0, 4130, -1228

7.2.2.7 Processing tuple variation store data

When a particular variation instance has been selected, an application needs to process the variation store data to derive interpolated values for that instance – interpolated grid coordinates for outline points, or interpolated CVT values. In the case of the 'gvar' table, this will be done glyph-by-glyph as needed. The application can process the TupleVariationHeaders to filter the tuple variation tables that are applicable for the current instance, or to calculate a scalar for each tuple variation table directly. Scalars can then be applied to deltas in each tuple variation table, and the net adjustments applied to the target items.

NOTE In the 'cvar' table, there is a logical delta for each CVT index given in the packed point number data. In the 'gvar' table, there are two logical deltas for each point number: one for the point's X coordinate, and one for the X coordinate. The delta data is organized with all of the deltas for X coordinates first, followed by deltas for Y coordinates.

NOTE In the ‘gvar’ table, if the data for a given glyph lists point numbers for some points in a contour but not others, then delta values for the omitted point numbers must be inferred. See the ‘gvar’ table description for details.

For details on determining applicability of a given tuple variation table, and on calculation of scalars and net adjustments to target items, see OFF Font variations overview.

Because point number and delta data are stored in a packed representation, the data shall be processed from the start in order to determine the presence of any particular point number, or to retrieve the delta for a particular item. For this reason, the format is best suited to processing all of the data in a given tuple variation table at once rather than processing data for individual target items. In the case of glyph outlines, this is reasonable since there is no common application scenario for interpolating an adjusted position of a single outline point.

The “phantom” points, which provide side-bearing and advance width information, are a possible exception to that generalization, however. (See "Instructing TrueType Glyphs" [24] for more background on phantom points.) In particular, some text-layout operations require glyph metrics (advance widths or side bearings) without necessarily requiring glyph outline data. Yet the tuple variation store formats used in the ‘gvar’ table require that interpolated outlines be computed in order to obtain the interpolated glyph metrics. The ‘HVAR’ table and ‘VVAR’ table provide an alternative way to represent horizontal and vertical glyph metric variation data, and these use the item variation store format which is specifically designed to be suitable for processing data for particular target items.

7.2.2.8 Differences between ‘gvar’ and ‘cvar’ tables

The following is a summary of key differences between tuple variation stores in the ‘gvar’ and ‘cvar’ tables.

- The ‘gvar’ table is a parent table for tuple variation stores, and contains one tuple variation store (the glyph variation data table) for each glyph ID. In contrast, the entire ‘cvar’ table is comprised of a single, slightly-extended (with version fields) tuple variation store.
- Because the ‘gvar’ table contains multiple tuple variation stores, sharing of data between tuple variation stores is possible, and is used for shared tuple records. Because the ‘cvar’ table has a single tuple variation store, no possibility of shared data arises.
- The tupleIndex field of TupleVariationHeader structures within a tuple variation store includes a flag that indicates whether the structure instance includes an embedded peak tuple record. In the ‘gvar’ table, this is optional. In the ‘cvar’ table, it is mandatory.
- The serialized data includes packed “point” numbers. In the ‘gvar’ table, these refer to glyph contour point numbers or, in the case of a composite glyph, to component indices. In the context of the ‘cvar’ table, these are indices for CVT entries.
- In the ‘gvar’ table, point numbers cover the points or components defined in a ‘glyf’ entry plus four additional “phantom” points that represent the glyph’s horizontal and vertical advance and side bearings. (See "Instructing TrueType Glyphs" [24] for more background on phantom points.) The last four point numbers for any glyph, including composite glyphs, are for the phantom points.
- In the ‘gvar’ table, if deltas are not provided for some points and the point indices are not represented in the point number data, then interpolated deltas for those points will in some cases be inferred. This is not done in the ‘cvar’ table, however.
- In the ‘gvar’ table, the serialized data for a given region has two logical deltas for each point number: one for the X coordinate, and one for the Y coordinate. Hence the total number of deltas is twice the count of control points. In the ‘cvar’ table, however, there is only one delta for each point number.

7.2.3 Item variation stores

Item variation stores are used for most variation data other than that used for TrueType glyph outlines, including the variation data in ‘MVAR’, ‘HVAR’ and ‘GDEF’ tables.

NOTE For CFF2 glyph outlines, delta values are interleaved directly within glyph outline description in the ‘CFF2’ table. The sets of regions which are associated with the delta sets are defined in an item variation store, contained as a subtable within the CFF2 table. See the ‘CFF2’ table description for additional details.

The item variation store formats organize sets of variation data into groupings by the target items. This makes the formats well-suited to computing interpolated instance values for particular font data items. This is useful

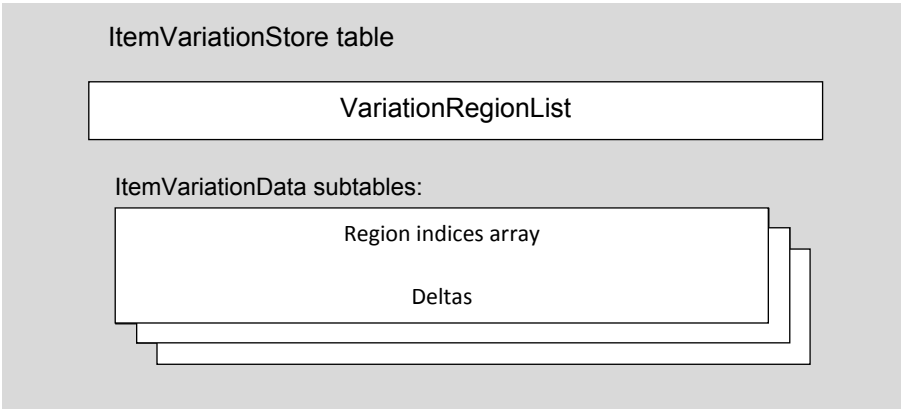
for certain text layout operations in which only certain data items are required, such as the advance widths of specific glyphs or anchor positions used in specific 'GPOS' lookup tables.

The different tables that use item variation stores have their own top-level formats. Each will include an offset to an itemVariationStore table, containing the variation data. This chapter describes the shared formats: the itemVariationStore and its component structures.

Variation data is comprised of delta adjustment values that apply to particular target items and that have effect for instances within particular regions of the font's variation space. Some mechanism is needed to associate delta values with target items. In the tuple variation store formats (described earlier in this subclause), data containing a set of deltas also includes a set of point number indices to identify the target items to which the deltas apply. In the item variation store, however, a block of delta values has implicit *delta-set* indices, and separate data outside the item variation store is provided that indicates the delta-set index associated with a particular target item. For example, the 'MVAR' table header includes an array of records that identify target font data items and the delta-set index for each item.

The itemVariationStore table includes a variation region list, which defines all of the different regions of the font's variation space for which variation data is defined. It also includes a set of itemVariationData subtables, each of which provides a portion of the total variation data. Each subtable is associated with some subset of the defined regions, and will include deltas used for one or more target items. Conceptually, the deltas form a two-dimensional array, with delta-set rows that include a delta for each of the regions referenced by that subtable. From this perspective, the table columns correspond to regions.

The item variation store includes a variation region list and an array of item variation data subtables. The following figure illustrates the overall structure.



Note that multiple subtables are necessary only if the number of distinct delta-set data exceeds 65,536. Multiple subtables may also be used, however, to provide more compact data representation. There are different ways that the delta data can be made more compact.

First, deltas with a value of zero have no impact on their target items. If there are several delta-set rows that have a zero delta for the same region, then those rows could be moved into a subtable that does not reference that region. As a result, there will be fewer delta values in each row, making the size of data for those rows smaller.

Also, some delta values require 16-bit representations, but some require only 8 bits. For a given subtable, deltas in each row correspond in order to the regions that are referenced, but the ordering of regions has no effect. Hence, regions and corresponding deltas within each row can be re-ordered. Thus, regions that require 16-bit delta representations can be ordered together. The itemVariationData format specifies a count of regions (columns) for which a 16-bit delta representation is used, with the remaining deltas in each row using 8 bits. By reordering columns, the size required for a given delta-set row can potentially be reduced. If a set of rows have similar requirements in regard to which columns have deltas requiring 16-bit versus 8-bit representations, then those rows can be moved into a subtable with a column order that allows a maximal number of deltas using 8-bit rather than 16-bit representations.

Note that there is minimal overhead for each subtable: 10 bytes (6 bytes in the subtable header and 4 bytes for the offset in the parent table) plus 2 bytes for each region that is referenced.

A complete delta-set index involves an outer-level index into the itemVariationData subtable array, plus an inner-level index to a delta-set row within that subtable. As noted above, delta-set indices are stored outside the variation store. Different parent tables that use an item variation store will store indices in different ways, and may utilize different schemes for how to represent the indices in an efficient manner. For example, the 'HVAR' and 'VVAR' tables allow the outer and inner indices to be combined into one-byte, two-byte, three-byte or four-byte representations depending on the indexing requirements of the variation store. For larger sets of variation data, such as may be needed for 'HVAR' or 'VVAR' tables, optimization of the indices data as well as the delta data may have a significant impact on overall size. Optimizing compilers may need to consider the impact on representation of indices in tandem as it optimizes the item variation store to achieve the best overall results.

7.2.3.1 Variation regions

As noted above, variation data is comprised of delta adjustment values that have effect over particular regions within the font's variation space. In a tuple variation store (described earlier in this chapter), the deltas are organized into groupings by region of applicability, with each grouping associated with a particular region. In contrast, the item variation store format organizes deltas into groupings by the target items to which they apply, with each grouping having deltas for several regions. Accordingly, the item variation store uses different formats for describing the regions in which a set of deltas apply.

For a given item variation store, a set of regions is specified using a VariationRegionList.

VariationRegionList

Type	Name	Description
uint16	axisCount	The number of variation axes for this font. This must be the same number as axisCount in the 'fvar' table.
uint16	regionCount	The number of variation region tables in the variation region list.
VariationRegion	variationRegions [regionCount]	Array of variation regions.

The regions can be in any order. Each region is defined using an array of RegionAxisCoordinates records, one for each axis defined in the 'fvar' table:

VariationRegion record

Type	Name	Description
RegionAxisCoordinates	regionAxes[axisCount]	Array of region axis coordinates records, in the order of axes given in the 'fvar table.

Each RegionAxisCoordinates record provides coordinate values for a region along a single axis:

RegionAxisCoordinates Record

Type	Name	Description
F2DOT14	startCoord	The region start coordinate value for the current axis.
F2DOT14	peakCoord	The region peak coordinate value for the current axis.
F2DOT14	endCoord	The region end coordinate value for the current axis.

The three values shall all be within the range -1.0 to +1.0. startCoord must be less than or equal to peakCoord, and peakCoord must be less than or equal to endCoord. The three values must be either all non-positive or all non-negative with one possible exception: if peakCoord is zero, then startCoord can be negative or 0 while endCoord can be positive or zero.

NOTE The following guidelines are used for setting the three values in different scenarios:

- In the case of a non-intermediate region for which the given axis should factor into the scalar calculation for the region, either startCoord and peakCoord are set to a negative value (typically, -1.0) and endCoord is set to zero, or startCoord is set to zero and peakCoord and endCoord are set to a positive value (typically +1.0).
- In the case of an intermediate region for which the given axis should factor into the scalar calculation for the region, startCoord, peakCoord and endCoord are all set to non-positive values or are all set to non-negative values.
- If the given axis should not factor into the scalar calculation for a region. This is achieved by setting peakCoord to zero. In this case, startCoord can be any non-positive value, and endCoord can be any non-negative value. It is recommended either that all three be set to zero, or that startCoord be set to -1.0 and endCoord be set to +1.0.

The full algorithm for interpolation of instance values is given in subclause 7.1 (Font variations overview). The logical algorithm involves computing per-axis scalar values for a given region and a given instance. The per-axis scalars for a region are then combined to yield an overall scalar for the region that is then applied to delta adjustment values. Given a selected variation instance, a per-axis scalar can be calculated for each RegionAxisCoordinates record. The overall scalar for a region can be calculated by combining the per-axis scalars for that region.

7.2.3.2 Item variation store and item variation data tables

The item variation store table has the following structure.

ItemVariationStore table

Type	Name	Description
uint16	format	Set to 1.
Offset32	variationRegionListOffset	Offset in bytes from the start of the item variation store to the variation region list
uint16	itemVariationDataCount	The number of item variation data subtables.
Offset32	itemVariationDataOffsets [itemVariationDataCount]	Offsets in bytes from the start of the item variation store to each item variation data subtable.

The item variation store includes an array of offsets to item variation data subtables. Each item variation data subtable includes deltas for some number of items, and some subset of regions. The regions are indicated by an array of indices into the variation region list.

ItemVariationData subtable

Type	Name	Description
uint16	itemCount	The number of delta sets for distinct items.
uint16	shortDeltaCount	The number of deltas in each delta set that use a 16-bit representation. Must be less than or equal to regionCount.
uint16	regionIndexCount	The number of variation regions referenced.
uint16	regionIndexes[regionCount]	Array of indices into the variation region list for the regions referenced by this item variation data table.
DeltaSet	deltaSets[itemCount]	Delta-set rows.

DeltaSet record

Type	Name	Description
uint8	deltaData [shortDeltaCount + regionCount]	Variation delta values – a table with itemCount rows and regionCount columns.

In a logical view, the item variation data table contains a two-dimensional array of delta values with itemCount rows and regionCount columns. In each delta-set row, the first shortDeltaCount elements are represented as

signed 16-bit values (int16), and the remaining regionCount – shortDeltaCount elements are represented as signed 8-bit values (int8). The length of the data for each row is shortDeltaCount + regionCount.

NOTE Delta values are each represented directly. They are not packed as in the tuple variation store.

7.2.3.3 Processing item variation store data

When a particular variation instance has been selected, an application needs to process the variation store data associated with particular target items to derive interpolated values for those items and that instance.

For a given target item, data outside the item variation store provides a delta-set outer-/inner-index pair for that item. The associations between target items and delta-set indices are represented in different ways in different parent tables; the specification of each font table should be referred to for the formats used.

To compute the interpolated instance value for a given target item, the application uses data in the parent table to get the delta-set index for that item. It uses the outer-level index portion to select an item variation data subtable within the item variation store, and the inner-level index portion to select a delta-set row within that subtable. The delta set contains one delta for each region referenced by the subtable, in order of the region indices given in the regionIndices array. The delta values in the delta set are read with the first shortDeltaCount elements read as int16, and the remaining regionCount – shortDeltaCount values read as int8. The application uses the regionIndices array for that subtable to identify applicable regions and to compute a scalar for each of these regions based on the selected instance. Each of the scalars is then applied to the corresponding delta within the delta set to derive a scaled adjustment. The scaled adjustments for the row are then combined to obtain the overall adjustment for the item.

Complete details on the interpolation algorithm logic are provided in subclause 7.1.

When a particular variation instance has been selected, an application will often need to interpolate values for several items that may use deltas in different item variation data subtables. All of the subtables will reference region definitions in the shared variation region list. When the instance has been selected, applications can pre-compute and cache a scalar for that instance for each region in the region list. Then when processing different target items, the cached scalar array can be used without needing to re-compute region scalars for each target item.

7.2.4 Design-variation axis tag registry

This registry defines design-variation axis tags for use in OpenType fonts. By providing registered tags with well-defined semantics and associated numeric scales for variation, this provides some degree of interoperability between different fonts from different vendors, or between fonts and applications.

Design-variation axis tags are used within the 'fvar' table in variable fonts, and also within the 'STAT' table. Note that they are relevant for non-variable fonts as well as variable fonts: even though a font might not be a variable font, it is still a design variant within its font family. The 'STAT'' table allows applications to understand relationships of font design variants within a given family, whether they are implemented as non-variable fonts or as variation instances of a variable font.

Syntactic requirements for design-variation axis tags

Design-variation axis tags are arrays of four unsigned bytes (uint8[4]) that can equivalently be interpreted as a string of four ASCII characters. Axis tags must begin with a letter (0x41 to 0x5A, 0x61 to 0x7A) and must use only letters, digits (0x30 to 0x39) or space (0x20). Space characters must only occur as trailing characters in tags that have fewer than four letters or digits.

Fonts may use tags defined in this registry, or may use foundry-defined tags. (Foundry-defined tags can also be referred to as “custom” or “private” tags.) Foundry-defined tags must begin with an uppercase letter (0x41 to 0x5A), and must use only uppercase letters or digits. Registered axis tags must not use that pattern, but can use any other valid pattern. This ensures that foundry-defined tags and registered tags are never conflicting.

Documentation of registered axis tags

For every axis tag defined in this registry, certain information is required or recommended:

- The tag specification must include a US English name for the axis that may be used as a display string in application user interfaces to refer to the axis, or as the basis of localized display strings.
- The tag specification must include a description of the intended meaning and design-variation behavior for the axis.
- The tag specification must include information regarding the numeric scale used for the axis. This must include a specification of the range of values that are valid for the axis. Depending on the nature of the axis, this may or may not be a bounded range. It must also provide information regarding the semantic interpretation for values, specifying either some objective measure or some convention by which values can be interpreted.
- Whenever appropriate, the tag specification should also indicate a numeric value that is recommended or required (by definition of the scale) for that axis in a “Regular” font.

NOTE The “Regular” font in a font family often has a sub-family name that omits axis qualifiers. For example, one foundry may create an optical-size font suited to 12 point size with “Text” included in the name as an indicator of the intended optical size, but another foundry may create a similar font without any indicator for optical size in the name. The choice for a recommended “Regular” axis value should be made with this in mind.

Additional information may also be provided, such as suggestions for programmatic selection of axis values that may be useful in applications.

The specification for the semantic interpretation of numeric values is required as a means to provide some degree of interoperability between different fonts, between fonts and software implementations, and between OpenType Font Variations and other specifications, such as font-weight values in CSS.

Note that interoperability is assumed to be attained in varying degrees, depending on the nature of an axis and the scale that it uses. For example, the scale for the Weight axis provides a limited degree of interoperability. Two different fonts with a Weight axis value of 700 (or “Bold”) may not result in the same amount of darkness or “color” when applied to the same text; but in both cases, a user can expect these to be darker than the “Regular” or “Semibold” fonts from each respective font family, and application developers can produce results that will be predictable for users if they associate that axis value with a particular state of a user-interface control or with a markup tag.

In contrast to this, the scale for the Optical size axis is designed to provide a much stronger degree of interoperability. For instance, two different fonts with an optical-size value of 20 are assumed to be best suited to text set at 20 points, because the scale is designed that way. If this axis had been defined with a different numeric scale, then an application might not be able to assume that two fonts with the same optical-size value are equally-well suited for a given context.

Not all axes will be equally amenable to a precise or objective measure. For example, there is no objective scale for an amount of italicness. But an Italic axis can be defined with a range from 0.0 to 1.0, representing whatever the font developer considers to be a non-italic design and a fully-italic design, and that is sufficient for applications to associate those numeric variation values with off/on states of an Italic toggle in a user interface to provide a meaningful and familiar experience. It also provides a useful basis of comparison between different fonts, which may be important, for instance, in font-fallback implementations: if the requested font face had an Italic axis setting of 1 but a font-fallback font must be used when displaying text, the application is able to select an appropriate Italic axis setting in the fallback font.

If an axis is intended to interact with programmatic mechanisms that automatically select axis values to provide some effect, then a more precise definition of the numeric scale and its interpretation may be needed. It must be clear to application and platform developers what independent variables should contribute as inputs for selection of axis values, and how the numeric values of the axis scale can be derived from those inputs.

For variable font implementations that support a given axis, the “Regular” value will often be a good choice for the default value of that axis in the 'fvar' variation axis record. The default values set in the 'fvar' table, however, are implementation-specific, and fonts are not required to use this “Regular” value as the axis default value.

How to register a design-variation axis tag

While font developers can always use foundry-defined axis tags however they might choose, we encourage font developers to use registered axis tags when implementing designs for which the registered axis is applicable. We welcome submissions for new design-variation axis tag registration.

Registration of an axis tag can be useful for two key purposes. One is to foster conventionality and familiarity for a kind of design variation. For example, by defining the 'opsz' tag for optical-size variation, to use for tailoring glyph outlines according to the font size, different vendors can incorporate this kind of design variation into fonts, and the variations in these various fonts can be presented to font users as the same kind of variation. The more fonts that are implemented using an 'opsz' axis, the more familiar designers and content authors will become with this kind of design variation. They will benefit from greater consistency in experience when fonts use the same concepts than if different fonts were using similar but different concepts.

Another key purpose for registration of axis tags is to facilitate interoperability between different fonts or between fonts and applications. For example, by specifying a numeric scale for the 'opsz' axis that corresponds to text size in points, this makes it possible for applications to implement mechanisms for automatic selection of optical-size variation that can work with any fonts that support variation in the 'opsz' axis.

The merits for adding a variation axis tag to the registry is primarily determined in relation to these two key purposes: What is the likelihood that an axis of design variation will be implemented in fonts from multiple vendors and found to be useful to designers; and what is the likelihood that applications will implement mechanisms that make use of an interoperable understanding of the axis.

To qualify for registration, a complete description of the axis must be provided, including each of the categories of information listed above. If the axis is intended to interact with mechanisms that select axis values programmatically, then the description must include a clear specification for the numeric scale. There must be reasonable indication of alignment with one or both of the two key purposes for registration described above, and a reasonable indication that the axis will be implemented in fonts from multiple vendors and supported in software platforms and applications. It is recommended that the party proposing the new registration seek input from and get consensus among multiple font and software vendors regarding the definition of the proposed axis and its merits.

7.2.4.1 Registered axis tags list

The following design-variation axes and tags have been registered; the linked pages provide the axis tag descriptions. These are listed in alphabetic order of the tags.

Axis Tag	Name
'ital'	Italic
'opsz'	Optical size
'slnt'	Slant
'wdth'	Width
'wght'	Weight

7.2.4.2 Registered axis tags definitions

Tag: 'ital'

Name: Italic

Description: Used to vary between non-italic and italic.

Valid numeric range: Values must be in the range 0 to 1.

Scale interpretation: A value of 0 can be interpreted as “Roman” (non-italic); a value of 1 can be interpreted as (fully) italic.

Recommended or required "Regular" value: 0 is required.

Additional information:

The Italic axis has long been treated as a kind of design variation within a font family. The 'ital' axis tag is used within a 'STAT' table of italic fonts to provide a complete characterization of a font in relation to its family within the 'STAT' table. The Italic axis can be used as a variation axis within a variable font, though this is not expected to be common.

The Italic axis is distinct from the Slant axis ('slnt'). Fonts may use one or the other, depending on the nature of the design, but should rarely use both. While an italic design often incorporates some slant in the design, use of the Italic axis does not require use of the Slant axis. An italic font should not be characterized in the 'STAT' table as being italic and also having some slant, unless the font family includes multiple italic designs with different amounts of slant.

Tag: 'opsz'

Name: Optical size

Description: Used to vary design to suit different text sizes.

Valid numeric range: Values must be strictly greater than zero.

Scale interpretation: Values can be interpreted as text size, in points.

Recommended or required "Regular" value: A value in the range 9 to 13 is recommended.

Suggested programmatic interactions: Applications may choose to select an optical-size variant automatically based on the text size.

Additional information:

The Optical size axis can be used as a variation axis within a variable font. It can also be used within a 'STAT' table in non-variable fonts within a family that has optical-size variants to provide a complete characterization of a font in relation to its family within the 'STAT' table. In the 'STAT' table of a non-variable font, a format 2 axis value table is recommended to characterize the range of text sizes for which the optical-size variant is intended.

The scale for the Optical size axis is text size in points. For these purposes, the text size is as determined by the document or application for its intended use; the actual physical size on a display may be different due to document or application zoom settings or intended viewing distance.

In applications that automatically select an Optical size variant, this should normally be done based on the text size with a default or "100%" zoom level, not on a combination of text size and zoom level.

Tag: 'slnt'

Name: Slant

Description: Used to vary between upright and slanted text.

Valid numeric range: Values must be greater than -90 and less than +90.

Scale interpretation: Values can be interpreted as the angle, in counter-clockwise degrees, of oblique slant from whatever the designer considers to be upright for that font design.

Recommended or required "Regular" value: 0 is required.

Additional information:

The Slant axis can be used as a variation axis within a variable font. It can also be used within a 'STAT' table in non-variable, oblique fonts to provide a complete characterization of a font in relation to its family within the 'STAT' table.

The Slant axis is distinct from the Italic axis ('ital'). Fonts may use one or the other, depending on the nature of the design, but should rarely use both. While an italic design often incorporates some slant in the design, use of the Italic axis does not require use of the Slant axis. An italic font should not be characterized in the 'STAT' table as being italic and also having some slant, unless the font family includes multiple italic designs with different amounts of slant.

Note that the scale for the Slant axis is interpreted as the angle of slant in counter-clockwise degrees from upright. This means that a typical, right-leaning oblique design will have a negative slant value. This matches the scale used for the italicAngle field in the 'post' table.

In a variable font that implements 'slnt' variations, the value in the italicAngle field of the 'post' table must match the default 'slnt' value specified in the 'fvar' table. For non-default instances of a variable font, the 'slnt' axis value can be used as the post.italicAngle value for the instance.

Tag: 'wdth'

Name: Width

Description: Used to vary width of text from narrower to wider.

Valid numeric range: Values must be strictly greater than zero.

Scale interpretation: Values can be interpreted as a percentage of whatever the font designer considers “normal width” for that font design.

Recommended or required “Regular” value: 100 is required.

Suggested programmatic interactions: Applications may choose to select a width variant in a variable font automatically in order to fit a span of text into a target width.

Additional information:

The Width axis has long been used in conjunction with face names such as “Condensed” or “Extended”. Change in glyph width is typically the primary aspect of the design that varies, though other secondary details such as stroke thickness may also be encompassed in this variation.

The Width axis can be used as a variation axis within a variable font. It can also be used within a 'STAT' table in non-variable fonts within a family that has width variants to provide a complete characterization of a font in relation to its family within the 'STAT' table.

The Width axis uses a scale that correlates with but is different from the scale used for the usWidthClass field of the 'OS/2' table. The description of usWeightClass in the 'OS/2' table documentation provides a table of mappings from usWidthClass values to “% of normal” values. Because usWidthClass is limited to nine integer values, is has much less granularity than the Width axis.

When mapping from 'wdth' values to usWidthClass, interpolate fractional values between the mapped values and then round, and clamp to the range 1 to 9.

In a variable font that implements 'wdth' variations, the value in the usWidthClass field of the 'OS/2' table must correspond to the default 'wdth' value specified in the 'fvar' table. For non-default instances of a variable font, the 'wdth' axis value can be used to derive the OS/2.usWidthClass value for that instance.

Percentage of normal width is a comparitive scale that will depend on the specific items being compared. The width of a line of text very much depends on the content of the text. No specific reference string is specified here as the basis for comparisons; a font designer can choose what they consider to be representative strings assigning a 'wdth' value to a design variant. Ideally, the 'wdth' value should provide a good estimate for most strings in the target languages of how the width of the string formatted with that 'wdth' variant compares to the width of the same string when formatted with the “normal” variant.

When using a variable font, applications may choose to make small, automated 'wdth' adjustments in order to fit a span of text to some target size. This might be done, for instance, to fit headings within a column, or to improve paragraphy layout. The relative change in 'wdth' value (ratio of the original 'wdth' to the adjusted 'wdth') may be used as a first approximation of the adjustment needed. Since the relative

change in width may depend on the actual text content, however, this may not provide the exact adjustment needed to obtain the desired width adjustment. The application will likely need to refine the adjustment over multiple attempts.

Tag: 'wght'

Name: Weight

Description: Used to vary stroke thicknesses or other design details to give variation from lighter to blacker.

Valid numeric range: Values must be in the range 1 to 1000.

Scale interpretation: Values can be interpreted in direct comparison to values for usWeightClass in the 'OS/2' table, or the CSS font-weight property.

Recommended or required "Regular" value: 400 is required.

Additional information:

The Weight axis has long been used in conjunction with face names such as "Regular", "Light" or "Bold". Change in stroke thickness is typically the primary aspect of the design that varies, though other secondary details such as glyph width or thick-thin contrast may also be encompassed in this variation.

The Weight axis can be used as a variation axis within a variable font. It can also be used within a 'STAT' table in non-variable fonts within a family that has weight variants to provide a complete characterization of a font in relation to its family within the 'STAT' table.

In a variable font that implements 'wght' variations, the value in the usWeightClass field of the 'OS/2' table must match the default 'wght' value specified in the 'fvar' table. For non-default instances of a variable font, the 'wght' axis value can be used as the OS/2.usWeightClass value for the instance.

7.3 Font variations tables

For an overview of OFF font variations and the specification of the interpolation algorithm used for variations, see OFF "Font variations overview". For details regarding which tables are required or optional in variable fonts, see "Variation data tables and miscellaneous requirements".

For information on common table formats used for variations, see OFF "Font variations common table formats".

Note that some variation-related formats may be used in tables other than the variations-specific tables listed above. In particular, the 'GDEF' or 'BASE' tables in a variable font can include variation data using common table formats. A 'CFF2' table in a variable font can also include variation data, though using formats that are specific to the 'CFF2' table.

7.3.1 avar – Axis variations table

The axis variations table ('avar') is an optional table used in variable fonts that use OFF Font Variations mechanisms. It can be used to modify aspects of how a design varies for different instances along a particular design-variation axis. Specifically, it allows modification of the coordinate normalization that is used when processing variation data for a particular variation instance.

The 'avar' table shall be used in combination with a font variations ('fvar') table and other required or optional tables used in variable fonts.

7.3.1.1 Overview

The 'fvar' table defines a range of design variations supported by a font — the font's variation space. It specifies certain axes of variation that are used, and a range of supported values for each axis using scales appropriate to the nature of each axis. When processing a font's variation data to derive glyph outlines or other values for a particular variation instance, the coordinates selected by the user must be mapped from the scales that are used to define the axes in the 'fvar' table to a normalized scale that is the same for every axis.

A default normalization mapping is defined that maps the minimum, default and maximum values specified for each axis in the 'fvar' table to -1, 0, and 1, respectively, and that maps other values in between by linear interpolation between those values. The default mapping can be represented by the following pseudo-code.

```
if (userValue < axisDefaultValue)
{
    defaultNormalizedValue = -(axisDefault - userValue) / (axisDefault - axisMin);
}
else if (userValue > axisDefaultValue)
{
    defaultNormalizedValue = (userValue - axisDefault) / (axisMax - axisDefault);
}
else
{
    defaultNormalizedValue = 0;
}
```

Take notice that, if the user selects an axis value that is outside the minimum/maximum range specified in the font, then the value used must be clamped to the minimum or maximum value.

The default normalization uses a predefined mapping of three positions along each axis to particular values, dividing the overall range of each axis into two segments. If an 'avar' table is present, then the output of the default normalization can be further modified by allowing mappings to be defined for additional positions along each scale, creating multiple segments, with other values within each segment interpolated. The following figure illustrates an example of such a modification.

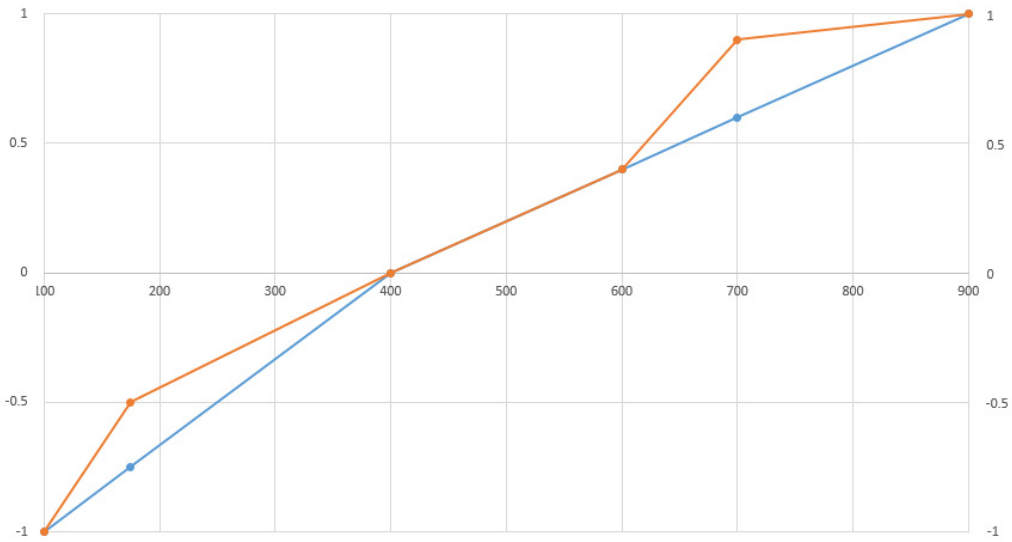


Figure 7.3: Example of 'avar'-modified normalization – horizontal axis is user scale, vertical axis is normalized scale

The conceptual effect of these additional scale mappings is to make the variation along an axis less linear. Values change linearly within each segment, but additional segments make the way that values change across the entire axis range less linear overall. The effect might also be described as compressing some portions of the scale while making other portions less compressed.

The visual effects of additional axis-value mappings in the 'avar' table is seen in how glyph outlines change as the user-scale values for an axis change. The same amount of change in the user-scale value may correspond to a subtle change in glyph outlines in one portion of the axis range, but more dramatic changes in the glyph outlines in another portion of the axis range.

Note that it may be possible to achieve the same or similar effects by adding glyph variation data for additional regions of the variation space. That approach requires more work and more font data, however, and tedious

design iteration may be needed to obtain the desired results. The 'avar' table may provide a simple and light-weight way to achieve a particular effect.

Note also that the variation data created by the font designer will also have a significant effect on whether user-scale values and glyph outlines change uniformly. When an 'avar' table is used, the 'avar' table and the glyph variation data (for TrueType or CFF2) are both factors in the variation behavior that the user will see.

7.3.1.2 Table formats

The 'avar' table is comprised of a small header plus segment maps for each axis.

Axis variation table

Type	Name	Description
uint16	majorVersion	Major version number of the axis variations table – set to 1.
uint16	minorVersion	Minor version number of the axis variations table – set to 0.
Uint16	<reserved>	Permanently reserved, set to 0.
uint16	axisCount	The number of variation axes for this font. This must be the same number as axisCount in the 'fvar' table.
SegmentMaps	axisSegmentMaps[axisCount]	The segment maps array – one segment map for each axis, in the order of axes specified in the ‘fvar’ table.

There must be one segment map for each axis defined in the 'fvar' table, and the segment maps for the different axes shall be given in the order of axes specified in the 'fvar' table. The segment map for each axis is comprised of a list of axis-value mapping records.

SegmentMaps record

Type	Name	Description
uint16	positionMapCount	The number of correspondence pairs for this axis.
AxisValueMap	axisValueMaps[positionMapCount]	The array of axis value map records for this axis.

Each axis value map record provides a single axis-value mapping correspondence.

AxisValueMap record

Type	Name	Description
F2DOT14	fromCoordinate	A normalized coordinate value obtained using default normalization.
F2DOT14	toCoordinate	The modified, normalized coordinate value.

Axis value maps can be provided for any axis, but are required only if the normalization mapping for an axis is being modified. If the segment map for a given axis has any value maps, then it must include at least three value maps: -1 to -1, 0 to 0, and 1 to 1. These value mappings are essential to the design of the variation mechanisms and are required even if no additional maps are specified for a given axis. If any of these is missing, then *no modification* to axis coordinate values will be made for that axis.

All of the axis value map records for a given axis shall have different fromCoordinate values, and axis value map records shall be arranged in increasing order of the fromCoordinate value. If the fromCoordinate value of a record is less than or equal to the fromCoordinate value of a previous record in the array, then the given record may be ignored.

Also, for any given record except the first, the toCoordinate value must be greater than or equal to the toCoordinate value of the preceding record. This requirement ensures that there are no retrograde behaviors as the user-scale value range is traversed. If a toCoordinate value of a record is less than that of the previous record, then the given record may be ignored.

7.3.1.3 Processing

Each pair of axis value map records for a given axis defines a segment within the range for that axis. Within a segment, intermediate values are interpolated linearly. For a given user-scale coordinate, the full normalization process, with 'avar' modifications applied, is as follows.

1. Compute the default normalized coordinate value, **defaultNormalizedValue**, as described above.
2. Using the SegmentMaps record for the given axis, scan the AxisValueMap records to find the first record that has a fromCoordinate value greater than or equal to **defaultNormalizedValue**. Designate this record as **endSeg**.
3. If **endSeg.fromCoordinate** equals **defaultNormalizedValue**, then the final, modified normalized value is **endSeg.toCoordinate**. Return this result and end.
4. The else case (**endSeg.fromCoordinate** is strictly greater than **defaultNormalizedValue** — endSeg cannot be the first map record, which is for -1): Designate the preceding record as **startSeg**.
5. The final, modified normalized value, **finalNormalizedValue**, is computed as follows:

$$\begin{aligned}\text{finalNormalizedValue} = {} & \text{startSeg.toCoordinate} \\ & + (\text{endSeg.toCoordinate} - \text{startSeg.toCoordinate}) \\ & \times \left(\frac{\text{defaultNormalizeValue} - \text{startSeg.fromCoordinate}}{\text{endSeg.fromCoordinate} - \text{startSeg.fromCoordinate}}\right)\end{aligned}$$

Take notice that certain requirements regarding the level of precision used and how rounding is handled must be observed by implementations.

7.3.1.4 Axis segments example

The following example illustrates how the 'avar' mappings work. This example is illustrated by the figure shown earlier in this chapter.

Suppose that the 'avar' table of a font has the following mappings for a particular axis:

Record index	*fromCoordinate*	*toCoordinate*
0	-1.0	-1.0
1	-0.75	-0.5
2	0	0
3	0.4	0.4
4	0.6	0.9
5	1	1

Suppose that the user selects an instance, and the default normalized value for this instance is -0.5. The relevant segment for this value is defined by record 1 and record 2. The final normalized value is computed as follows:

$$\text{finalNormalizedValue} = -0.5 + \big(0 - (-0.5)\big) \times \left(\frac{-.5 - (-0.75)}{0 - (-0.75)}\right)$$

$$= -0.3333$$

The following table shows how several normalized coordinate values would be modified by this 'avar' data:

Default normalized value	Final normalized value
-1.0	-1.0
-0.75	-0.5
-0.5	-0.3333
-0.25	-0.1667
0	0
0.25	0.25
0.5	0.65
0.75	0.9375
1.0	1.0

7.3.2 cvar – CVT variations table

The control value table (CVT) variations table is used in variable fonts to provide variation data for CVT values.

Fonts that use the Truetype outline format for glyphs and that have hinting instructions will typically also have a CVT table. The CVT table provides an indexed list of control values that can be referenced by instructions. Example values are the height of a serif, x-height, or the width of upper case stems. When glyph outline points are adjusted by instructions to improve rasterization at a particular PPEM size, the control values may be used by the instructions to provide design-distance values for those adjustments – typically, values that need to be kept constant across all glyphs in the font for a given PPEM size.

Within a variable font, the numeric value of particular control values may need to be adjusted for different variation instances, to match the changes to outlines for different instances. The CVT variations table provides variation data for that purpose. By using interpolation to derive adjusted CVT values for a particular instance, instructions can obtain instance-appropriate values, and the same instructions can be used for all variations.

The CVT variations table must be used in combination with TrueType outlines and a 'cvt ' table, and also in combination with a font variations ('fvar') table and other required or optional tables used in variable fonts.

7.3.2.1 Table format

The 'cvar' table uses a variant of the tuple variation store format. The 'gvar' table uses a very-slightly different variant of this format for each glyph.

In terms of overall structure, the ‘cvar’ table begins with a header, which is followed by serialized variation data.

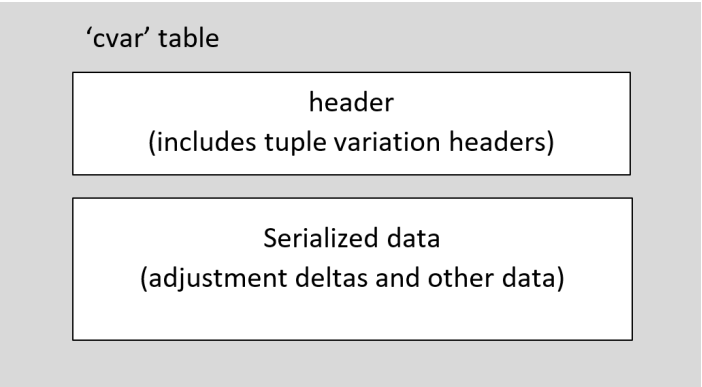


Figure 7.4: High-level organization of the 'cvar' table

The variation data includes logical groupings of data that apply to different regions of the variation space – *tuple-variation data* tables. These logical groupings are stored in two parts: a header, and serialized data. The 'cvar' header includes an array of tuple-variation data headers, each of which is associated with a particular portion of the serialized data.

The serialized data includes adjustment delta values and also packed "point number" data that identify the CVT entries to which the deltas apply. The serialized data for a given region can include point number data that applies to that specific tuple-variation data, but there can also be a shared set of point number data, stored at the start of the serialized data, that can be used in relation to multiple tuple-variation data tables.

The format of the 'cvar' header is as follows.

'cvar' table header

Type	Name	Description
uint16	majorVersion	Major version number of the CVT variations table — set to 1.
uint16	minorVersion	Minor version number of the CVT variations table — set to 0.
uint16	tupleVariationCount	A packed field. The high 4 bits are flags, and the low 12 bits are the number of tuple-variation data tables for this glyph. The count can be any number between 1 and 4095.
Offset16	dataOffset	Offset from the start of the 'cvar' table to the serialized data.
TupleVariationHeader	tupleVariationHeaders [tupleVariationCount]	Array of tuple variation headers.

Complete details regarding the tupleVariationCount field, the flags used in the tupleVariationCount, the TupleVariationHeader format and the format of the serialized data are provided in the OFF "Font variations common table formats". Details regarding how the data are processed to derive interpolated CVT values for particular instances are provided in that subclause and in the OFF "Font variations overview".

As noted above, the format of the 'cvar' table is closely related to formats used in the 'gvar' table. The following are key differences to note:

— The TupleVariationHeader structure includes a particular field, tupleIndex. This is a packed field that includes flag bits, one of which indicates whether the structure includes an embedded peak tuple record. In the 'gvar' table, this is optional, and the flag does not always need to be set. In the 'cvar' table, however, the embedded peak tuple record is mandatory, and this flag shall always be set.

— The serialized data includes packed “point” numbers. In the context of the 'gvar' table, these are indices for the outline points of a particular glyph. In the 'cvar' table, these are indices for CVT values within the 'cvt ' table.

— In the 'gvar' table, there are two logical deltas for each enumerated outline point: one for the X coordinate, and one for the Y coordinate. Hence, the total number of logical deltas is twice the count of point numbers. In the 'cvar' table, there is exactly one logical delta for each point number.

Note that the CVT values are all FWORDs, and that the total number of CVT values is determined by the length of the 'cvt ' table. Hence, CVT index values range from 0 to floor(cvtLength / sizeof(FWORD)) – 1.

7.3.3 fvar – Font variations table

Font Variations allow a font designer to incorporate multiple faces within a font family into a single font resource. Variable fonts can provide great flexibility for content authors and designers while also allowing the font data to be represented in an efficient format.

A variable font allows for continuous variation along some given design axis, such as weight:

Figure 7.5: Continuous variation along a design axis

Conceptually, variable fonts define one or more axes over which design characteristics can vary. Weight is one possible axis of variation, but many different kinds of variation are possible. Variable fonts are not limited to a single axis of variation, but can combine two or more different axes of variation. For example, the following illustrates a combination of weight and width variation:

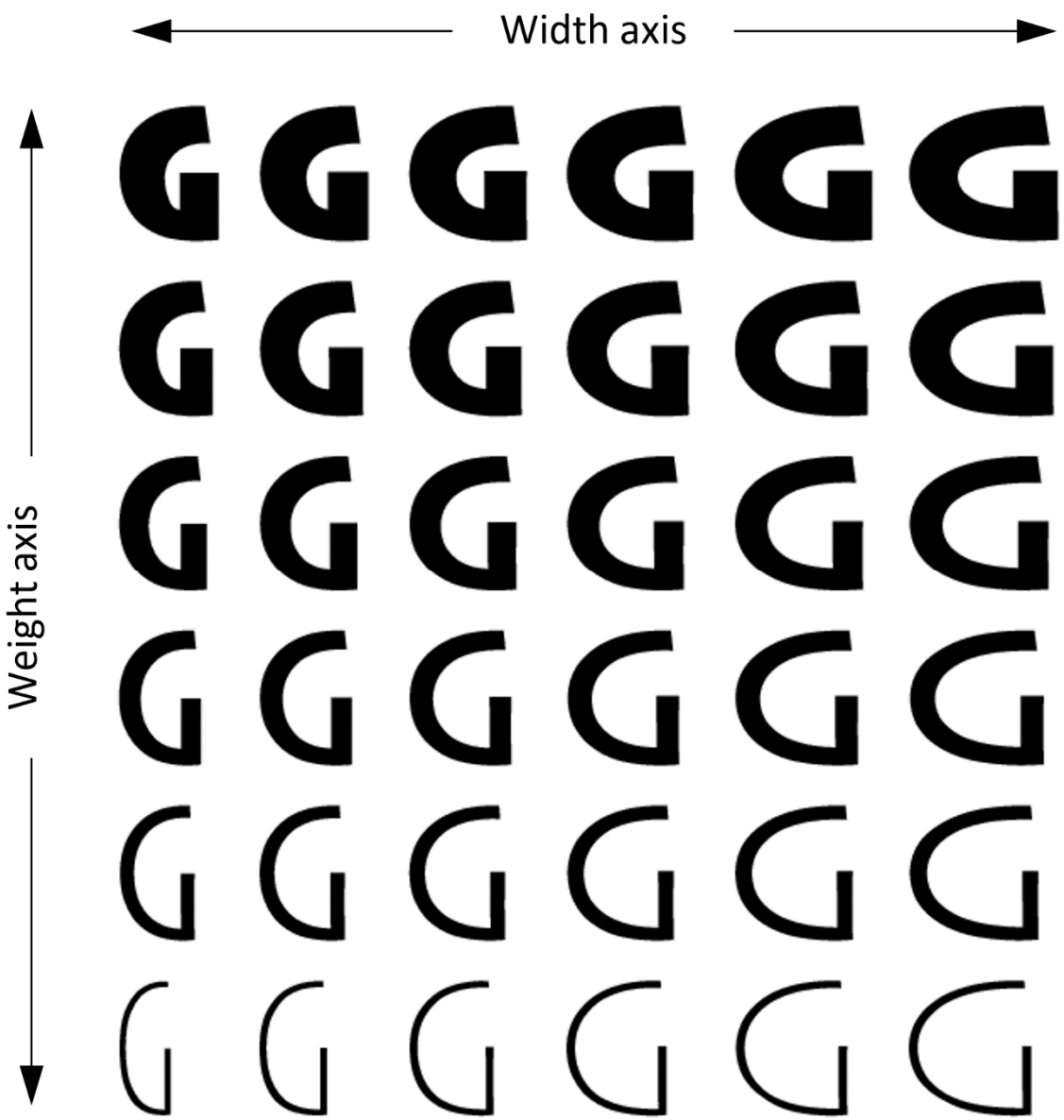


Figure 7.6: Continuous variation along multiple design axes

The set of axes supported by a font define a variation space for that font, supporting a potentially-vast number of design-variation instances at positions across that space.

The font variations table ('fvar') provides the global definition of variations supported within the font. It specifies the axes of variation that are used and the ranges of variation for each axis. It also allows the font designer to specify certain coordinate positions within the font's variation space as named instances. Named instances have designer-provided names, effectively equivalent to sub-family names, that applications can use as a short list of “pre-chosen” design variants they can offer to users.

All variable fonts shall include a font variations table, as well as other required or optional tables used in variable fonts.

Note that some of the information in the font variations table also needs to be reflected in the style attributes ('STAT') table, which is also required in all variable fonts. In particular, each axis and each named instance specified in the font variations table shall have matching axis records and axis value tables in the style attributes table.

7.3.3.1 Table formats

The font variations table consists of a table header, followed by an array of variation axis records, followed by an array of named-instance records:

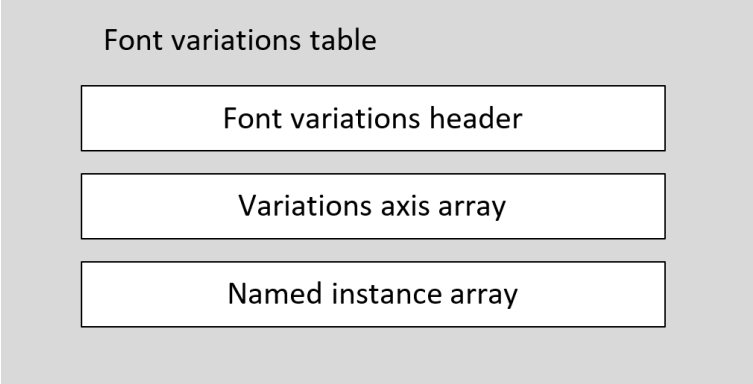


Figure 7.7: High-level organization of the font variations table

'fvar' Header

The format of the font variations table header is as follows.

Font variations header

Type	Name	Description
uint16	majorVersion	Major version number of the font variations table – set to 1.
uint16	minorVersion	Minor version number of the font variations table – set to 0.
Offset16	axesArrayOffset	Offset in bytes from the beginning of the table to the start of the variation axis array – set to 16 (0x0010) for this version.
uint16	(reserved)	This field is permanently reserved. Set to 2.
uint16	axisCount	The number of variation axes in the font (the number of records in the axes array).
uint16	axisSize	The size in bytes of each VariationAxisRecord – set to 20 (0x0014) for this version.
uint16	instanceCount	The number of named instances defined in the font (the number of records in the instances array).
uint16	instanceSize	The size in bytes of each InstanceRecord – set to either *axisCount * sizeof(Fixed) + 4*, or *axisCount * sizeof(Fixed) + 6*.

The header is followed by axes and instances arrays. The location of the axes array is specified in the axisArrayOffset field; the instances array directly follows the axes array.

VariationAxisRecord	axes[axisCount]	The variation axis array.
InstanceRecord	instances[instanceCount]	The named instance array.

NOTE The axisSize and instanceSize fields indicate the size of the VariationAxisRecord and InstanceRecord structures. In this version of the 'fvar' table, the InstanceRecord structure has an optional field, and so two different size formulations are possible. Future minor-version updates of the 'fvar' table may define compatible extensions to either formats. Implementations must use the axisSize and instanceSize fields to determine the start of each record.

The set of axes that make up the font's variation space are defined by an array of variation axis records. The number of records, and the number of axes, is determined by the axisCount field. A functional variable font must have an axisCount value that is greater than zero. If axisCount is zero, then the font is not functional as a variable font, and must be treated as a non-variable font; any variation-specific tables or data is ignored.

VariationAxisRecord

The format of the variation axis record is as follows:

VariationAxisRecord

Type	Name	Description
Tag	axisTag	Tag identifying the design variation for the axis.
Fixed	minValue	The minimum coordinate value for the axis.
Fixed	defaultValue	The default coordinate value for the axis.
Fixed	maxValue	The maximum coordinate value for the axis.
uint16	flags	Axis qualifiers – see details below.
uint16	axisNameID	The name ID for entries in the 'name' table that provide a display name for this axis.

Each axis has a tag that identifies the design variation for the axis. For example, the tag 'wght' designates a weight variation. Tags are registered for commonly-used design axes, but foundry-defined tags may also be used. Registered tags define valid ranges of coordinate values for the axis across all fonts. The variation axis record defines minimum and maximum values supported by the font, which may be more limited that the valid ranges defined for a registered tag.

NOTE Axis values given in the variation axis record use user scale coordinates that are specific to each axis tag. The user scale for each registered tag is described with the definition of each tag. In most other font tables that contain variation-related data, axis coordinate values are expressed using normalized coordinate scales. For more information regarding user scales and normalized scales, and a specification of the normalization process, see the "Coordinate Scales and Normalization" section in the OFF "Font Variations Overview".

Axis tags must conform to certain requirements to be valid. Also, the valid patterns for registered tags and for foundry-defined tags are mutually exclusive. This is required to ensure there is never a future conflict between foundry-defined tags in existing fonts and newly-registered tags.

For more details on axis tags and definitions for registered design-variation axes, see "Design-variations axis tags registry".

The default value interacts with the glyph and glyph variations tables in a particular way: the variation instance that has the default coordinate value for each axis will use glyph outlines as defined in the glyph table, without any variations from the glyph variations table applied. This instance is referred to as the default instance.

Flags can be assigned to indicate certain uses or behaviors for a given axis, independent of the specific axis tag and its definition. If no flags are set, then no assumptions are to be made beyond the definition for a registered axis. The following flags are defined.

Mask	Name	Description
0x0001	HIDDEN_AXIS	The axis should not be exposed directly in user interfaces.
0xFFFE	Reserved	Reserved for future use – set to 0.

The HIDDEN_AXIS flag is provided to indicate a recommendation by the font developer that the axis not be exposed directly to end users in application user interfaces. Reasons for setting this flag might include that the axis is intended only for programmatic interaction, or is intended for font-internal use by the font developer. If this flag is set, the axis should not be exposed to users in application user interfaces except in specialized scenarios, such as a font inspection utility. The flag should not affect handling of named instances, which should always be exposed in text-formatting user interfaces. If this flag is not set, then applications may expose the given axis in a default user interface or, based on the nature of the axis, may choose to expose it in an advanced-user interface.

The axisNameID field provides a name ID that can be used to obtain strings from the 'name' table that can be used to refer to the axis in application user interfaces. The name ID must be greater than 255 and less than 32768. For registered axis tags, a conventional US English axis name is provided; it is recommended that that name, or localized derivative names, be used in application user interfaces to provide greater consistency in user experience between different fonts.

InstanceRecord

The instance record format includes an array of n-tuple coordinate arrays that define position within the font's variation space. The n-tuple array has the following format:

Tuple Record (Fixed):

Type	Name	Description
Fixed	coordinates[axisCount]	Coordinate array specifying a position within the font's variation space.

The format of the instance record is as follows:

InstanceRecord:

Type	Name	Description
uint16	subfamilyNameID	The name ID for entries in the 'name' table that provide subfamily names for this instance.
uint16	flags	Reserved for future use – set to zero.
Tuple	coordinates	The coordinate array for this instance.
uint16	postscriptNameID	Optional. The name ID for entries in the 'name' table that provide PostScript names for this instance.

The postScriptNameID field is optional, but should be included in all variable fonts, and may be required in some platforms. Note that all of the instance records in a given font must be the same size, with all either including or omitting the postScriptNameID field.

The subfamilyNameID field provides a name ID that can be used to obtain strings from the 'name' table that can be treated as equivalent to name ID 17 (typographic subfamily) strings for the given instance. Values of 2 or 17 can be used; otherwise, values must be greater than 255 and less than 32768. The values 2 or 17 should only be used if the named instance corresponds to the font's default instance.

The postScriptNameID field provides a name ID that can be used to obtain strings from the 'name' table that can be treated as equivalent to name ID 6 (PostScript name) strings for the given instance. Values of 6 and 0xFFFF can be used; otherwise, values must be greater than 255 and less than 32768. If the value is 0xFFFF, then the value is ignored, and no PostScript name equivalent is provided for the instance. The value 6 should only be used if the named instance corresponds to the font's default instance.

All of the instance records in a font should have distinct coordinates and distinct subfamilyNameID and postScriptName ID values. If two or more records share the same coordinates, the same nameID values or the same postScriptNameID values, then all but the first can be ignored.

The default instance of a font is that instance for which the coordinate value of each axis is the defaultValue specified in the corresponding variation axis record. An instance record is not required for the default instance, though an instance record can be provided. When enumerating named instances, the default instance should be enumerated even if there is no corresponding instance record. If an instance record is included for the default instance (that is, an instance record has coordinates set to default values), then the nameID value should be set to either 2 or 17, and the postScriptNameID value should be set to 6.

NOTE Since an instance record for the default instance is not required, a variable font that has no instance records defined in the 'fvar' table (instanceCount is zero) still has one named instance.

7.3.3.2 Variation Instance Selection

When formatting text using a variable font, applications must select a particular variation instance; that is, specific, in-range values must be specified for each of the axes defined in the font variation table. An instance may be selected by reference to a named instanced defined in an instance record, or by using a set of arbitrary axis values for the various axes. If a value is not specified for any particular axis, the default value for that axis defined in the font is used. If an application specifies a value for an axis that is less than the minValue defined in the font, then minValue must be used. Similarly, if an application specifies a value greater than the maxValue defined in the font, then maxValue must be used.

7.3.3.3 Example

This example is for a hypothetical font with family name “SelawikV” that has two axes of variation, for weight and width. This table summarizes the description of the axes for the font:

Axis tag	Mininum value	Default value	Maximum value	Axis name ID
‘wght’	300	400	700	256
‘wdth’	62.5	100	150	257

This font also has the following named instances:

Instance subfamily name	Subfamily name ID	PostScript name	PostScript name ID	‘wght’ value	‘wdth’ value
Regular	258	SelawikV-Regular	262	400	100
Bold	259	SelawikV-Bold	263	700	100
Condensed	260	SelawikV-Condensed	264	400	75
Condensed Bold	261	SelawikV-CondensedBold	265	700	75

The 'fvar' table is constructed as follows:

Hex data	Field	Comment
	Header	
0001	majorVersion	
0000	minorVersion	
0010	axesArrayOffset	16 bytes – combined size of fields before the axes array
0002	countSizePairs	2 count/size pairs: axis, instance
0002	axisCount	2 axes ('wght', 'wdth')
0014	axisSize	Size of each variation axis record is 20 bytes.
0004	instanceCount	4 named instances.
000E	instanceSize	Size of instance records is 14 bytes.
	First variation axis record	
77676874	axisTag	Axis tag 'wght'.
012C0000	minValue	Minimum 'wght' value is 300 (Fixed format).
01900000	defaultValue	Default 'wght' value is 400.
02BC0000	maxValue	Maximum 'wght' value is 700.
0000	flags	
0100	axisNameID	Display names for axis use name ID 256
	Second variation axis record	
77647468	axisTag	Axis tag 'wdth'
003E8000	minValue	Minimum 'wdth' value is 62.5 (Fixed format).
00640000	defaultValue	Default 'wdth' value is 100.
00960000	maxValue	Maximum 'wdth' value is 150.
0000	flags	
0101	axisNameID	Display names for axis use name ID 257.
	First instance record	
0102	subfamilyNameID	Instance subfamily name "Regular" uses name ID 258.
0000	flags	

01900000	coordinates[0]	‘wght’ coordinate is 400.
00640000	coordinates[1]	‘wdth’ coordinate is 100.
0106	postscriptNameID	Instance PostScript name "SelawikV-Regular" uses name ID 262.
	Second instance record	
0103	subfamilyNameID	Instance subfamily name "Bold" uses name ID 259.
0000	flags	
02BC0000	coordinates[0]	'wght' coordinate is 700.
00640000	coordinates[1]	'wdth' coordinate is 100.
0107	postscriptNameID	Instance PostScript name "SelawikV-Bold" uses name ID 263.
	Third instance record	
0104	subfamilyNameID	Instance subfamily name "Condensed" uses name ID 260.
0000	flags	
01900000	coordinates[0]	'wght' coordinate is 400.
004B0000	coordinates[1]	'wdth' coordinate is 75.
0108	postscriptNameID	Instance PostScript name "SelawikV-Condensed" uses name ID 264.
	Fourth instance record	
0105	subfamilyNameID	Instance subfamily name "Condensed Bold" uses name ID 261.
0000	flags	
02BC0000	coordinates[0]	'wght' coordinate is 700.
004B0000	coordinates[1]	'wdth' coordinate is 75.
0109	postscriptNameID	Instance PostScript name "SelawikV-CondensedBold" uses name ID 265.

The total size of the table is 112 bytes.

7.3.4 gvar – Glyph variations table

Font Variations allow a font designer to incorporate multiple faces within a font family into a single font resource. In a variable font, the font variations ('fvar') table defines a set of design variations supported by the font, and then various tables provide data that specify how different font values, such as X-height or X and Y coordinates for glyph outline points, are adjusted for different variation instances. The glyph variations ('gvar') table provides all of the variation data that describe how TrueType glyph outlines in a 'glyf' table change across the font’s variation space.

The glyph variations table shall be used in combination with TrueType outlines — a glyph ('glyf') table – and also in combination with a font variations ('fvar') table and other required or optional tables used in variable fonts.

The 'gvar' table contains *glyph variation data* sub-tables with variation data for each glyph in the 'glyf' table. The glyph variation data is a specific variant of the *tuple variation store* format. Another variant of the tuple variation store is also used in the 'cvar' table.

Variation data is comprised of many adjustment-delta values. These deltas apply to particular target items, such as the X or Y coordinate of some glyph outline point, and are applicable for instances within a particular region of the font's variation space. The tuple variation store format organizes deltas into groupings by region of applicability, with a different group of data for each region. As glyph outlines often comprise the largest volume of data in a font, the tuple variation store format uses run-length encoding and other optimization mechanisms to provide efficient representation of the variation data.

Each region-specific grouping of data includes data covering all of the outline points for the given glyph. This means that the tuple variation store formats are suited to unpacking and processing delta values for all outline points at once, rather than for random outline points. In most application scenarios, glyph outline processing involves the entire glyph outline at once, so this bias in the format is generally not a particular limitation.

A notable exception, however, is the use of horizontal or vertical glyph metrics in text-layout operations that occur prior to rendering. The TrueType rasterizer dynamically generates "phantom" points for each glyph that represent horizontal and vertical advance widths and side bearings, and the variation data within the 'gvar' table includes data for these phantom points. (See "Instructing TrueType Glyphs" [24] for more background on phantom points.) Thus, a text-layout implementation could utilize the 'gvar' table to obtain interpolated glyph metrics for a given variation instance. Doing so, however, would require invocation of the rasterizer and processing of data for all outline points of each glyph rather than just the glyph-metric phantom points. As an alternative, the horizontal metrics variations ('HVAR') and vertical metrics variations ('VVAR') tables can provide variation data for glyph metrics that can be processed without invoking the rasterizer, and that use different formats that are better suited to processing data for particular items — advances or side bearings for specific glyphs. For this reason, it is recommended that variable fonts include an 'HVAR' table, and also a 'VVAR' table if the font has 'vhea' and 'vmtx' tables to support vertical layout.

7.3.4.1 Glyph variations table format

The glyph variations table is comprised of a header followed by GlyphVariationData subtables for each glyph that describe the ways that each glyph is transformed across the font's variation space.

Each glyph variation data table includes sets of data that reference various regions within the font's variation space. Each region is defined using one or three tuple records, with a "peak" tuple record required. In many cases, a region referenced by one glyph will also be referenced by many other glyphs. As an optimization, the 'gvar' table allows for a shared set of tuple records that can be referenced by the tuple variation store data for any glyph.

The high-level structure of the 'gvar' table is as follows:

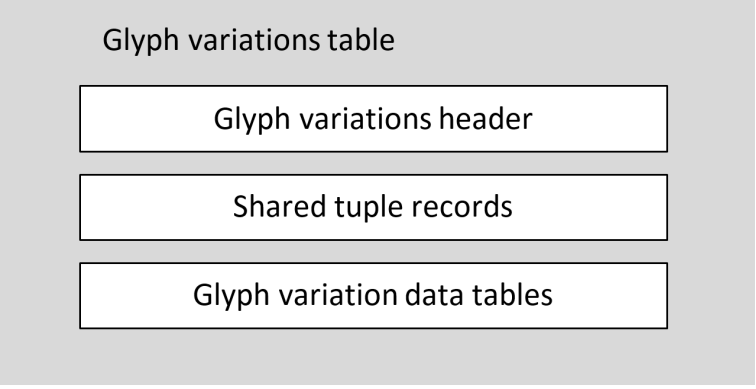


Figure 7.8: High-level organization of 'gvar' table

The header includes offsets to the start of the shared tuples data, and to the start of the glyph variation data tables.

Each glyph variation data table provides variation data for a particular glyph. These are variable in size. For this reason, the header also includes an array of offsets for each glyph variation data table from the start of the glyph variation data table array. There is one offset corresponding to each glyph ID, plus one extra offset. (Note that the same scheme is also used in the index to location ('loca') table.) The difference between two consecutive offsets in the array indicates the size of a given table, with an extra offset in the array to indicate the size of the last table. Some sizes derived in this way may be zero, in which case there is no glyph variation data for that particular glyph, and the same outline is used for that glyph ID across the entire variation space.

7.3.4.1.1 'gvar' header

The glyph variations table header format is as follows:

'gvar' header

Type	Name	Description
uint16	majorVersion	Major version number of the glyph variations table – set to 1.
uint16	minorVersion	Minor version number of the glyph variations table – set to 0.
uint16	axisCount	The number of variation axes for this font. This must be the same number as axisCount in the 'fvar' table.
uint16	sharedTupleCount	The number of shared tuple records. Shared tuple records can be referenced within glyph variation data tables for multiple glyphs, as opposed to other tuple records stored directly within a glyph variation data table.
Offset32	sharedTuplesOffset	Offset from the start of this table to the shared tuple records.
uint16	glyphCount	The number of glyphs in this font. This must match the number of glyphs stored elsewhere in the font.
uint16	flags	Bit-field that gives the format of the offset array that follows. If bit 1 is clear, the offsets are uint16; if bit 1 is set, the offsets are uint32.
Offset32	glyphVariationDataArrayOffset	Offset from the start of this table to the array of GlyphVariationData tables.
Offset16 or Offset32	glyphVariationDataOffsets [glyphCount + 1]	Offsets from the start of the GlyphVariationData array to each GlyphVariationData table.

If the short format (uint16) is used for offsets, the value stored is the offset divided by 2. Hence, the actual offset for the location of the GlyphVariationData table within the font will be the value stored in the offsets array multiplied by 2.

7.3.4.1.2 Shared tuples array

The shared tuples array provides a set of variation-space positions that can be referenced by variation data for any glyph. The shared tuples array follows the glyphVariationData offsets array at the end of the 'gvar' header. This data is simply an array of tuple records, each representing a position in the font's variation space.

Shared tuples array:

Type	Name	Description
TupleRecord	sharedTuples[sharedTupleCount]	Array of tuple records shared across all glyph variation data tables.

Tuple records that are in the shared array or that are contained directly within a given glyph variation data table use 2.14 values to represent normalized coordinate values.

7.3.4.1.3 The glyphVariationData table array

The glyphVariationData table array follows the 'gvar' header and shared tuples array. Each glyphVariationData table describes the variation data for a single glyph in the font.

The glyph variation data table is a specific form of the tuple variation store format. It is comprised of a header, followed by serialized data.

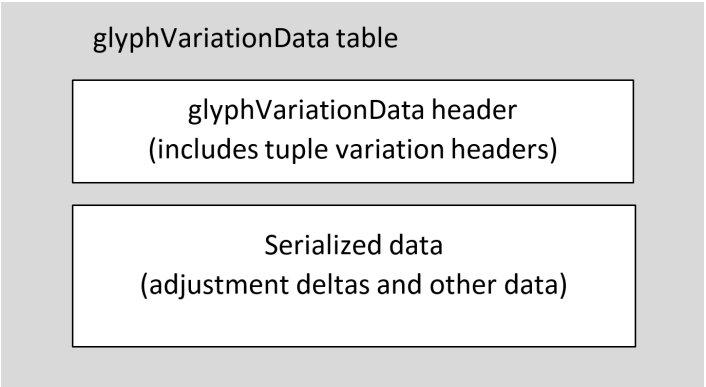


Figure 7.9: High-level organization of glyphVariationData table

The variation data includes logical groupings of data that apply to different regions of the variation space – *tuple variation data* tables. These logical groupings are stored in two parts: a header, and serialized data. The glyph variation data header includes an array of tuple variation-data headers, each of which is associated with a particular portion of the serialized data.

The serialized data includes adjustment delta values and also packed “point” numbers that identify the points in the glyph outline to which the deltas apply. In the case of a composite glyph, these numbers are component indices rather than point number indices. The serialized data for a given region can include point number data that applies to that specific tuple-variation data, but there can also be a shared set of point number data, stored at the start of the serialized data. Shared point number data can be used in relation to multiple tuple-variation data tables for the given glyph.

The glyphVariationData header is structured as follows:

GlyphVariationData header:

Type	Name	Description
uint16	tupleVariationCount	A packed field. The high 4 bits are flags, and the low 12 bits are the number of tuple variation tables for this glyph. The number of tuple variation tables can be any number between 1 and 4095.
Offset16	dataOffset	Offset from the start of the GlyphVariationData table to the serialized data
TupleVariationHeader	tupleVariationHeaders [tupleCount]	Array of tuple variation headers.

7.3.4.2 Processing the 'gvar' table

When processing glyph data in a variable font for a particular variation instance, default glyph outline data will be obtained from 'glyf' table, which is combined with the corresponding glyph variation data subtable within the 'gvar' table.

The tuple variation headers within the selected glyph variation data table will each specify a particular region of applicability within the font's variation space. These will be compared with the coordinates for the selected variation instance to determine which of the tuple-variation data tables are applicable, and to calculate a scalar value for each. These comparisons and scalar calculations are done using normalized-scale coordinate values.

For each of the tuple-variation data tables that are applicable, the point number and delta data will be unpacked and processed. The data for applicable regions can be processed in any order. Derived delta values will correspond to particular point numbers derived from the packed point number data. For a given point number, the computed scalar is applied to the X coordinate and Y coordinate deltas as a coefficient, and then resulting delta adjustments applied to the X and Y coordinates of the point.

There are two aspects of processing that are specific to the 'gvar' table. First, in the case of composite glyphs, point numbers refer either to components or to phantom points. Adjustments of phantom points are handled as for regular points, but adjustments to components are handled differently. Additional information regarding processing of variation data for composite glyphs is provided below.

Secondly, within a given tuple-variation data table, deltas may be provided for all of the glyph's points (and phantom points), or only for some points. If deltas are provided for only some point numbers in a glyph outline, then delta values for un-referenced points may need to be inferred using interpolation. This is additional processing specific to the 'gvar' table, and is described below.

7.3.4.3 Point Numbers and processing for composite glyphs

The general discussion above and in the description of interpolation in the Overview chapter assumes simple glyphs. Certain special considerations apply to composite glyphs.

The variation data for composite glyphs also use packed point number data representing a series of numbers, but the numbers in this case, apart from the last four "phantom" point numbers, refer to the components that make up the glyph rather than to outline points. The glyph components of a composite glyph are assigned these pseudo-point numbers according to the order the components are given in the glyph entry, starting with 0. The four phantom point numbers representing side bearings are still added at the end, as for simple glyphs.

For example, consider a composite glyph for "é" made up of two components: the base glyph "e", and a glyph for the accent, positioned at a specified offset. Pseudo- and phantom point numbers would be as follows:

Point number	Element
0	Base glyph
1	Accent glyph
2	Left side bearing point
3	Right side bearing point
4	Top side bearing point
5	Bottom side bearing point

Packed point number data for this glyph could include numbers referencing any or all of these elements.

The adjustment deltas for component glyphs adjust the placement of the component. If a glyph component is not referenced in the packed point numbers, then its position is not adjusted. Each component glyph within a

composite will have its own glyph entry that, itself, has its own variation data. The processing of component glyphs must begin with the most deeply-nested, non-composite glyphs.

The position of a component can be represented in one of two ways: directly using X and Y offset values, or indirectly using point numbers in the parent and component glyphs that get aligned. Which method is used is determined by a bit in the flags field of a composite glyph description: if ARGS_ARE_XY_VALUES (bit 1) is set, then X and Y offsets are used; if that bit is clear, then point numbers are used. If the position of a component is represented using X and Y offsets – the ARGS_ARE_XY_VALUES flag is set – then adjustment deltas can be applied to those offsets. However, if the position of a component is represented using point numbers – the ARGS_ARE_XY_VALUES flag is not set – then adjustment deltas have no effect on that component and should not be specified.

In addition to a component's placement, the composite glyph description can also specify a scale value or a 2 × 2 matrix transformation to be applied to the component. Adjustment deltas do not have any effect on scaling or 2 × 2 transformations applied to a component. The deltas only affect the positioning of the component.

If any component entry in a composite glyph has the USE_MY_METRICS flag set, then the hinted glyph metrics for the composite as a whole are taken from that component, rather than using the values given for the composite glyph itself. That is, the positions of phantom points for the composite glyph are set to the hinted positions of the component's phantom points. If more than one component glyph has this flag set, then the metrics for the composite glyph are taken from the last component that has this flag set. In a variable font, if a component entry has this flag set, then the phantom point positions for the composite glyph are set to the interpolated and hinted positions of the component's phantom point, and delta values for the composite glyph's phantom points are not used.

NOTE When a composite glyph has a component with the USE_MY_METRICS flag set, the 'hmtx' and 'HVAR' data for the composite glyph are used in the same manner in which the 'hmtx' data would be used for a non-variable font. The 'hmtx' and 'HVAR' data should be set to appropriate values for the composite glyph, though the hinted phantom point positions may not exactly match the linearly-scaled metrics obtained from the 'hmtx' and 'HVAR' data.

The process for transforming a composite glyph is as follows:

Let componentCount be the number of components in the composite glyph. Let components[] be the component descriptions of a composite glyph (base 0 index), and let C be a component entry in a composite glyph description.

Using the standard variation interpolation algorithm, process the variation data for the glyph to obtain net X and Y delta adjustments for each of the components and phantom points.

Apply the net X and Y delta adjustments to the phantom points (p = componentCount to componentCount + 3; note that the phantom point positions may be later overridden by a component):

For AW and LSB points, apply the net X delta adjustments, and ignore Y deltas.

For AH and TSB points, apply the net Y delta adjustments, and ignore X deltas.

For p = 0 to componentCount - 1:

C = components[p]

For the glyph at C.glyphIndex, apply the variation interpolation process and run the hinting program.

If ARGS_ARE_XY_VALUES is set in C.flags:

Apply net X and Y delta adjustments for index p to the X and Y positioning offsets for C:

X position offset = C.argument1 + netAdjustmentX
Y position offset = C.argument2 + netAdjustmentY

Else (ARGS_ARE_XY_VALUES is not set):

Ignore any delta adjustments provided for this component.

Apply scaling to C as applicable — if positioning offsets are scaled, it is the delta-adjusted offsets that are scaled.

If USE_MY_METRICS is set in C.flags:

Set the positions of the composite's phantom points equal to the hinted positions of the phantom points for C.

Loop (next p)

For example, consider glyph 128 of the Skia font, which is the glyph for "Ä". The glyph entry has two component entries, both with ARGS_ARE_XY_VALUES set. The advance width in the 'hmtx' table is 1358, and the left side bearing is 16; this puts left side bearing and right side bearing phantom points (point indices 2 and 3) at (0, 0) and (1358, 0). Relevant details from the composite glyph description are as follows:

Component	glyphIndex	argument1 (X offset)	argument2 (Y offset)
0	65	0	0
1	315	286	0

Glyph 128 has tuple variation data for various regions within the variation space; three will be considered for this example, with normalized coordinate positions as follows:

Region	(weight, width)
R1	(1, 0)
R2	(0, 1)
R3	(1, 1)

The data for R1 the following deltas:

	pt 0	pt 1	pt 2	pt 3
X	0	69	58	145
Y	0	0	0	0

The data for R2 the following deltas:

	pt 0	pt 1	pt 2	pt 3
X	0	53	38	351
Y	0	0	0	0

The data for R3 the following deltas:

	pt 0	pt 1	pt 2	pt 3
X	0	21	-6	25
Y	0	0	0	0

As all of the Y delta values are zero, there is no adjustment to Y coordinate values.

Now consider a variation instance with coordinates (0.2, 0.7). Variation data for all three of the regions above will have an effect, with scalar values as follow:

Region	Scalar
R1	0.2
R2	0.7
R3	0.14

The net adjustment to X coordinate values for component offsets and phantom points will be as follows:

Item	Derived X coordinate adjustment
Component 1 X offset	0.2 × 69 + 0.7 × 53 + 0.14 × 21 = 53.84
Left side bearing point X coordinate	0.2 × 58 + 0.7 × 38 + 0.14 × -6 = 37.36
Right side bearing point X coordinate	0.2 × 145 + 0.7 × 351 + 0.14 × 25 = 278.2

Combining these adjustments with the default values in the 'glyf' entry yields the following:

Component 0 (X, Y) offset	(0, 0)
Component 1 (X, Y) offset	(286 + 53.84, 0) = (339.84, 0)
Left side bearing phantom point	(0 + 37.36, 0) = (37.36, 0)
Right side bearing phantom point	(1358 + 278.2, 0) = (1636.2, 0)

7.3.4.4 Inferred deltas for un-referenced point numbers

The tuple variation data for a given glyph and region (a given tuple variation data set) may include deltas for all outline points, or for only some. The packed point number data identify the points for which deltas are provided. If some points are omitted from the list of point numbers, then the data does not explicitly include delta values for them, and deltas may need to be inferred. This is done for a given glyph on a region-by-region basis based on the point numbers specified with each set of tuple variation data.

NOTE Inferring of deltas for un-referenced points applies only to simple glyphs, not to composite glyphs.

A single set of point-number data is used for both X- and Y-direction deltas. If a point has explicit deltas, then it has explicit deltas for both X and Y directions. If the point requires inferred deltas, then both X and Y deltas are inferred. The values of inferred X and Y deltas are calculated separately.

The scalar calculated for a given region and variation instance is applied to the inferred deltas to obtain scaled delta adjustments that are applied to the point coordinates, just as for explicit deltas.

The process of calculating inferred variation deltas is somewhat comparable to the TrueType *Interpolate Untouched Points (IUP)* instruction. Both calculate an adjustment for an unaffected point based on the adjustments to adjacent, affected points. In the case of the IUP instruction, however, the calculated adjustment is based on the position of adjacent points *after other instructions have applied*. In contrast, calculation of inferred variation deltas is based on the *default positions* of points and the unscaled delta values for a given region. It is not impacted by the order in which the tuple-variation data for different regions is processed, and the inferred deltas can be pre-computed before any processing for a specific instance is done. Also, the calculations used for deriving the value of inferred deltas are slightly different from the calculations used for the IUP instruction.

Calculation of inferred deltas is done for a given glyph and a given region on a contour-by-contour basis.

For a given contour, if the point number list does not include any of the points in that contour, then none of the points in the contour are affected and no inferred deltas need to be computed.

If the point number list includes some but not all of the points in a given contour, then inferred deltas must be derived for the points that were not included in the point number list, as follows.

First, for any un-referenced point, identify the nearest points before and after, in point number order, that are referenced. Note that the same referenced points will be used for calculating both X and Y inferred deltas. If there is no lower point number from that contour that was referenced, then the highest, referenced point number from that contour is used. Similarly, if no higher point number from that contour was referenced, then the lowest, referenced point number is used.

Once the adjacent, referenced points are identified, then inferred-delta calculation is done separately for X and Y directions.

Next, the (X or Y) grid coordinate values of the adjacent, referenced points are compared. If these coordinates are the same, then the delta values for the adjacent points are compared: if the delta values are the same, then this value is used as the inferred delta for the target, un-referenced point. If the delta values are different, then the inferred delta for the target point is zero.

NOTE If exactly one point from the contour is referenced in the point number list, then every point in that contour uses the same X and Y delta values as that point. This follows as a specific case of the above: for all other points that are not referenced, the one referenced point is at once both the preceding adjacent point and the following adjacent point. Hence, the adjacent points have the same coordinate value and the same delta, and therefore the un-referenced points get an inferred delta of the same value.

If the coordinates of the adjacent, reference points are different, then the coordinate for the same (X or Y) direction of the target point is compared to those coordinates. If the coordinate of the target point is between the coordinates of the adjacent points, then a delta is interpolated, as described below. But if the coordinate of the target point is not between the coordinates of the adjacent points, then the inferred delta is the delta for whichever of the adjacent points is closer in the given direction.

The following pseudo-code summarizes the above details.

```
if precedingPoint.coord = followingPoint.coord
{
    if precedingPoint.delta = followingPoint.delta
        targetPoint.delta = precedingPoint.delta
    else
        targetPoint.delta = 0
}
else  /* precedingPoint.coord <> followingPoint.coord */
{
    if targetPoint.coord <= min(precedingPoint.coord, followingPoint.coord)
    {
        if precedingPoint.coord < followingPoint.coord
            targetPoint.delta = precedingPoint.delta
        else  /* followingPoint.coord < precedingPoint.coord */
            targetPoint.delta = followingPoint.delta
    }
    else if targetPoint.coord >= max(precedingPoint.coord, followingPoint.coord)
    {
        if precedingPoint.coord > followingPoint.coord, then
            targetPoint.delta = precedingPoint.delta
        else  /* followingPoint.coord > precedingPoint.coord */
            targetPoint.delta = followingPoint.delta
    }
    else  /* target point coordinate is between adjacent point coordinates */
    {
```



```
            /* target point delta is derived from the adjacent point deltas
               using linear interpolation */
        }
    }
```

When the coordinates of the two adjacent points are different and the coordinate of the target point is between those coordinates, then a delta for the target point is computed by linear interpolation of the deltas for the two adjacent points. The following describes this calculation for either X or Y direction.

NOTE The logical flow of the algorithm to this point implies that the coordinates of the two adjacent points are different. This avoids a division by zero in the following calculations that would otherwise occur.

Select one of the two adjacent points as the reference point, and let the other be the comparison point. It doesn't matter which is which. Let **refCoord** be the grid coordinate for the current direction of the former, and let **compCoord** be the grid coordinate for the latter. Let **targetCoord** be the grid coordinate for the target, un-referenced point. Calculate a proportion, **proportion**, as follows:

$$\text{proportion} = \frac{\text{targetCoord} - \text{refCoord}}{\text{compCoord} - \text{refCoord}}$$

Now let **deltaRef** and **deltaComp** be the unscaled adjustment-delta values in the variation data for the reference and comparison points. The inferred delta for the target point, **deltaTarget**, is calculated as follows:

$$\text{deltaTarget} = (1 - \text{proportion}) \times \text{deltaRef} + \text{proportion} \times \text{deltaComp}$$

The following example illustrates the process for obtaining inferred deltas. Suppose P1, P2 and P3 are points in the same contour, with coordinate positions as shown below, and that P1 and P3 are referenced in the point number data while P2 is not.

Note that P1 and P3 have different X and Y coordinates. Also note that, in the X direction, P2 is between the two adjacent points, while in the Y direction it is not. For the Y direction, the inferred delta will be the delta of the closer point, P3.

For the X direction, let P1 be the reference point; P3 is the comparison point. A proportion is calculated:

$$\text{proportion} = \frac{260 - 245}{305 - 245} = 0.25$$

Now suppose that P1 and P3 have X and Y deltas in the variation data as follows:

Point	X delta	Y delta
P1	+28	-62
P3	-42	-57

The inferred X delta for P2, deltaX_P2, is calculated as follows:

$$\text{deltaX}_{P2} = (1 - 0.25) \times 28 + 0.25 \times -42 = 10.5$$

The inferred Y delta for P2 is the value of the Y delta for P3. Thus, the deltas for all three points is obtained:

Point	X delta	Y delta
P1	+28	-62
P2	+10.5	-57
P3	-42	-57

These delta values can now be used in the interpolation algorithm, with a scalar applied to each based on the region coordinates and the coordinates for the current variation instance.

7.3.5 HVAR – Horizontal metrics variations table

The 'HVAR' table is used in variable fonts to provide variations for horizontal glyph metrics values. This can be used to provide variation data for advance widths in the ‘hmtx’ table. In fonts with TrueType outlines, it can also be used to provide variation data for left and right side bearings obtained from the ‘hmtx’ table and glyph bounding box.

For a general overview of OFF Font variations and terminology related to variations, see subclause 7.1.

In a font with TrueType outlines, the rasterizer will generate “phantom” points that represent left, right, top and bottom side bearings. (See "Instructing TrueType Glyphs" [24] for more background on phantom points.) In a TrueType variable font, the glyph variations (‘gvar’) table will include variation data for the phantom points of each glyph, allowing glyph metrics to be interpolated for different variation instances as part of deriving the interpolated glyph outlines. For this reason, the 'HVAR' table is not required in variable fonts that have TrueType outlines. For text-layout operations that require glyph metrics but not actual glyph outlines, however, there can be significant performance benefits by being able to obtain adjusted glyph metrics for an instance without needing to interpolate glyph outlines. For this reason, it is recommended that an 'HVAR' table be included in variable fonts that have TrueType outlines.

The CFF2 rasterizer does not generate phantom points as in the TrueType rasterizer. For this reason, an 'HVAR' table is required to handle any variation in horizontal glyph metrics in a variable font with CFF2 outlines.

Take note that, in a variable font with TrueType outlines, the left side bearing for each glyph must equal xMin, and bit 1 in the flags field of the 'head' table must be set.

The 'HVAR' table contains an *item variation store* table to represent variation data. The item variation store and constituent formats are described in subclause 7.2. The item variation store is also used in the 'VVAR', 'GDEF' and certain other tables, but is different from the formats for variation data used in the 'cvar' or 'gvar' tables.

The item variation store format uses *delta-set indices* to reference variation delta data for particular target, font-data items to which they are applied, such as the advance width of a particular glyph. Data external to the item variation store identifies the delta-set index to be used for each given target item. Within the 'HVAR' table, glyph IDs can be used as implicit indices for advance width variations, or an optional delta-set index mapping table can be used that explicitly provides delta-set indices for advance widths to be associated with each glyph ID.

An advance width mapping table adds additional data within the 'HVAR' table, but it also makes it possible to use a more compact representation of the data in the item variation store. For example, if multiple glyphs have the same advance widths, the mapping table allows all of them to reference a single delta set within the store. Additional optimizations within the item variation store are possible. See the Common Table Formats chapter for more discussion about size optimization. In general, inclusion of an advance width mapping is recommended.

Optional mapping tables can also be used to provide delta-set indices for glyph side bearings. In variable fonts with TrueType outlines, variation data for side bearings is recommended. If variation data for side bearings is provided, it should include data for both left and right side bearings. Mapping tables for left and right side bearings must also be included.

7.3.5.1 Related and co-requisite tables

The 'HVAR' table is used only in variable fonts. It shall be used in combination with a horizontal metrics ('hmtx') table, and also in combination with a font variations ('fvar') table, and other required or optional tables used in variable fonts. See subclause 7.1.6 in the OFF Font variations overview for general information.

For variable fonts that have TrueType outlines, the 'HVAR' table is optional but recommended. For variable fonts that have CFF2 outlines, the 'HVAR' table is required if there is any variation in glyph advance widths across the variation space.

NOTE The 'hdmx' table is not used in variable fonts.

7.3.5.2 Table formats

The horizontal metrics variations table has the following format:

Horizontal metrics variations table

Type	Name	Description
uint16	majorVersion	Major version number of the metrics variations table – set to 1.
uint16	minorVersion	Minor version number of the metrics variations table – set to 0.
Offset32	itemVariationStoreOffset	Offset in bytes from the start of this table to the item variation store table.
Offset32	advanceWidthMappingOffset	Offset in bytes from the start of this table to the delta-set index mapping for advance widths (may be NULL).
Offset32	lsbMappingOffset	Offset in bytes from the start of this table to the delta-set index mapping for left side bearings (may be NULL).
Offset32	rsbMappingOffset	Offset in bytes from the start of this table to the delta-set index mapping for right side bearings (may be NULL).

The item variation store table is documented in subclause 7.2.

Mapping tables are optional. If a given mapping table is not provided, the offset is set to NULL.

Variation data for advance widths is required. A delta-set index mapping table for advance widths can be provided, but is optional. If a mapping table is not provided, glyph indices are used as implicit delta-set indices. To access the delta set for the advance of given glyph, the delta-set outer-level index is zero, and the glyph ID is used as the inner-level index.

Variation data for side bearings are optional. If included, mapping tables are required to provide the delta-set index for each glyph.

The delta-set index mapping table has the following format:

DeltaSetIndexMap table

Type	Name	Description
uint16	entryFormat	A packed field that describes the compressed representation of delta-set indices. See details below.
uint16	mapCount	The number of mapping entries.
uint8	mapData[variable]	The delta-set index mapping data. See details below.

The mapCount field indicates the number of delta-set index mapping entries. Glyph IDs are used as the index into the mapping array. If a given glyph ID is greater than mapCount – 1, then the last entry is used.

Each mapping entry represents a delta-set outer-level index and inner-level index combination. Logically, each of these indices is a 16-bit, unsigned value. These are represented in a packed format that uses one, two, three or four bytes. The entryFormat field is a packed field that describes the compressed representation used in the mapData field of the given deltaSetIndexMap table. The format of the entryFormat field is as follows:

EntryFormat Field Masks

Mask	Name	Description
0x000F	INNER_INDEX_BIT_COUNT_MASK	Mask for the low 4 bits, which give the count of bits minus one that are used in each entry for the inner-level index.
0x0030	MAP_ENTRY_SIZE_MASK	Mask for bits that indicate the size in bytes minus one of each entry.
0xFFC0	Reserved	Reserved for future use – set to 0.

The size of each mapping entry is ((entryFormat & mapEntrySizeMask) >> 4 + 1). The total size of the map data array is entrySize * mapCount.

For a given entry, the outer-level and inner-level indices can be obtained as follows:

outerIndex = entry >> (entryFormat & innerIndexBitCountMask)

innerIndex = entry & ((1 << (entryFormat & innerIndexBitCountMask)) – 1)

7.3.5.3 Processing

When performing text layout using a particular variation instance of a variable font, the application will need to obtain adjusted glyph metrics for that instance. The application obtains default values from the 'hmtx' and 'glyf' tables, and uses the 'HVAR' table to obtain interpolated adjustment values that are applied to the defaults.

Delta-set indices are obtained based on the glyph ID. If there is no delta-set index mapping table for advance widths, then glyph IDs implicitly provide the indices: for a given glyph ID, the delta-set outer-level index is zero, and the glyph ID is the delta-set inner-level index.

If delta-set index mappings are provided, the outer- and inner-level indices are combined in a packed format that uses one to four bytes for each mapping entry. Each mapping table provides information describing the packed format; this is used to extract the separate outer- and inner-level indices.

The indices are used to reference a delta set within the item variation store for the target advance width or side bearing. The two-level organization of data within the item variation store is described in subclause 7.2. Each delta set includes different deltas that apply to variation instances falling within different regions of the

variation space. The process by which the deltas are processed to derive an interpolated value for a given target item is described in subclause 7.1.

7.3.6 MVAR – Metrics variations table

The metrics variations table is used in variable fonts to provide variations for font-wide metric values found in the OS/2 table and other font tables. For a general overview of OFF Font Variations and terminology related to variations, see subclause 7.1.

The metrics variations table contains an *item variation store* structure to represent variation data. The item variation store and constituent formats are described in subclause 7.2. The item variation store is also used in the 'HVAR' and 'GDEF' tables, but is different from the formats for variation data used in the 'cvar' or 'gvar' tables.

The item variation store format uses *delta-set indices* to reference variation delta data for particular target font-data items to which they are applied. Data external to the item variation store identifies the delta-set index to be used for each given target item. Within the 'MVAR' table, an array of *value tag* records identifies a set of target items, and provides the delta-set index used for each. The target items are identified by four-byte tags, with a given tag representing some font-wide value found in another table. For example, the tag 'hasc' represents the OS/2.sTypoAscender value. More details on tags are provided below.

The item variation store format uses a two-level organization for variation data: a store can have multiple *item variation data* subtables, and each subtable has multiple delta-set rows. A delta-set index is a two-part index: an outer index that selects a particular item variation data subtable, and an inner index that selects a particular delta-set row within that subtable. A value record specifies both the outer and inner portions of the delta-set index.

NOTE Apple platforms allow for use of a font metrics ('fmtx') table to specify various font-wide metric values by reference to the X or Y coordinates of contour points for a specified “magic” glyph. OFF Font variations does not use the font metrics table.

The metrics variations table shall be used in combination with a font variations ('fvar') table and other required or optional tables used in variable fonts. See "Variation data tables and miscellaneous requirements" for additional details.

7.3.6.1 Table formats

The metrics variations table has the following format:

Metrics variations table

Type	Name	Description
uint16	majorVersion	Major version number of the metrics variations table – set to 1.
uint16	minorVersion	Minor version number of the metrics variations table – set to 0.
uint16	(reserved)	Not used; set to 0.
uint16	valueRecordSize	The size in bytes of each value record.
uint16	valueRecordCount	The number of value records.
Offset16	itemVariationStoreOffset	Offset in bytes from the start of this table to the item variation store table. If valueRecordCount is zero, set to zero; if valueRecordCount is greater than zero, must be greater than zero.

ValueRecord	valueRecords[valueRecordCount]	Array of value tag records that identify target items and the associated delta-set index for each. The valueTag records must be in binary order of their valueTag field.

The valueRecordSize field indicates the size of each value record. Future, minor version updates of the 'MVAR' table may define compatible extensions to the value record format with additional fields. Implementations shall use the valueRecordSize field to determine the start of each record.

The valueRecords array is an array of value records that identify the target, font-wide measures for which variation adjustment data is provided (target items), and outer and inner delta-set indices for each item into the item variation store data.

ValueRecord:

Type	Name	Description
Tag	valueTag	Four-byte tag identifying a font-wide measure.
uint16	deltaSetOuterIndex	A delta-set outer index — used to select an item variation data subtable within the item variation store.
uint16	deltaSetInnerIndex	A delta-set inner index — used to select a delta-set row within an item variation data subtable.

The value records must be given in binary order of the valueTag values. Each tag identifies a font-wide measure found in some other font table. For example, if a value record has a value tag of 'hasc', this corresponds to the OS/2.sTypoAscender field. Details on the tags used within the 'MVAR' table are provided below.

7.3.6.2 Processing

When reading a value within a variable font, such as the OS/2.sCapHeight value (the target item), the value tags array in the metrics variations table is scanned to find the tag that corresponds to that target item. Records in the array are stored in binary order of values in the valueTag fields. If the tag does not occur in the tag array, then the item is constant across the font's variation space. If the tag does occur, however, then the delta-set index is used to reference a set of deltas within the item variation store. The two-level organization of data within the item variation store is described in subclause 7.2. Each delta set includes different deltas that apply to variation instances falling within different regions of the variation space. The process by which the deltas are processed to derive an interpolated value for a given target item is described in the OFF Font variations overview.

7.3.6.3 Value tags

Four-byte tags are used to represent particular metric or other values. For example, the tag 'hasc' (horizontal ascent) is used to represent the OS/2.sTypoAscender value. Tags are defined for various values found in the OS/2 and Windows metrics ('OS/2') table, the horizontal header ('hhea') table, the grid-fitting and scan-conversion ('gasp') table, the PostScript ('post') table, and the vertical metrics header ('vhea') table.

NOTE The OS/2.usWeightClass, OS/2.usWidthClass and post.italicAngle values are not supported by variation data in the 'MVAR' table. This is because values for these three fields correspond directly to input axis values for the 'wght', 'wdth' and 'slnt' variation axes. See the discussion of these axes in the "Design-variation axis tags registry" for details on the relationship between these fields and the corresponding design axes.

Tags in the metrics variations table are case sensitive. Tags defined in this table use only lowercase letters or digits.

Tags that are used in a font's metrics variations table should be those that are documented in this table specification. A font may also use privately-defined tags, which have semantics known only by private

agreement. Private-use tags must begin with an uppercase letter and use only uppercase letters or digits. If a private-use tag is used in a given font, any application that does not recognize that tag should ignore it.

The following tags are defined:

Value tags, ordered by logical groupings

Value	Mnemonic	Value represented
'hasc'	Horizontal ascender	OS/2.sTypoAscender
'hdsc'	Horizontal descender	OS/2.sTypoDescender
'hlgp'	Horizontal line gap	OS/2.sTypoLineGap
'hcla'	Horizontal clipping ascent	OS/2.usWinAscent
'hcld'	Horizontal clipping decent	OS/2.usWinDescent
'vasc'	Vertical ascender	vhea.ascent
'vdsc'	Vertical descender	vhea.descent
'vlgp'	Vertical line gap	vhea.lineGap
'hcrs'	Horizontal caret rise	hhea.caretSlopeRise
'hcrn'	Horizontal caret run	hhea.caretSlopeRun
'hcof'	Horizontal caret offset	hhea.caretOffset
'vcrs'	Vertical caret rise	vhea.caretSlopeRise
'vcrn'	Vertical caret run	vhea.caretSlopeRun
'vcof'	Vertical caret offset	vhea.caretOffset
'xhgt'	X height	OS/2.sxHeight
'cpht'	Cap height	OS/2.sCapHeight
'sbxs'	Subscript em x size	OS/2.ySubscriptXSize
'sbys'	Subscript em y size	OS/2.ySubscriptYSize
'sbxo'	Subscript em x offset	OS/2.ySubscriptXOffset
'sbyo'	Subscript em y offset	OS/2.ySubscriptYOffset
'spxs'	Superscript em x size	OS/2.ySuperscriptXSize
'spys'	Superscript em y size	OS/2.ySuperscriptYSize
'spxo'	Superscript em x offset	OS/2.ySuperscriptXOffset
'spyo'	Superscript em y offset	OS/2.ySuperscriptYOffset
'strs'	Strikeout size	OS/2.yStrikeoutSize
'stro'	Strikeout offset	OS/2.yStrikeoutPosition

'unds'	Underline size	post.underlineThickness
'undo'	Underline offset	post.underlinePosition
'gsp0'	gaspRange[0]	gasp.gaspRange[0].rangeMaxPPEM
'gsp1'	gaspRange[1]	gasp.gaspRange[1].rangeMaxPPEM
'gsp2'	gaspRange[2]	gasp.gaspRange[2].rangeMaxPPEM
'gsp3'	gaspRange[3]	gasp.gaspRange[3].rangeMaxPPEM
'gsp4'	gaspRange[4]	gasp.gaspRange[4].rangeMaxPPEM
'gsp5'	gaspRange[5]	gasp.gaspRange[5].rangeMaxPPEM
'gsp6'	gaspRange[6]	gasp.gaspRange[6].rangeMaxPPEM
'gsp7'	gaspRange[7]	gasp.gaspRange[7].rangeMaxPPEM
'gsp8'	gaspRange[8]	gasp.gaspRange[8].rangeMaxPPEM
'gsp9'	gaspRange[9]	gasp.gaspRange[9].rangeMaxPPEM

Value tags, in alphabetical order of tags:

Value	Mnemonic	Value represented
'cpht	Cap height	OS/2.sCapHeight
'gsp0'	gaspRange[0]	gasp.gaspRange[0].rangeMaxPPEM
'gsp1'	gaspRange[1]	gasp.gaspRange[1].rangeMaxPPEM
'gsp2'	gaspRange[2]	gasp.gaspRange[2].rangeMaxPPEM
'gsp3'	gaspRange[3]	gasp.gaspRange[3].rangeMaxPPEM
'gsp4'	gaspRange[4]	gasp.gaspRange[4].rangeMaxPPEM
'gsp5'	gaspRange[5]	gasp.gaspRange[5].rangeMaxPPEM
'gsp6'	gaspRange[6]	gasp.gaspRange[6].rangeMaxPPEM
'gsp7'	gaspRange[7]	gasp.gaspRange[7].rangeMaxPPEM
'gsp8'	gaspRange[8]	gasp.gaspRange[8].rangeMaxPPEM
'gsp9'	gaspRange[9]	gasp.gaspRange[9].rangeMaxPPEM
'hasc'	Horizontal ascender	OS/2.sTypoAscender
'hcla'	Horizontal clipping ascent	OS/2.usWinAscent
'hcld'	Horizontal clipping decent	OS/2.usWinDescent
'hcof'	Horizontal caret offset	hhea.caretOffset

'hcrn'	Horizontal caret run	hhea.caretSlopeRun
'hcrs'	Horizontal caret rise	hhea.caretSlopeRise
'hdsc'	Horizontal descender	OS/2.sTypoDescender
'hlgp'	Horizontal line gap	OS/2.sTypoLineGap
'sbxo'	Subscript em x offset	OS/2.ySubscriptXOffset
'sbxs'	Subscript em x size	OS/2.ySubscriptXSize
'sbyo'	Subscript em y offset	OS/2.ySubscriptYOffset
'sbys'	Subscript em y size	OS/2.ySubscriptYSize
'spxo'	Superscript em x offset	OS/2.ySuperscriptXOffset
'spxs'	Superscript em x size	OS/2.ySuperscriptXSize
'spyo'	Superscript em y offset	OS/2.ySuperscriptYOffset
'spys'	Superscript em y size	OS/2.ySuperscriptYSize
'stro'	Strikeout offset	OS/2.yStrikeoutPosition
'strs'	Strikeout size	OS/2.yStrikeoutSize
'undo'	Underline offset	post.underlinePosition
'unds'	Underline size	post.underlineThickness
'vasc'	Vertical ascender	vhea.ascent
'vcof'	Vertical caret offset	vhea.caretOffset
'vcrn'	Vertical caret run	vhea.caretSlopeRun
'vcrs'	Vertical caret rise	vhea.caretSlopeRise
'vdsc'	Vertical descender	vhea.descent
'vlgp'	Vertical line gap	vhea.lineGap
'xhgt'	X height	OS/2.sxHeight

Note that the tags 'gsp0' to 'gsp9' are used to provide variation data for the rangeMaxPPEM member of records in a grid-fitting and scan-conversion procedure ('gasp') table. The last 'gasp' table entry always uses a rangeMaxPPEM value of 0xFFFF. The maximum number of value records for 'gasp' entries must never be more than one less the number of entries in the 'gasp' table.

7.3.7 STAT – Style attributes table

The style attributes table describes design attributes that distinguish font-style variants within a font family. It also provides associations between those attributes and name elements that may be used to present font options within application user interfaces.

A font family is a set of font faces that share key aspects of design. These aspects of design are common to all of the fonts in the family, and differentiate that family from other font families. But the fonts within a family also differ from one another in particular ways: differences in stroke thicknesses, differences in contrast, etc.;

or combinations of such differences. In this way, the fonts within a family are style variants of the family design. A given family will have a particular set of attribute types by which the member fonts differ: the axes of variation.

Style variants within a family may be implemented as static variants in discrete fonts, such as Arial Light or Arial Bold. They may also be implemented as dynamic variations in a variable font, using a font variations ('fvar') table and related tables; for example, weight or width variations in the Skia font. A single font may also combine dynamic variations using 'fvar' and related tables along with static style attributes. For example, a family may have variants in relation to weight attributes and also a Roman versus italic distinction, with weight implemented as a dynamic variation using 'fvar' and related tables, but with Roman and italic as static design attributes — a Roman font with weight variation, and a corresponding italic font with weight variation.

The style attributes table allows software implementations to understand relationships among the various fonts or design-variation instances within a family. It provides strings that can be used to create user interfaces with controls for individual style attributes, or used to compose strings that may be required for use in legacy applications.

NOTE The style attributes table provides a characterization of a font relative to the entire family to which it belongs, not just that one font in isolation. This characterization may involve attributes that users would not associate with the font itself. For example, part of the characterization of a Regular font would include an italic-axis value (non-italic) if the font family includes italic members, even though that font is not an italic font. (See Example 2, below.) In the case of variable fonts, the set of axes that are relevant may be a superset of the axes used in the 'fvar' table, if those axes are relevant for the entire family. For example, a family may include a Roman variable font with weight variation, and a paired italic font, also with weight variation. The italic axis is relevant to the complete characterization of both fonts, since the italic axis is relevant to the family as a whole. (See Example 5, below.)

A style attributes table is required in all variable fonts, and is optional for non-variable fonts. For a general overview of OFF Font Variations, see subclause 7.1.

The style attributes table is also recommended for all new, non-variable fonts, especially if fonts have style attributes in axes other than weight, width, or slope.

When used in multi-axis fonts, the style attributes table can accomplish the intended purpose of name IDs 21 and 22, but using a more general and flexible mechanism. For fonts used only in newer applications, strings for name IDs 21 and 22 may not be required if a style attributes table is present. When fonts need to work in older applications as well, however, name IDs 21 and 22 should continue to be used when applicable.

7.3.7.1 Style attributes header

The style attributes table begins with a header comprised of a major/minor version plus arrays of design-axis and axis-value records.

Style attributes header

Type	Name	Description
uint16	majorVersion	Major version number of the style attributes table – set to 1.
uint16	minorVersion	Minor version number of the style attributes table – set to 2.
uint16	designAxisSize	The size in bytes of each axis record.
uint16	designAxisCount	The number of design axis records. In a font with an ‘fvar’ table, this value must be greater than or equal to the axisCount value in the ‘fvar’ table.
Offset32	offsetToDesignAxes	Offset in bytes from the beginning of the 'STAT' table to the start of the design axes array.

uint16	axisValueCount	The number of axis value tables.
Offset32	offsetToAxisValueOffsets	Offset in bytes from the beginning of the 'STAT' table to the start of the design axes value offsets array.
uint16	elidedFallbackNameID	Name ID used as fallback when projection of names into a particular font model produces a subfamily name containing only elidable elements.

In version 1.0 of the style attributes table, the elidedFallbackNameId field was omitted. Use of version 1.0 is deprecated. Version 1.1 adds the elidedFallbackNameId field. Version 1.2 adds support for the format 4 axis value table; otherwise, version 1.2 and version 1.1 are the same.

The elidedFallbackNameId field provides a name that can be used when composing a name if all of the axis-value names are elidable. For example, "Normal" weight and "Roman" slant may both be marked as elidable axis-value names, and so a composed name for normal weight and Roman slant may result in an empty string. The elidedFallbackNameId is used to provide an alternative name ID to use in this case, such as "Regular". In many fonts, this may reference name ID 17 or name ID 2.

The header is followed by the design axes and axis value offsets arrays, the location of which are provided by offset fields.

AxisRecord	designAxes [designAxisCount]	The design-axes array.
Offset16	axisValueOffsets [axisValueCount]	Array of offsets to axis value tables, in bytes from the start of the axis value offsets array.

The designAxisSize field indicates the size of each axis record. Future minor-version updates of the 'STAT' table may define compatible extensions to the axis record format with additional fields. Implementations must use the designAxisSize designAxisSize field to determine the start of each record.

7.3.7.2 Axis records

The axis record provides information about a single design axis.

AxisRecord

Type	Name	Description
Tag	axisTag	A tag identifying the axis of design variation.
uint16	axisNameID	The name ID for entries in the 'name' table that provide a display string for this axis.
uint16	axisOrdering	A value that applications can use to determine primary sorting of face names, or for ordering of descriptors when composing family or face names.

Each axis record has a tag designating the axis. Take note that tag values must follow the rules for tags described in the "Design-variation axis tags registry".

The axisNameID field provides a name ID that can be used to obtain strings from the 'name' table that can be used to refer to the axis in application user interfaces. The name ID must be greater than 255 and less than 32768.

In a variable font, there must be an axis record for every axis defined in the font variations table, and it shall use the same name ID used in the font variations table. If a variation axis is omitted from the 'STAT' table, or if

different name IDs are used, applications may have unexpected behavior in the use of 'STAT' table data. However, the order of axis records in the 'STAT' table is arbitrary and does not need to match the order of records in the 'fvar' table.

If a variable font has additional axes that are not implemented as dynamic-variation axes in the font variations table, but that are relevant for the font or the family of which it is a member, axis records should also be included.

A non-variable font should include axis records for any axes that are relevant for the font or the family of which it is a member, especially if axes other than weight, width and slope are used.

NOTE For every axis declared using an axis record, the axis should be a variation axis defined in an 'fvar' table, or else it should be reflected in the font's sub-family name (name ID 17 or name ID 2) except in the case that the font's value on that axis is a "normal" value that is suppressed from the sub-family name.

The axis ordering field allows the font developer to influence ordering of axes in user interfaces, and the ordering of axis-value descriptors when face names are projected into different font-family models. Lower values are assumed to have an earlier sort order or higher priority than higher values. Values need not be contiguous, and records are not required to be well-ordered in relation to the axis ordering field. No two records should have the same value. Beyond this, no strict requirements are specified here regarding how these values are used in applications.

One way in which axisOrdering values might be used is in presenting an enumerated list of face names within a family: rather than listing faces in an arbitrary order, they might be sorted using these values to determine primary, secondary, etc. orderings. For example, if width is given a lower value than weight, then all weights with one particular width would be listed before the different weights with the next width, and so on.

Another way in which axisOrdering might be used is in composing names. A particular scenario for this is generating family and subfamily names that conform to expectations of applications that use four-member regular, bold, italic, bold italic (R/B/I/BI)or weight/width/slope (WWS) models for font families. In non-variable fonts, the designer can include name ID pairs 1/2 and 21/22 to support these models, but there is no other provision for this in a variable font. The style attributes table provides individual axis value descriptor strings that can be combined in different ways as needed to fit the requirements for name IDs 1/2 or 21/22. But when doing so, there is no independent specification for how the different elements should be ordered in a family name. For example, "Arial Serif Caption" and "Arial Caption Serif" are both possible as a name ID 21 equivalent. The axisOrdering value could be used in applications to choose the ordering when composing such names.

To ensure consistency in how face names are presented to users, the axis ordering given in axis records should be consistent across different fonts within a family, and with the ordering of axis-value descriptors used in the typographic subfamily (name ID 17). In a variable font, the axis ordering should also be consistent with the ordering of axis-value descriptors used in the strings referenced by named instances defined in the 'fvar' table.

Some legacy applications may serialize face names in document markup to indicate text formatting. Such applications can use the axis ordering field to generate names for serialization purposes in order to improve the likelihood of successful interchange with other applications that read the same document formats. However, use of face names in serialized markup is not recommended. This includes use of strings referenced by axis value records within the STAT table. Instead, applications should use the typographic family name (name ID 16) plus a design vector comprised of axis tag-value pairs, or one of the fully-qualified, non-localized names: the unique font identifier (name ID 3), or the Postscript name (name ID 6).

Different fonts belonging to the same family should have matching axis record values. However, if a set of fonts for a family are released, and then at some later time the family is extended with additional fonts using new axes of variation, the previously-shipped fonts do not need to be updated with the additional axis records; the newer fonts will provide the axis details.

7.3.7.3 Axis value tables

Axis value tables provide details regarding a specific style-attribute value on some specific axis of design variation, or a combination of design-variation axis values, and the relationship of those values to name elements. This information can be useful for presenting fonts in application user interfaces.

A particular use of axis value tables is to assist platforms in supporting typographic families that involve a wide range of design variations in older applications that assume more limited options for variation within a family. Specifically, axis value tables can be used to transform typographic family and subfamily names into alternate names appropriate to different family models. For example, some applications assume a WWS family model (a family can only include weight, width, or Italic or Oblique variants); and some applications may assume an R/B/I/BI family model (a family can include only Regular, Bold, Italic or Bold Italic variants). This use for axis value tables is similar in purpose to the alternate family/subfamily name ID pairs in the 'name' table:

— Name IDs 1 and 2 for an R/B/I/BI family model.

— Name IDs 21 and 22 for a WWS family model.

— Name IDs 16 and 17 for an unrestricted, typographic family model.

In a variable font, however, the names provided for name instances include only name ID 16 and 17 equivalents; there are are no name ID 1/2 or name ID 21/22 equivalents. Thus, axis value tables are especially important in variable fonts, to allow the named instances to be supported in older applications, particularly when any variations beyond weight, width, or italic or oblique are used.

In a variable font, every element of typographic subfamily names for all of the named instances defined in the 'fvar' table should be reflected in an axis value table. Additional axis value tables may be included for name elements that do not appear in any named instances; these may be used in some font-picking user interfaces, but are not required for mapping typography family/subfamily names into legacy naming models. Some variable fonts may include axes that are not reflected in subfamily names for any named instances — that is, variants along these axes are selectable only by means of numeric axis values. In such cases, there is no requirement to assign names or to create axis value tables for values on these axes.

In many cases, a name element will be associated with a particular value on a single axis. For example, “Bold” representing a specific weight-axis value; or “Condensed” representing a specific width value. Such name elements are assumed to be combinable with name elements associated with values on other axes, as in an instance name “Bold Condensed”. Name elements of this type are referred to as analytic names.

In other cases, a name element may be associated with a particular combination of values on multiple axes, and not be amenable to analysis into simpler, independent elements. For example, a variable font for lettering might use several custom axes to provide different stroke or swash-terminal modifications, and named instances may be defined for certain combinations of values for these axes with non-decomposable names for those combinations; for example, “Florid” for a particular combination of stroke- and termination-axes values. Name elements of this type are referred to as non-analytic. Note that a font with non-analytic names might also use an axis such as weight that uses analytic names, leading to some named instances that combine non-analytic and analytic elements, such as “Florid Bold” and “Florid Semibold”.

NOTE If an application transforms family and subfamily names to be compatible with an R/B/I/BI family model, then any subfamily name elements that are not “Regular”, “Bold”, “Italic” or “Oblique” will be moved into a family name. For example, given a family name “Selawik” and a subfamily name “Condensed Bold”, this will be transformed into the R/B/I/BI model as family name “Selawik Condensed” and subfamily name “Bold”. (“Selawik” and “Selawik Condensed” will be treated as separate families in the R/B/I/BI model.) Similarly, when names are transformed to the WWS family model, any subfamily name elements that do not represent a weight value, a width value, or italic or oblique will be moved into family names. For this reason, it is recommended that width, weight and italic/oblique descriptors always be placed last in a subfamily name, to minimize differences in how names will appear in different appliations that use different family models.

Note that a variable font may have some distinguishing, subfamily attributes that are static, not implemented as a variation, but that are relevant in relation to the complete typographic family. For example, a weight-variation font may have a paired, italic weight-variation font. Axis value records should also be provided for any such distinctions that are relevant within a family.

Axis value tables are particularly important for variable fonts, but can also be used in non-variable fonts. Note that newer platform implementations may utilize axis value tables, if included in a non-variable font, in preference to name ID 1/2 or name ID 21/22 pairs, for supporting older applications. In a non-variable font, axis value records can be provided for any style-variant distinction that is applicable to the font and relevant for the font family to which it belongs.

When used in non-variable fonts, axis value tables for particular values should be implemented consistently across fonts in the family. In particular, two different fonts within a family may share certain style attributes in common, and axis value tables for these values should be implemented consistently. For example, Bold Condensed and Bold Semi-Condensed fonts both have the same weight attribute, Bold, and should have matching axis value tables for Bold. If they are not consistent, some applications may exhibit unexpected behavior in font-related operations, such as how the fonts are presented in user interfaces, or how the fonts are chosen in fallback or other font-selection operations.

Four different axis value table formats are defined. Formats 1, 2 and 3 are appropriate for single-axis values associated with analytic name elements. Format 4 is used for multi-axis value combinations associated with non-analytic name elements. Additional formats may be supported in the future, with a minor-version update of the 'STAT' table.

NOTE 1 Use of format 1, format 2, or format 3 axis value tables is especially recommended for values on 'wght' and 'wdth' axes, and also for “Italic” (value 1.0 on the 'ital' axis) or “Oblique” (a non-zero value on the 'slnt' axis) variants.

NOTE 2 Use of format 4 axis value tables is especially recommended for non-analytic sub-family names and the corresponding axis-value combinations in static-font families or in variable fonts that also involve 'wght' or 'wdth' axes, or “Italic” or “Oblique” variants.

For any format of axis value table, the first field is read to determine the format. If the format is not recognized, then the axis value table can be ignored.

Each format includes a valueNameID field, which references a display string to be associated with the numeric axis value or combination of axis values. Defined name IDs, such as name IDs 2, 17 or 22, can be used if those have the appropriate string for an axis value. Otherwise, name IDs must be greater than 255 and less than 32768.

The different formats all include fields with numeric axis values. For any axis with a registered axis tag, the numeric values in the axis value tables will be interpreted in the same manner as they would be interpreted in the font variations table. In particular, they will be interpreted according to the user-coordinate scale and conventions specified for that axis. Similarly, if a variable font has a variation axis defined using a non-registered, custom tag, it is assumed that the values in axis value tables and in the font variations table are interpreted using the same custom scale, even if it is not conventionally defined.

In some cases, an attribute using a new design axis may be introduced into a family after other fonts have been released, with the new attribute not anticipated in any way in the initial fonts. For example, a font designer might initially create Regular, Bold and Italic variants of a design, and then later add width variants such as Condensed. In that case, the initial fonts might not have identified that they represent the “Normal” attribute on the width scale. The newer fonts should include axis value records that describe the earlier fonts. A flag is defined to designate such axis value records; this is described in detail below.

No two tables should provide information for the same axis value.

Axis value table, format 1

AxisValueFormat1:

Type	Name	Description
uint16	format	Format identifier – set to 1
uint16	axisIndex	Index into the axis record array identifying the axis of design variation to which the axis value record applies.
uint16	flags	Flags – see below for details.
uint16	valueNameID	The name ID for entries in the 'name' table that provide a display string for this attribute value.
Fixed	value	A numeric value for this attribute value.

A format 1 table is used simply to associate a specific axis value with a name.

Axis value table, format 2

AxisValueFormat2:

Type	Name	Description
uint16	format	Format identifier – set to 2
uint16	axisIndex	Index into the axis record array identifying the axis of design variation to which the axis value record applies.
uint16	flags	Flags – see below for details.
uint16	valueNameID	The name ID for entries in the 'name' table that provide a display string for this attribute value.
Fixed	nominalValue	A nominal numeric value for this attribute value.
Fixed	rangeMinValue	The minimum value for a range associated with the specified name ID.
Fixed	rangeMaxValue	The maximum value for a range associated with the specified name ID.

A format 2 table can be used if a given name is associated with a particular axis value, but is also associated with a range of values. For example, in a family that supports optical size variations, “Subhead” may be used in relation to a range of sizes. The rangeMinValue and rangeMaxValue fields are used to define that range. In a variable font, a named instance has specific coordinates for each axis. The nominalValue field allows some specific, nominal value to be associated with a name, to align with the named instances defined in the font variations table, while the rangeMinValue and rangeMaxValue fields allow the same name also to be associated with a range of axis values.

Some design axes may be open ended, having an effective minimum value of negative infinity, or an effective maximum value of positive infinity. To represent an effective minimum of negative infinity, set rangeMinValue to 0x80000000. To represent an effective maximum of positive infinity, set rangeMaxValue to 0x7FFFFFFF.

Two format 2 tables for a given axis should not have ranges with overlap greater than zero. If a font has two format 2 tables for a given axis, T1 and T2, with overlapping ranges, the following rules will apply:

- If the range of T1 overlaps the higher end of the range of T2 with a greater max value than T2 (T1.rangeMaxValue > T2.rangeMaxValue and T1.rangeMinValue <= T2.rangeMaxValue), then T1 is used for all values within its range, including the portion that overlaps the range of T2.
- If the range of T2 is contained entirely within the range of T1 (T2.rangeMinValue >= T1.rangeMinValue and T2.rangeMaxValue <= T1.rangeMaValue), then T2 is ignored.

In the case of two tables with identical ranges for the same axis, it will be up to the implementation which is used and which is ignored.

Axis value table, format 3

AxisValueFormat3:

Type	Name	Description
uint16	format	Format identifier – set to 3
uint16	axisIndex	Index into the axis record array identifying the axis of design variation to which the axis value record applies.
uint16	flags	Flags – see below for details.
uint16	valueNameID	The name ID for entries in the 'name' table that provide a display string for this attribute value.
Fixed	value	A numeric value for this attribute value.
Fixed	linkedValue	The numeric value for a style-linked mapping from this value.

A format 3 table can be used to indicate another value on the same axis that is to be treated as a style-linked counterpart to the current value. This is primarily intended for “bold” style linking on a weight axis. These mappings may be used in applications to determine which style within a family should be selected when a user selects a “Bold” formatting option. A mapping is defined from a “non-bold” value to its “bold” counterpart. It is not necessary to provide a “bold” mapping for every weight value; mappings should be provided for lighter weights, but heavier weights (typically, semibold or above) would already be considered “bold” and would not require a “bold” mapping.

NOTE Applications are not required to use these style-linked mappings when implementing text formatting user interfaces. This data can be provided in a font for the benefit of applications that choose to do so. If a given application does not apply such style mappings for the given axis, then the linkedValue field is ignored.

Axis value table, format 4

AxisValueFormat4:

Type	Name	Description
uint16	format	Format identifier – set to 4
uint16	axisCount	The total number of axes contributing to this axis-values combination.
uint16	flags	Flags – see below for details.
uint16	valueNameID	The name ID for entries in the 'name' table that provide a display string for this combination of axis values.
AxisValue	axisValues[axisCount]	Array of AxisValue records that provide the combination of axis values, one for each contributing axis.

The axisValues array uses AxisValue records, which have the following format.

AxisValue record:

Type	Name	Description
uint16	axisIndex	Zero-base index into the axis record array identifying the axis to which this value applies. Must be less than designAxisCount.
Fixed	value	A numeric value for this attribute value.

Each AxisValue record shall have a different axisIndex value. The records can be in any order.

Flags

The following axis value table flags are defined:

Mask	Name	Description
0x0001	OLDER_SIBLING_FONT_ATTRIBUTE	If set, this axis value table provides axis value information that is applicable to *other* fonts within the same font family. This is used if the other fonts were released earlier and did not include information about values for some axis. If newer versions of the other fonts include the information themselves and are present, then this record is ignored.
0x0002	ELIDABLE_AXIS_VALUE_NAME	If set, it indicates that the axis value represents the “normal” value for the axis and may be omitted when composing name strings.

When the OlderSiblingFontAttribute flag is used, implementations may use the information provided to determine behavior associated with a *different* font in the same family. If a previously-released family is extended with fonts for style variations from a new axis of design variation, then all of them should include a OlderSiblingFontAttribute table for the “normal” value of earlier fonts. The values in the different fonts should match; if they do not, application behavior may be unpredictable.

NOTE 1 When the OlderSiblingFontAttribute flag is set, that axis value table is intended to provide default information about other fonts in the same family, but not about the font in which that axis value table is contained. The font should contain different axis value tables that do not use this flag to make declarations about itself.

The ElidableAxisValueName flag can be used to designate a “normal” value for an axis that should not normally appear in a face name. For example, the designer may prefer that face names not include “Normal” width or “Regular” weight. If this flag is set, applications are permitted to omit these descriptors from face names, though they may also include them in certain scenarios.

NOTE 2 Fonts should provide axis value tables for “normal” axis values even if they should not normally be reflected in face names.

NOTE 3 If a font or a variable-font instance is selected for which all axis values have the ElidableAxisValueName flag set, then applications may keep the name for the weight axis, if present, to use as a constructed subfamily name, with names for all other axis values omitted.

When the OlderSiblingFontAttribute flag is set, this will typically be providing information regarding the “normal” value on some newly-introduced axis. In this case, the ElidableAxisValueName flag may also be set, as desired. When applied to the earlier fonts, those likely would not have included any descriptors for the new axis, and so the effects of the ElidableAxisValueName flag are implicitly assumed.

If multiple axis value tables have the same axis index, then one of the following should be true:

- The font is a variation font, and the axis is defined in the font variations table as a variation axis.
- The OlderSiblingFontAttribute flag is set in one of the records.

Two different fonts within a family may share certain style attributes in common. For example, Bold Condensed and Bold Semi Condensed fonts both have the same weight attribute, Bold. Axis value tables for particular values should be implemented consistently across a family. If they are not consistent, applications may exhibit unpredictable behaviors.

7.3.7.4 Examples

The following examples illustrate data provided by a style attributes table for various font scenarios.

7.3.7.4.1 Example 1: Non-variation font family with different weight variants

Suppose a font family has Regular, Bold and Heavy weight variants. These fonts would have matching axis records:

Axis tag	Axis Name	Axis ordering
‘wght’	Weight	0

The three fonts would have axis value data as follows:

Font	Axis tag	Value	Name string	Flag	Other data
Font 1	‘wght’	400	Regular	ElidableAxisValueName	linkedValue = 700
Font 2	‘wght’	700	Bold		
Font 3	‘wght’	900	Heavy		

7.3.7.4.2 Example 2: Non-variation family with weight values plus italic

Suppose the font family from example 1 also has italic variants. The fonts would have matching axis records reflecting weight and italic axes:

Axis tag	Axis Name	Axis ordering
‘wght’	Weight	0
‘ital’	Italic	1

Each of the three non-italic fonts would include an additional axis value record to reflect the non-italic attribute. The six fonts would have data as follows:

Font	Axis tag	Value	Name string	Flag	Other data
Font 1	‘wght’	400	Regular	ElidableAxisValueName	linkedValue = 700
Font 1	‘ital’	0	Regular	ElidableAxisValueName	
Font 2	‘wght’	700	Bold		
Font 2	‘ital’	0	Regular	ElidableAxisValueName	
Font 3	‘wght’	900	Heavy		
Font 3	‘ital’	0	Regular	ElidableAxisValueName	
Font 4	‘wght’	400	Regular	ElidableAxisValueName	linkedValue = 700
Font 4	‘ital’	1	Italic		
Font 5	‘wght’	700	Bold		
Font 5	‘ital’	1	Italic		
Font 6	‘wght’	900	Heavy		
Font 6	‘ital’	1	Italic		

7.3.7.4.3 Example 3: Non-variation family with weight and variants, later extended to add width variants

Suppose the font family from example 2 is later extended with different width variants. The new fonts in the family would include matching axis records reflecting three axes:

Axis tag	Axis Name	Axis ordering
‘wdth’	Width	0
‘wght’	Weight	1
‘ital’	Italic	2

Newer versions of the initially-released fonts would also include the additional axis record. When newer fonts co-exist with the original version of the earlier fonts, the ordering from the more-complete axis records in the newer fonts is used.

To allow for situations in which one of the newer fonts co-exists with the older fonts, which did not reference the width axis, the newer fonts should each include an axis record to describe the “Normal” width, which is inferred onto the earlier fonts.

Axis tag	Value	Name string	Flag	Other data
‘wdth’	100	Normal	OlderSiblingFontAttribute	

7.3.7.4.4 Example 4: A weight/width variation font

Consider a family comprised of a single variation font with weight and width variations. This font would have axis records for the two variation axes:

Axis tag	Axis Name	Axis ordering
‘wght’	Weight	0
‘wdth’	Width	1

Suppose the variation font has 6 named instances that correspond to three different weights for each of two widths. The style attributes table should include axis value records for at least those three weights and those two widths, but could also include records for other weight or width values. The font may include the following axis value records:

Axis tag	Value	Name string	Flag	Other data
‘wght’	300	Light		linkedValue = 600
‘wght’	400	Regular	ElidableAxisValueName	linkedValue = 700
‘wght’	600	Semibold		
‘wght’	700	Bold		
‘wght’	900	Black		
‘wdth’	62.6	Extra-Condensed		
‘wdth’	75	Condensed		
‘wdth’	100	Normal	ElidableAxisValueName	
‘wdth’	125	Expanded		
‘wdth’	150	Extra-Expanded		

7.3.7.4.5 Example 5: A family comprised of a non-italic variation font plus an italic variation font

Consider a family comprised of a non-italic, weight/width variation font plus a corresponding italic, weight/width variation font. Each font would have axis records for three axes:

Axis tag	Axis Name	Axis ordering
‘wght’	Weight	0
‘wdth’	Width	1
‘ital’	Italic	2

In addition to axis value records for various weight or width values, the first font would also include a record to reflect the non-italic attribute:

Axis tag	Value	Name string	Flag	Other data
‘ital’	0	Normal	ElidableAxisValueName	

The second font would include a record to reflect the italic attribute, in addition to the records for various weight and width values:

Axis tag	Value	Name string	Flag	Other data
‘ital’	1	Italic		

The pattern in this example can be applied to other cases involving a family with style variations implemented using a combination of font-variation (‘fvar’) mechanisms plus static, non-variation designs. The axis records in each font would span both variation and non-variation axes. Axis value records in a given font would include multiple values for axes implemented using variation mechanisms, plus single records for the relevant attribute values of other axes.

7.3.7.4.6 Example 6: A varible font with non-analytic subfamily names associated with multiple axis values

Consider a variable font that uses several custom axes, 'TRM1', 'TRM2', 'STK1', 'STK2', and also the registered 'wght' axis. Suppose that this font has named instances “Florid” and “Jagged” that involve particular combinations of values for the custom axes; and additional named instances that correspond to those two named instances but with other 'wght' values: “Florid Bold”, “Florid Semibold”, etc. The font would have axis value tables for 'wght' values, with data such as the following:

Axis tag	Value	Name string	Flag	Other data
‘wght’	400	Regular	ElidableAxisValueName	linkedValue = 700
‘wght’	700	Bold		
‘wght’	900	Heavy		

The font would also have format 4 axis value tables corresponding to “Florid” and “Jagged”, with data such as the following:

Name string	AxisValue records
Florid	'TRM1' = 250 'TRM2' = 1000 'STK1' = 550 'STK2' = 0
Jagged	'TRM1' = 900 'TRM2' = 4500 'STK1' = 0 'STK2' = 310

7.3.8 VVAR – Vertical metrics variations table

The 'VVAR' table is used in variable fonts to provide variations for vertical glyph metric values. This can be used to provide variation data for advance heights in the 'vmtx' table. In fonts with TrueType outlines, it can also be used to provide variation data for top and bottom side bearings obtained from the 'vmtx' table and glyph bounding box. In addition, it can also be used in fonts that have CFF2 outlines to provide vertical-origin variation data.

For a general overview of OFF Font variations and terminology related to variations, see subclause 7.1.

In a font with TrueType outlines, the rasterizer will generate “phantom” points that represent left, right, top and bottom side bearings. (See "Instructing TrueType Glyphs" [24] for more background on phantom points.) In a TrueType variable font, the glyph variations ('gvar') table will include variation data for the phantom points of each glyph, allowing glyph metrics to be interpolated for different variation instances as part of deriving the interpolated glyph outlines. For this reason, the 'VVAR' table is not required in variable fonts that have TrueType outlines. For text-layout operations that require glyph metrics but not actual glyph outlines, however,

there can be significant performance benefits by being able to obtain adjusted glyph metrics for an instance without needing to interpolate glyph outlines. For this reason, it is recommended that a 'VVAR' table be included in variable fonts that have TrueType outlines and that support vertical layout.

The CFF2 rasterizer does not generate phantom points as in the TrueType rasterizer. For this reason, an 'VVAR' table is required to handle any variation in vertical glyph metrics in a variable font with CFF2 outlines.

The format and processing of the 'VVAR' table is analogous to the horizontal metrics variations ('HVAR') table.

7.3.8.1 Related and co-requisite tables

The 'VVAR' table is used only in variable fonts that support vertical layout. It must be used in combination with a vertical metrics ('vmtx') table, and also in combination with a font variations ('fvar') table, and other required or optional tables used in variable fonts. See Variation Data Tables and Miscellaneous Requirements in the Font Variations Overview chapter for general information.

For variable fonts that have TrueType outlines and that support vertical layout, the 'VVAR' table is optional but recommended. For variable fonts that have CFF2 outlines and that support vertical layout, the 'VVAR' table is required if there is any variation in glyph advance heights across the variation space.

NOTE The 'VDMX' table is not used in variable fonts.

7.3.8.2 Table formats

The vertical metrics variations table has the following format:

Vertical metrics variations table

Type	Name	Description
uint16	majorVersion	Major version number of the vertical metrics variations table — set to 1.
uint16	minorVersion	Minor version number of the vertical metrics variations table — set to 0.
Offset32	itemVariationStoreOffset	Offset in bytes from the start of this table to the item variation store table.
Offset32	advanceHeightMappingOffset	Offset in bytes from the start of this table to the delta-set index mapping for advance heights (may be NULL).
Offset32	tsbMappingOffet	Offset in bytes from the start of this table to the delta-set index mapping for top side bearings (may be NULL).
Offset32	bsbMappingOffset	Offset in bytes from the start of this table to the delta-set index mapping for bottom side bearings (may be NULL).
Offset32	vOrgMappingOffset	Offset in bytes from the start of this table to the delta-set index mapping for Y coordinates of vertical origins (may be NULL).

The item variation store table is documented in subclause 7.2.

Mapping tables are optional. If a given mapping table is not provided, the offset is set to NULL.

Variation data for advance heights is required. A delta-set index mapping table for advance heights can be provided, but is optional. If a mapping table is not provided, glyph indices are used as implicit delta-set indices, as in the HVAR table.

Variation data for side bearings are optional. If included, mapping tables are required to provide the delta-set index for each glyph.

Mappings and variation data for vertical origins are not used in fonts with TrueType outlines, but can be included in variable fonts with CFF2 outlines if there is variability in the Y coordinates of glyph vertical origins, the default values of which are recorded in the 'VORG' table. A mapping table is required for vertical-origin variation data.

See the horizontal metrics variations ('HVAR') table description for remaining details.

8 Recommendations for OFF fonts

This clause outlines recommendations for creating OFF fonts.

8.1 Byte ordering

All OFF fonts use Motorola-style byte ordering (Big Endian).

8.2 '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.

8.3 Mixing outline formats

It is not recommended to mix outline formats within a single font. Choose the format that meets your feature requirements.

8.4 Filenames

OFF fonts may have the extension .OTF, .TTF, .OTC or .TTC, depending on the kind of outlines in the font and the desired backward compatibility.

- A file containing a single font resource with TrueType outlines should have either .OTF or .TTF as the extension. The choice between .OTF and .TTF may depend on the desire for backward compatibility on older systems or with previous versions of the font.
- A file containing a single font resource with only CFF outline data (no TrueType outlines) should have an .OTF extension.
- A font collection file (one that contains multiple font resources) should have either .OTC or .TTC as the extension, regardless of whether or not layout tables are present in any of the font resources, and regardless of the kind of outline data used. The .TTC extension may be used for font collection files containing font resources that use CFF outline data if needed for backward compatibility with older software that was not aware of the .OTC extension.
- A variable font that uses OFF font variations mechanisms and associated tables should use the extensions .OTF, .TTF, .OTC or .TTC following the above guidance. If there is a need to provide some indication wihtin a filename that the file contains a variable font, a recommended convention is to append “VF” (with a preceding delimiter character) at the end of the file name (before the extension) — e.g., “Selawik-VF.ttf”.

In all cases, software must determine the kind of outlines present in a font not from the filename extension but from the contents of the file.

8.5 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 32-bit boundaries. Any extra space after a table (and before the next 32-bit boundary) should be padded with zeros.

8.6 Glyph 0: the .notdef glyph

Glyph 0 must be assigned to a .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 Glyph ID 0 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.

8.7 '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.

8.8 '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.

8.9 'cvt' table

Should be defined only if required by font instructions.

8.10 'fpgm' table

Should be defined only if required by TrueType font instructions.

8.11 '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 It 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.

8.12 '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.

8.13 '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.

8.14 'hhea' table

All data required. It is suggested that monospaced fonts set numberOfHMetrics to three (see hmtx).

8.15 'hmtx' table

All data required. It is suggested that monospaced fonts have three entries in the numberOfHMetrics field.

8.16 '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).

The OFF specification allows CFF OT fonts to express their kerning in a kern table. Many OFF text layout engines support this. Windows GDI's CFF OT driver, however, ignores the kern table in a CFF OT font when it prepares kerning pairs to report via its pair kerning API.

When a kern table and GPOS table are both present in a font, and an OFF layout engine is requested to apply kerning to a run of text of a particular script and language system: (a) If the number of kern feature lookups in the resolved language system in the GPOS table is zero, then the kern table should be applied, followed by any remaining GPOS features requested. (b) If the number of kern feature lookups in the resolved language system in the GPOS table is non-zero, then all GPOS lookups, including the kern lookups, should be applied in the usual way and the kern table data ignored.

If a kern table present but no GPOS table is present in the font, then an OFF layout engine should apply the kern table to the text, regardless of the resolved language system of the text.

If compatibility with legacy environments is not a concern, font vendors are encouraged to record kerning in the GPOS table's kern feature and not in the kern table.

OFF Font Variations mechanisms do not include any way to represent variation of data within a kern table. Therefore, kerning in a variable should be implemented using the GPOS table.

8.17 '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.

8.18 '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.

8.19 'maxp' table

All data required for a font with TrueType outlines. Fonts with CFF or CFF2 data must only fill the numGlyphs field.

8.20 '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, and the referenced string data must be encoded in UTF-16.

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 Microsoft records, language ID = 0 for Macintosh records); additional language strings for the Microsoft 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. URL's 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. If the font is 'Regular' as indicated in string 2, then sometimes only the family name contained in string 1 is used for the full font name. In many contexts, the full font name is what will be exposed to users.

In variable fonts, the Typographic Family and Typographic Subfamily names (name IDs 16 and 17) are required. Applications that support OFF font variations will typically present to users the Typographic Family name along with the Typographic Subfamily name or alternative subfamily names for named instances, as specified in the font variations ('fvar') table. In some situations, a font vendor may want to make available a variable font as well as some set of non-variable fonts corresponding to the named instances of the variable font. In such situations, the vendor may want to have distinct family names for the family implemented as a variable font and the family implemented using several non-variable fonts. In that case, a suggested convention is to append “VF” at the end of the variable-font family name. Note, however, that this will result in distinct families, and content formatted with the one may not display as intended in some contexts if only the other is available. For example, if a document is formatted using the regular and bold instances of a variable font with family name "Selawik VF" and then the document is viewed in a context in which only the non-variable fonts Selawik Regular and Selawik Bold are available, the viewing application will generally not be able to associate the non-variable fonts available to it with the formatting declarations in the content.

OFF fonts that include a name with name ID of 6 should 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, may be ignored.

When translated to ASCII, these two name strings must be identical and restricted to the printable ASCII subset, codes 33 through 126, except for the 10 characters: '[', ']', '(', ')', '{', '}', '<', '>', '/', '%'. Some implementations have a 63-character length limit; however, a 127-character length limit is recommended.

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 requirement 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.

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 requirement 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.

8.21 '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 'Latin Pictorial' (value = 5).

In a variable font that uses OFF Font Variation mechanisms, there is no way to represent different PANOSE values for different instances supported by the font. The PANOSE values can be set based on the default instance

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 9.8 "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.

8.22 '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" [3], 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.

NOTE The PostScript glyph name must be no longer than 31 characters, include only uppercase or lowercase English letters, European digits, the period or the underscore, i.e. from the set [A-Za-z0-9_.] and start with a letter, except the special glyph names “,notdef” and “.null” which start with a period.

8.23 'prep' table

Should be defined only if required by the TrueType font instructions.

8.24 '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.

8.25 TrueType Collections

The process of building TTC files involves paying close attention to 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 – i.e. their pre-programs, function definitions, and control values must not conflict.

9 General recommendations

9.1 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)

9.2 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 'Latin Pictorial' (value = 5).

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.

9.3 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)

9.4 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.

9.5 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).

9.6 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.

9.7 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 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 Reference [11] 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.

9.8 Stroke reduction in variable fonts

When designing a font family to support a number of variations, there may be cases in which it is desirable to make significant, structural changes to particular glyphs for certain variations. A common example is stroke reduction for heavier weights or narrower widths, simplifying the structure of a glyph so that counters do not become filled and disappear at smaller text sizes. Within a variable font, two techniques might be used to implement a stroke-reduction effect:

— Use a pair of slightly-overlapping intermediate regions within the variation data for a glyph in order to introduce deltas for particular contour points that result in the desired structural change and that apply for only problem ranges on one or more axes.

— Use an OFF Layout Required Variation Alternates feature in combination with a FeatureVariations table wihthin the 'GSUB' table to perform a glyph substitution when a variation instance is selected in some range along one or more axes.

While both techniques are possible, it should be noted that the first technique, using overlapping intermediate-regions, can be tricky to implement and may result in unexpected or undesired results if an instance is selected using arbitrary axis values in the range over which the transition occurs. The second technique is recommended as it will generally be easier to implement and maintain, and provides the font designer better control over behavior near the point of transition.

9.9 Families with optical size variants

In families that have fonts for different optical sizes or in variable fonts that support the optical size ('opsz') design axis, a 'STAT' table with format 2 axis value tables should be used to indicate text size ranges for which the different optical-size variants or variable-font named instances are recommended. This supersedes the use of the *usLowerOpticalPointSize* and *usUpperOpticalPointSize* fields in the 'OS/2' table, and the OFF Layout 'size' feature.

When creating an axis value table to correspond to the font or named instance that is intended for the largest text sizes, the upper text size limit should be, effectively, infinity. To represent this in a format 2 axis value table, set the rangeMaxValue to 0x7FFFFFFF.

Annex A
(informative)

Font Class and Font Subclass parameters

A.1 General

This annex defines the Font Class and the Font Subclass parameter values to be used in the classification of font designs by the font designer or supplier. This information is stored in the sFamilyClass field of a font's OS/2 table.

A.2 sFamilyClass

Format: int16

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 assigned by IBM to each font family. This parameter is intended for use in selecting an alternate font when the requested font is not available. The font class is the most general and the font subclass is the most specific. The high byte of this field contains the family class, while the low byte contains the family subclass.

These values classify a font design as to its appearance, but do not identify the specific font family, typeface variation, designer, supplier, size, or metric table differences. It should be noted that some font designs may be classified equally well into more than IBM Font Class or Subclass. Such designs should be matched to a classification for which substitution of another font design from the same class or subclass would generally result in a similar appearance of the presented document.

A.3 Class ID=0 No Classification

This class ID is used to indicate that the associated font has no design classification or that the design classification is not of significance to the creator or user of the font resource.

A.4 Class ID=1 Oldstyle Serifs

This style is generally based upon the Latin printing style of the 15th to 17th century, with a mild diagonal contrast in stroke emphasis (lighter in upper left to lower right, heavier in upper right to lower left) and bracketed serifs. This Class reflects the ISO Serif Class, Oldstyle and Legibility Subclasses as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.4.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design subclassification is not of significance to the creator or user of the font resource.

A.4.2 Subclass ID = 1 : IBM Rounded Legibility

This style is generally characterized by a large x-height, with short ascenders and descenders. Specifically, it is distinguished by a medium resolution, hand tuned, bitmap rendition of the more general rounded legibility subclass. This IBM Subclass is not strictly specified in the ISO/IEC 9541-1 Amendment 1 standard.

A.4.3 Subclass ID = 2 : Garalde

This style is generally characterized by a medium x-height, with tall ascenders. An example of this font style is the ITC Garamond family. This IBM Subclass reflects the ISO Serif Class, Oldstyle Subclass, and Garalde Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.4.4 Subclass ID = 3 : Venetian

This style is generally characterized by a medium x-height, with a relatively monotone appearance and sweeping tails based on the designs of the early Venetian printers. This IBM Subclass is not strictly specified in the ISO/IEC 9541-1 Amendment 1 standard.

A.4.5 Subclass ID = 4 : Modified Venetian

This style is generally characterized by a large x-height, with a relatively monotone appearance and sweeping tails based on the designs of the early Venetian printers. An example of this font style is the Allied Linotype Palatino family. This IBM Subclass reflects the ISO Serif Class, Transitional Subclass, and Modified Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.4.6 Subclass ID = 5 : Dutch Modern

This style is generally characterized by a large x-height, with wedge shaped serifs and a circular appearance to the bowls similar to the Dutch Traditional Subclass below, but with lighter stokes. This IBM Subclass is not strictly specified in the ISO/IEC 9541-1 Amendment 1 standard.

A.4.7 Subclass ID = 6 : Dutch Traditional

This style is generally characterized by a large x-height, with wedge shaped serifs and a circular appearance of the bowls. An example of this font style is the IBM Press Roman family. This IBM Subclass reflects the ISO Serif Class and Legibility Subclass as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.4.8 Subclass ID = 7 : Contemporary

This style is generally characterized by a small x-height, with light stokes and serifs. An example of this font style is the University family. This IBM Subclass reflects the ISO Serif Class and Contemporary Subclass as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.4.9 Subclass ID = 8 : Calligraphic

This style is generally characterized by the fine hand writing style of calligraphy, while retaining the characteristic Oldstyle appearance. This IBM Subclass is not reflected in the ISO/IEC 9541-1 Amendment 1 standard.

A.4.10 Subclass ID = 9-14 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.4.11 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.5 Class ID=2 Transitional Serifs

This style is generally based upon the Latin printing style of the 18th to 19th century, with a pronounced vertical contrast in stroke emphasis (vertical strokes being heavier than the horizontal strokes) and bracketed serifs. This IBM Class reflects the ISO Serif Class, Transitional Subclass as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.5.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

A.5.2 Subclass ID = 1 : Direct Line

This style is generally characterized by a medium x-height, with fine serifs, noticeable contrast, and capitol letters of approximately the same width. An example of this font style is the Monotype Baskerville family. This IBM Subclass reflects the ISO Serif Class, Transitional Subclass, and Direct Line Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.5.3 Subclass ID = 2 : Script

This style is generally characterized by a hand written script appearance while retaining the Transitional Direct Line style. An example of this font style is the IBM Nasseem (Arabic) family. This IBM Subclass is not strictly specified in the ISO/IEC 9541-1 Amendment 1 standard, though the ISO Serif Class, Transitional Subclass, and Direct Line Specific Group would be a close approximation.

A.5.4 Subclass ID = 3-14 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.5.5 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.6 Class ID=3 Modern Serifs

This style is generally based upon the Latin printing style of the 20th century, with an extreme contrast between the thick and thin portion of the strokes. This IBM Class reflects the ISO Serif Class, Modern Subclass as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.6.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

A.6.2 Subclass ID = 1 : Italian

This style is generally characterized by a medium x-height, with thin hairline serifs. An example of this font style is the Monotype Bodoni family. This IBM Subclass reflects the ISO Serif Class, Modern Subclass, and Italian Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.6.3 Subclass ID = 2 : Script

This style is generally characterized by a hand written script appearance while retaining the Modern Italian style. An example of this font style is the IBM Narkissim (Hebrew) family. This IBM Subclass is not strictly specified in the ISO/IEC 9541-1 Amendment 1 standard, though the ISO Serif Class, Modern Subclass, and Italian Specific Group would be a close approximation.

A.6.4 Subclass ID = 3-14 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.6.5 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.7 Class ID=4 Clarendon Serifs

This style is a variation of the Oldstyle Serifs and the Transitional Serifs, with a mild vertical stroke contrast and bracketed serifs. This IBM Class reflects the ISO Serif Class, Square Serif Subclass as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.7.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

A.7.2 Subclass ID = 1 : Clarendon

This style is generally characterized by a large x-height, with serifs and strokes of equal weight. An example of this font style is the Allied Linotype Clarendon family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Clarendon Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.7.3 Subclass ID = 2 : Modern

This style is generally characterized by a large x-height, with serifs of a lighter weight than the strokes and the strokes of a lighter weight than the Traditional. An example of this font style is the Monotype Century Schoolbook family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Clarendon Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.7.4 Subclass ID = 3 : Traditional

This style is generally characterized by a large x-height, with serifs of a lighter weight than the strokes. An example of this font style is the Monotype Century family.This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Clarendon Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.7.5 Subclass ID = 4 : Newspaper

This style is generally characterized by a large x-height, with a simpler style of design and serifs of a lighter weight than the strokes. An example of this font style is the Allied Linotype Excelsior Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Clarendon Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.7.6 Subclass ID = 5 : Stub Serif

This style is generally characterized by a large x-height, with short stub serifs and relatively bold stems. An example of this font style is the Cheltenham Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Short Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.7.7 Subclass ID = 6 : Monotone

This style is generally characterized by a large x-height, with monotone stems. An example of this font style is the ITC Korinna Family. This IBM Subclass is not strictly specified in the ISO/IEC 9541-1 Amendment 1 standard.

A.7.8 Subclass ID = 7 : Typewriter

This style is generally characterized by a large x-height, with moderate stroke thickness characteristic of a typewriter. An example of this font style is the Prestige Elite Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Typewriter Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.7.9 Subclass ID = 8-14: (reserved for future use)

These subclass IDs are reserved for future assignment.

A.7.10 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.8 Class ID=5 Slab Serifs

This style is characterized by serifs with a square transition between the strokes and the serifs (no brackets). This IBM Class reflects the ISO Serif Class, Square Serif Subclass (except the Clarendon Specific Group) as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.8.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

A.8.2 Subclass ID = 1 : Monotone

This style is generally characterized by a large x-height, with serifs and strokes of equal weight. An example of this font style is the ITC Lubalin Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.8.3 Subclass ID = 2 : Humanist

This style is generally characterized by a medium x-height, with serifs of lighter weight that the strokes. An example of this font style is the Candida Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.8.4 Subclass ID = 3 : Geometric

This style is generally characterized by a large x-height, with serifs and strokes of equal weight and a geometric (circles and lines) design. An example of this font style is the Monotype Rockwell Family. This IBM

Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.8.5 Subclass ID = 4 : Swiss

This style is generally characterized by a large x-height, with serifs and strokes of equal weight and an emphasis on the white space of the characters. An example of this font style is the Allied Linotype Serifa Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.8.6 Subclass ID = 5 : Typewriter

This style is generally characterized by a large x-height, with serifs and strokes of equal but moderate thickness, and a geometric design. An example of this font style is the IBM Courier Family. This IBM Subclass is not strictly specified in the ISO/IEC 9541-1 Amendment 1 standard, though the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group would be a close approximation.

A.8.7 Subclass ID = 6-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

A.8.8 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.9 Class ID=6 (reserved for future use)

This class ID is reserved for future assignment.

A.10 Class ID=7 Freeform Serifs

This style includes serifs, but which expresses a design freedom that does not generally fit within the other serif design classifications. This IBM Class reflects the remaining ISO Serif Class subclasses as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.10.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

A.10.2 Subclass ID = 1 : Modern

This style is generally characterized by a medium x-height, with light contrast in the strokes and a round full design. An example of this font style is the ITC Souvenir Family. This IBM Subclass is not reflected in the ISO/IEC 9541-1 Amendment 1 standard.

A.10.3 Subclass ID = 2-14 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.10.4 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.11 Class ID=8 Sans Serifs

This style includes most basic letter forms (excluding Scripts and Ornamentals) that do not have serifs on the strokes. This IBM Class reflects the ISO Sans Serif Class as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.11.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

A.11.2 Subclass ID = 1 : IBM Neo-grotesque Gothic

This style is generally characterized by a large x-height, with uniform stroke width and a simple one story design distinguished by a medium resolution, hand tuned, bitmap rendition of the more general Neo-grotesque Gothic Subclass. An example of this font style is the IBM Sonoran Sans Serif family. This IBM Subclass is not strictly specified in the ISO/IEC 9541-1 Amendment 1 standard.

A.11.3 Subclass ID = 2 : Humanist

This style is generally characterized by a medium x-height, with light contrast in the strokes and a classic Roman letterform. An example of this font style is the Allied Linotype Optima family. This IBM Subclass reflects the ISO Sans Serif Class, Humanist Subclass as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.11.4 Subclass ID = 3 : Low-x Round Geometric

This style is generally characterized by a low x-height, with monotone stroke weight and a round geometric design. An example of this font style is the Fundicion Tipograficia Neufville Futura family. This IBM Subclass reflects the ISO Sans Serif Class, Geometric Subclass, Round Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.11.5 Subclass ID = 4 : High-x Round Geometric

This style is generally characterized by a high x-height, with uniform stroke weight and a round geometric design. An example of this font style is the ITC Avant Garde Gothic family. This IBM Subclass reflects the ISO Sans Serif Class, Geometric Subclass, Round Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.11.6 Subclass ID = 5 : Neo-grotesque Gothic

This style is generally characterized by a high x-height, with uniform stroke width and a simple one story design. An example of this font style is the Allied Linotype Helvetica family. This IBM Subclass reflects the ISO Sans Serif Class, Gothic Subclass, Neo-grotesque Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.11.7 Subclass ID = 6 : Modified Neo-grotesque Gothic

This style is similar to the Neo-grotesque Gothic style, with design variations to the G and Q. An example of this font style is the Allied Linotype Univers family. This IBM Subclass is not strictly specified in the ISO/IEC

9541-1 Amendment 1 standard, though the ISO Sans Serif Class, Gothic Subclass, Neo-grotesque Specific Group would be a close approximation.

A.11.8 Subclass ID = 7-8 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.11.9 Subclass ID = 9 : Typewriter Gothic

This style is similar to the Neo-grotesque Gothic style, with moderate stroke thickness characteristic of a typewriter. An example of this font style is the IBM Letter Gothic family. This IBM Subclass reflects the ISO Sans Serif Class, Gothic Subclass, Typewriter Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.11.10 Subclass ID = 10 : Matrix

This style is generally a simple design characteristic of a dot matrix printer. An example of this font style is the IBM Matrix Gothic family. This IBM Subclass is not reflected in the ISO/IEC 9541-1 Amendment 1 standard.

A.11.11 Subclass ID = 11-14 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.11.12 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.12 Class ID=9 Ornamentals

This style includes highly decorated or stylized character shapes that are typically used in headlines. This IBM Class reflects the ISO Ornamental Class and the ISO Blackletter Class as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.12.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

A.12.2 Subclass ID = 1 : Engraver

This style is characterized by fine lines or lines engraved on the stems. An example of this font style is the Copperplate family. This IBM Subclass reflects the ISO Ornamental Class and Inline Subclass, or the Serif Class and Engraving Subclass as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.12.3 Subclass ID = 2 : Black Letter

This style is generally based upon the printing style of the German monasteries and printers of the 12th to 15th centuries. An example of this font style is the Old English family. This IBM Subclass reflects the ISO Blackletters Class as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.12.4 Subclass ID = 3 : Decorative

This style is characterized by ornamental designs (typically from nature, such as leaves, flowers, animals, etc.) incorporated into the stems and strokes of the characters. An example of this font style is the Saphire

family. This IBM Subclass reflects the ISO Ornamental Class and Decorative Subclass as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.12.5 Subclass ID = 4 : Three Dimensional

This style is characterized by a three dimensional (raised) appearance of the characters created by shading or geometric effects. An example of this font style is the Thorne Shaded family. This IBM Subclass reflects the ISO Ornamental Class and Three Dimensional Subclass as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.12.6 Subclass ID = 5-14 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.12.7 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.13 Class ID=10 Scripts

This style includes those typefaces that are designed to simulate handwriting. This IBM Class reflects the ISO Script Class and Uncial Class as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.13.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

A.13.2 Subclass ID = 1 : Uncial

This style is characterized by unjoined (nonconnecting) characters that are generally based on the hand writing style of Europe in the 6th to 9th centuries. An example of this font style is the Libra family. This IBM Subclass reflects the ISO Uncial Class as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.13.3 Subclass ID = 2 : Brush Joined

This style is characterized by joined (connecting) characters that have the appearance of being painted with a brush, with moderate contrast between thick and thin strokes. An example of this font style is the Mistral family. This IBM Subclass reflects the ISO Script Class, Joined Subclass, and Informal Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.13.4 Subclass ID = 3 : Formal Joined

This style is characterized by joined (connecting) characters that have a printed (or drawn with a stiff brush) appearance with extreme contrast between the thick and thin strokes. An example of this font style is the Coronet family. This IBM Subclass reflects the ISO Script Class, Joined Subclass, and Formal Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.13.5 Subclass ID = 4 : Monotone Joined

This style is characterized by joined (connecting) characters that have a uniform appearance with little or no contrast in the strokes. An example of this font style is the Kaufmann family. This IBM Subclass reflects the ISO Script Class, Joined Subclass, and Monotone Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.13.6 Subclass ID = 5 : Calligraphic

This style is characterized by beautifully hand drawn, unjoined (non-connecting) characters that have an appearance of being drawn with a broad edge pen. An example of this font style is the Thompson Quillscript family. This IBM Subclass reflects the ISO Script Class, Unjoined Subclass, and Calligraphic Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.13.7 Subclass ID = 6 : Brush Unjoined

This style is characterized by unjoined (non-connecting) characters that have the appearance of being painted with a brush, with moderate contrast between thick and thin strokes. An example of this font style is the Saltino family. This IBM Subclass reflects the ISO Script Class, Unjoined Subclass, and Brush Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.13.8 Subclass ID = 7 : Formal Unjoined

This style is characterized by unjoined (non-connecting) characters that have a printed (or drawn with a stiff brush) appearance with extreme contrast between the thick and thin strokes. An example of this font style is the Virtuosa family. This IBM Subclass reflects the ISO Script Class, Unjoined Subclass, and Formal Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.13.9 Subclass ID = 8 : Monotone Unjoined

This style is characterized by unjoined (non-connecting) characters that have a uniform appearance with little or no contrast in the strokes. An example of this font style is the Gilles Gothic family. This IBM Subclass reflects the ISO Script Class, Unjoined Subclass, and Monotone Specific Group as documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.13.10 Subclass ID = 9-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

A.13.11 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.14 Class ID=11 (reserved for future use)

This class ID is reserved for future assignment.

A.15 Class ID=12 Symbolic

This style is generally design independent, making it suitable for Pi and special characters (icons, dingbats, technical symbols, etc.) that may be used equally well with any font. This IBM Class reflects various ISO Specific Groups, as noted below and documented in the ISO/IEC 9541-1 Amendment 1 standard.

A.15.1 Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

A.15.2 Subclass ID = 1-2 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.15.3 Subclass ID = 3 : Mixed Serif

This style is characterized by either both or a combination of serif and sans serif designs on those characters of the font for which design is important (e.g., superscript and subscript characters, numbers, copyright or trademark symbols, etc.). An example of this font style is found in the IBM Symbol family. This IBM Subclass is not reflected in the ISO/IEC 9541-1 Amendment 1 standard.

A.15.4 Subclass ID = 4-5 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.15.5 Subclass ID = 6 : Oldstyle Serif

This style is characterized by a Oldstyle Serif IBM Class design on those characters of the font for which design is important (e.g., superscript and subscript characters, numbers, copyright or trademark symbols, etc.). An example of this font style is found in the IBM Sonoran Pi Serif family. This IBM Subclass is not directly reflected in the ISO/IEC 9541-1 Amendment 1 standard, though it is indirectly by the ISO Serif Class and Legibility Subclass (implies that all characters of the font exhibit the design appearance, while only a subset of the characters actually exhibit the design).

A.15.6 Subclass ID = 7 : Neo-grotesque Sans Serif

This style is characterized by a Neo-grotesque Sans Serif IBM Font Class and Subclass design on those characters of the font for which design is important (e.g., superscript and subscript characters, numbers, copyright or trademark symbols, etc.). An example of this font style is found in the IBM Sonoran Pi Sans Serif family. This IBM Subclass is not directly reflected in the ISO/IEC 9541-1 Amendment 1 standard, though it is indirectly by the ISO Sans Serif Class and Gothic Subclass (implies that all characters of the font exhibit the design appearance, while only a subset of the characters actually exhibit the design).

A.15.7 Subclass ID = 8-14 : (reserved for future use)

These subclass IDs are reserved for future assignment.

A.15.8 Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

A.16 Class ID=13 Reserved

A.17 Class ID=14 Reserved

Annex B
(informative)

Earlier versions of OS/2 – OS/2 and Windows metrics

B.1 OS/2 - OS/2 and Windows metrics (version 0)

NOTE This is maintained for purposes of being able to validate version 0 OS/2 tables. Please note that the description of the OS/2 table version 0 in the original Apple TrueType specification [7] differs from this document – the fields 'sTypoAscender', 'sTypoDescender', 'sTypoLineGap', 'usWinAscent' and 'usWinDescent' are missing in the Apple TrueType specification.

The OS/2 table consists of a set of metrics that are required in OFF fonts. The layout of version 0 of this table is as follows:

Type	Name of Entry	Comments
uint16	version	0x0000
int16	xAvgCharWidth	
uint16	usWeightClass	
uint16	usWidthClass	
uint16	fsType	
int16	ySubscriptXSize	
int16	ySubscriptYSize	
int16	ySubscriptXOffset	
int16	ySubscriptYOffset	
int16	ySuperscriptXSize	
int16	ySuperscriptYSize	
int16	ySuperscriptXOffset	
int16	ySuperscriptYOffset	
int16	yStrikeoutSize	
int16	yStrikeoutPosition	
int16	sFamilyClass	
uint8	panose[10]	
uint32	ulCharRange[4]	Bits 0-31
Tag	achVendID[4]	
uint16	fsSelection	
uint16	usFirstCharIndex	
uint16	usLastCharIndex	

int16	sTypoAscender	
int16	sTypoDescender	
int16	sTypoLineGap	
uint16	usWinAscent	
uint16	usWinDescent	

version

Format: uint16

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 zero (0).

xAvgCharWidth

Format: int16

Units: Pels / em units

Title: Average weighted escapement.

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

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

Letter	Weight Factor	Letter	Weight Factor
a	64	o	56
b	14	p	17
c	27	q	4

d	35	r	49
e	100	s	56
f	20	t	71
g	14	u	31
h	42	v	10
i	63	w	18
j	3	x	3
k	6	y	18
l	35	z	2
m	20	space	166
n	56		

usWeightClass

Format: uint16

Title: Weight class.

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

Comments:

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

usWidthClass

Format: uint16

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

fsType

Format: uint16

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.

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

Bit	Bit Mask	Description
	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.
4-15		Reserved, must be zero.

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

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

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

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

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

ySubscriptXSize

Format: int16

Units: Font design units

Title: Subscript horizontal font size.

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

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

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

ySubscriptYSize

Format: int16

Units: Font design units

Title: Subscript vertical font size.

Description: The recommended vertical size in font design units for subscripts for this font.

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

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

ySubscriptXOffset

Format: int16

Units: Font design units

Title: Subscript x offset.

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

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

ySubscriptYOffset

Format: int16

Units: Font design units

Title: Subscript y offset.

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

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

ySuperscriptXSize

Format: int16

Units: Font design units

Title: Superscript horizontal font size.

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

Comments: If a font has two recommended sizes for subscripts, e.g., numerics and other, the numeric sizes should be stressed. This size field maps to the em square size of the font being used for a subscript. The horizontal font size specifies a font designer's recommended horizontal font size for

superscript characters associated with this font. If a font does not include all of the required superscript characters for an application, and the application can substitute characters by scaling the character of a font or by substituting characters from another font, this parameter specifies the recommended em square for those superscript characters.

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

ySuperscriptYSize

Format: int16

Units: Font design units

Title: Superscript vertical font size.

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

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

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

ySuperscriptXOffset

Format: int16

Units: Font design units

Title: Superscript x offset.

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

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

ySuperscriptYOffset

Format: int16

Units: Font design units

Title: Superscript y offset.

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

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

yStrikeoutSize

Format: int16

Units: Font design units

Title: Strikeout size.

Description: Width of the strikeout stroke in font design units.

Comments: This field should normally be the width of the em dash for the current font. If the size is one, the strikeout line will be the line represented by the strikeout position field. If the value is two, the strikeout line will be the line represented by the strikeout position and the line immediately *above* the strikeout position. For a Roman font with a 2048 em square, 102 is suggested.

yStrikeoutPosition

Format: int16

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.

sFamilyClass

Format: int16

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 assigned by IBM to each font family. This parameter is intended for use in selecting an alternate font when the requested font is not available. The font class is the most general and the font subclass is the most specific. The high byte of this field contains the family class, while the low byte contains the family subclass.

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.

PANOSE		
Type	**Name**	**Description**
uint8	bFamilyType;	Family Type
uint8	bSerifStyle;	Serif Style
uint8	bWeight;	Weight
uint8	bProportion;	Proportion
uint8	bContrast;	Contrast
uint8	bStrokeVariation;	Stroke Variation
uint8	bArmStyle;	Arm Style
uint8	bLetterform;	Letterform
uint8	bMidline;	Midline
uint8	bXHeight;	X Height

Family Type	
Value	**Description**
0	Any
1	No Fit
2	Text and Display
3	Script
4	Decorative
5	Pictorial

Serif Style	
Value	**Description**
0	Any
1	No Fit
2	Cove
3	Obtuse Cove
4	Square Cove
5	Obtuse Square Cove
6	Square
7	Thin
8	Bone
9	Exaggerated
10	Triangle
11	Normal Sans
12	Obtuse Sans
13	Perp Sans
14	Flared
15	Rounded

Weight	
Value	**Description**
0	Any
1	No Fit
2	Very Light
3	Light
4	Thin
5	Book
6	Medium

7	Demi
8	Bold
9	Heavy
10	Black
11	Nord

Proportion	
Value	**Description**
0	Any
1	No Fit
2	Old Style
3	Modern
4	Even Width
5	Expanded
6	Condensed
7	Very Expanded
8	Very Condensed
9	Monospaced

Contrast	
Value	**Description**
0	Any
1	No Fit
2	None
3	Very Low
4	Low
5	Medium Low
6	Medium
7	Medium High
8	High
9	Very High

Stroke Variation	
Value	**Description**
0	Any
1	No Fit
2	Gradual/Diagonal
3	Gradual/Transitional

4	Gradual/Vertical
5	Gradual/Horizontal
6	Rapid/Vertical
7	Rapid/Horizontal
8	Instant/Vertical

Arm Style	
Value	**Description**
0	Any
1	No Fit
2	Straight Arms/Horizontal
3	Straight Arms/Wedge
4	Straight Arms/Vertical
5	Straight Arms/Single Serif
6	Straight Arms/Double Serif
7	Non-Straight Arms/Horizontal
8	Non-Straight Arms/Wedge
9	Non-Straight Arms/Vertical
10	Non-Straight Arms/Single Serif
11	Non-Straight Arms/Double Serif

Letterform	
Value	**Description**
0	Any
1	No Fit
2	Normal/Contact
3	Normal/Weighted
4	Normal/Boxed
5	Normal/Flattened
6	Normal/Rounded
7	Normal/Off Center
8	Normal/Square
9	Oblique/Contact
10	Oblique/Weighted
11	Oblique/Boxed
12	Oblique/Flattened
13	Oblique/Rounded
14	Oblique/Off Center

15	Oblique/Square

Midline	
Value	**Description**
0	Any
1	No Fit
2	Standard/Trimmed
3	Standard/Pointed
4	Standard/Serifed
5	High/Trimmed
6	High/Pointed
7	High/Serifed
8	Constant/Trimmed
9	Constant/Pointed
10	Constant/Serifed
11	Low/Trimmed
12	Low/Pointed
13	Low/Serifed

X-height	
Value	**Description**
0	Any
1	No Fit
2	Constant/Small
3	Constant/Standard
4	Constant/Large
5	Ducking/Small
6	Ducking/Standard
7	Ducking/Large

ulCharRange

Format: 16-byte unsigned long array (4 elements)

Title: Character Range

Description: This field is split conceptually into two bit fields of 96 and 32 bits each. The low 96 bits are used to specify the Unicode blocks encompassed by the font file. The high 32 bits are used to specify the character or script sets that are covered by the font file. The actual bit assignments are not yet completed; presently, all bits must be set to zero (0).

achVendID

Format: 4-byte Tag

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 bibliolgraphy.

fsSelection

Format: 2-byte bit field.

Title: Font selection flags.

Description: Contains information concerning the nature of the font patterns, as follows:

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

Comments: All undefined bits must be zero.

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

If bit 6 is set, then bits 0 and 5 must be clear, else the behavior is undefined. As noted above, the settings of bits 0 and 1 must be reflected in

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

usFirstCharIndex

Format: uint16uint16

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.

usLastCharIndex

Format: 2-byte uint16

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

sTypoAscender

Format: int16

Description: The typographic ascender for this font. 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 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. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table.

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.

sTypoDescender

Format: int16

Description: The typographic descender for this font. Remember that this is not the same as the Descender value in the 'hhea' table, 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. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table.

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 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.

sTypoLineGap

Format: int16

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 sTypoLineGap 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 chapter, "Recommendations for Windows Fonts). These new metrics, when combined with the character design widths, will allow applications to lay out documents in a typographically correct and portable fashion. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table.

usWinAscent

Format: uint16

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.

usWinDescent

Format: uint16

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.

B.2 OS/2 - OS/2 and Windows metrics (version 1)

NOTE This is maintained for purposes of being able to validate version 1 OS/2 tables.

The OS/2 table consists of a set of metrics that are required in OFF fonts. The layout of version 1 of this table is as follows:

Type	Name of Entry	Comments
uint16	version	0x0001
int16	xAvgCharWidth	
uint16	usWeightClass	
uint16	usWidthClass	
uint16	fsType	
int16	ySubscriptXSize	
int16	ySubscriptYSize	
int16	ySubscriptXOffset	
int16	ySubscriptYOffset	
int16	ySuperscriptXSize	
int16	ySuperscriptYSize	
int16	ySuperscriptXOffset	
int16	ySuperscriptYOffset	
int16	yStrikeoutSize	
int16	yStrikeoutPosition	
int16	sFamilyClass	
uint8	panose[10]	
uint32	ulUnicodeRange1	Bits 0-31
uint32	ulUnicodeRange2	Bits 32-63

uint32	ulUnicodeRange3	Bits 64-95
uint32	ulUnicodeRange4	Bits 96-127
Tag	achVendID[4]	
uint16	fsSelection	
uint16	usFirstCharIndex	
uint16	usLastCharIndex	
int16	sTypoAscender	
int16	sTypoDescender	
int16	sTypoLineGap	
uint16	usWinAscent	
uint16	usWinDescent	
uint32	ulCodePageRange1	Bits 0-31
uint32	ulCodePageRange2	Bits 32-63

version

Format: uint16

Units: n/a

Title: OS/2 table version number.

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

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

xAvgCharWidth

Format: int16

Units: Pels / em units

Title: Average weighted escapement.

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

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

is calculated according to this formula: Sum the individual character widths multiplied by the following weighting factors and then divide by 1000. For example:

Letter	Weight Factor	Letter	Weight Factor
a	64	O	56
b	14	P	17
c	27	Q	4
d	35	R	49
e	100	S	56
f	20	T	71
g	14	U	31
h	42	V	10
i	63	W	18
j	3	X	3
k	6	Y	18
l	35	Z	2
m	20	space	166
n	56		

usWeightClass

Format: uint16

Title: Weight class.

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

Comments:

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

usWidthClass

Format: uint16

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

fsType

Format: uint16

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

Bit	Bit Mask	Description
	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.
4-15		Reserved, must be zero.

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

Restricted License embedding (0x0002): Fonts that have this bit set **must not be modified, embedded or exchanged in any manner** without first obtaining permission of the legal owner. Caution:

NOTE For Restricted License embedding to take effect, it must be the only level of embedding selected (as noted in the previous paragraph).

Preview & Print embedding (0x0004): Fonts with this bit set indicate that they may be embedded within documents but must only be installed temporarily on the remote system. Any document which includes a Preview & Print embedded font must be opened “read-only;” the

application must not allow the user to edit the document; it can only be viewed and/or printed.

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

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

ySubscriptXSize

Format: int16

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 $1/10^{th}$ the size of the normal character.

ySubscriptYSize

Format: int16

Units: Font design units

Title: Subscript vertical font size.

Description: The recommended vertical size in font design units for subscripts for this font.

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

recommended horizontal EmInc for those subscript characters.

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

ySubscriptXOffset

Format: int16

Units: Font design units

Title: Subscript x offset.

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

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

ySubscriptYOffset

Format: int16

Units: Font design units

Title: Subscript y offset.

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

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

ySuperscriptXSize

Format: int16

Units: Font design units

Title: Superscript horizontal font size.

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

Comments: If a font has two recommended sizes for subscripts, e.g., numerics and other, the numeric sizes should be stressed. This size field maps to the em square size of the font being used for a subscript. The horizontal font size specifies a font designer's recommended horizontal font size for

superscript characters associated with this font. If a font does not include all of the required superscript characters for an application, and the application can substitute characters by scaling the character of a font or by substituting characters from another font, this parameter specifies the recommended em square for those superscript characters.

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

ySuperscriptYSize

Format: int16

Units: Font design units

Title: Superscript vertical font size.

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

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

ySuperscriptXOffset

Format: int16

Units: Font design units

Title: Superscript x offset.

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

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

ySuperscriptYOffset

Format: int16

Units: Font design units

Title: Superscript y offset.

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

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

yStrikeoutSize

Format: int16

Units: Font design units

Title: Strikeout size.

Description: Width of the strikeout stroke in font design units.

Comments: This field should normally be the width of the em dash for the current font. If the size is one, the strikeout line will be the line represented by the strikeout position field. If the value is two, the strikeout line will be the line represented by the strikeout position and the line immediately *above* the strikeout position. For a Roman font with a 2048 em square, 102 is suggested.

yStrikeoutPosition

Format: int16

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.

sFamilyClass

Format: int16

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 assigned by IBM to each font family. This parameter is intended for use in selecting an alternate font when the requested font is not available. The font class is the most general and the font subclass is the most specific. The high byte of this field contains the family class, while the low byte contains the family subclass.

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.

PANOSE		
Type	**Name**	**Description**
uint8	bFamilyType;	Family Type
uint8	bSerifStyle;	Serif Style
uint8	bWeight;	Weight
uint8	bProportion;	Proportion
uint8	bContrast;	Contrast
uint8	bStrokeVariation;	Stroke Variation
uint8	bArmStyle;	Arm Style
uint8	bLetterform;	Letterform
uint8	bMidline;	Midline
uint8	bXHeight;	X Height

Family Type	
Value	**Description**
0	Any
1	No Fit
2	Text and Display
3	Script
4	Decorative
5	Pictorial

Serif Style	
Value	**Description**
0	Any
1	No Fit

2	Cove
3	Obtuse Cove
4	Square Cove
5	Obtuse Square Cove
6	Square
7	Thin
8	Bone
9	Exaggerated
10	Triangle
11	Normal Sans
12	Obtuse Sans
13	Perp Sans
14	Flared
15	Rounded

Weight	
Value	**Description**
0	Any
1	No Fit
2	Very Light
3	Light
4	Thin
5	Book
6	Medium
7	Demi
8	Bold
9	Heavy
10	Black
11	Nord

Proportion	
Value	**Description**
0	Any
1	No Fit
2	Old Style
3	Modern
4	Even Width
5	Expanded
6	Condensed

7	Very Expanded
8	Very Condensed
9	Monospaced

Contrast	
Value	**Description**
0	Any
1	No Fit
2	None
3	Very Low
4	Low
5	Medium Low
6	Medium
7	Medium High
8	High
9	Very High

Stroke Variation	
Value	**Description**
0	Any
1	No Fit
2	Gradual/Diagonal
3	Gradual/Transitional
4	Gradual/Vertical
5	Gradual/Horizontal
6	Rapid/Vertical
7	Rapid/Horizontal
8	Instant/Vertical

Arm Style	
Value	**Description**
0	Any
1	No Fit
2	Straight Arms/Horizontal
3	Straight Arms/Wedge
4	Straight Arms/Vertical
5	Straight Arms/Single Serif
6	Straight Arms/Double Serif

7	Non-Straight Arms/Horizontal
8	Non-Straight Arms/Wedge
9	Non-Straight Arms/Vertical
10	Non-Straight Arms/Single Serif
11	Non-Straight Arms/Double Serif

Letterform	
Value	Description
0	Any
1	No Fit
2	Normal/Contact
3	Normal/Weighted
4	Normal/Boxed
5	Normal/Flattened
6	Normal/Rounded
7	Normal/Off Center
8	Normal/Square
9	Oblique/Contact
10	Oblique/Weighted
11	Oblique/Boxed
12	Oblique/Flattened
13	Oblique/Rounded
14	Oblique/Off Center
15	Oblique/Square

Midline	
Value	**Description**
0	Any
1	No Fit
2	Standard/Trimmed
3	Standard/Pointed
4	Standard/Serifed
5	High/Trimmed
6	High/Pointed
7	High/Serifed
8	Constant/Trimmed
9	Constant/Pointed
10	Constant/Serifed

11	Low/Trimmed
12	Low/Pointed
13	Low/Serifed

X-height	
Value	**Description**
0	Any
1	No Fit
2	Constant/Small
3	Constant/Standard
4	Constant/Large
5	Ducking/Small
6	Ducking/Standard
7	Ducking/Large

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

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

Title: Unicode Character Range

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

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

Bit	Description
0	Basic Latin
1	Latin-1 Supplement
2	Latin Extended-A
3	Latin Extended-B
4	IPA Extensions
5	Spacing Modifier Letters
6	Combining Diacritical Marks

7	Basic Greek
8	Greek Symbols and Coptic
9	Cyrillic
10	Armenian
11	Basic Hebrew
12	Hebrew Extended (A and B blocks combined)
13	Basic Arabic
14	Arabic Extended
15	Devanagari
16	Bengali
17	Gurmukhi
18	Gujarati
19	Oriya
20	Tamil
21	Telugu
22	Kannada
23	Malayalam
24	Thai
25	Lao
26	Basic Georgian
27	Georgian Extended
28	Hangul Jamo
29	Latin Extended Additional
30	Greek Extended
31	General Punctuation
32	Superscripts And Subscripts
33	Currency Symbols
34	Combining Diacritical Marks For Symbols
35	Letterlike Symbols
36	Number Forms
37	Arrows
38	Mathematical Operators
39	Miscellaneous Technical
40	Control Pictures
41	Optical Character Recognition
42	Enclosed Alphanumerics
43	Box Drawing
44	Block Elements

45	Geometric Shapes
46	Miscellaneous Symbols
47	Dingbats
48	CJK Symbols And Punctuation
49	Hiragana
50	Katakana
51	Bopomofo
52	Hangul Compatibility Jamo
53	CJK Miscellaneous
54	Enclosed CJK Letters And Months
55	CJK Compatibility
56	Hangul
57	Reserved for Unicode SubRanges
58	Reserved for Unicode SubRanges
59	CJK Unified Ideographs
60	Private Use Area
61	CJK Compatibility Ideographs
62	Alphabetic Presentation Forms
63	Arabic Presentation Forms-A
64	Combining Half Marks
65	CJK Compatibility Forms
66	Small Form Variants
67	Arabic Presentation Forms-B
68	Halfwidth And Fullwidth Forms
69	Specials
70-127	Reserved for Unicode SubRanges

achVendID

Format: 4-byte Tag

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 bibliolgraphy.

fsSelection

Format: 2-byte bit field.

Title: Font selection flags.

Description: Contains information concerning the nature of the font patterns, as follows:

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

Comments: All undefined bits must be zero.

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

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

usFirstCharIndex

Format: 2-byte USHORT

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

usLastCharIndex

Format: uint16

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

sTypoAscender

Format: int16

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 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. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table.

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.

sTypoDescender

Format: int16

Description: The typographic descender for this font. Remember that this is not the same as the Descender value in the 'hhea' table 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. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table.

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 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.

sTypoLineGap

Format: int16

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 sTypoLineGap 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 chapter, "Recommendations for Windows Fonts). These new metrics, when combined with the character design widths, will allow applications to lay out documents in a typographically correct and portable fashion. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table.

usWinAscent

Format: uint16

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.

usWinDescent

Format: uint16

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.

ulCodePageRange1 Bits 0-31
ulCodePageRange2 Bits 32-63

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

Title: Code Page Character Range

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

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

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

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

49	866	MS-DOS Russian
50	865	MS-DOS Nordic
51	864	Arabic
52	863	MS-DOS Canadian French
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

B.3 OS/2 - OS/2 and Windows metrics (version 3)

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

NOTE This is maintained for purposes of being able to validate version 3 OS/2 tables.

Type	Name of Entry	Comments
uint16	Version	0x0003
int16	xAvgCharWidth	
uint16	usWeightClass	
uint16	usWidthClass	
uint16	fsType	
int16	ySubscriptXSize	
int16	ySubscriptYSize	
int16	ySubscriptXOffset	
int16	ySubscriptYOffset	
int16	ySuperscriptXSize	
int16	ySuperscriptYSize	
int16	ySuperscriptXOffset	
int16	ySuperscriptYOffset	
int16	yStrikeoutSize	
int16	yStrikeoutPosition	
int16	sFamilyClass	

uint8	Panose[10]	
uint32	ulUnicodeRange1	Bits 0-31
uint32	ulUnicodeRange2	Bits 32-63 version 0x0001 and later
uint32	ulUnicodeRange3	Bits 64-95 version 0x0001 and later
uint32	ulUnicodeRange4	Bits 96-127 version 0x0001 and later
Tag	achVendID[4]	
uint16	fsSelection	
uint16	usFirstCharIndex	
uint16	usLastCharIndex	
int16	sTypoAscender	
int16	sTypoDescender	
int16	sTypoLineGap	
uint16	usWinAscent	
uint16	usWinDescent	
uint32	ulCodePageRange1	Bits 0-31 version 0x0001 and later
uint32	ulCodePageRange2	Bits 32-63 version 0x0001 and later
int16	sxHeight	version 0x0002 and later
int16	sCapHeight	version 0x0002 and later
uint16	usDefaultChar	version 0x0002 and later
uint16	usBreakChar	version 0x0002 and later
uint16	usMaxContext	version 0x0002 and later

version

Format: uint16

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 three (3). See Annex A.

xAvgCharWidth

Format: int16

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 A.

usWeightClass

Format: uint16

Title: Weight class.

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

Comments:

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

usWidthClass

Format: uint16

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

fsType

Format: uint16

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.

ySubscriptXSize

Format: int16

Units: Font design units

Title: Subscript horizontal font size.

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

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

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

ySubscriptYSize

Format: int16

Units: Font design units

Title: Subscript vertical font size.

Description: The recommended vertical size in font design units for subscripts for this font.

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

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

ySubscriptXOffset

Format: int16

Units: Font design units

Title: Subscript x offset.

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

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

ySubscriptYOffset

Format: int16

Units: Font design units

Title: Subscript y offset.

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

Comments: The Subscript Y offset parameter specifies a font designer's recommended vertical offset from

the character baseline to the character baseline for subscript characters associated with this font. Values are expressed as a positive offset below the character baseline. If a font does not include all of the required subscript for an application, this parameter specifies the recommended vertical distance below the character baseline for those subscript characters.

ySuperscriptXSize

Format: int16

Units: Font design units

Title: Superscript horizontal font size.

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

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

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

ySuperscriptYSize

Format: int16

Units: Font design units

Title: Superscript vertical font size.

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

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

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

ySuperscriptXOffset

Format: int16

Units: Font design units

Title: Superscript x offset.

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

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

ySuperscriptYOffset

Format: int16

Units: Font design units

Title: Superscript y offset.

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

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

yStrikeoutSize

Format: int16

Units: Font design units

Title: Strikeout size.

Description: Width of the strikeout stroke in font design units.

Comments: This field should normally be the width of the em dash for the current font. If the size is one, the strikeout line will be the line represented by the strikeout position field. If the value is two, the strikeout line will be the line represented by the strikeout position and the line immediately *above* the strikeout position. For a Roman font with a 2048 em square, 102 is suggested.

yStrikeoutPosition

Format: int16

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.

sFamilyClass

Format: int16

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 A. 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.

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
uint8	bFamilyType;
uint8	bSerifStyle;
uint8	bWeight;
uint8	bProportion;
uint8	bContrast;
uint8	bStrokeVariation;
uint8	bArmStyle;
uint8	bLetterform;
uint8	bMidline;
uint8	bXHeight;

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 (Windows platform). If the bit is set (1) then the Unicode range is considered functional. If the bit is clear (0) then the range is not considered functional. Each of the bits is treated as an independent flag and the bits can be set in any combination. The determination of "functional" is left up to the font designer, although character set selection should attempt to be functional by ranges if at all possible.

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

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

24	Thai
25	Lao
26	Georgian
27	Reserved for Unicode SubRanges
28	Hangul Jamo
29	Latin Extended Additional
30	Greek Extended
31	General Punctuation
32	Superscripts And Subscripts
33	Currency Symbols
34	Combining Diacritical Marks For Symbols
35	Letterlike Symbols
36	Number Forms
37	Arrows
	Supplemental Arrows-A
	Supplemental Arrows-B
38	Mathematical Operators
	Supplemental Mathematical Operators
	Miscellaneous Mathematical Symbols-A
	Miscellaneous Mathematical Symbols-B
39	Miscellaneous Technical
40	Control Pictures
41	Optical Character Recognition
42	Enclosed Alphanumerics
43	Box Drawing
44	Block Elements
45	Geometric Shapes
46	Miscellaneous Symbols
47	Dingbats
48	CJK Symbols And Punctuation
49	Hiragana
50	Katakana
	Katakana Phonetic Extensions
51	Bopomofo
	Bopomofo Extended
52	Hangul Compatibility Jamo
53	Reserved for Unicode SubRanges
54	Enclosed CJK Letters And Months

55	CJK Compatibility
56	Hangul Syllables
57	Non-Plane 0 *
58	Reserved for Unicode SubRanges
59	CJK Unified Ideographs
	CJK Radicals Supplement
	Kangxi Radicals
	Ideographic Description Characters
	CJK Unified Ideograph Extension A
	CJK Unified Ideographs Extension B
	Kanbun
60	Private Use Area
61	CJK Compatibility Ideographs
	CJK Compatibility Ideographs Supplement
62	Alphabetic Presentation Forms
63	Arabic Presentation Forms-A
64	Combining Half Marks
65	CJK Compatibility Forms
66	Small Form Variants
67	Arabic Presentation Forms-B
68	Halfwidth And Fullwidth Forms
69	Specials
70	Tibetan
71	Syriac
72	Thaana
73	Sinhala
74	Myanmar
75	Ethiopic
76	Cherokee
77	Unified Canadian Aboriginal Syllabics
78	Ogham
79	Runic
80	Khmer
81	Mongolian
82	Braille Patterns
83	Yi Syllables
	Yi Radicals

84	Tagalog
	Hanunoo
	Buhid
	Tagbanwa
85	Old Italic
86	Gothic
87	Deseret
88	Byzantine Musical Symbols
	Musical Symbols
89	Mathematical Alphanumeric Symbols
90	Private Use (plane 15)
	Private Use (plane 16)
91	Variation Selectors
92	Tags
93-127	Reserved for Unicode SubRanges

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

achVendID

Format: 4-byte Tag

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 bibliolgraphy.

fsSelection

Format: 2-byte bit field.

Title: Font selection flags.

Description: Contains information concerning the nature of the font patterns, as follows:

Bit #	macStyle bit	C definition	Description
0	bit 1	ITALIC	Font contains Italic characters, otherwise they are upright.

1		UNDERSCORE	Characters are underscored.
2		NEGATIVE	Characters have their foreground and background reversed.
3		OUTLINED	Outline (hollow) characters, otherwise they are solid.
4		STRIKEOUT	Characters are overstruck.
5	bit 0	BOLD	Characters are emboldened.
6		REGULAR	Characters are in the standard weight/style for the font.

Comments: All undefined bits must be zero.

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

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

usFirstCharIndex

Format: uint16

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.

usLastCharIndex

Format: uint16

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.

sTypoAscender

Format: int16

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 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.

sTypoDescender

Format: int16

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 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.

sTypoLineGap

Format: int16

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 sTypoLineGap 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.

usWinAscent

Format: uint16

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.

usWinDescent

Format: uint16

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.

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

sxHeight

Format: int16

Description: This metric specifies the distance between the baseline and the approximate height of non-ascending lowercase letters measured in font design units. 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.

sCapHeight

Format: int16

Description: This metric specifies the distance between the baseline and the approximate height of uppercase letters measured in font design units. 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.

usDefaultChar

Format: uint16

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).

usBreakChar

Format: uint16

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).

usMaxContext

Format: uint16

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.

B.4 OS/2 - OS/2 and Windows metrics (version 4)

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

NOTE This is maintained for purposes of being able to validate version 4 OS/2 tables.

Type	Name of Entry	Comments
uint16	Version	0x0004
int16	xAvgCharWidth	
uint16	usWeightClass	
uint16	usWidthClass	
uint16	fsType	
int16	ySubscriptXSize	
int16	ySubscriptYSize	
int16	ySubscriptXOffset	
int16	ySubscriptYOffset	
int16	ySuperscriptXSize	
int16	ySuperscriptYSize	
int16	ySuperscriptXOffset	
int16	ySuperscriptYOffset	
int16	yStrikeoutSize	
int16	yStrikeoutPosition	
int16	sFamilyClass	
uint8	Panose[10]	
uint32	ulUnicodeRange1	Bits 0-31
uint32	ulUnicodeRange2	Bits 32-63 version 0x0001 and later
uint32	ulUnicodeRange3	Bits 64-95 version 0x0001 and later
uint32	ulUnicodeRange4	Bits 96-127 version 0x0001 and later
Tag	achVendID[4]	
uint16	fsSelection	
uint16	usFirstCharIndex	
uint16	usLastCharIndex	
int16	sTypoAscender	
int16	sTypoDescender	
int16	sTypoLineGap	
uint16	usWinAscent	
uint16	usWinDescent	

uint32	ulCodePageRange1	Bits 0-31 version 0x0001 and later
uint32	ulCodePageRange2	Bits 32-63 version 0x0001 and later
int16	sxHeight	version 0x0002 and later
int16	sCapHeight	version 0x0002 and later
uint16	usDefaultChar	version 0x0002 and later
uint16	usBreakChar	version 0x0002 and later
uint16	usMaxContext	version 0x0002 and later

version

Format: uint16

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 B.

xAvgCharWidth

Format: int16

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 A.

usWeightClass

Format: uint16

Title: Weight class.

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

Comments:

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

usWidthClass

Format: uint16

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

fsType

Format: uint16

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.

ySubscriptXSize

Format: int16

Units: Font design units

Title: Subscript horizontal font size.

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

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

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

ySubscriptYSize

Format: int16

Units: Font design units

Title: Subscript vertical font size.

Description: The recommended vertical size in font design units for subscripts for this font.

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

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

ySubscriptXOffset

Format: int16

Units: Font design units

Title: Subscript x offset.

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

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

ySubscriptYOffset

Format: int16

Units: Font design units

Title: Subscript y offset.

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

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

ySuperscriptXSize

Format: int16

Units: Font design units

Title: Superscript horizontal font size.

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

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

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

ySuperscriptYSize

Format: int16

Units: Font design units

Title: Superscript vertical font size.

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

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

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

ySuperscriptXOffset

Format: int16

Units: Font design units

Title: Superscript x offset.

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

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

ySuperscriptYOffset

Format: int16

Units: Font design units

Title: Superscript y offset.

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

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

yStrikeoutSize

Format: int16

Units: Font design units

Title: Strikeout size.

Description: Width of the strikeout stroke in font design units.

Comments: This field should normally be the width of the em dash for the current font. If the size is one, the strikeout line will be the line represented by the strikeout position field. If the value is two, the strikeout line will be the line represented by the strikeout position and the line immediately *above* the strikeout position. For a Roman font with a 2048 em square, 102 is suggested.

yStrikeoutPosition

Format: int16

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.

sFamilyClass

Format: int16

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 A. 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.

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'.

Comments: The specification for assigning PANOSE values [14] can be found in bibliography.

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)	F0000-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.

achVendID

Format: 4-byte Tag

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 bibliolgraphy.

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 5 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.

usFirstCharIndex

Format: uint16

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.

usLastCharIndex

Format: uint16

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.

sTypoAscender

Format: int16

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 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.

sTypoDescender

Format: int16

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.

sTypoLineGap

Format: int16

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 sTypoLineGap 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.

usWinAscent

Format: uint16

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.

usWinDescent

Format: uint16

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.

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
32-46		Reserved for OEM
47		Reserved
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

sxHeight

Format: int16

Description: This metric specifies the distance between the baseline and the approximate height of non-ascending lowercase letters measured in font design units. 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.

sCapHeight

Format: int16

Description: This metric specifies the distance between the baseline and the approximate height of uppercase letters measured in font design units. 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.

usDefaultChar

Format: uint16

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.

usBreakChar

Format: uint16

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.

usMaxContext

Format: uint16

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.

Annex C
(informative)

OFF Mirroring Pairs List

This file is a copy of the Bidi_Mirroring_Glyph Property of Unicode 5.1 (http://www.unicode.org/Public/5.1.0/ucd/BidiMirroring.txt), with header comments changed and the commented list at the end removed. Consult the URL above for specifications for the format of the data.

The data in this Annex will not be revised.

See the section "Left-to-right and right-to-left text" in subclause 6.1.4 for a description of how this file is to be used by a text layout engine.

```
0028; 0029 # LEFT PARENTHESIS
0029; 0028 # RIGHT PARENTHESIS
003C; 003E # LESS-THAN SIGN
003E; 003C # GREATER-THAN SIGN
005B; 005D # LEFT SQUARE BRACKET
005D; 005B # RIGHT SQUARE BRACKET
007B; 007D # LEFT CURLY BRACKET
007D; 007B # RIGHT CURLY BRACKET
00AB; 00BB # LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
00BB; 00AB # RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
0F3A; 0F3B # TIBETAN MARK GUG RTAGS GYON
0F3B; 0F3A # TIBETAN MARK GUG RTAGS GYAS
0F3C; 0F3D # TIBETAN MARK ANG KHANG GYON
0F3D; 0F3C # TIBETAN MARK ANG KHANG GYAS
169B; 169C # OGHAM FEATHER MARK
169C; 169B # OGHAM REVERSED FEATHER MARK
2039; 203A # SINGLE LEFT-POINTING ANGLE QUOTATION MARK
203A; 2039 # SINGLE RIGHT-POINTING ANGLE QUOTATION MARK
2045; 2046 # LEFT SQUARE BRACKET WITH QUILL
2046; 2045 # RIGHT SQUARE BRACKET WITH QUILL
207D; 207E # SUPERSCRIPT LEFT PARENTHESIS
207E; 207D # SUPERSCRIPT RIGHT PARENTHESIS
208D; 208E # SUBSCRIPT LEFT PARENTHESIS
208E; 208D # SUBSCRIPT RIGHT PARENTHESIS
2208; 220B # ELEMENT OF
2209; 220C # NOT AN ELEMENT OF
220A; 220D # SMALL ELEMENT OF
220B; 2208 # CONTAINS AS MEMBER
220C; 2209 # DOES NOT CONTAIN AS MEMBER
220D; 220A # SMALL CONTAINS AS MEMBER
2215; 29F5 # DIVISION SLASH
223C; 223D # TILDE OPERATOR
223D; 223C # REVERSED TILDE
2243; 22CD # ASYMPTOTICALLY EQUAL TO
2252; 2253 # APPROXIMATELY EQUAL TO OR THE IMAGE OF
2253; 2252 # IMAGE OF OR APPROXIMATELY EQUAL TO
2254; 2255 # COLON EQUALS
2255; 2254 # EQUALS COLON
2264; 2265 # LESS-THAN OR EQUAL TO
2265; 2264 # GREATER-THAN OR EQUAL TO
2266; 2267 # LESS-THAN OVER EQUAL TO
2267; 2266 # GREATER-THAN OVER EQUAL TO
```

```
2268; 2269 # [BEST FIT] LESS-THAN BUT NOT EQUAL TO
2269; 2268 # [BEST FIT] GREATER-THAN BUT NOT EQUAL TO
226A; 226B # MUCH LESS-THAN
226B; 226A # MUCH GREATER-THAN
226E; 226F # [BEST FIT] NOT LESS-THAN
226F; 226E # [BEST FIT] NOT GREATER-THAN
2270; 2271 # [BEST FIT] NEITHER LESS-THAN NOR EQUAL TO
2271; 2270 # [BEST FIT] NEITHER GREATER-THAN NOR EQUAL TO
2272; 2273 # [BEST FIT] LESS-THAN OR EQUIVALENT TO
2273; 2272 # [BEST FIT] GREATER-THAN OR EQUIVALENT TO
2274; 2275 # [BEST FIT] NEITHER LESS-THAN NOR EQUIVALENT TO
2275; 2274 # [BEST FIT] NEITHER GREATER-THAN NOR EQUIVALENT TO
2276; 2277 # LESS-THAN OR GREATER-THAN
2277; 2276 # GREATER-THAN OR LESS-THAN
2278; 2279 # [BEST FIT] NEITHER LESS-THAN NOR GREATER-THAN
2279; 2278 # [BEST FIT] NEITHER GREATER-THAN NOR LESS-THAN
227A; 227B # PRECEDES
227B; 227A # SUCCEEDS
227C; 227D # PRECEDES OR EQUAL TO
227D; 227C # SUCCEEDS OR EQUAL TO
227E; 227F # [BEST FIT] PRECEDES OR EQUIVALENT TO
227F; 227E # [BEST FIT] SUCCEEDS OR EQUIVALENT TO
2280; 2281 # [BEST FIT] DOES NOT PRECEDE
2281; 2280 # [BEST FIT] DOES NOT SUCCEED
2282; 2283 # SUBSET OF
2283; 2282 # SUPERSET OF
2284; 2285 # [BEST FIT] NOT A SUBSET OF
2285; 2284 # [BEST FIT] NOT A SUPERSET OF
2286; 2287 # SUBSET OF OR EQUAL TO
2287; 2286 # SUPERSET OF OR EQUAL TO
2288; 2289 # [BEST FIT] NEITHER A SUBSET OF NOR EQUAL TO
2289; 2288 # [BEST FIT] NEITHER A SUPERSET OF NOR EQUAL TO
228A; 228B # [BEST FIT] SUBSET OF WITH NOT EQUAL TO
228B; 228A # [BEST FIT] SUPERSET OF WITH NOT EQUAL TO
228F; 2290 # SQUARE IMAGE OF
2290; 228F # SQUARE ORIGINAL OF
2291; 2292 # SQUARE IMAGE OF OR EQUAL TO
2292; 2291 # SQUARE ORIGINAL OF OR EQUAL TO
2298; 29B8 # CIRCLED DIVISION SLASH
22A2; 22A3 # RIGHT TACK
22A3; 22A2 # LEFT TACK
22A6; 2ADE # ASSERTION
22A8; 2AE4 # TRUE
22A9; 2AE3 # FORCES
22AB; 2AE5 # DOUBLE VERTICAL BAR DOUBLE RIGHT TURNSTILE
22B0; 22B1 # PRECEDES UNDER RELATION
22B1; 22B0 # SUCCEEDS UNDER RELATION
22B2; 22B3 # NORMAL SUBGROUP OF
22B3; 22B2 # CONTAINS AS NORMAL SUBGROUP
22B4; 22B5 # NORMAL SUBGROUP OF OR EQUAL TO
22B5; 22B4 # CONTAINS AS NORMAL SUBGROUP OR EQUAL TO
22B6; 22B7 # ORIGINAL OF
22B7; 22B6 # IMAGE OF
22C9; 22CA # LEFT NORMAL FACTOR SEMIDIRECT PRODUCT
22CA; 22C9 # RIGHT NORMAL FACTOR SEMIDIRECT PRODUCT
22CB; 22CC # LEFT SEMIDIRECT PRODUCT
22CC; 22CB # RIGHT SEMIDIRECT PRODUCT
22CD; 2243 # REVERSED TILDE EQUALS
22D0; 22D1 # DOUBLE SUBSET
22D1; 22D0 # DOUBLE SUPERSET
22D6; 22D7 # LESS-THAN WITH DOT
22D7; 22D6 # GREATER-THAN WITH DOT
```

```
22D8; 22D9 # VERY MUCH LESS-THAN
22D9; 22D8 # VERY MUCH GREATER-THAN
22DA; 22DB # LESS-THAN EQUAL TO OR GREATER-THAN
22DB; 22DA # GREATER-THAN EQUAL TO OR LESS-THAN
22DC; 22DD # EQUAL TO OR LESS-THAN
22DD; 22DC # EQUAL TO OR GREATER-THAN
22DE; 22DF # EQUAL TO OR PRECEDES
22DF; 22DE # EQUAL TO OR SUCCEEDS
22E0; 22E1 # [BEST FIT] DOES NOT PRECEDE OR EQUAL
22E1; 22E0 # [BEST FIT] DOES NOT SUCCEED OR EQUAL
22E2; 22E3 # [BEST FIT] NOT SQUARE IMAGE OF OR EQUAL TO
22E3; 22E2 # [BEST FIT] NOT SQUARE ORIGINAL OF OR EQUAL TO
22E4; 22E5 # [BEST FIT] SQUARE IMAGE OF OR NOT EQUAL TO
22E5; 22E4 # [BEST FIT] SQUARE ORIGINAL OF OR NOT EQUAL TO
22E6; 22E7 # [BEST FIT] LESS-THAN BUT NOT EQUIVALENT TO
22E7; 22E6 # [BEST FIT] GREATER-THAN BUT NOT EQUIVALENT TO
22E8; 22E9 # [BEST FIT] PRECEDES BUT NOT EQUIVALENT TO
22E9; 22E8 # [BEST FIT] SUCCEEDS BUT NOT EQUIVALENT TO
22EA; 22EB # [BEST FIT] NOT NORMAL SUBGROUP OF
22EB; 22EA # [BEST FIT] DOES NOT CONTAIN AS NORMAL SUBGROUP
22EC; 22ED # [BEST FIT] NOT NORMAL SUBGROUP OF OR EQUAL TO
22ED; 22EC # [BEST FIT] DOES NOT CONTAIN AS NORMAL SUBGROUP OR EQUAL
22F0; 22F1 # UP RIGHT DIAGONAL ELLIPSIS
22F1; 22F0 # DOWN RIGHT DIAGONAL ELLIPSIS
22F2; 22FA # ELEMENT OF WITH LONG HORIZONTAL STROKE
22F3; 22FB # ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE
22F4; 22FC # SMALL ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE
22F6; 22FD # ELEMENT OF WITH OVERBAR
22F7; 22FE # SMALL ELEMENT OF WITH OVERBAR
22FA; 22F2 # CONTAINS WITH LONG HORIZONTAL STROKE
22FB; 22F3 # CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE
22FC; 22F4 # SMALL CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE
22FD; 22F6 # CONTAINS WITH OVERBAR
22FE; 22F7 # SMALL CONTAINS WITH OVERBAR
2308; 2309 # LEFT CEILING
2309; 2308 # RIGHT CEILING
230A; 230B # LEFT FLOOR
230B; 230A # RIGHT FLOOR
2329; 232A # LEFT-POINTING ANGLE BRACKET
232A; 2329 # RIGHT-POINTING ANGLE BRACKET
2768; 2769 # MEDIUM LEFT PARENTHESIS ORNAMENT
2769; 2768 # MEDIUM RIGHT PARENTHESIS ORNAMENT
276A; 276B # MEDIUM FLATTENED LEFT PARENTHESIS ORNAMENT
276B; 276A # MEDIUM FLATTENED RIGHT PARENTHESIS ORNAMENT
276C; 276D # MEDIUM LEFT-POINTING ANGLE BRACKET ORNAMENT
276D; 276C # MEDIUM RIGHT-POINTING ANGLE BRACKET ORNAMENT
276E; 276F # HEAVY LEFT-POINTING ANGLE QUOTATION MARK ORNAMENT
276F; 276E # HEAVY RIGHT-POINTING ANGLE QUOTATION MARK ORNAMENT
2770; 2771 # HEAVY LEFT-POINTING ANGLE BRACKET ORNAMENT
2771; 2770 # HEAVY RIGHT-POINTING ANGLE BRACKET ORNAMENT
2772; 2773 # LIGHT LEFT TORTOISE SHELL BRACKET
2773; 2772 # LIGHT RIGHT TORTOISE SHELL BRACKET
2774; 2775 # MEDIUM LEFT CURLY BRACKET ORNAMENT
2775; 2774 # MEDIUM RIGHT CURLY BRACKET ORNAMENT
27C3; 27C4 # OPEN SUBSET
27C4; 27C3 # OPEN SUPERSET
27C5; 27C6 # LEFT S-SHAPED BAG DELIMITER
27C6; 27C5 # RIGHT S-SHAPED BAG DELIMITER
27C8; 27C9 # REVERSE SOLIDUS PRECEDING SUBSET
27C9; 27C8 # SUPERSET PRECEDING SOLIDUS
27D5; 27D6 # LEFT OUTER JOIN
27D6; 27D5 # RIGHT OUTER JOIN
```



```
27DD; 27DE # LONG RIGHT TACK
27DE; 27DD # LONG LEFT TACK
27E2; 27E3 # WHITE CONCAVE-SIDED DIAMOND WITH LEFTWARDS TICK
27E3; 27E2 # WHITE CONCAVE-SIDED DIAMOND WITH RIGHTWARDS TICK
27E4; 27E5 # WHITE SQUARE WITH LEFTWARDS TICK
27E5; 27E4 # WHITE SQUARE WITH RIGHTWARDS TICK
27E6; 27E7 # MATHEMATICAL LEFT WHITE SQUARE BRACKET
27E7; 27E6 # MATHEMATICAL RIGHT WHITE SQUARE BRACKET
27E8; 27E9 # MATHEMATICAL LEFT ANGLE BRACKET
27E9; 27E8 # MATHEMATICAL RIGHT ANGLE BRACKET
27EA; 27EB # MATHEMATICAL LEFT DOUBLE ANGLE BRACKET
27EB; 27EA # MATHEMATICAL RIGHT DOUBLE ANGLE BRACKET
27EC; 27ED # MATHEMATICAL LEFT WHITE TORTOISE SHELL BRACKET
27ED; 27EC # MATHEMATICAL RIGHT WHITE TORTOISE SHELL BRACKET
27EE; 27EF # MATHEMATICAL LEFT FLATTENED PARENTHESIS
27EF; 27EE # MATHEMATICAL RIGHT FLATTENED PARENTHESIS
2983; 2984 # LEFT WHITE CURLY BRACKET
2984; 2983 # RIGHT WHITE CURLY BRACKET
2985; 2986 # LEFT WHITE PARENTHESIS
2986; 2985 # RIGHT WHITE PARENTHESIS
2987; 2988 # Z NOTATION LEFT IMAGE BRACKET
2988; 2987 # Z NOTATION RIGHT IMAGE BRACKET
2989; 298A # Z NOTATION LEFT BINDING BRACKET
298A; 2989 # Z NOTATION RIGHT BINDING BRACKET
298B; 298C # LEFT SQUARE BRACKET WITH UNDERBAR
298C; 298B # RIGHT SQUARE BRACKET WITH UNDERBAR
298D; 2990 # LEFT SQUARE BRACKET WITH TICK IN TOP CORNER
298E; 298F # RIGHT SQUARE BRACKET WITH TICK IN BOTTOM CORNER
298F; 298E # LEFT SQUARE BRACKET WITH TICK IN BOTTOM CORNER
2990; 298D # RIGHT SQUARE BRACKET WITH TICK IN TOP CORNER
2991; 2992 # LEFT ANGLE BRACKET WITH DOT
2992; 2991 # RIGHT ANGLE BRACKET WITH DOT
2993; 2994 # LEFT ARC LESS-THAN BRACKET
2994; 2993 # RIGHT ARC GREATER-THAN BRACKET
2995; 2996 # DOUBLE LEFT ARC GREATER-THAN BRACKET
2996; 2995 # DOUBLE RIGHT ARC LESS-THAN BRACKET
2997; 2998 # LEFT BLACK TORTOISE SHELL BRACKET
2998; 2997 # RIGHT BLACK TORTOISE SHELL BRACKET
29B8; 2298 # CIRCLED REVERSE SOLIDUS
29C0; 29C1 # CIRCLED LESS-THAN
29C1; 29C0 # CIRCLED GREATER-THAN
29C4; 29C5 # SQUARED RISING DIAGONAL SLASH
29C5; 29C4 # SQUARED FALLING DIAGONAL SLASH
29CF; 29D0 # LEFT TRIANGLE BESIDE VERTICAL BAR
29D0; 29CF # VERTICAL BAR BESIDE RIGHT TRIANGLE
29D1; 29D2 # BOWTIE WITH LEFT HALF BLACK
29D2; 29D1 # BOWTIE WITH RIGHT HALF BLACK
29D4; 29D5 # TIMES WITH LEFT HALF BLACK
29D5; 29D4 # TIMES WITH RIGHT HALF BLACK
29D8; 29D9 # LEFT WIGGLY FENCE
29D9; 29D8 # RIGHT WIGGLY FENCE
29DA; 29DB # LEFT DOUBLE WIGGLY FENCE
29DB; 29DA # RIGHT DOUBLE WIGGLY FENCE
29F5; 2215 # REVERSE SOLIDUS OPERATOR
29F8; 29F9 # BIG SOLIDUS
29F9; 29F8 # BIG REVERSE SOLIDUS
29FC; 29FD # LEFT-POINTING CURVED ANGLE BRACKET
29FD; 29FC # RIGHT-POINTING CURVED ANGLE BRACKET
2A2B; 2A2C # MINUS SIGN WITH FALLING DOTS
2A2C; 2A2B # MINUS SIGN WITH RISING DOTS
2A2D; 2A2E # PLUS SIGN IN LEFT HALF CIRCLE
2A2E; 2A2D # PLUS SIGN IN RIGHT HALF CIRCLE
```

```
2A34; 2A35 # MULTIPLICATION SIGN IN LEFT HALF CIRCLE
2A35; 2A34 # MULTIPLICATION SIGN IN RIGHT HALF CIRCLE
2A3C; 2A3D # INTERIOR PRODUCT
2A3D; 2A3C # RIGHTHAND INTERIOR PRODUCT
2A64; 2A65 # Z NOTATION DOMAIN ANTIRESTRICTION
2A65; 2A64 # Z NOTATION RANGE ANTIRESTRICTION
2A79; 2A7A # LESS-THAN WITH CIRCLE INSIDE
2A7A; 2A79 # GREATER-THAN WITH CIRCLE INSIDE
2A7D; 2A7E # LESS-THAN OR SLANTED EQUAL TO
2A7E; 2A7D # GREATER-THAN OR SLANTED EQUAL TO
2A7F; 2A80 # LESS-THAN OR SLANTED EQUAL TO WITH DOT INSIDE
2A80; 2A7F # GREATER-THAN OR SLANTED EQUAL TO WITH DOT INSIDE
2A81; 2A82 # LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE
2A82; 2A81 # GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE
2A83; 2A84 # LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE RIGHT
2A84; 2A83 # GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE LEFT
2A8B; 2A8C # LESS-THAN ABOVE DOUBLE-LINE EQUAL ABOVE GREATER-THAN
2A8C; 2A8B # GREATER-THAN ABOVE DOUBLE-LINE EQUAL ABOVE LESS-THAN
2A91; 2A92 # LESS-THAN ABOVE GREATER-THAN ABOVE DOUBLE-LINE EQUAL
2A92; 2A91 # GREATER-THAN ABOVE LESS-THAN ABOVE DOUBLE-LINE EQUAL
2A93; 2A94 # LESS-THAN ABOVE SLANTED EQUAL ABOVE GREATER-THAN ABOVE SLANTED EQUAL
2A94; 2A93 # GREATER-THAN ABOVE SLANTED EQUAL ABOVE LESS-THAN ABOVE SLANTED EQUAL
2A95; 2A96 # SLANTED EQUAL TO OR LESS-THAN
2A96; 2A95 # SLANTED EQUAL TO OR GREATER-THAN
2A97; 2A98 # SLANTED EQUAL TO OR LESS-THAN WITH DOT INSIDE
2A98; 2A97 # SLANTED EQUAL TO OR GREATER-THAN WITH DOT INSIDE
2A99; 2A9A # DOUBLE-LINE EQUAL TO OR LESS-THAN
2A9A; 2A99 # DOUBLE-LINE EQUAL TO OR GREATER-THAN
2A9B; 2A9C # DOUBLE-LINE SLANTED EQUAL TO OR LESS-THAN
2A9C; 2A9B # DOUBLE-LINE SLANTED EQUAL TO OR GREATER-THAN
2AA1; 2AA2 # DOUBLE NESTED LESS-THAN
2AA2; 2AA1 # DOUBLE NESTED GREATER-THAN
2AA6; 2AA7 # LESS-THAN CLOSED BY CURVE
2AA7; 2AA6 # GREATER-THAN CLOSED BY CURVE
2AA8; 2AA9 # LESS-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL
2AA9; 2AA8 # GREATER-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL
2AAA; 2AAB # SMALLER THAN
2AAB; 2AAA # LARGER THAN
2AAC; 2AAD # SMALLER THAN OR EQUAL TO
2AAD; 2AAC # LARGER THAN OR EQUAL TO
2AAF; 2AB0 # PRECEDES ABOVE SINGLE-LINE EQUALS SIGN
2AB0; 2AAF # SUCCEEDS ABOVE SINGLE-LINE EQUALS SIGN
2AB3; 2AB4 # PRECEDES ABOVE EQUALS SIGN
2AB4; 2AB3 # SUCCEEDS ABOVE EQUALS SIGN
2ABB; 2ABC # DOUBLE PRECEDES
2ABC; 2ABB # DOUBLE SUCCEEDS
2ABD; 2ABE # SUBSET WITH DOT
2ABE; 2ABD # SUPERSET WITH DOT
2ABF; 2AC0 # SUBSET WITH PLUS SIGN BELOW
2AC0; 2ABF # SUPERSET WITH PLUS SIGN BELOW
2AC1; 2AC2 # SUBSET WITH MULTIPLICATION SIGN BELOW
2AC2; 2AC1 # SUPERSET WITH MULTIPLICATION SIGN BELOW
2AC3; 2AC4 # SUBSET OF OR EQUAL TO WITH DOT ABOVE
2AC4; 2AC3 # SUPERSET OF OR EQUAL TO WITH DOT ABOVE
2AC5; 2AC6 # SUBSET OF ABOVE EQUALS SIGN
2AC6; 2AC5 # SUPERSET OF ABOVE EQUALS SIGN
2ACD; 2ACE # SQUARE LEFT OPEN BOX OPERATOR
2ACE; 2ACD # SQUARE RIGHT OPEN BOX OPERATOR
2ACF; 2AD0 # CLOSED SUBSET
2AD0; 2ACF # CLOSED SUPERSET
2AD1; 2AD2 # CLOSED SUBSET OR EQUAL TO
2AD2; 2AD1 # CLOSED SUPERSET OR EQUAL TO
```

```
2AD3; 2AD4 # SUBSET ABOVE SUPERSET
2AD4; 2AD3 # SUPERSET ABOVE SUBSET
2AD5; 2AD6 # SUBSET ABOVE SUBSET
2AD6; 2AD5 # SUPERSET ABOVE SUPERSET
2ADE; 22A6 # SHORT LEFT TACK
2AE3; 22A9 # DOUBLE VERTICAL BAR LEFT TURNSTILE
2AE4; 22A8 # VERTICAL BAR DOUBLE LEFT TURNSTILE
2AE5; 22AB # DOUBLE VERTICAL BAR DOUBLE LEFT TURNSTILE
2AEC; 2AED # DOUBLE STROKE NOT SIGN
2AED; 2AEC # REVERSED DOUBLE STROKE NOT SIGN
2AF7; 2AF8 # TRIPLE NESTED LESS-THAN
2AF8; 2AF7 # TRIPLE NESTED GREATER-THAN
2AF9; 2AFA # DOUBLE-LINE SLANTED LESS-THAN OR EQUAL TO
2AFA; 2AF9 # DOUBLE-LINE SLANTED GREATER-THAN OR EQUAL TO
2E02; 2E03 # LEFT SUBSTITUTION BRACKET
2E03; 2E02 # RIGHT SUBSTITUTION BRACKET
2E04; 2E05 # LEFT DOTTED SUBSTITUTION BRACKET
2E05; 2E04 # RIGHT DOTTED SUBSTITUTION BRACKET
2E09; 2E0A # LEFT TRANSPOSITION BRACKET
2E0A; 2E09 # RIGHT TRANSPOSITION BRACKET
2E0C; 2E0D # LEFT RAISED OMISSION BRACKET
2E0D; 2E0C # RIGHT RAISED OMISSION BRACKET
2E1C; 2E1D # LEFT LOW PARAPHRASE BRACKET
2E1D; 2E1C # RIGHT LOW PARAPHRASE BRACKET
2E20; 2E21 # LEFT VERTICAL BAR WITH QUILL
2E21; 2E20 # RIGHT VERTICAL BAR WITH QUILL
2E22; 2E23 # TOP LEFT HALF BRACKET
2E23; 2E22 # TOP RIGHT HALF BRACKET
2E24; 2E25 # BOTTOM LEFT HALF BRACKET
2E25; 2E24 # BOTTOM RIGHT HALF BRACKET
2E26; 2E27 # LEFT SIDEWAYS U BRACKET
2E27; 2E26 # RIGHT SIDEWAYS U BRACKET
2E28; 2E29 # LEFT DOUBLE PARENTHESIS
2E29; 2E28 # RIGHT DOUBLE PARENTHESIS
3008; 3009 # LEFT ANGLE BRACKET
3009; 3008 # RIGHT ANGLE BRACKET
300A; 300B # LEFT DOUBLE ANGLE BRACKET
300B; 300A # RIGHT DOUBLE ANGLE BRACKET
300C; 300D # [BEST FIT] LEFT CORNER BRACKET
300D; 300C # [BEST FIT] RIGHT CORNER BRACKET
300E; 300F # [BEST FIT] LEFT WHITE CORNER BRACKET
300F; 300E # [BEST FIT] RIGHT WHITE CORNER BRACKET
3010; 3011 # LEFT BLACK LENTICULAR BRACKET
3011; 3010 # RIGHT BLACK LENTICULAR BRACKET
3014; 3015 # LEFT TORTOISE SHELL BRACKET
3015; 3014 # RIGHT TORTOISE SHELL BRACKET
3016; 3017 # LEFT WHITE LENTICULAR BRACKET
3017; 3016 # RIGHT WHITE LENTICULAR BRACKET
3018; 3019 # LEFT WHITE TORTOISE SHELL BRACKET
3019; 3018 # RIGHT WHITE TORTOISE SHELL BRACKET
301A; 301B # LEFT WHITE SQUARE BRACKET
301B; 301A # RIGHT WHITE SQUARE BRACKET
FE59; FE5A # SMALL LEFT PARENTHESIS
FE5A; FE59 # SMALL RIGHT PARENTHESIS
FE5B; FE5C # SMALL LEFT CURLY BRACKET
FE5C; FE5B # SMALL RIGHT CURLY BRACKET
FE5D; FE5E # SMALL LEFT TORTOISE SHELL BRACKET
FE5E; FE5D # SMALL RIGHT TORTOISE SHELL BRACKET
FE64; FE65 # SMALL LESS-THAN SIGN
FE65; FE64 # SMALL GREATER-THAN SIGN
FF08; FF09 # FULLWIDTH LEFT PARENTHESIS
FF09; FF08 # FULLWIDTH RIGHT PARENTHESIS
```

```
FF1C; FF1E # FULLWIDTH LESS-THAN SIGN
FF1E; FF1C # FULLWIDTH GREATER-THAN SIGN
FF3B; FF3D # FULLWIDTH LEFT SQUARE BRACKET
FF3D; FF3B # FULLWIDTH RIGHT SQUARE BRACKET
FF5B; FF5D # FULLWIDTH LEFT CURLY BRACKET
FF5D; FF5B # FULLWIDTH RIGHT CURLY BRACKET
FF5F; FF60 # FULLWIDTH LEFT WHITE PARENTHESIS
FF60; FF5F # FULLWIDTH RIGHT WHITE PARENTHESIS
FF62; FF63 # [BEST FIT] HALFWIDTH LEFT CORNER BRACKET
FF63; FF62 # [BEST FIT] HALFWIDTH RIGHT CORNER BRACKET
```

Annex D
(informative)

The CFF2 CharString Format

D.1 Overview

The CFF2 CharString format provides a method for compact encoding of glyph procedures in an outline font program. CFF2 CharStrings are intended for use only with a 'CFF2' font table in an OFF font file.

The CFF2 CharString format is closely descended from the Type 2 CharString format [4]. The motivation for developing the CFF2 CharString format was two-fold:

- Reduce the size of CFF fonts by removing all the data which is duplicated elsewhere in the OFF font, or is not used in the context of OFF font files.
- Add support for OFF font variations. (See subclause 7.1 for a general overview of OFF font variations and a complete list of the tables required to support a variable font.)

Accordingly, the CFF2 CharString format has added some new operators, and has removed many more operators. However, the encoding of operators and operands is the same between Type 2 and CFF2 CharStrings; interpreters will need relatively little change to process both formats. See subclause E.5.3 (Changes from Type 2 CharStrings) for a complete list of the differences between Type 2 and CFF2 CharString formats.

This document only describes how CFF2 CharStrings are encoded, and does not attempt to explain the reasons for choosing various options. CFF2 CharStrings are based on Type 1 font concepts, and this Annex assumes familiarity with the Type 1 font format specification. For more information, please see “Adobe Type 1 Font Format”. Also, familiarity with the 'CFF2' table format is assumed; please see subclause 5.4.2 for details.

D.2 CFF2 CharStrings

The following sections describe the general concepts of encoding a CFF2 CharString.

D.2.1 Hints

The CFF2 CharString format supports six hint operators: **hstem**, **vstem**, **hstemhm**, **vstemhm**, **hintmask**, and **cntrmask**. The hint information must be declared at the beginning of a CharString (see subclause E.3.1 "CFF2 CharString organization") using the **hstem**, **vstem**, **hstemhm**, and **vstemhm** operators, each of which may each take arguments for multiple stem hintsThis subclause.

CFF2 hint operators aid the rasterizer in recognizing and controlling stems and counter areas within a glyph. A stem generally consists of two positions (edges) and the associated width. Edge stem hints help to control character features where there is only a single edge (see subclause E.4.3 "Hint operators".). The CFF2 CharString format includes edge hints, which are equivalent to the Type 1 concept of ghost hints (see the section on ghost hints, in the *Adobe Type 1 Font Format*). They are used to locate an edge rather than a stem that has two edges. A stem width value of -20 is reserved for a top or right edge, and a value of -21 for a bottom or left edge. The operation of hints with other negative width values is undefined.

hintmask

The **hintmask** operator has the same function as that described in “Changing Hints within a Character”, section 8.1 of the *Adobe Type 1 Font Format*. It provides a means for activating or deactivating stem hints so that only a set of non-overlapping hints are active at one time. The **hintmask** operator is followed by one or more data bytes that specify the stem hints which are to be active for the subsequent path construction. The

number of data bytes must be exactly the number needed to represent the number of stems in the original stem list (those stems specified by the **hstem**, **vstem**, **hstemhm**, and **vstemhm** commands), using one bit in the data bytes for each stem in the original stem list. Bits with a value of one indicate stems that are active, and a value of zero indicates stems that are inactive.

cntrmask

The **cntrmask** (countermask) hint causes arbitrary but nonoverlapping collections of counter spaces in a character to be controlled in a manner similar to how stem widths are controlled by the stem hint commands. (For more information, see Adobe Technical Note #5015, "The Type 1 Font Format Supplement" [30].) The cntrmask operator is followed by one or more data bytes that specify the index number of the stem hints on both sides of a counter space. The number of data bytes must be exactly the number needed to represent the number of stems in the original stem list (those stems specified by the hstem, vstem, hstemhm, or vstemhm commands), using one bit in the data bytes for each stem in the original stem list.

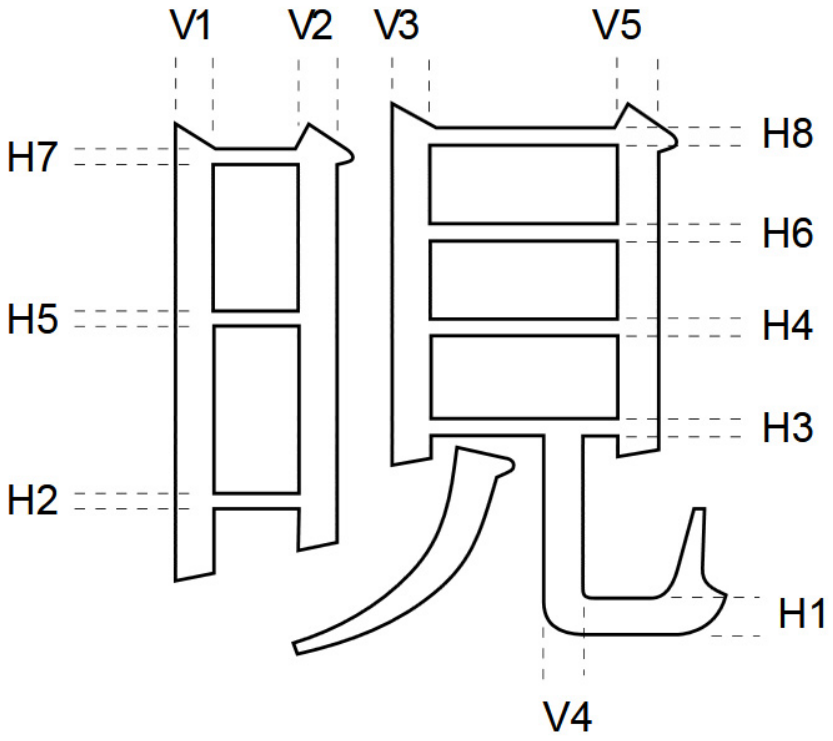


Figure E1 – Counter control example

For the example shown in Figure E1, the stem list for the glyph would be:

- **H1 H2 H3 H4 H5 H6 H7 H8 V1 V2 V3 V4 V5**

and the following cntrmask commands would be used to control the counter spaces between those stems:

- **cntrmask** 0xB5 0xE8 (H1 H3 H4 H6 H8 V1 V2 V3 V5)
- **cntrmask** 0x4A 0x00(H2 H5 H7)

The bits set in the data bytes indicate that the corresponding stem hints delimit the desired set of counters. Hints specified in the first command have a higher priority than those in the second command. Notice that the V4 stem does not delimit an appropriate counter space, and hence is not referenced in this example.

Note that hints are just that, hints, or recommendations. They are additional guidelines to an intelligent rasterizer.

If the font's **LanguageGroup** is not equal to 1 (a LanguageGroup value of 1 indicates complex Asian language glyphs), the **cntrmask** operator, with three stems, can be used in place of the **hstem3** and **vstem3** hints in the Type 1 format, as long as the related conditions specified in the Type 1 specification are met. For more information on Counter Control hints, see Adobe Technical Note #5015, "Type 1 Font Format Supplement".

D.2.2 The Flex mechanism

The flex mechanism is provided to improve the rendering of shallow curves, representing them as line segments at small sizes rather than as small humps or dents in the character shape. It is essentially a path construction mechanism: the arguments describe the construction of two curves, with an additional argument that is used as a hint for when the curves should be rendered as a straight line at smaller sizes and resolutions.

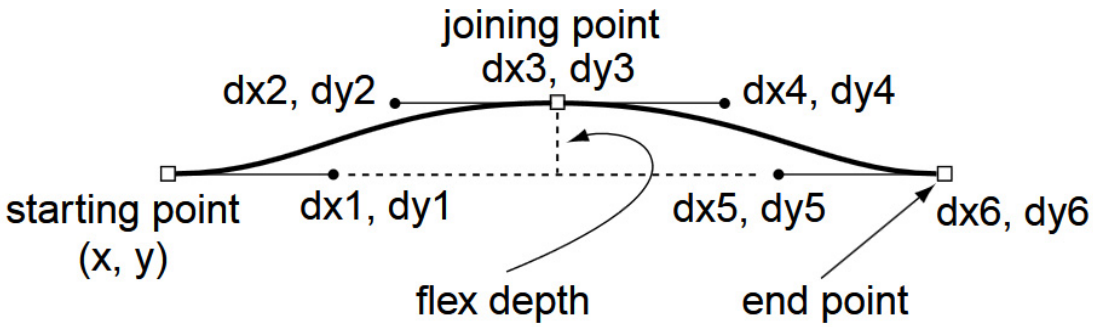


Figure E2 – Flex hint example

The CFF2 flex mechanism is general; there are no restrictions on what type or orientation of curve may be expressed with a flex operator. The **flex** operator is used for the general case; special cases can use the **flex1**, **hflex**, or **hflex1** operators for a more efficient encoding. Figure E2 shows an example of the flex mechanism used for a horizontal curve, and Figure E3 shows an example of flex curves at non-standard angles.

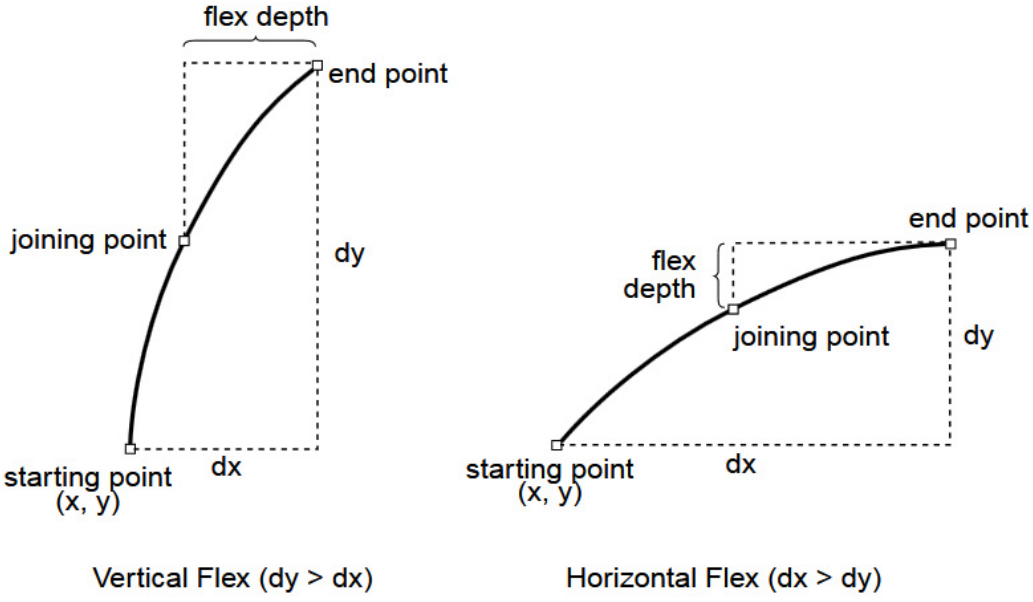


Figure E3 – Flex depth calculations

The flex operators can be used for any curved character feature, in any orientation or depth that meets the following requirements:

- The character feature must be capable of being represented as exactly two curves, drawn by two rrcurveto operators.
- The curves must meet at a common point called the joining point.
- The length of the combined curve must exceed its depth.

D.2.3 Subroutines

A CFF2 font program can use subroutines to reduce the storage requirements by combining the program statements that describe common elements of the characters in the font.

A subroutine is typically a sequence of CharString bytes representing a sub-program that occurs in more than one place in a font's CharString data. Local subroutines may be called from within the same Font DICT and global subroutines may be shared across Font DICTs.

Subroutines may contain sections of CharStrings, and are encoded the same as CFF2 CharStrings. They are called with the **callsubr** (for a local subroutine) or **callgsubr** (for a global subroutine) operator, using a biased index into the local or global Subrs array as the argument.

CharString subroutines may call other subroutines, to the depth allowed by the implementation limits (see subclause E.5.2 "CFF2 CharString implementation limits" for details). A subroutine returns when the interpreter reaches the end of its byte-string.

D.3 CharString encoding

A CFF2 CharString program is a sequence of unsigned 8-bit bytes that encode numbers and operators. The byte value specifies a operator, a number, or subsequent bytes that are to be interpreted in a specific manner.

The bytes are decoded into numbers and operators. The CFF2 CharString interpreter is required to count the number of arguments on the argument stack. It can thus detect additional sets of arguments for a single operator. The stack depth implementation limit is specified in subclause E.5.2 "CFF2 CharString implementation limits".

A number, decoded from a CharString, is pushed onto the CFF2 argument stack. An operator expects its arguments in order from this argument stack with all arguments generally taken from the bottom of the stack (first argument bottom-most); however, some operators, particularly the subroutine operators, normally work from the top of the stack. If an operator returns results, they are pushed onto the CFF2 argument stack (last result topmost).

In the following discussion, all numeric constants are decimal numbers, except where indicated.

D.3.1 CFF2 CharString organization

The sequence and form of a CFF2 CharString program may be represented as:

{hs* vs* cm* hm* mt subpath}? {mt subpath}*

Where:

hs = **hstem** or **hstemhm** command

vs = **vstem** or **vstemhm** command

cm = **cntrmask** operator

hm = **hintmask** operator

mt = moveto (i.e. any of the moveto) operators

subpath = refers to the construction of a subpath (one complete closed contour), which may include **hintmask** operators where appropriate.

and the following symbols indicate specific usage:

* * zero or more occurrences are allowed
* ? zero or one occurrences are allowed
* + one or more occurrences are allowed
* { } indicates grouping

Stated in words, the constraints on the sequence of operators in a CharString are as follows:

CFF2 CharStrings must be structured with operators, or classes of operators, sequenced in the following specific order:

1. **Hints**: zero or more of each of the following hint operators, in exactly the following order: **hstem**, **hstemhm**, **vstem**, **vstemhm**, **cntrmask**, **hintmask**. Each entry is optional, and each may be expressed by one or more occurrences of the operator. The hint operators cntrmask and/or hintmask must not occur if the CharString has no stem hints.

2. **Path Construction**: Zero or more path construction operators are used to draw the path of the character; the second and all subsequent subpaths must also begin with one of the moveto operators. The hintmask operator may be used as needed.

Any operand for the hint and path construction operators may be replaced by the **blend** operator and its operands. If a **vsindex** operator is used, it must be used only once, and prior to any **blend** operator. Blend operators may not be nested. Hint mask bytes and subroutine indices may not be blended

NOTE CharStrings may contain subr and gsubr calls as desired at any point between complete tokens. This means that a subr (gsubr) call must not occur between the bytes of a multibyte commandsnumber or operator, nor between the bytes of a multibyte command (for example, **hintmask** *and* ***cntrmask***).

D.3.2 CharString number encoding

A CharString byte containing the values from 32 through 254 inclusive indicates an integer. These values are decoded in three ranges (also see table below):

- A CharString byte containing a value, v, between 32 and 246 inclusive, specifies the integer v - 139. Thus, the integer values from −107 through 107 inclusive may be encoded in a single byte.
- A CharString byte containing a value, v, between 247 and 250 inclusive, indicates an integer involving the next byte, w, according to the formula:

  ```
  (v - 247) * 256 + w + 108
  ```

  The integer values between 108 and 1131 inclusive can be encoded in 2 bytes in this manner.
- A CharString byte containing a value, v, between 251 and 254 inclusive, indicates an integer involving the next byte, w, according to the formula:

  ```
  - [(v - 251) * 256] - w - 108
  ```

  The integer values between −1131 and −108 inclusive can be encoded in 2 bytes in this manner.

If the CharString byte contains the value 255, the next four bytes contain a Fixed value(a signed fixed-point number with 16 fractional bits).

NOTE The CFF2 interpretation of a number encoded in five-bytes (those with an initial byte value of 255) differs from how it is interpreted in the Type 1 format.

In addition to the numeric representations listed above, numbers between -32768 and +32767 can be represented using the operator (28) followed by an int16. This allows a more compact representation of large numbers which occur occasionally in fonts, but perhaps more importantly, this will allow more compact encoding of numbers which may be used as arguments to **callsubr** and **callgsubr**.

CFF2 CharString Encoding Values

CharString Byte Value	Interpretation	Number Range Represented	Bytes Required
0 to 11	operators	operators 0 to 11	1
12	escape: next byte interpreted as additional operators	additional 0 to 255 range for operator codes	2
13 to 18	operators	operators 13 to 18	1
19, 20	operators (**hintmask** and **cntrmask**)	operators 19, 20	2 or more
21 to 27	operators	operators 21 to 27	1
28	next 2 bytes are an int16	-32768 to +32767	3
29 to 31	Operators	operators 29 to 31	1
32 to 246	result = v-139	-107 to +107	1
247 to 250	with next byte, w, result = (v - 247) * 256 + w + 108	+108 to +1131	2
251 to 254	with next byte, w, result = -[(v - 251) * 256] - w -108	-108 to -1131	2
255	next 4 bytes are a Fixed value	16-bit signed integer with 16 bits of fraction	5

D.3.3 CharString operator encoding

CharString operators are encoded in one or two bytes. Not all possible operator encoding values are defined (see subclause E.5.1 "CFF2 CharString command codes" for a list of operator encoding values). When an unrecognized operator is encountered, it is ignored and the stack is cleared.

If an operator byte contains the value 12, then the value in the next byte specifies an operator. This escape mechanism allows many extra operators to be encoded.

D.4 CharString operators

CFF2 CharString operators are divided into four groups, classified by function:

1. path construction;
2. hints;
3. subroutine;
4. variation data support.

The following definitions use a format similar to that used in the *PostScript Language Reference Manual*. Parentheses following the operator name either include the operator value that represents this operator in a CharString byte, or the two values (beginning with 12) that represent a two-byte operator.

Many operators take their arguments from the bottom-most entries in the CFF2 argument stack; this behavior is indicated by the stack bottom symbol '|-' appearing to the left of the first argument. Operators that clear the argument stack are indicated by the stack bottom symbol '|-' in the result position of the operator definition.

Because of this stack-clearing behavior, in general, arguments are not accumulated on the CFF2 argument stack for later removal by a sequence of operators, arguments generally may be supplied only for the next operator. Notable exceptions occur with subroutine calls and with the **blend** operator. All stack operations must observe the stack limit (see subclause E.5.2 "CFF2 CharString implementation limits" for details).

D.4.1 Path construction operators

In a CFF2 CharString, a path is constructed by sequential application of one or more path construction operators. The current point is initially the (0, 0) point of the character coordinate system. The operators listed in this subclause cause the current point to change, either by a *moveto* operation, or by appending one or more curve or line segments to the current point. Upon completion of the operation, the current point is updated to the position to which the move was made, or to the last point on the segment or segments.

Many of the operators can take multiple sets of arguments, which indicate a series of path construction operations. The number of operations is limited only by the limit on the stack size (see subclause E.5.2 "CFF2 CharString implementation limits").

All Bézier curve path segments are drawn using six arguments, dxa, dya, dxb, dyb, dxc, dyc; where dxa and dya are relative to the current point, and all subsequent arguments are relative to the previous point. A number of the curve operators take advantage of the situation where some tangent points are horizontal or vertical (and hence the value is zero), thus reducing the number of arguments needed.

The *flex* operators are considered path construction commands because they specify the drawing of two curves. There is also an additional argument that serves as a hint as to when to render the curves as a straight line at small sizes and low resolutions.

The following are three types of *moveto* operators. For the initial *moveto* operator in a CharString, the arguments are relative to the (0, 0) point in the character's coordinate system; subsequent *moveto* operators' arguments are relative to the current point.

Every character path and subpath must begin with one of the *moveto* operators. If the current path is open when a *moveto* operator is encountered, the path is closed by performing a *lineto* operation to the previous *moveto* coordinates before performing the *moveto* operation. The inserted *lineto* operation does not change the current point.

rmoveto |- dx1 dy1 **rmoveto** (21) |-

Moves the current point to a position at the relative coordinates *(dx1, dy1)*.

hmoveto |- dx1 **hmoveto** (22) |-

Moves the current point *dx1* units in the horizontal direction.

vmoveto |- dy1 **vmoveto** (4) |-

Moves the current point *dy1* units in the vertical direction.

rlineto |- {dxa dya}+ **rlineto** (5) |-

Appends a line from the current point to a position at the relative coordinates *dxa*, *dya*. Additional **rlineto** operations are performed for all subsequent argument pairs. The number of lines is determined from the number of arguments on the stack.

hlineto |- dx1 {dya dxb}* **hlineto** (6) |-

|- {dxa dyb}+ **hlineto** (6) |-

Appends a horizontal line of length *dx1* to the current point. With an odd number of arguments, subsequent argument pairs are interpreted as alternating values of *dy* and *dx*, for which additional *lineto* operators draw alternating vertical and horizontal lines. With an even number of arguments, the arguments are

interpreted as alternating horizontal and vertical lines. The number of lines is determined from the number of arguments on the stack.

vlineto |- dy1 {dxa dyb}* **vlineto** (7) |-

|- {dya dxb}+ **vlineto** (7) |-

Appends a vertical line of length *dy1* to the current point. With an odd number of arguments, subsequent argument pairs are interpreted as alternating values of *dx* and *dy*, for which additional *lineto* operators draw alternating horizontal and vertical lines. With an even number of arguments, the arguments are interpreted as alternating vertical and horizontal lines. The number of lines is determined from the number of arguments on the stack.

rrcurveto |- {dxa dya dxb dyb dxc dyc}+ **rrcurveto** (8) |-

Appends a Bézier curve, defined by *dxa...dyc*, to the current point. For each subsequent set of six arguments, an additional curve is appended to the current point. The number of curve segments is determined from the number of arguments on the number stack and is limited only by the size of the number stack.

hhcurveto |- dy1? {dxa dxb dyb dxc}+ **hhcurveto** (27) |-

Appends one or more Bézier curves, as described by the *dxa...dxc* set of arguments, to the current point. For each curve, if there are 4 arguments, the curve starts and ends horizontal. The first curve need not start horizontal (the odd argument case). Note the argument order for the odd argument case.

hvcurveto |- dx1 dx2 dy2 dy3 {dya dxb dyb dxc dxd dxe dye dyf}* dxf? **hvcurveto** (31) |-

|- {dxa dxb dyb dyc dyd dxe dye dxf}+ dyf? **hvcurveto** (31) |-

Appends one or more Bézier curves to the current point. The tangent for the first Bézier must be horizontal, and the second must be vertical (except as noted below).

If there is a multiple of four arguments, the curve starts horizontal and ends vertical. Note that the curves alternate between start horizontal, end vertical, and start vertical, and end horizontal. The last curve (the odd argument case) need not end horizontal/vertical.

rcurveline |- {dxa dya dxb dyb dxc dyc}+ dxd dyd **rcurveline** (24) |-

Is equivalent to one **rrcurveto** for each set of six arguments *dxa...dyc*, followed by exactly one **rlineto** using the *dxd, dyd* arguments. The number of curves is determined from the count on the argument stack.

rlinecurve |- {dxa dya}+ dxb dyb dxc dyc dxd dyd **rlinecurve** (25) |-

Is equivalent to one **rlineto** for each pair of arguments beyond the six arguments *dxb...dyd* needed for the one **rrcurveto** command. The number of lines is determined from the count of items on the argument stack.

vhcurveto |- dy1 dx2 dy2 dx3 {dxa dxb dyb dyc dyd dxe dye dxf}* dyf? **vhcurveto** (30) |-

|- {dya dxb dyb dxc dxd dxe dye dyf}+ dxf? **vhcurveto** (30) |-

Appends one or more Bézier curves to the current point, where the first tangent is vertical and the second tangent is horizontal. This command is the complement of **hvcurveto**; see the description of **hvcurveto** for more information.

vvcurveto |- dx1? {dya dxb dyb dyc}+ **vvcurveto** (26) |-

Appends one or more curves to the current point. If the argument count is a multiple of four, the curve starts and ends vertical. If the argument count is odd, the first curve does not begin with a vertical tangent.

flex |- dx1 dy1 dx2 dy2 dx3 dy3 dx4 dy4 dx5 dy5 dx6 dy6 fd **flex** (12 35) |-

Causes two Bézier curves, as described by the arguments (as shown in Figure E2), to be rendered as a straight line when the *flex depth* is less than fd /100 device pixels, and as curved lines when the flex depth is greater than or equal to fd/100 device pixels. The flex depth for a horizontal curve, as shown in Figure E2, is the distance from the join point to the line connecting the start and end points on the curve. If the curve is not exactly horizontal or vertical, it must be determined whether the curve is more horizontal or vertical by the method described in the **flex1** description, below, and as illustrated in Figure E3.

NOTE In cases where some of the points have the same x or y coordinate as other points in the curves, arguments may be omitted by using one of the following forms of the **flex** operator: **hflex**, **hflex1**, or **flex1**.

hflex |- dx1 dx2 dy2 dx3 dx4 dx5 dx6 **hflex** (12 34) |-

Causes the two curves described by the arguments *dx1...dx6* to be rendered as a straight line when the flex depth is less than 0.5 (that is, fd is 50) device pixels, and as curved lines when the flex depth is greater than or equal to 0.5 device pixels.

hflex is used when the following are all true:

a) The starting and ending points, first and last control points have the same y value.
b) The joining point and the neighbor control points have the same y value.
c) The flex depth is 50.

hflex1 |- dx1 dy1 dx2 dy2 dx3 dx4 dx5 dy5 dx6 **hflex1** (12 36) |-

Causes the two curves described by the arguments to be rendered as a straight line when the flex depth is less than 0.5 device pixels, and as curved lines when the flex depth is greater than or equal to 0.5 device pixels.

hflex1 is used if the conditions for hflex are not met but all of the following are true:

a) The starting and ending points have the same y value.
b) The joining point and the neighbor control points have the same y value.
c) The flex depth is 50.

flex1 |- dx1 dy1 dx2 dy2 dx3 dy3 dx4 dy4 dx5 dy5 d6 **flex1** (12 37) |-

Causes the two curves described by the arguments to be rendered as a straight line when the flex depth is less than 0.5 device pixels, and as curved lines when the flex depth is greater than or equal to 0.5 device pixels.

The d6 argument will be either a dx or dy value, depending on the curve (see Figure E3). To determine the correct value, compute the distance from the starting point (x, y), the first point of the first curve, to the last flex control point (dx5, dy5) by summing all the arguments except d6; call this (dx, dy). If abs(dx) > abs(dy), then the last point's x-value is given by d6, and its y-value is equal to y. Otherwise, the last point's x-value is equal to x and its y-value is given by d6.

flex1 is used if the conditions for **hflex** and **hflex1** are not met but all of the

following are true:

a) The starting and ending points have the same x or y value.
b) The flex depth is 50.

D.4.2 Finishing a CharString outline

CFF2 CharStrings differ from Type 2 CharStrings in that there is no operator for finishing a CharString outline definition.The end of the CharString byte string implies the end of the last subpath, and serves the same purpose as the Type 2 **endchar** operator. If the last operator in a CharString is a **call(g)subr**, then the CharString ends when that subroutine ends.

The smallest legal CharString is simply an empty byte string.

D.4.3 Hint operators

All hints must be declared at the beginning of the CharString program, after the width (see subclause E.3.1 "CFF2 CharString organization" for details).

hstem |- y dy {dya dyb}* **hstem** (1) |-

Specifies one or more horizontal stem hints (see the following section for more information about horizontal stem hints). This allows multiple pairs of numbers, limited by the stack depth, to be used as arguments to a single **hstem** operator.

It is required that the stems are encoded in ascending order (defined by increasing bottom edge). The encoded values are all relative; in the first pair, y is relative to 0, and dy specifies the distance from y. The first value of each subsequent pair is relative to the last edge defined by the previous pair.

A width of -20 specifies the top edge of an edge hint, and -21 specifies the bottom edge of an edge hint. All other negative widths have undefined meaning.

Figure E4 shows an example of the encoding of a character stem that uses top and bottom edge hints. The edge stem hint serves to control the position of the edge of the stem in situations where controlling the stem width is not the primary purpose.

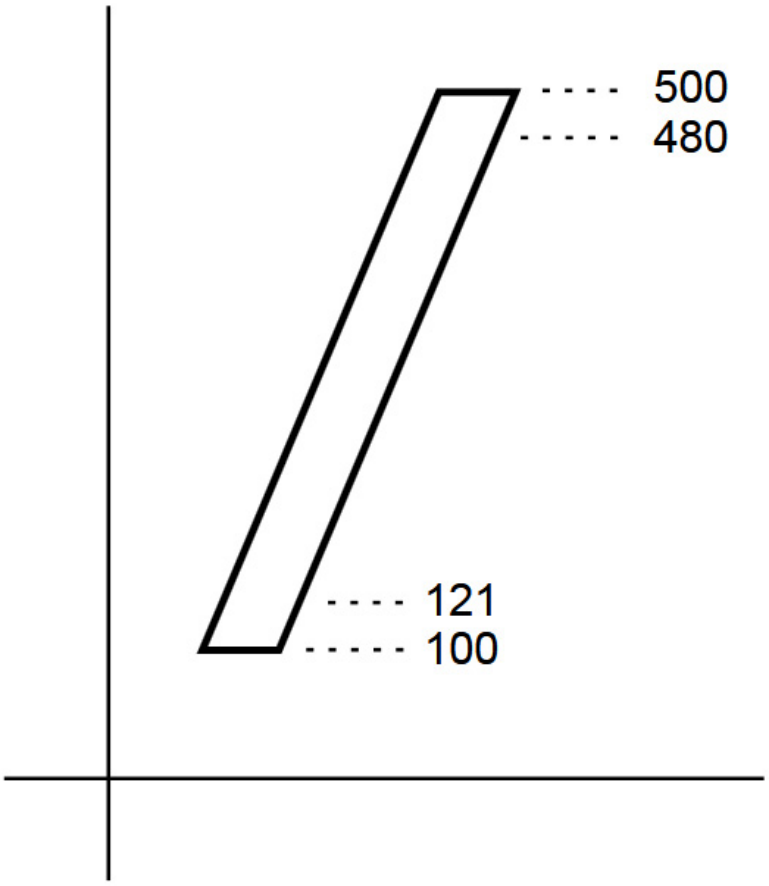


Figure E4 – Encoding of edge hints

The encoding for the edge stem hints shown in Figure E4 would be:

```
121 –21 400 –20 hstem
```

vstem |- x dx {dxa dxb}* **vstem** (3) |-

Specifies one or more vertical stem hints between the *x* coordinates *x* and *x*+*dx*, where *x* is relative to the origin of the coordinate axes.

It is required that the stems are encoded in ascending order (defined by increasing left edge). The encoded values are all relative; in the first pair, x is relative to 0, and dx specifies the distance from x. The first value of each subsequent pair is relative to the last edge defined by the previous pair.

A width of -20 specifies the right edge of an edge hint, and -21 specifies the left edge of an edge hint. All other negative widths have undefined meaning.

It is important to note that **hstem** hints must not overlap other **hstem** hints, and similarly, **vstem** hints must not overlap other **vstem** hints. If overlapping hints are needed, then the overlapping hints must be defined by using the **hstemhm** or **vstemhm** operator rather than the **hstem** or **vstem** operator. In addition, the hints must be selectively applied by using the **hintmask** operator to select the active hint so that no overlapping

horizontal or vertical hints are active at the same time. See below for details on the **hintmask** operator and the example in Figure E6.

Figure E5 below shows an example of overlapping hints on a sample feature of a character outline.

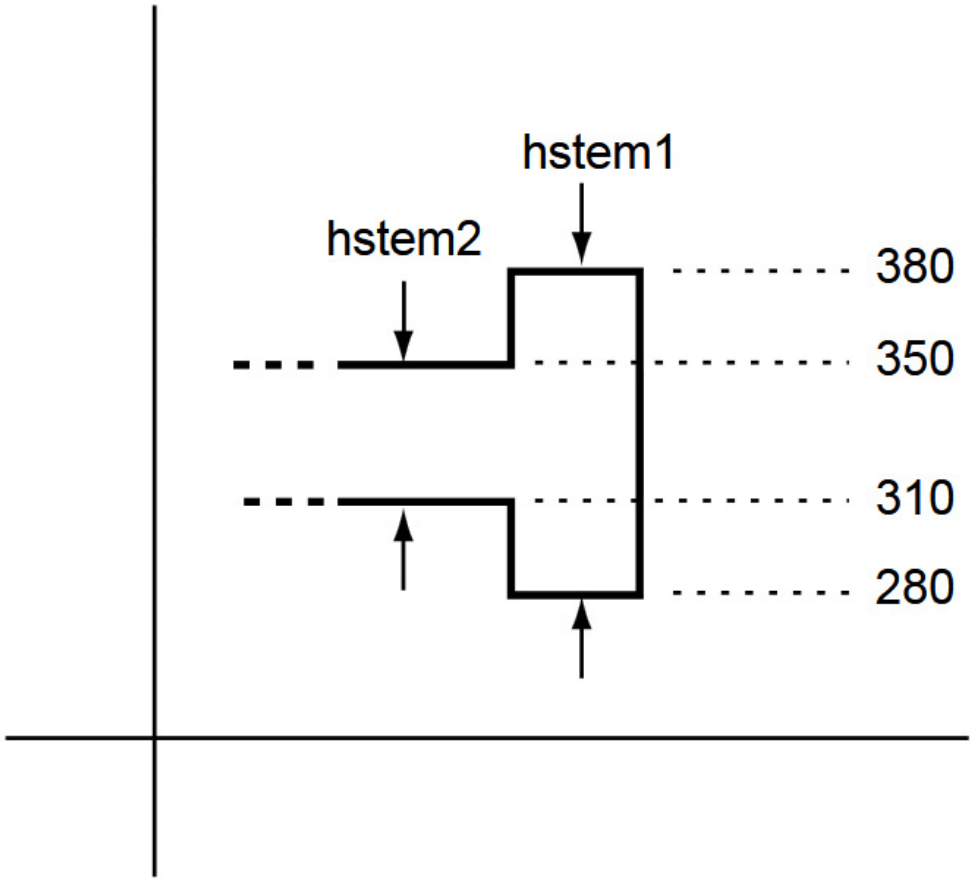


Figure E5 – Encoding of overlapping hints

The encoding for the example shown in Figure E5 would be:

```
280 100 –70 40 hstemhm
```

Again, note that the **hintmask** operator, not shown here, must also be used in conjunction with these hints.

hstemhm |- y dy {dya dyb}* **hstemhm** (18) |-

Has the same meaning as **hstem** (1), except that it must be used in place of **hstem** if the CharString contains one or more **hintmask** operators.

vstemhm |- x dx {dxa dxb}* **vstemhm** (23) |-

Has the same meaning as **vstem** (3), except that it must be used in place of vstem if the CharString contains one or more **hintmask** operators.

hintmask |- **hintmask** (19) mask |-

Specifies which hints are active and which are not active. If any hints overlap, **hintmask** must be used to establish a non-overlapping subset of hints. **hintmask** may occur any number of times in a CharString. Path operators occurring after a **hintmask** are influenced by the new hint set, but the current point is not moved. If stem hint zones overlap and are not properly managed by use of the hintmask operator, the results are undefined.

The *mask* data bytes are defined as follows:

- The number of data bytes is exactly the number needed, one bit per hint, to reference the number of stem hints declared at the beginning of the CharString program.
- Each bit of the mask, starting with the most-significant bit of the first byte, represents the corresponding hint zone in the order in which the hints were declared at the beginning of the CharString.
- For each bit in the mask, a value of '1' specifies that the corresponding hint shall be active. A bit value of '0' specifies that the hint shall be inactive.

Unused bits in the mask, if any, must be zero.

If **hstem** and **vstem** hints are both declared at the beginning of a CharString, and this sequence is followed directly by the **hintmask** or **cntrmask** operators, then the **vstem** hint operator (or, if applicable, the **vstemhm** operator) need not be included. For example, Figure E6 shows part of a character with **hstem** and **vstem** hints.

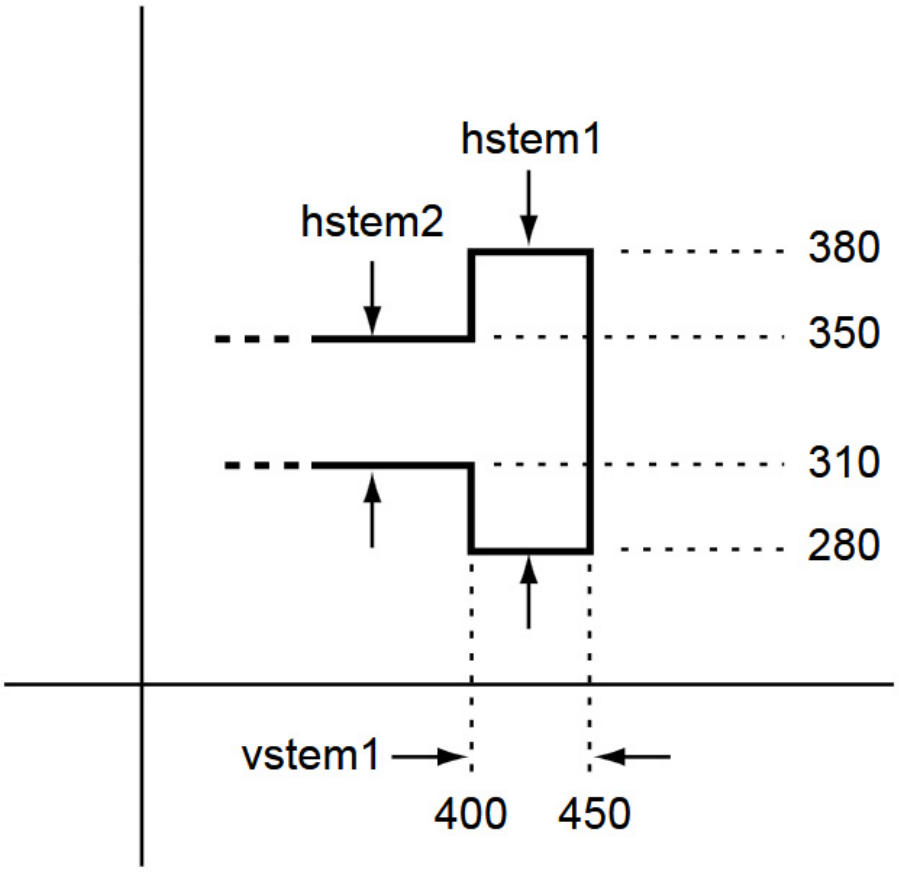


Figure E6 – Hint encoding example

Figure E6 shows two overlapping **hstem** hints: hstem1, from 280 to 380; and hstem2, from 310 to 350. Because these overlap, use of the **hintmask** operator will be required to select one or the other as active. It also shows one **vstem** hint: vstem1, from 400 to 450. Suppose that only these three hints are defined, and that the outline is being constructed in counterclockwise direction, beginning with the horizontal line segment ending at (400, 310). In this case, the first hint group that should be active is hstem2 plus vstem1. For this scenario, the hints would be specified as follows:

```
280 100 −70 40 hstemhm 400 50 hintmask 0x60
```

Note that the **hstemhm** is used to indicate that hint substitution (using hintmask) will be used. The hints are defined in the order hstem1, hstem2, then vstem1. The **hintmask** operand, 0x60 (01100000), indicates which hints are active at the beginning of the path construction: the second and third hints in order — hstem2 and vstem1.

cntrmask |- **cntrmask** (20) mask |-

Specifies the counter spaces to be controlled, and their relative priority. The *mask* bits in the bytes, following the operator, reference the stem hint declarations; the most significant bit of the first byte refers to the first stem hint declared, through to the last hint declaration. The counters to be controlled are those that are delimited by the referenced stem hints. Bits set to 1 in the first **cntrmask** command have top priority; subsequent **cntrmask** commands specify lower priority counters (see Figure E1 and the accompanying example).

D.4.4 Subroutine operators

The numbering of subroutines is encoded more compactly by using the negative half of the number space, which effectively doubles the number of compactly encodable subroutine numbers. The bias applied depends on the number of subrs (gsubrs). If the number of subrs (gsubrs) is less than 1240, the bias is 107. Otherwise if it is less than 33900, it is 1131; otherwise it is 32768. This bias is added to the encoded subr (gsubr) number to find the appropriate entry in the subr (gsubr) array.

callsubr subr# **callsubr** (10)

Calls a CharString subroutine with index subr# (*actually the subr number plus the subroutine bias number, as described in* subclause *E.2.3*) in the Subrs array. Each element of the Subrs array is a CharString encoded like any other CharString. The subroutine call removes only the subr# and operator from the stack, so arguments pushed on the stack are available to any operators in the subroutine. Similarly, values pushed from inside the subroutine are available after the subroutine returns to the caller. Calling an undefined subr (gsubr) has undefined results.

These subroutines are generally used to encode sequences of path operators that are repeated throughout the font program, for example, serif outline sequences. Subroutine calls may be nested to the depth specified in the implementation limits in subclause E.5.2.

callgsubr globalsubr# **callgsubr** (29)

Operates in the same manner as **callsubr** except that it calls a global subroutine.

D.4.5 Variation data operators

In order to support variation data in CFF2 CharStrings, two new operators are added in CFF2 CharStrings: **vsindex** and **blend**.

A variable font holds data representing the equivalent of several distinct design variations, and uses algorithms for interpolation — or *blending* — between these designs to derive a continuous range of design instances. This allows an entire family of fonts to be represented by a single variable font. For example, a variable font may contain data equivalent to Light and Heavy designs from a family, which can then be interpolated to derive instances for any weight in a continuous range between Light and Heavy.

See subclause 7.1 for general background on OFF font variations, details on the tables used to support a variable font, terminology, and a specification of the interpolation algorithm used to blend values to derive specific design instances.

Outline data for a variable font in the CFF2 format are built much like a non-variable CFF2 table would be built, with exactly the same structure and operators as would be used for the default design representation. However, wherever a value occurs in the default design, the single value for the one design is supplemented with a set of delta values, followed by the **blend** operator. (For efficiency, a single blend operator may follow a series of such delta sets, rather than after each individual set.) Unlike other CharString operators, **blend** does not clear the stack when it is processed. The result of the **blend** operator remains on the stack to be processed by the following operator.

Within a variable font, different glyphs can use different sets of regions and associated delta values for the blending operation. When processing a given glyph, the interpreter must determine which set to use. These sets are stored in the CFF2 table in an *ItemVariationStore* structure. The ItemVariationStore contains one or more *ItemVariationData subtables*, each of which contains a list of Variation Regions. The first ItemVariationData subtable (index 0) is used by default, when no other subtable has been specified. When an ItemVariationData subtable other than the default is needed for a set of delta values, the **vsindex** operator is used. When this operator is used in a Private DICT to set a non-default itemVariationData index, this then becomes the default Item Variation Data index for not only the Private DICT, but also for all CharStrings that reference that Private DICT. When the **vsindex** operator is used in a CharString, it supersedes any **vsindex** from the private DICT. All private DICTs and CharStrings in a CFF2 table share the same ItemVariationStore.

Syntax for Font Variations support operators.

vsindex |- ivs **vsindex** (15) |-

Selects the ItemVariationData subtable to be used for blending in this CharString; the ivs argument is the ItemVariationData index. When used, **vsindex** must precede the first **blend** operator, and may occur only once in the CharString. If the **vsindex** operator is not present in the CharString, then the ItemVariationData index is inherited from the Private DICT **vsindex** value. If the **vsindex** operator is not present in a Private DICT, then the default value is 0.

blend num(0)…num(n-1), delta(0,0)…delta(k-1,0), delta(0,1)…delta(k-1,1) … delta(0,n-1)…delta(k-1,n-1) n **blend** (16) val(0)…val(n-1)

For *k* regions, produces *n* interpolated result value(s) from *n*(k + 1)* operands.

The **blend** operator is used in conjunction with other operators to produce interpolated input values for the other operator that are applicable for the currently-selected variation instance. The variation-instance design vector is specified in the interpreter, external to the font program, and the interpreter applies the design vector when processing the **blend** operator to calculate the appropriate interpolated values.

Blending is supported only for design space coordinate values, such as the operands for the hint and path construction operators. The last operand on the stack, *n*, specifies the number of operands that will be left on the stack for the next operator. (For example, if the **blend** operator is used in conjunction with the **hflex** operator, which requires 6 operands, then n would be set to 6.) This operand also informs the handler for the **blend** operator that the operator is preceded by n+1 sets of operands.

The first set of operands, num(0)…num(n-1), contains the n operands for the following CharsString operator that are applicable to the default variation instance.

There are n additional sets of operands, delta(0,*i*)…delta(k-1,*i*), one corresponding to each of the values in the first set of operands. Each set contains the variation adjustment deltas for the given operand, with each delta being associated with a different region of the design-variation space. (See the “Variation Space, Default Instances and Adjustment Deltas” and “Variation Data” sections in subclause 7.1 for more information regarding regions and associated adjustment deltas.) Each of these additional operand sets has *k* operands, where k represents the number of regions for which variation adjustment deltas are defined. This value is determined by the ItemVariationData subtable that is currently active for the CharString (see

vsindex).

Thus, the first set contains n operands, and the following sets contain n*k operands, giving a total of n*(k+1) operands that precede the final n operand.

The **blend** handler will compare the currently-selected variation-instance design vector with the coordinates for each region to compute an interpolation scalar factor for each given region. This is then applied to the adjustment deltas for the region to compute a net adjustment delta to be applied to the default value. For more details on the interpolation algorithm, see subclause 7.1.7 in the OFF Font variations overview.

As an example of the **blend** operator, consider use of **blend** to interpolate inputs to an **rmoveto** operator. The **rmoveto** operator takes two operands, x and y, and so the *n* argument for the **blend** operator will be set to 2.

Suppose, also, that the font has weight and width variation axes, and that the default variation instance of the font is Regular. In addition, suppose that this CharString uses deltas for three regions, the extreme points of which correspond to Light and Bold (extremes for the weight axis), and also Condensed (extreme for the width axis, opposite of Regular which has normal width). Thus, the value of *k* is 3.

Now suppose that the x and y arguments for the **removeto** operator that would be needed for the Regular, Light, Bold and Condensed variation instances are as follows:

```
Regular: 100 200 rmoveto
Light: 100 150 rmoveto
Bold: 100 300 rmoveto
Condensed: 50 100 rmoveto
```

For the **blend** operator, the first set of arguments will be the **rmoveto** arguments required for the default instance:

```
100 200
```

The following sets of **blend** arguments are the set of deltas for the three regions, with one set of deltas for each of the **rmoveto** arguments. As an example of a delta, the x argument for Regular — the default instance — is 100, and the x argument required for Condensed is 50, and so the x-argument delta for the 3rd region (index 2) is -50. Thus, the set of deltas for the first **rmoveto** operand for all three regions are:

```
0 0 -50
```

And the set of deltas for the second **rmoveto** operand for all three regions are:

```
-50 100 -100
```

Combining this together, the CharString sequence for the **blend** and **rmoveto** operator combination will be as follows; parentheses are added to demarcate each of the **blend** operand sets:

```
(100 200) (0 0 -50) (-50 100 -100) 2 blend rmoveto
```

When the **blend** operator is processed for a particular variation instance, the handler will calculate scalar coefficients for each region and then apply each scalar to the deltas for the corresponding region. For instance, if the selected instance is midway between Regular and Light on the weight axis and normal width on the width axis, then the scalar for region 0 will be 0.5, while the scalars for regions 1 and 2 will each be 0.0. The **blend** operator will interpolate the **rmoveto** arguments as follows:

```
x = 100 + (0 * 0.5) + (0 * 0.0) + (-50 * 0.0) = 100 y = 200 + (-50 * 0.5) +
(100 * 0.0) + (-100 * 0.0) = 175
```

These **blend** results are pushed onto the stack. Thus, the **rmoveto** operation that will be performed is:

```
100 175 rmoveto
```

D.5 Supplemental information

D.5.1 CFF2 CharString command codes

One-byte CFF2 Operators			
Dec	**Hex**	**Operator**	**Note**
0	00	<reserved>	
1	01	**hstem**	
2	02	<reserved>	
3	03	**vstem**	
4	04	**vmoveto**	
5	05	**rlineto**	
6	06	**hlineto**	
7	07	**vlineto**	
8	08	**rrcurveto**	
9	09	<reserved>	
10	0a	**callsubr**	
11	0b	<reserved>	
12	0c	escape	First byte of a 2-byte operator.
13	0d	<reserved>	
14	0e	<reserved>	
15	0f	**vsindex**	
16	10	**blend**	
17	11	<reserved>	
18	12	**hstemhm**	
19	13	**hintmask**	
20	14	**cntrmask**	
21	15	**rmoveto**	
22	16	**hmoveto**	
23	17	**vstemhm**	
24	18	**rcurveline**	
25	19	**rlinecurve**	
26	1a	**vvcurveto**	

27	1b	**hhcurveto**	
28	1c	<numbers>	First byte of a 3-byte sequence specifying an unsigned integer value (next two bytes are a uint16).
29	1d	**callgsubr**	
30	1e	**vhcurveto**	
31	1f	**hvcurveto**	
32 to 246	20 to f6	<numbers>	
247 to 254	f7 to fe	<numbers>	First byte of a 2-byte sequence specifying a number.
255	ff	<number>	First byte of a 5-byte sequence specifying a Fixed value.

Two-byte CFF2 Operators		
Dec	**Hex**	**Operator**
12 0 to 12 33	0c 00 to 0c 21	<reserved>
12 34	0c 22	**hflex**
12 35	0c 23	**flex**
12 36	0c 24	**hflex1**
12 37	0c 25	**flex1**
12 38 to 12 255	0c 26 to 0c ff	<reserved>

D.5.2 CFF2 CharString implementation limits

The following are the implementation limits of the CFF2 CharString interpreter:

Description	Limit
Argument stack	513
Number of stem hints (H/V total)	96
Subr nesting, stack limit	10
CharString length	65535
maximum (g)subrs count	65536

D.5.3 Changes from Type 2 CharStrings

- CFF2 CharStrings do not contain a value for width.
- The CharString operator set is extended in CFF2 to include the **blend** and **vsindex** operators. These work as described in subclause E.4.5 (Variation data operators). The CFF2 CharString operator code for **blend** is 16, and for **vsindex** is 15.
- The Type 2 operators **endchar** and **return** are removed.
- The Type 2 logic, storage, and math operators are removed.
- For CFF2 fonts, the fill rule for CharStrings must always be the nonzero winding number rule, rather than the even-odd rule. Overlap subpaths are permitted.
- The stack depth is increased from 48 to 513.

Annex E
(informative)

CFF2 DICT Encoding

One-byte CFF2 DICT Operators

Dec	Hex	Operator	Note
0 to 5	00 to 05	<reserved>	
6	06	**BlueValues**	
7	07	**OtherBlues**	
8	08	**FamilyBlues**	
9	09	**FamilyOtherBlues**	
10	0a	**StdHW**	
11	0b	**StdVW**	
12	0c	escape	First byte of a 2-byte operator.
13 to 16	0d to 10	<reserved>	
17	11	**CharStrings**	
18	12	**Private**	
19	13	**Subrs**	
20 to 21	14 to 15	<reserved>	
22	16	**vsindex**	
23	17	**blend**	
24	18	**vstore**	
25 to 27	19 to 1b	<reserved>	
28	1c	<numbers>	First byte of a 3-byte sequence specifying a signed integer value (following two bytes are an int16).
29	1d	<numbers>	First byte of a 5-byte sequence specifying a signed integer value (following four bytes are an int32).
30	1e	**BCD**	
31	1f	<reserved>	
32 to 246	20 to f6	<numbers>	
247 to 254	f7 to fe	<numbers>	First byte of a 2-byte sequence specifying a number.
255	ff	<reserved>	

NOTE Operator code 25 (0x19) was previously assigned as a **maxstack** operator. This was never required in CFF2 fonts and is no longer supported.

Two-byte CFF2 DICT Operators

Dec	Hex	Operator
12 0 to 12 6	0c 00 to 0c 06	<reserved>
12 7	0c 07	**FontMatrix**
12 8	0c 08	<reserved>
12 9	0c 09	**BlueScale**
12 10	0c 0a	**BlueShift**
12 11	0c 0b	**BlueFuzz**
12 12	0c 0c	**StemSnapH**
12 13	0c 0d	**StemSnapV**
12 14 to 12 16	0c 0e to 0c 10	<reserved>
12 17	0c 11	**LanguageGroup**
12 18	0c 12	**ExpansionFactor**
12 19 to 12 35	0c 13 to 0c 23	<reserved>
12 36	0c 24	**FDArray**
12 37	0c 25	**FDSelect**
12 38 to 12 255	0c 26 to 0c ff	<reserved>

Annex F
(informative)

Registration of Media Type: `application/font-sfnt`

The content of this Annex (presented below in its entirety) represents a text of new media type registration for a 'font-sfnt" subtype of the "application" tree. Since the new top-level media type "font" tree is now available as introduced by the RFC 8081 (https://tools.ietf.org/html/rfc8081), we recommend using the "font/sfnt" subtype as a generic font media type.

The use of "application/font-sfnt" media type is now DEPRECATED and the text of its registration below provided only for historic references.

This appendix registers a new MIME media type, in conformance with http://www.ietf.org/rfc/rfc4288.txt.

Type name:

`application`

Subtype name:

`font-sfnt`

Required parameters:

None

Optional parameters:

```
1)    Name:   Outlines
      Value:  TTF, CFF

2)    Name:   Layout
      Value:  OTF, AAT, SIL
```

Encoding considerations:

binary

Security considerations:

Fonts are collections of different tables containing data structures that represent different types of information, including glyph outlines in various data formats, hinting instructions, metrics and layout information for multiple languages and writing systems, rules for glyph substitution and positioning, etc. Depending on the data format used to represent the glyph data the font may contain either TrueType or PostScript outlines and their respective hint instructions. There are many existing, already standardized font table tags and formats that allow an unspecified number of entries containing predefined data fields for storage of variable length binary data.

Many existing (TrueType, OpenType and OFF, SIL Graphite, WOFF and many other) font formats are based on the table-based SFNT (scalable font) format which is extremely flexible, highly extensible and offers an opportunity to introduce additional table structures when needed, in a way that would not affect existing font rendering engines and text layout implementations. However, this very extensibility may present specific security concerns – the flexibility and ease of adding new data structures makes it easy for any arbitrary data

to be hidden inside a font file. There is a significant risk that the flexibility of font data structures may be exploited to hide malicious binary content disguised as a font data component.

Fonts may contain 'hints' – programmatic instructions that are executed by the font engine for the alignment of graphical elements of glyph outlines with the target display pixel grid. Depending on the font technology utilized in the creation of a font these hints may represent active code interpreted and executed by the font rasterizer. Even though hints operate within the confines of the glyph outline conversion system and have no access outside the font rendering engine, hint instructions can be, however, quite complex, and a maliciously designed complex font could cause undue resource consumption (e.g. memory or CPU cycles) on a machine interpreting it. Indeed, fonts are sufficiently complex, and most (if not all) interpreters cannot be completely protected from malicious fonts without undue performance penalties.

Widespread use of fonts as necessary component of visual content presentation warrants that a careful attention should be given to security considerations whenever a font is either embedded into an electronic document or transmitted alongside media content.

Interoperability considerations:

As it was noted in the first paragraph of the "Security considerations" section, the same font format wrapper can be used to encode fonts with different types of glyph data represented as either TrueType or PostScript (CFF) outlines. Existing font rendering engines may not be able to process some of the particular outline formats, and downloading a font resource that contains unsupported glyph data format would result in inability of application to render and display text. Therefore, it would be extremely useful to clearly identify the format of the glyph outline data within a font using an optional parameter, and allow applications to make decisions about downloading a particular font resource sooner. Similar, another optional parameter is suggested to identify the type of text shaping and layout mechanism that is supported by a font. Please note that as new outline formats and text shaping mechanisms may be defined in the future, the set of allowed values for two optional parameters defined by this application may be extended.

Published specification:

As of the date of this submission, the main published specification is ISO/IEC 14496-22:2009 "Open Font Format". This media type registration is extracted from the ISO/IEC 14496-22/AMD2 of Open Font Format (OFF) specification being developed by ISO/IEC SC29/WG11.

Applications that use this media type:

Any and all applications that are able to create, edit or display textual media content.

Additional information:

Magic number(s):

The TrueType fonts and OFF / OpenType fonts containing TrueType outlines should use 0x00010000 as the 'sfnt' version number.

The OFF / OpenType fonts containing CFF data should use the tag 'OTTO' as 'sfnt' version number.

File extension(s):

Font file extensions used for OFF / OpenType fonts: .ttf, .otf

Typically, .ttf extension is only used for fonts containing TrueType outlines, while .otf extension can be used for any OpenType/OFF font, either with TrueType or CFF outlilnes.

Macintosh file type code(s):

(no code specified)

Intended usage:

COMMON

Restrictions on usage:

None

Author:

The ISO/IEC 14496-22 "Open Font Format" specification is a product of the ISO/IEC JTC1 SC29/WG11.

Change controller:

The ISO/IEC has change control over the above-mentioned standard.

Bibliography

[1] List of Locale ID (LCID) Values as Assigned by Microsoft - http://www.microsoft.com/globaldev/reference/lcid-all.mspx

[2] Apple's TrueType Manual, chapter 6: The 'post' table - https://developer.apple.com/fonts/TTRefMan/RM06/Chap6post.html

[3] Adobe Glyph List specification - http://sourceforge.net/adobe/aglfn/wiki/AGL%20Specification/

[4] Type 2 Charstring format, <http://partners.adobe.com/public/developer/en/font/5177.Type2.pdf>

[5] The Compact Font Format specification, <http://partners.adobe.com/public/developer/en/font/5176.CFF.pdf>

[6] Vendor registry - http://www.microsoft.com/typography/links/vendorlist.aspx

[7] Apple's TrueType Reference Manual - http://developer.apple.com/fonts/TTRefMan/

[8] TrueType Font Files, Technical Specification V1.66 http://www.microsoft.com/typography/SpecificationsOverview.mspx

[9] OpenType Layout Font Specification http://www.microsoft.com/typography/otspec/TTOCHAP1.htm

[10] Script-specific Development: http://www.microsoft.com/typography/SpecificationsOverview.mspx

[11] Feature Tags: http://www.microsoft.com/typography/developers/OpenType/featuretags.aspx

[12] PKCS#7 signatures: ftp://ftp.rsa.com/pub/pkcs/ascii/pkcs-7.asc

[13] Counter-signatures: ftp://ftp.rsa.com/pub/pkcs/ascii/pkcs-9.asc

[14] PANOSE Specification: http://www.panose.com/

[15] Portable Network Graphics (PNG) specification: http://www.w3.org/TR/PNG/

[16] Conformance requirements for SVG viewers: http://www.w3.org/TR/SVG11/conform.html#ConformingSVGViewers

[17] SVG Integration: http://www.w3.org/TR/svg-integration/

[18] CSS Custom Properties for Cascading Variables Module Level 1 specification: http://www.w3.org/TR/css-variables-1/

[19] Scalable Vector Graphics (SVG) 2 specification: http://www.w3.org/TR/SVG2

[20] Unicode Character Database file for joining-script properties http://www.unicode.org/Public/UCD/latest/ucd/ArabicShaping.txt

[21] Requirements for Japanese Text Layout (JLREQ): https://www.w3.org/TR/jlreq/

[22] Unicode Character Database (Unicode Annex #44) http://www.unicode.org/reports/tr44/tr44-18.html

[23] Unicode Technical Report #50: Unicode Vertical Text Layout. http://www.unicode.org/reports/tr50/

[24] The WOFF2 specification: http://www.w3.org/TR/WOFF2/

[25] IETF BCP 47 specification, "Tags for Identifying Languages". http://tools.ietf.org/html/bcp47

[26] IANA Language Subtag Registry.
http://www.iana.org/assignments/language-subtag-registry/language-subtag-registry

[27] Adobe Technical Note #5902: “PostScript Name Generation for Variation Fonts”.
http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/font/pdfs/5902.AdobePSNameGeneration.html

[28] CFF2 changes from CFF 1.0 https://www.microsoft.com/typography/otspec/cff2.htm#appendixD

[29] Example CFF2 Font:
https://www.microsoft.com/typography/otspec/cff2.htm#appendixA

[30] Adobe Technical Note #5015: "Type 1 Font Format Supplement".
http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/font/pdfs/5015.Type1_Supp.pdf

ICS 35.040.40

Price based on 628 pages