

**INTERNATIONAL ORGANISATION FOR STANDARDISATION  
ORGANISATION INTERNATIONALE DE NORMALISATION  
ISO/IEC JTC 1/SC 29/WG 3  
CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC 1/SC 29/WG 3 m 65472**

**Milford, Ontario, Canada – November 2023**

**Title: Proposed changes to accommodate larger character sets and to add new features**

**Author: Dave Crossland (Google Inc., [dcrossland@google.com](mailto:dcrossland@google.com)), Behdad Esfahbod ([behdad@behdad.org](mailto:behdad@behdad.org)), Laurence Penney ([lorp@lorp.org](mailto:lorp@lorp.org)), Liam Quin (Delightful Computing, [liam@delightfulcomputing.com](mailto:liam@delightfulcomputing.com)), Rod Sheeter (Google Inc., [rsheetter@google.com](mailto:rsheetter@google.com))**

## Contents

### Table of Contents

Contents.....	2
Introduction.....	2
maxp—Maximum profile [5.1.6].....	5
GLYP—Glyph data [insert before 5.2.4].....	6
LOCA—Index to location [new table inserted before 5.2.5 ‘loca’].....	19
GVAR—Glyph variations table (a new table based on ‘gvar’) [7.3.4.1.1].....	21
cmap—Character to glyph index mapping table [5.1.2].....	23
HHEA—Horizontal header [5.1.4].....	24
HMTX—Horizontal metrics [insert before 5.1.5 hmtx].....	26
VHEA—Vertical header table [insert before 5.6.9 ‘vhea’].....	27
VMTX—Vertical metric table.....	32
VORG—Vertical origin table [5.3.4].....	34
sbix—Standard bitmap graphics table [5.5.7].....	34
GDEF—The glyph definition table [6.3.2].....	35
GPOS—The glyph positioning table [6.3.3].....	36
GSUB—The glyph substitution table [6.3.4].....	36

### Introduction

This PDF document was produced on 20<sup>th</sup> November 2023 at 3pm.

Changes since the version from the start of October:

- GSUB/GPOS/GDEF headers now increase only the minor version and are compatible with the previous versions. New 32-bit fields are added at the end of the headers.
- Sub-tables are renamed from having 24 at the end to having 2 at the end.
- Top-level offsets are 32-bit and not 24-bit.
- Some minor editorial changes, and some smaller technical fixes as a result of technical comments both in meetings and sent privately.

This proposal introduces new tables GLYP and LOCA. These mirror the existing glyf and loca tables but allow more glyphs (24-bit glyph indices instead of 16-bit). The new tables also to permit adding new features to glyph definitions while still being able to produce fonts that work with older software.

Other new tables are added as a consequence, as described below.

The proposal enables the following:

1. Fonts can be made that work in both older and newer software, although older software will not of course support new features or access more than 65,535 glyphs in a font. Fonts can also be made that will not work in older software (e.g. have a 'GLYP' table but do not contain 'glyf', 'CFF', or 'CFF2'). This leaves compatibility questions in the hands of font designers and font tools.
2. The maximum number of glyphs in a font is increased from 65,535 (16 bits) to 16,777,216 (24 bits).

Note: The entire Unicode space is approximately 21 bits. However, the new table is versioned, so that if 24-bits becomes a problem the number could be raised and multi-gigabyte font files could be introduced.

In addition to increasing the maximum number of glyphs, and hence extending the glyph ID to be a 24-bit number, this proposal also address some limitations in the glyph data itself, as follows:

3. The glyph path data is extended to allow individual glyphs to contain both cubic and quadratic Bézier curves.
4. The glyph table format is also extended to support Variable Composite glyphs, which can refer to variable components. These variable components can have transformation matrices applied to them, as well as variable font axis values.

*Note:* The table names GLYX and LOCX have also been suggested.

In this proposal, the number of glyphs in the GLYP table is the same as the number of entries in the LOCA table, minus one because of the end pointer. Moreover, tables that encode glyph indices as two-byte numbers are either given new versions of structures that encode 24-bit glyph indices, or are replaced with new upper-case tables.

For tables where a new version with a new table tag is introduced, that is, where there is a table with the same name in both upper and lower case, if an implementation understands the new upper-case-named version, the corresponding lower-case-named table shall be ignored.

Tables are also extended where necessary to use 24-bit offsets instead of 16-bit offsets to allow for the larger tables that are necessary for the larger number of glyphs.

Where a new table has been introduced, such as GLYP or LOCA, the corresponding older table need not be present. Older software will not then be able to use the font unless it has a CFF alternative. But older software that does not understand more than the first 65535 glyphs in the font will be unable to access all the glyphs in any case, and might not understand variable components or mixed cubic and quadratic paths; sometimes this may be acceptable, in which case glyf and loca tables should be provided.

In the text that follows, new or changed text is **highlighted**, but entirely new sections are not.

Summary of new and changed tables (in alphabetical order):

Name	Status	Notes
cmap	extended	Format 16 is added, with 24-bit glyph IDs
GDEF	changed	Updated to avoid overflow, and for 24-bit Glyph IDs
<a href="#">GLYP</a>	new	Augmented version of glyf; length; variable composites; cubic segments

GPOS	changed	Updated to avoid overflow, and for 24-bit Glyph IDs
GSUB	changed	Updated to avoid overflow, and for 24-bit Glyph IDs
GVAR	new	Augmented version of gvar to allow more entries and to point to GLYF
HHEA	new	Same as hhea but with 32-bit numberOfHMetrics
HMTX	new	Augmented version of hmtx
<a href="#">LOCA</a>	new	Augmented version of loca
<a href="#">maxp</a>	unchanged	Previous proposal required numGlyphs to be set to 65535 for a font with more than that number of glyphs. Minor editorial note added to clarify this.
sbix	changed	Updated to derive size from GLYF
VHEA	New	Same as vhea but with 32-bit numberOfVMetrics
VMTX	new	Augmented version of vmtx
VORG	changed	Updated for 24-bit Glyph Ids and to allow more entries

## Per-table changes

### maxp—Maximum profile [5.1.6]

This table is unchanged except for an editorial note to add, highlighted in yellow.

This table establishes the memory requirements for this font. Fonts with CFF or CFF2 outlines shall use Version 0.5 of this table, specifying only the numGlyphs field. Fonts with TrueType outlines shall use Version 1.0 of this table, where all data is required.

#### Version 0.5

Type	Name	Description
Version16Dot16	version	0x00005000 for version 0.5
uint16	numGlyphs	The number of glyphs in the font.

#### Version 1.0

Type	Name	Description
Version16Dot16	version	0x00010000 for version 1.0.
uint16	numGlyphs	The number of glyphs in the font.  This is the number of glyphs in the 'glyf' table, if any.  <i>Note:</i> The separate GLYP table does not rely on this number; the number of entries in the GLYP table is determined by the size in bytes of the LOCA table, taking into account the value of indexToLocFormat.
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.

## GLYF—Glyph data [insert before 5.2.4]

### [5.2.4.1] 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. For details regarding scaling, grid-fitting and rasterization of TrueType outlines, see TrueType Fundamentals<sup>[35]</sup>.

The 'GLYF' table contains information that describes the glyphs in the font in a format based on the TrueType outline format.

For compatibility with older software, a 'glyf' table may also be present, in the same format as the 'GLYF' table, and contains outlines and hinting instructions suitable for older software. For details regarding scaling, grid-fitting and rasterization of TrueType outlines, see TrueType Fundamentals<sup>[35]</sup>. The 'glyf' table is restricted to contain no more than 65535 entries, and some flags and features from 'GLYF' entries are not supported; these are documented in the description of 'GLYF'.

If both 'GLYF' and 'glyf' tables are present, software that can process the 'GLYF' table shall ignore the 'glyf' table. The 'LOCA' table shall always be present if there is a non-empty 'GLYF' table.

The number of entries of the 'GLYF' table is equal to the number of entries in the 'LOCA' table, minus one to account for the pointer to the end of the last entry.

In some tables, some offset fields are duplicated but with a 2 at the end of their name, and are 32-bit instead of 16-bit. Where these are non-zero, software that can process the 'GLYF' table shall ignore the fields without the trailing '2' in their names, and shall use the '2' versions instead. The '2' versions shall be used where values exceed 65535, or where separate offset lists are needed for GLYF-aware software.

### 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. Glyph data blocks of length zero represent empty glyphs without contours, such as spacing glyphs.

If the font contains a 'glyf' table, the number of glyph definitions in it is equal to the numGlyphs field in

the 'maxp' table.

The 'GLYP' 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 'GLYP' table.

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 'GLYP' table must be in glyph ID order.

Each glyph description uses one of **three** formats:

- Simple glyph descriptions specify a glyph outline directly using Bézier control points.
- Composite glyph descriptions specify a glyph outline indirectly by referencing one or more glyph IDs to use as components.
- Variable composite glyph descriptions also specify a glyph outline indirectly, but can include transformations to be applied to those components, and participate in font variations. Variable composite glyph descriptions shall only be used in the 'GLYP' table, not the 'glyf' table.

In all cases, if the glyph is not empty the glyph description shall begin with a glyph header.

## Glyph headers

Each glyph description shall begin with a 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, and -2 for variable 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:** Variable Composite Glyphs are allowed only in the 'GLYP' table, not in 'glyf', for backwards compatibility with older software.

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). These values are obtained directly from the point coordinate data for the glyph, comparing all on-curve and off-curve points. Phantom points computed by the rasterizer are not relevant. Note that the bounding box defined by control points is guaranteed to contain the outline, but might not be tight to the outline.

The scaler will perform better if the glyph coordinates have been created such that the xMin for each glyph is equal to the left side bearing (lsb) for that glyph. 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 the *flags* field in the 'head' table.

**NOTE** 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 'GLYP' 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 Reference [36] for more background on phantom points.

**NOTE** Side bearings for glyph definitions in the older 'glyf' table are obtained from the 'hmtx' lower-case named table.

In a variable font, the minimum and maximum x or y values of control points can vary, and a tight bounding rectangle containing the outline or all points for an instance could be smaller or larger than for the default instance (that is, for the glyph description in this table). The xMin, yMin, xMax and yMax values might or might not encompass the derived outline for an instance. Also, the 'GVAR' (or 'gvar') table does not provide deltas for these values. If an application requires a bounding rectangle for a non-default instance of a glyph, the derived point data (with deltas applied) should be processed to determine a bounding rectangle.

#### Simple glyph description [5.2.4.1.1]

This is the information used to describe a glyph if numberOfContours is greater than or equal to 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 instruction byte code for the glyph.
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[ <i>variable</i> ]	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 1 In the 'glyph' 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. The number of points is determined by the last entry in the endPtsOfContours array. 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, and the sign is determined by the X_IS_SAME_OR_POSITIVE_X_SHORT_VECTOR flag. If not set, its interpretation depends on the X_IS_SAME_OR_POSITIVE_X_SHORT_VECTOR flag: If that other flag is set, the x-coordinate is the same as the previous x-coordinate, and no element is added to the xCoordinates array. If both flags are not set, the corresponding element in the xCoordinates array is two bytes and interpreted as a signed integer. See the description of the X_IS_SAME_OR_POSITIVE_X_SHORT_VECTOR flag for additional information.
0x04	Y_SHORT_VECTOR	Bit 2: If set, the corresponding y-coordinate is 1 byte long, and the sign is determined by the Y_IS_SAME_OR_POSITIVE_Y_SHORT_VECTOR flag. If not set, its interpretation depends on the Y_IS_SAME_OR_POSITIVE_Y_SHORT_VECTOR flag: If that other flag is set, the y-coordinate is the same as the previous y-coordinate, and no element is added to the yCoordinates array. If both flags are not set, the corresponding element in the yCoordinates array is two bytes and interpreted as a signed integer. See the

		description of the Y_IS_SAME_OR_POSITIVE_Y_SHORT_VECTOR flag for additional information.
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 equaling positive and 0 negative. If X_SHORT_VECTOR 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 equaling 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 the discussion of “Overlapping contours” in Apple’s specification <sup>[7]</sup> for details regarding behavior in Apple platforms.) When used, it shall be set on the first flag byte for the glyph. See additional details below.
0x80	CUBIC	Bit 7: Off-curve point belongs to a cubic-Bezier segment.  The CUBIC flag shall be used only in the ‘GLYP’ table, not the ‘glyph’ table. In the ‘glyph’ table it shall be set to zero.

If the CUBIC flag is non-zero, the corresponding off-curve point belongs to a Cubic Bézier path segment, and all of the following conditions shall be met:

- The number of consecutive cubic off-curve points within a contour (without wrap-around) is even.
- Either all the off-curve points between any two on-curve points (with wrap-around) have the CUBIC flag clear, or they all have the CUBIC flag set.
- The CUBIC flag shall only be set to 1 on off-curve points. For on-curve points it is reserved and shall be set to zero.

Every consecutive two off-curve points that have the CUBIC bit set define a cubic Bézier segment. Within any consecutive set of cubic off-curve points within a contour (with wrap-around), an implied on-curve point is inserted by the font processor at the mid-point between every second off-curve point and the next one.

If there are no on-curve points and all (even number of) off-curve points are CUBIC, the first off-curve point shall be considered the first control-point of a cubic Bézier curve, and the font processor shall insert implied on-curve points between the every second point and the next one as usual.

### **Filling Algorithm and Overlapping Contours**

A non-zero-fill algorithm is needed to avoid dropouts when contours overlap. 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. Implementations that always use a non-zero-fill algorithm will ignore this flag. Note that some implementations might check this flag specifically in non-variable fonts, but always use a non-zero-fill algorithm for variable fonts. This flag can be used in order to provide broad interoperability of fonts — particularly non-variable fonts — when glyphs have overlapping contours.

Variable fonts often make use of overlapping contours. This has implications for tools that generate static-font data for a specific instance of a variable font, if broad interoperability of the derived font is desired: if a glyph has overlapping contours in the given instance, then the tool should either set the OVERLAP\_SIMPLE flag in the derived glyph data, or else should merge contours to remove overlap of separate contours.

NOTE 2 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.

#### **[5.2.4.1.2] Composite glyph description**

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

NOTE 1 A numberOfContours value of -1 is recommended to indicate a composite glyph.

A composite, or compound, glyph describes an outline indirectly by referencing other glyphs that get incorporated into the composite glyph as components. This is useful when the same contours are needed for multiple glyphs as it provides consistency for contours that are repeated across multiple glyphs and can also provide significant size reduction.

To add clarity in explaining composite glyphs, the terms parent and child will be used, a composite glyph description being the parent, and the other glyphs referenced as components being the children.

Composite glyphs may be nested within other composite glyphs—that is, a composite glyph parent may include other composite glyphs as child components. Thus, a composite glyph description is a directed graph. This graph shall be acyclic, with every path through the graph leading to a simple glyph as a leaf node. The maxComponentDepth field in the 'maxp' table is set to indicate the maximum nesting depth across all composite glyphs in a font. There is no minimum nesting depth that must be supported. For fonts to be compatible with the widest range of implementations, nesting of composites should be avoided.

NOTE 2 Some PostScript devices (and possibly other implementations) do not correctly render glyphs that have nested composite descriptions. A composite glyph description that has nested composites can be flattened to reference only simple glyphs as child components. This can lose some benefits of de-duplication of information, but can still retain significant size-saving benefits as well as providing broader compatibility.

The data for a composite glyph description is comprised of a sequence of data blocks for each child component glyph. A flag within the data for each component is used to indicate if there are additional components in the sequence. The sequence is processed in the order given, with the contours from each child glyph incorporated into the parent. As the contours from a child are incorporated, its control points are renumbered to follow sequentially after all points previously incorporated into the parent.

Each glyph has an outline positioned within the font design grid based on the x and y coordinates of its control points. When incorporated as a child into a composite glyph, the parent can control placement of the child outline within the parent's design grid. This can be done in two different ways: by specifying a vector offset added to (x, y) coordinates of the child's control points, or by specifying one control point from the child's outline that is aligned with a specified control point in the parent. The second mechanism assumes some outlines have already been incorporated into the parent, so cannot be used for the first component glyph.

The parent can also specify a scale or other affine transform to be applied to a child glyph as it is incorporated into the parent. The transform can affect an offset vector used to position the child glyph; see below for additional details.

Each component glyph can include instructions that apply to its outline. A parent composite glyph description can include instructions that apply to the composite as a whole, after instructions for each child have been performed. Instructions for the parent composite apply to the accumulated contour data from components with points renumbered, as described above.

Before each child is incorporated into the parent, it is processed, with phantom points defined and hinting instructions performed. Thus, if placement of the child is done by alignment of points, the child's phantom points can be used for this alignment, and instructions in child glyphs that affected their points will already have been performed.

The data block for each child component starts with two uint16 values: a flags field, and a glyph ID. These are followed by two argument fields, though the size and interpretation of the arguments varies according to the flags that are set. Optional fields describing a transformation can follow the arguments, depending on the flags.

### *Component glyph record*

Type	Name	Description
uint16	flags	component flag
uint16 or uint24	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
[transform data]		optional transform data—see below

The C pseudo-code fragment below shows how the sequence of composite glyph records is stored and parsed; definitions for the flag bits follow this fragment:

```
do {
    uint16 flags;
    uint24 glyphIndex;
    if ( flags & ARG_1_AND_2_ARE_WORDS) {
        (int16 or FWORD) argument1;
```

```

        (int16 or FWORD) argument2;
    } else {
        uint16 argland2; /* (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_INSTRUCTIONS){
    uint16 numInstr
    uint8 instr[numInstr]

```

The following composite glyph flags are defined:

Mask	Flags	Description
0x0001	ARG_1_AND_2_ARE_WORDS	Bit 0: If this is set, the arguments are 16-bit (uint16 or int16); otherwise, they are bytes (uint8 or int8).
0x0002	ARGS_ARE_XY_VALUES	Bit 1: If this is set, the arguments are signed xy values; otherwise, they are unsigned point numbers.
0x0004	ROUND_XY_TO_GRID	Bit 2: If set and ARGS_ARE_XY_VALUES is also set, the xy values are rounded to the nearest grid line. Ignored if ARGS_ARE_XY_VALUES is not set.
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: There is a 2 by 2 transformation that will be used to scale the component.
0x0100	WE_HAVE_INSTRUCTIONS	Bit 8: Following the last component are instructions for the composite glyph.
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 component glyph. This works for hinted and unhinted glyphs.
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, component glyphs may overlap without having this flag set. It can affect behaviors in some platforms, however. (See Apple’s specification <sup>[7]</sup> for details regarding behavior in Apple platforms.) When used, it shall 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: The composite is designed to have the component offset scaled. Ignored if ARGS_ARE_XY_VALUES is not set.
0x1000	UNSCALED_COMPONENT_OFFSET	Bit 12: The composite is designed not to have the component offset scaled. Ignored if ARGS_ARE_XY_VALUES is not set.
0x2000	GID_IS_24_BIT	Bit 13: Set to allow encoding 24bit glyph indices in composite glyphs.
0xC010	RESERVED	Bits 4, 14 and 15 are reserved and shall be set to 0.

The argument1 and argument2 fields of the component glyph record are used to determine the placement of the child component glyph within the parent composite glyph. They are interpreted either as an offset vector or as points from the parent and the child, according to whether the ARGS\_ARE\_XY\_VALUES flag is set. This flag must always be set for the first component of a composite glyph.

If ARGS\_ARE\_XY\_VALUES is set, then argument1 and argument2 are interpreted as units in the design coordinate system and an offset vector  $(x, y) = (\text{argument1}, \text{argument2})$  is added to the coordinates of each control point of the component glyph. In a variable font, the offset vector can be modified by deltas in the 'gvar' table; see 7.3.4.3 for details. If a scale or transform matrix is provided, the offset vector might or might not be subject to the transformation; see the discussion below of the SCALED\_COMPONENT\_OFFSET and UNSCALED\_COMPONENT\_OFFSET flags for details.

If ARGS\_ARE\_XY\_VALUES is set and the ROUND\_XY\_TO\_GRID flag is also set, the offset vector (after any transformation and variation deltas are applied) is grid-fitted, with the x and y values rounded to the nearest pixel grid line.

If ARGS\_ARE\_XY\_VALUES is not set, then argument1 is a point number in the parent glyph (from contours incorporated and re-numbered from previous component glyphs); and argument2 is a point number (prior to re-numbering) from the child component glyph. Phantom points from the parent or the child may be referenced. The child component glyph is positioned within the parent glyph by aligning the two points. If a scale or transform matrix is provided, the transformation is applied to the child’s point before the points are aligned.

In a variable font, when a component is positioned by alignment of points, deltas are applied to component glyphs before this alignment is done. Any deltas specified for the parent composite glyph to be applied to components positioned by point alignment are ignored. See 7.3.4.3 for details.

The WE\_HAVE\_A\_SCALE, WE\_HAVE\_AN\_X\_AND\_Y\_SCALE and WE\_HAVE\_A\_TWO\_BY\_TWO flags are mutually exclusive: no more than one of these may be set. If WE\_HAVE\_A\_SCALE is set, one additional

F2DOT14 value is appended to the component glyph data; if WE\_HAVE\_AN\_X\_AND\_Y\_SCALE is set, two F2DOT14 values are appended; if WE\_HAVE\_A\_TWO\_BY\_TWO, four F2DOT14 values are appended. The child component glyph is transformed as it is incorporated into the parent composite glyph, prior to grid-fitting of the parent. The transform can affect an offset vector used to position the child; see discussion below of the SCALED\_COMPONENT\_OFFSET and UNSCALED\_COMPONENT\_OFFSET flags for details.

The WE\_HAVE\_INSTRUCTIONS flag is used to indicate that the parent composite glyph has instructions, in addition to instructions for any of the child component glyphs. If the flag is set on any component glyph, then a uint16 value is read immediately after the last component glyph to get the byte length for instructions.

The purpose of USE\_MY\_METRICS is to force the lsb and rsb to take on values obtained from the component glyph. 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 that the behavior of the USE\_MY\_METRICS operation is undefined for rotated component glyphs.

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 may 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.

### Variable Composite Glyph Descriptions [new sub-section]

A Variable Composite glyph starts with the standard glyph header with a *numberOfContours* of -2, followed by a number of Variable Component records. Each Variable Component record is at least five bytes long.

#### Variable Component Record

type	name	notes
uint16	flags	see below
uint8	numAxes	Number of axes to follow
GlyphID16 or GlyphID24	gid	This is a GlyphID16 if bit 12 of <i>flags</i> is clear, else GlyphID24
uint8 or uint16	axisIndices[numAxes]	This is a uint16 if bit 1 of <i>flags</i> is set, else a uint8
F2DOT14	axisValues[numAxes]	The axis value for each axis
FWORD	TranslateX	Optional, only present if bit 3 of <i>flags</i> is set

type	name	notes
FWORD	TranslateY	Optional, only present if bit 4 of <i>flags</i> is set
F4DOT12	Rotation	Optional, only present if bit 5 of <i>flags</i> is set
F6DOT10	ScaleX	Optional, only present if bit 6 of <i>flags</i> is set
F6DOT10	ScaleY	Optional, only present if bit 7 of <i>flags</i> is set
F4DOT12	SkewX	Optional, only present if bit 8 of <i>flags</i> is set
F4DOT12	SkewY	Optional, only present if bit 9 of <i>flags</i> is set
FWORD	TCenterX	Optional, only present if bit 10 of <i>flags</i> is set
FWORD	TCenterY	Optional, only present if bit 11 of <i>flags</i> is set

### Variable Component Transformation

The transformation data consists of individual optional fields, which can be used to construct a transformation matrix.

*Transformation fields:*

name	default value
TranslateX	0
TranslateY	0
Rotation	0
ScaleX	1
ScaleY	1
SkewX	0
SkewY	0
TCenterX	0
TCenterY	0

The TCenterX and TCenterY values represent the “center of transformation”.

Details of how to build a transformation matrix, as pseudo-Python code:

```
# Using fontTools.misc.transform.Transform
```



```

t = Transform() # Identity
t = t.translate(TranslateX + TCenterX, TranslateY + TCenterY)
t = t.rotate(Rotation * math.pi)
t = t.scale(ScaleX, ScaleY)
t = t.skew(-SkewX * math.pi, SkewY * math.pi)
t = t.translate(-TCenterX, -TCenterY)

```

#### Variable Component Flags

bit number	meaning
0	Use my metrics
1	axis indices are shorts (clear = bytes, set = shorts)
2	if ScaleY is missing: take value from ScaleX
3	have TranslateX
4	have TranslateY
5	have Rotation
6	have ScaleX
7	have ScaleY
8	have SkewX
9	have SkewY
10	have TCenterX
11	have TCenterY
12	gid is 24 bit
13	axis indices have variation
14	reset unspecified axes
15	(reserved, set to 0)

#### Processing

Variations of composite glyphs shall be processed as follows:

For each composite glyph, a vector of coordinate points is prepared, and its variation applied from the 'gvar' table. The coordinate points for the composite glyph are a concatenation of those for each variable

component in order. For the purposes of 'GVAR' delta IUP calculations, each point is considered to be in its own contour.

The coordinate points for each variable component consist of those for each axis value if flag bit 13 (axis indices have variation) is set, represented as the X value of a coordinate point; followed by up to five points representing the transformation.

The five possible points encode, in order, in their X, Y components, the following transformation components:

1. (TranslateX, TranslateY),
2. (Rotation, 0)
3. (ScaleX, ScaleY)
4. (SkewX, SkewY)
5. (TcenterX, TcenterY)

Only the transformation components present according to the flag bits are encoded. For example, the point (TranslateX, TranslateY) is encoded if and only if either of flag 3 (have TranslateX) or 4 (have TranslateY) is set. If that point is encoded, the variations from it will be applied to the transformation components even if a specific component has its flag bit off.

*Example:* if only flag 3 is set (have TranslateX), any variations for TranslateY are still applied to the TranslateY of the transformation.

The component glyphs to be loaded use the coordinate values specified (with any variations applied if present). For any unspecified axis, the value used depends on flag bit 14 (reset unspecified axes). If the flag is set, then the normalized value zero is used. If the flag is clear the axis values from current glyph being processed (which itself might recursively come from the font or its own parent glyphs) are used.

*Example:* if the font variations have wght=.25 (normalized), and the current glyph being processed is using wght=.5 because it was referenced from another VarComposite glyph itself, when referring to a component that does not specify the wght axis, if flag bit 14 (reset unspecified axes) is set, then the value of wght=0 (default) will be used. However, if flag bit 14 is clear, wght=.5 (from current glyph) will be used.

**Note:** While it is the behavior of the 'glyf' table that glyphs loaded are shifted to align their LSB to that specified in the 'hmtx' table, much like regular Composite glyphs, this does not apply to component glyphs being loaded as part of a VariableComposite glyph.

**Note:** A static (non-variable) font that uses VarComposite glyphs, *would not* have fvar or avar tables but *would* have the 'gvar' (or 'GVAR') table. This combination is possible because the gvar table encodes its own number-of-axes, and the axes in this proposal are specified in the normalized space.

### Comparing CFF2 with CFF and glyf [5.3.3.13]

The formats intended for storing monochrome glyph outlines in OFF fonts are the 'GLYP' and 'glyf' tables (subclause 5.3.4), the CFF table (subclause 5.4.2), and the CFF2 table.

CFF2 and CFF use cubic (3rd order) Bézier curves to represent glyph outlines, whereas the 'glyf' table

uses quadratic (2nd order) Bézier curves. The 'GLYP' table supports a mixture of quadratic and cubic Bézier curves.

CFF2 and CFF also use a different conceptual model for “hints” than the 'GLYP' and 'glyf' tables. The tables also differ in relation to support of variations and in how variation data is stored.

The following table provides a summary comparison of the 'CFF2', 'CFF', 'GLYP' 'glyf' tables. Note that some of these differences might not be exposed in high-level font editing software or in runtime programming interfaces.

Consideration	GLYP	glyf	CFF	CFF2
curves	quadratic (2nd order) and cubic (3rd order)	quadratic (2nd order)	cubic (3rd order)	cubic (3rd order)
coordinate precision	1 FUnit		1/65536 FUnit	
hinting	TrueType instructions move outline points by controlled amounts		alignment zones apply to all glyphs, stem locations are declared in each glyph	
decoding	not stack-based (except TrueType instructions)		mostly stack-based	
Font variations	yes: outline variation data is stored in 'gvar' (subclause 7.3.4); hint variation data is stored in 'cvar' (subclause 7.3.2)		no	yes: variation data for outlines and hints is stored within the CFF2 table
data redundancy	low		moderate	low
overlapping contours	yes		no	yes
variable components	yes	no	no	

## LOCA—Index to location [new table inserted before 5.2.5 'loca']

The index to location table (LOCA) provides a mapping from glyph indices to offsets, which are the byte locations of TrueType glyph descriptions in the 'GLYP' table, relative to the beginning of that table (“local” offsets). In order to compute the length of the last glyph description, after the entry pointing to the last valid glyph description, there is a final entry that points to the end of the glyph data.

The 'LOCA' table is same as 'loca' defined in the next section, except how size is determined, and that in practice it can be longer. Entries in the 'loca' table point into the 'glyf' table; entries in the 'LOCA' table point into the 'GLYP' table.

If both 'LOCA' and 'loca' tables are present, the 'loca' table shall be ignored. The 'loca' table is used by older software that cannot process 'LOCA'.

The number of entries in LOCA is determined by dividing the size in bytes of the LOCA table by two or by four, depending on the `IndexToLocFormat` in the font header:

- For a Format 0 'loca' or 'LOCA' table (`head.indexToLocFormat = 0`), the number of entries is determined by the length of the table divided by 2.
- For a Format 1 'loca' or 'LOCA' table (`head.indexToLocFormat = 1`), the number of entries is determined by length of the table divided by 4.

**Note:** Since the format of both 'LOCA' and 'loca' is determined by the value of `IndexToLocFormat` in the font header, if both tables are present they must be in the same format as each other.

The table is an array of `n` offsets, where `n` is the number of glyphs in the font plus one. There are two formats for the array that use different sizes for the offsets, as described below. The format used is determined by the `indexToLocFormat` field in the 'head' table (5.1.3).

NOTE: With the 'loca' table, the total size of the array must equal the size of the offsets times the value of the `numGlyphs` field in the 'maxp' table (5.1.6) plus one. This does not apply to 'GLYP'.

Offsets must be two-byte aligned and must be in ascending order, with `loca[n] <= loca[n+1]`. By definition, glyph index zero points to the "missing character", which is the glyph 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 glyph, then the first and second offsets should have the same value. This also applies to any other glyphs without an outline, such as the space character: if a glyph has no outline, then `loca[n] = loca[n+1]`.

There are two formats of this table: the short format, and the long format. The format is specified in the `indexToLocFormat` entry in the 'head' table.

#### Short format

Type	Name	Description
Offset16	<code>offsets[numGlyphs + 1]</code>	The local offset divided by 2 is stored.

#### Long format

Type	Name	Description
Offset32	<code>offsets[numGlyphs + 1]</code>	The actual local offset is stored.

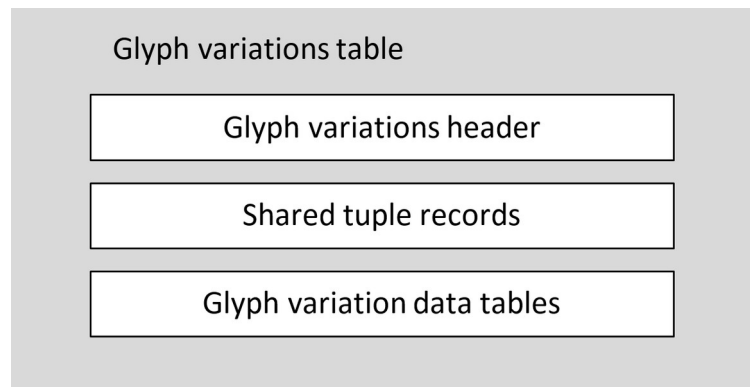
## GVAR—Glyph variations table (a new table based on 'gvar') [7.3.4.1.1]

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.

#### 'gvar' header [7.3.4.1.1]

The glyph variations table header format is as follows. **Note that the only difference between GVAR and gvar is that glyphCount is a uint32 in GVAR, not a uint16.**

*'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.

uint32	glyphCount	The number of glyphs in the GLYP table that have variation data..
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 (Offset16) 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.

[Remainder of GVAR section is identical to the rest of gvar]

## cmap—Character to glyph index mapping table [5.1.2]

### Table overview [5.1.2.1]

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 field 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. Subtable format 16 was added to allow more than 65535 glyphs in a font; it is similar to format 14 but uses 24-bit glyph indices.

Of the seven available formats, not all are commonly used today. Formats 4, 12, and 16 are appropriate for most new fonts, depending on the Unicode character repertoire supported. Format 14 is used in many applications for support of Unicode variation sequences. Format 16 is needed for larger fonts. Some platforms also make use for format 13 for a last-resort fallback font. Other subtable formats are not recommended for use in new fonts. Application developers, however, should anticipate that any of the formats may be used in fonts.

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

### 'cmap' Header [5.1.2.2]

This section is unchanged. A new sub-section, 5.1.2.5.11, is added after 5.1.2.5.10 Format 14: Unicode variation sequences:

#### [5.1.2.5.11] Format 16: U24-bit CMAP table

Subtable format 16 is the same as subtable format 14, except that it uses 24-bit index values in the UVSMapping Records:

##### UVSMapping24 Record:

Type	Name	Description
uint24	unicodeValue	Base Unicode value of the UVS
uint24	glyphID	Glyph ID of the UVS

This is intended for use with the 'GLYP' table.

### HHEA—Horizontal header [5.1.4]

This table is intended for use with the 'GLYP' table. The lower-case named 'hhea' table shall be used in conjunction with the lower-case named 'glyf' table.

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—see remarks below.
FWORD	descender	Typographic descent—see remarks below.
FWORD	lineGap	Typographic line gap. Negative lineGap values are treated as zero in some legacy platform implementations.
UFWORD	advanceWidthMax	Maximum advance width value in 'hmtx' table.
FWORD	minLeftSidebearing	Minimum left sidebearing value in 'hmtx' table for glyphs with contours (empty glyphs should be ignored).
FWORD	minRightSidebearing	Minimum right sidebearing value; calculated as $\min(\text{aw} - (\text{lsb} + \text{xMax} - \text{xMin}))$ for glyphs with contours (empty glyphs

		should be ignored).
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.
uint32	numberOfHMetrics	Number of hMetric entries in 'hmtx' table

The ascender, descender and linegap values in this table are Apple specific; see the 'hhea' table specification of Apple's TrueType Reference Manual [7] for details regarding Apple platforms. The sTypoAscender, sTypoDescender and sTypoLineGap fields in the OS/2 table are used on the Windows platform, and are recommended for new text-layout implementations. Font developers should evaluate behavior in target applications that may use fields in this table or in the OS/2 table to ensure consistent layout. See the descriptions of the OS/2 fields (5.1.8.23 – 5.1.8.25) for additional details.

### '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](#).

## HMTX—Horizontal metrics [insert before 5.1.5 hmtx]

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 ('GLYPH', 'CFF' or 'CFF2') and the horizontal metrics table. The horizontal metrics ('HMTX') table provides glyph advance widths and left side bearings.



Note: if the 'GLYP' table is present, any 'glyf' table shall be ignored, along with 'hmtx'.

If the GLYP is not present in a font, 'HMTX' shall be ignored, and 'htmx' shall be used.

In a font with TrueType outline data, the 'GLYP' 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 'GLYP' 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 'GLYP' 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 'GLYP' 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, an 'hmtx' or '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' or '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 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 left side bearing 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' or 'hmtx' tables and the 'HVAR' table.

NOTE: for maximum interoperability the 'hmtx' table rather than HMTX should be used with CFF; however, 'hmtx' is restricted to 16-bit glyph IDs.

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.

NOTE: The 'HHEA' table is for use with HMTX and GLYP. The 'hhea' table is used with the 'glyf' table. The primary difference between 'HMTX' and 'hmtx' is that 'HMTX' obtains numberOfHMetrics from 'HHEA' rather than from 'hhea'.

If numberOfHMetrics is less than the total number of glyphs in the 'GLYP' table, then the hMetrics array is followed by an array for the left side bearing values of the remaining glyphs.

#### 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.

FWORD	leftSideBearings  [numGlyphs - numberOfHMetrics]	Left side bearings for glyph IDs greater than or equal to numberOfHMetrics.
-------	---	---

*LongHorMetric Record:*

Type	Name	Description
UFWORD	advanceWidth	Advance width, in font design units.
FWORD	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 'GLYP' 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:

$$\text{rsb} = \text{aw} - (\text{lsb} + \text{xMax} - \text{xMin})$$

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:

$$\text{pp1} = \text{xMin} - \text{lsb}$$

$$\text{pp2} = \text{pp1} + \text{aw}$$

If a glyph has no contours, xMax/xMin are not defined. The left side bearing indicated in the 'HMTX' table for such glyphs should be zero.

## VHEA—Vertical header table [insert before 5.6.9 ‘vhea’]

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).

**Note:** The 'VHEA' and 'VMTX' tables, like the 'HHEA' and 'HMTX' counterparts, are used with the 'GLYP' table. If a 'GLYP' table is used, any 'hhea' or 'hmtx' lower-case named tables shall be ignored.

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*

<b>Version 1.0 Type</b>	<b>Name</b>	<b>Description</b>
Version16Dot16	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 $\max(\text{tsb} + (\text{yMax} - \text{yMin}))$
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.
uint32	numOfLongVerMetrics	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
Version16Dot16	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 FUnits from the ideographic em-box center baseline for the vertical axis to the right edge of the ideographic em-box.  It is usually set to (head.unitsPerEm)/2. For example, a font with an em of 1000 FUnits will set this field to 500. See the §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 FUnits from the ideographic em-box center baseline for the vertical axis to the left edge of the ideographic em-box.  It 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	minTopSideBearing	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	minBottomSideBearing	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 $\max(\text{tsb} + (\text{yMax} - \text{yMin}))$
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.
uint24	numOfLongVerMetrics	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](#).

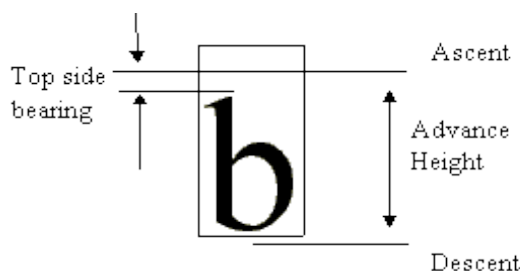
### *Vertical Header Table Example*

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	$\max(\text{tsb} + (\text{yMax} - \text{yMin})) = 2036$ .
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.

34/3	258	numOfLongVerMetrics	Number of advance heights in the vertical metrics table is 258.

## 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 5.11 – Vertical Metrics**

OFF vertical 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').

**Note:** Fonts using the upper-case named 'GLYP' table shall use 'HHEA', 'HMTX', 'VHEA' and 'VMTX' tables as needed. The upper-case named tables take precedence over the lower-case equivalents.

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 implementations 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 ('vkern') 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.

### VORG—Vertical origin table [5.3.4]

Replace the two tables under Vertical Origin Table Format as follows:

#### Vertical Origin Table Format

Type	Name	Description
uint16	majorVersion	Major version (starting at 1). Set to 1 or 2.



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 or uint32	numVertOriginYMetrics	Number of elements in the vertOriginYMetrics array. This is uint16 when MajorVersion is 1, and uint24 otherwise.

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 or uint24	glyphIndex	Glyph index. This shall be uint16 if MajorVersion is 1. It shall be uint24 otherwise.
int16	vertOriginY	Y coordinate, in the font's design coordinate system, of the vertical origin of glyph with index glyphIndex.

## sbix—Standard bitmap graphics table [5.5.7]

The number of glyphs referenced by the table is the number of glyphs in the font as defined earlier. Same applies to non-OpenType tables kerx and morx.

### Change

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.

### To

A font that includes an 'sbix' table may also include outline glyph data in a 'GLYF' '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.

## Table dependencies [5.5.7.4]

### Change

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.

To:

The glyph count is derived from the size of the 'GLYP' table when present, or 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, or in 'hmtx' and 'vmtx' for use with the lower-case named 'glyf' table.

## GDEF—The glyph definition table [6.3.2]

The main GDEF struct is augmented with a version 2 to alleviate offset-overflows when classDef and other structs grow large:

Before “**Glyph Class Definition table**” and after the table *GDEF Header, Version 1.3*, add:

*GDEF Header, Version 1.4*

Type	Name	Description
uint16	majorVersion	Major version of the GDEF table, = 1
uint16	minorVersion	Minor version of the GDEF table, = 4
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)
Offset32	glyphClassDefOffset2	Offset to class definition table for glyph type, from beginning of GDEF header (may be NULL)
Offset32	attachListOffset2	Offset to attachment point list table, from beginning of GDEF header (may be NULL)
Offset32	ligCaretListOffset2	Offset to ligature caret list table, from beginning of GDEF header (may be NULL)
Offset32	markAttachClassDefOffset2	Offset to class definition table for mark attachment type, from beginning of GDEF header (may be NULL)
Offset32	markGlyphSetsDefOffset2	Offset to the table of mark glyph set definitions, from beginning of GDEF header (may be NULL)

When `minorVersion` is 4, the offsets whose names end in 2, such as `attachListOffset2`, are available for use when values greater than 65535 are needed. If such an offset is non-zero, the corresponding 16-bit offset shall be ignored.

## GPOS—The glyph positioning table [6.3.3]

We define version two, to allow for additional glyphs and to avoid overflow.

### Changes:

After GPOS Header, Version 1.1, just before 6.3.3.3 GPOS lookup type descriptions add:

#### *GPOS Header, Version 1.2*

Type	Name	Description
uint16	majorVersion	Major version of the GPOS table, = 2
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
Offset32	featureVariationsOffset	Offset to FeatureVariations table, from beginning of GPOS table (may be NULL)
Offset32	scriptListOffset2	Offset to ScriptList table, for use when values greater than 65535 are needed.
Offset32	featureListOffset2	Offset to FeatureList table, for use when values greater than 65535 are needed.
Offset32	lookupListOffset2	Offset to FeatureList table, for use when values greater than 65535 are needed.

NOTE: The *featureVariationsOffset* remains 32-bits, the same as in the GPOS Header Version 1.1.

Note: The offsets whose names end in 2, such as *scriptListOffset2*, are for use when values greater than 65535 are needed.

[Typo note: “coverageOffet” appears in *SinglePosFormat1* subtable, missing an ‘s’]

At the end of 6.3.3.1 Lookup Type 2: Pair adjustment positioning subtable, just before 6.3.3.2 Lookup Type 2: Pair adjustment positioning subtable, add:

### Single Adjustment Positioning: Format 3: Single Positioning Value

Format 3 is the same as Format 1, but uses 24 bits.

*SinglePosFormat3 subtable:*

Type	Name	Description
uint16	posFormat	Format identifier: format = 3
Offset24	coverageOffset	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 4: Array of positioning values

Format 4 is the same as Format 3, but uses 24 bits.

*SinglePosFormat4 subtable:*

Type	Name	Description
uint16	posFormat	Format identifier: format = 4
Offset24	coverageOffset	Offset to Coverage table, from beginning of SinglePos subtable.
uint16	valueFormat	Defines the types of data in the ValueRecords.
uint24	valueCount	Number of ValueRecords – must equal glyphCount in the Coverage table.
ValueRecord	valueRecords [valueCount]	Array of ValueRecords – positioning values applied to glyphs.

At the end of section 6.3.3.2 Lookup Type 2: Pair adjustment positioning subtable, just before the start of 6.3.3.3 Lookup Type 3: Cursive attachment positioning subtable, add:

### Pair Adjustment Positioning Format 3: Adjustments for glyph pairs

Format 3 is the same as Format 1, but uses 24-bit glyph identifiers:

#### *PairPosFormat3 subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 3
Offset24	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).
uint24	pairSetCount	Number of PairSet tables.
Offset24	pairSetOffsets [pairSetCount]	Array of offsets to PairSet2 tables. Offsets are from the beginning of PairPos subtable, ordered by Coverage Index.

A PairSet2 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.

In Format 3, the PairSet2 table uses 24 bits for the number of pairs, to avoid overflow.

#### *PairSet2 table*

Type	Name	Description
uint24	pairValueCount	Number of PairValueRecords
PairValueRecord	pairValueRecords [pairValueCount]	Array of PairValueRecords, ordered by glyph ID of the second glyph.

### Pair Adjustment Positioning Format 4: Class pair adjustment

Format 4 is the same as Format 2 except for using 24 bits instead of 16 where needed.

#### *PairPosFormat4 subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 4

Offset24	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)
Offset24	classDef1Offset	Offset to ClassDef table, from beginning of PairPos subtable – for the first glyph of the pair
Offset24	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

At the end of 6.3.3.3.3 Lookup Type 3: Cursive attachment positioning subtable, add:

### **Cursive attachment positioning Format2: Cursive attachment**

Format 2 is the same as Format 2, except for using 24-bit numbers where needed.

#### *CursivePosFormat2 subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 2
Offset24	coverageOffset	Offset to Coverage table, from beginning of CursivePos subtable.
uint24	entryExitCount	Number of EntryExit2 records.
EntryExitRecord24	entryExitRecord[entryExitCount]	Array of EntryExit2 records, in Coverage Index order.

The EntryExit 2 Records are also in a slightly different format, again using 24-bit numbers to avoid overflow in large fonts:

#### *EntryExitRecord2*

Type	Name	Description
Offset24	entryAnchorOffset	Offset to EntryAnchor table, from beginning of

		CursivePos subtable (may be NULL).
Offset24	exitAnchorOffset	Offset to ExitAnchor table, from beginning of CursivePos subtable (may be NULL).

At the end of 6.3.3.4 Lookup Type 4: Mark-to-Base attachment positioning subtable, just before 6.3.3.5 Lookup Type 5: Mark-to-Ligature attachment positioning subtable, insert:

*MarkBasePosFormat2 subtable*

Type	Name	Description
uint16	posFormat	Format identifier-format = 2
Offset24	markCoverageOffset	Offset to MarkCoverage table, from beginning of MarkBasePos subtable
Offset24	baseCoverageOffset	Offset to BaseCoverage table, from beginning of MarkBasePos subtable
uint16	markClassCount	Number of classes defined for marks
Offset24	markArrayOffset	Offset to MarkArray2 table, from the beginning of the MarkBasePos subtable
Offset24	baseArrayOffset	Offset to BaseArray2 table, from the beginning of the MarkBasePos subtable

The BaseArray table is similarly updated to use 24-bit numbers in Format 2:

*BaseArray2 table*

Type	Name	Description
uint24	baseCount	Number of BaseRecord2 records
BaseRecord2	baseRecords[baseCount]	Array of BaseRecord2 records in order of baseCoverage Index

The Base Record itself must use 24-bit offsets:

*BaseRecord2*

Type	Name	Description
Offset24	baseAnchorOffset[markClassCount]	Array of offsets (one per mark class) to Anchor tables. Offsets are from the beginning of the BaseArray2 table,

		ordered by class.
--	--	-------------------

A MarkArray2 table is defined for use in 24-bit contexts:

*MarkArray2 table*

Type	Name	Description
uint24	markCount	Number of MarkRecord2 records
MarkRecord2	markRecords [markCount]	Array of MarkRecord2 records, ordered by corresponding glyphs in the associated mark Coverage order

*MarkRecord2*

Type	Name	Description
uint16	markClass	Class defined for the associated mark
Offset24	markAnchorOffset	Offset to Anchor table, from beginning of MarkArray2 table

At the end of subsection 6.3.3.3.5 Lookup Type 5: Mark-to-Ligature attachment positioning subtable, just before 6.3.3.6 Lookup Type 6 Mark-to-Mark attachment positioning subtable, insert:

### **Mark-to-Ligature attachment positioning Format2: Mark-to-Ligature Attachment**

Format 2 is the same as Format 1 except that it uses 24-bit numbers where needed.

NOTE: The number of classes remains 16-bit.

*MarkLigPosFormat2 subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 2
Offset24	markCoverageOffset	Offset to markCoverage table, from the beginning of the MarkLigPos subtable
Offset24	ligatureCoverageOffset	Offset to the ligatureCoverage table, from the beginning of the MarkLigPos subtable



uint16	markClassCount	Number of defined mark classes
Offset24	markArrayOffset	Offset to the MarkArray table, from the beginning of the MarkLigPos subtable
Offset24	ligatureArrayOffset	Offset to the LigatureArray2 table, from the beginning of the MarkLigPos subtable

The LigatureArray table is also updated:

#### *LigatureArray2 table*

Type	Name	Description
uint24	ligatureCount	Number of LigatureAttach table offsets
Offset24	ligatureAttachOffsets [ligatureCount]	Array of offsets to LigatureAttach tables. Offsets are from the beginning of the LigatureArray table, ordered by ligatureCoverage Index

The LigatureAttach table is unchanged from Format 1, imposing a limit of 65535 component records in a single ligature. However, the Component Record in Format 2 uses 24-bit numbers to avoid overflow. Note that the markClassCount field in the *MarkLigPosFormat2* table remains 16-bit:

#### *ComponentRecord2*

Type	Name	Description
Offset24	ligatureAnchorOffsets [markClassCount]	Array of offsets (one per class) to Anchor tables. Offsets are from beginning of LigatureAttach table, ordered by class (may be NULL)

At the end of 6.3.3.6 Lookup Type 6: Mark-to-Mark attachment positioning subtable, just before Lookup Type 7: Contextual positioning subtables, insert the following:

### **Mark-to-Mark attachment positioning Format2: Mark-to-Mark attachment**

The MarkMarkPosrFormat2 subtable is based on Format 1, but uses 24-bit numbers:

#### *MarkMarkPosFormat2 subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 2

Offset24	mark1CoverageOffset	Offset to Combining Mark Coverage table, from beginning of MarkMarkPos subtable
Offset24	mark2CoverageOffset	Offset to Base Mark Coverage table, from beginning of MarkMarkPos subtable
uint16	markClassCount	Number of Combining Mark classes defined
Offset24	mark1ArrayOffset	Offset to MarkArray2 table for Mark1, from the beginning of the MarkMarkPos subtable
Offset24	mark2ArrayOffset	Offset to Mark2Array2 table for Mark2, from the beginning of the MarkMarkPos subtable

The Mark2Array2 table contains one Mark2Record2 for each mark2Coverage table. It stores the records in the same order as the mark2Coverage Index.

#### *Mark2Array2 table*

Type	Name	Description
uint16	mark2Count	Number of Mark2 records
Mark2Record2	mark2Records [mark2Count]	Array of 24-bit Mark2Records, in Coverage order.

The individual Mark2Record entries in Format 2 use a 24-bit offset to avoid potential overflow:

#### *Mark2Record2*

Type	Name	Description
Offset24	mark2AnchorOffsets [markClassCount]	Array of offsets (one per class) to Anchor tables. Offsets are from beginning of Mark2Array2 table, in class order

At the very end of section 6.3.3.7 Lookup Type 7, Contextual positioning subtables, just before 6.3.3.8 LookupType 8: Chaining contextual positioning subtable] please insert the following:

### **Context Positioning Subtable Format 4: Simple Glyph Contexts**

Format 4 is the same as Format 1 but with 24-bit offsetst. As with Format 1, Format 4 defines the context for a glyph positioning operation as a particular sequence of glyphs.

#### *ContextPosFormat4 subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 4
Offset24	coverageOffset	Offset to Coverage table, from beginning of ContextPos subtable
uint16	posRuleSetCount	Number of PosRuleSet2 tables
Offset24	posRuleSetOffsets [posRuleSetCount]	Array of offsets to PosRuleSet2 tables. Offsets are from beginning of ContextPos subtable, ordered by Coverage Index

There is one PosRuleSet2 table for each glyph in the Coverage table. Each PosRuleSet2 table corresponds to a given glyph in the Coverage table, and describes all of the contexts that begin with that glyph.

A PosRuleSet2 table consists of an array of offsets to PosRule2 tables (posRuleOffsets), ordered by preference, and a count of the PosRule2 tables defined in the set (posRuleCount). The PosRuleSet2 table is the same as for Format 1, but refers to PosRule2 subtables.

#### *PosRuleSet2 Table*

Type	Name	Description
uint16	posRuleCount	Number of PosRule2 tables
Offset16	posRuleOffsets [posRuleCount]	Array of offsets to PosRule tables. Offsets are from the beginning of PosRuleSet2, ordered by preference.

The PosRule Table itself is extended to accommodate 24-bit glyph IDs:

#### *PosRule24Table*

Type	Name	Description
uint16	glyphCount	Number of glyphs in the Input glyph sequence
uint16	posCount	Number of PosLookupRecords
uint24	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 5: Using Class-based Glyph Contexts**

Format 5 is the same as Format 2, but uses 24-bit offsets to avoid possible overflow.

### *ContextPosFormat5 Subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 5
Offset24	coverageOffset	Offset to Coverage table, from beginning of ContextPos subtable
Offset24	classDefOffset	Offset to ClassDef table, from beginning of ContextPos subtable
uint24	posClassSetCount	Number of PosClassSet2 tables
Offset24	posClassSetOffsets [posClassSetCount]	Array of offsets to PosClassSet24 tables. Offsets are from the beginning of ContextPos subtable, ordered by class (may be NULL)

### *PosClassSet2 Table*

Type	Name	Description
uint16	posClassRuleCount	Number of PosClassRule tables
Offset24	posClassRuleOffsets[posClassRuleCount]	Array of offsets to PosClassRule tables. Offsets are from beginning of PosClassSet, ordered by preference.

The PosClassRule subtable is unchanged from Format 2:

### *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 6: Coverage-based glyph contexts

Format 6 is the same as Format 3, but using 24-bit numbers.

*ContextPosFormat5 subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 6
uint16	glyphCount	Number of glyphs in the input sequence
uint16	posCount	Number of PosLookupRecords
Offset24	coverageOffsets [glyphCount]	Array of offsets to Coverage tables, from the beginning of ContextPos subtable
PosLookupRecord	posLookupRecords [posCount]	Array of positioning lookups, in design order

At the end of subsection 6.3.3.3.8 Lookup Type 8: Chaining contextual positioning subtable, and just before 6.3.3.3.9 Lookup Type 9, Extension Positioning, insert:

**Chaining context positioning Format 4: Simple glyph contexts**

Format 4 is based on Format 1 but with 24-bit glyph IDs.

*ChainContextPosFormat4 subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 4
Offset24	coverageOffset	Offset to Coverage table, from beginning of ContextPos subtable
uint16	chainPosRuleSetCount	Number of ChainPosRuleSet2 tables
Offset24	chainPosRuleSetOffsets [chainPosRuleSetCount]	Array of offsets to ChainPosRuleSet2 tables. Offsets are from the beginning of the ChainContextPos subtable, ordered by Coverage Index

*ChainPosRuleSet24 table*

Type	Name	Description
uint24	chainPosRuleCount	Number of ChainPosRule2 tables
Offset24	chainPosRuleOffsets [chainPosRuleCount]	Array of offsets to ChainPosRule2 tables. Offsets are from the beginning of the ChainPosRuleSet2, ordered

		by preference
--	--	---------------

*ChainPosRule24 table*

Type	Name	Description
uint16	backtrackGlyphCount	Total number of glyphs in the backtrack sequence
uint24	backtrackSequence [backtrackGlyphCount]	Array of backtracking glyph IDs
uint16	inputGlyphCount	Total number of glyphs in the input sequence - includes the first glyph
uint24	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 5: Class-based glyph contexts

Format 5 is based on Format 2, but uses 24-bit offsets.

*ChainContextPosFormat5 subtable*

Type	Name	Description
uint16	posFormat	Format identifier: format = 5
Offset24	coverageOffset	Offset to Coverage table, from beginning of ChainContextPos subtable
Offset24	backtrackClassDefOffset	Offset to ClassDef table containing backtrack sequence context, from beginning of ChainContextPos subtable
Offset24	inputClassDefOffset	Offset to ClassDef table containing input sequence context, from beginning of ChainContextPos subtable
Offset24	lookaheadClassDefOffset	Offset to ClassDef table containing lookahead sequence context, from beginning of ChainContextPos subtable

uint16	chainPosClassSetCnt	Number of ChainPosClassSet2 tables
Offset24	chainPosClassSetOffsets [chainPosClassSetCnt]	Array of offsets to ChainPosClassSet2 tables. Offsets are from the beginning of the 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.

#### *ChainPosClassSet24 table*

Type	Name	Description
uint16	chainPosClassRuleCount	Number of ChainPosClassRule2 tables
Offset24	chainPosClassRuleOffsets [chainPosClassRuleCount]	Array of offsets to ChainPosClassRule2 tables. Offsets are from beginning of the ChainPosClassSet2 subtable, ordered by preference

The ChainPosClassRule subtable is unchanged from Format 2:

#### *ChainPosClassRule24 table*

Type	Name	Description
uint16	backtrackGlyphCount	Total number of glyphs in the backtrack sequence
uint24	backtrackSequence [backtrackGlyphCount]	Array of backtrack-sequence classes
uint16	inputGlyphCount	Total number of classes in the input sequence - includes the first class
uint24	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
uint24	lookAheadSequence [lookAheadGlyphCount]	Array of lookahead-sequence classes
uint16	posCount	Number of PosLookupRecords
PosLookupRecord	posLookupRecords [posCount]	Array of PosLookupRecords, in design order

## GSUB—The glyph substitution table [6.3.4]

After GSUB Header, Version 1.1 (just before 6.3.4.3 GSUB lookup type descriptions) add:

*GSUB Header, Version 1.2*

Type	Name	Description
uint16	majorVersion	Major version of the GSUB table, = 1
uint16	minorVersion	Minor version of the GSUB table, = 2
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)
Offset32	scriptList2	Offset to ScriptList table for use when offsets larger than 65535 are needed.
Offset32	featureList2	Offset to FeatureList table for use when offsets larger than 65535 are needed.
Offset32	lookupList2	Offset to LookupList table for use when offsets larger than 65535 are needed.

When minorVersion is 2, the offsets whose names end in 2, such as attachListOffset2, are available for use when values greater than 65535 are needed. If such an offset is non-zero, the corresponding 16-bit offset shall be ignored.

After SingleSubstFormat2 subtable, and just before 6.3.3.3.2 LookupType 2, add:

### Single substitution Format 3

Format 3 is based on Format 1. In Format 1, the delta addition process affects only the lowest 16 bits of the glyph identifier. In Format 3, the lower 24-bits are affected.

*SingleSubstFormat3 subtable:*

Type	Name	Description
uint16	substFormat	Format identifier: format = 3



Offset24	coverageOffset	Offset to Coverage table, from beginning of substitution subtable
int24	deltaGlyphID	Add to original glyph ID to get substitute glyph ID

### Single substitution Format 4

Format 4 is based on Format 2. In Format 2, the delta addition process affects only the lowest 16 bits of the glyph identifier. In Format 4, the lower 24-bits are affected.

*SingleSubstFormat4 subtable: Specified output glyph indices*

Type	Name	Description
uint16	substFormat	Format identifier: format = 4
Offset24	coverageOffset	Offset to Coverage table, from beginning of Substitution table
uint24	glyphCount	Number of glyph IDs in the substituteGlyphIDs array
uint24	substituteGlyphIDs [glyphCount]	Array of substitute glyph IDs – ordered by Coverage index

At the end of subsection 6.3.4.3.2 LookupType 2: Multiple substitution subtable, just before 6.3.4.3.3 LookupType 3, add:

### Multiple Substitution Format2: Multiple output glyphs

The Multiple Substitution Format2 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.

Format 2 is identical to Format 1, except that 24-bit glyph IDs are supported.

*MultipleSubstFormat2 subtable:*

Type	Name	Description
uint16	substFormat	Format identifier: format = 2
Offset24	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint24	sequenceCount	Number of 24-bit Sequence2 table offsets in the sequenceOffsets array
Offset24	sequenceOffsets	Array of offsets to 24-bit Sequence2 tables. Offsets are from

	[sequenceCount]	beginning of substitution subtable, ordered by Coverage index
--	-----------------	---

### *Sequence2 table*

Type	Name	Description
uint16	glyphCount	Number of glyph IDs in the substituteGlyphIDs array. This shall always be greater than 0.
uint24	substituteGlyphIDs [glyphCount]	String of glyph IDs to substitute

In section 6.3.4.3.3 LookupType 3 just before the end (LookupType 4: Ligature substitution subtable is next), insert after the AlternateSet table:

### **Alternate Substitution Format2: Alternative output glyphs**

The Alternate Substitution Format2 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 (alternateSetOffsets). It is identical to the Format1 subtable, except for using 24-bit numbers.

### *AlternateSubstFormat2 subtable:*

Type	Name	Description
uint16	substFormat	Format identifier: format = 2
Offset24	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint24	alternateSetCount	Number of AlternateSet2 tables
Offset24	alternateSetOffsets [alternateSetCount]	Array of offsets to AlternateSet2 tables. Offsets are from beginning of substitution table, ordered by Coverage index

### *AlternateSet24 table*

Type	Name	Description
uint16	glyphCount	Number of glyph IDs in the alternateGlyphIDs array
uint24	alternateGlyphIDs [glyphCount]	Array of alternate glyph IDs, in arbitrary order

At the end of subsection 6.3.4.3.4 LookupType 4: Ligature substitution subtable, just before section 6.3.4.3.5 Substitution Lookup Record, add:

### **Ligature Substitution Format2: All ligature substitutions in a script**

The Ligature Substitution Format2 subtable 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. The Format2 version is identical to the Format1 version except for supporting 24-bit instead of 16-bit numbers.

*LigatureSubstFormat2 subtable:*

Type	Name	Description
uint16	substFormat	Format identifier: format = 2
Offset24	coverageOffset	Offset to Coverage table, from beginning of Substitution table
uint24	ligatureSetCount	Number of LigatureSet2 tables
Offset24	ligatureSetOffsets [ligatureSetCount]	Array of offsets to LigatureSet2 tables. Offsets are from the beginning of the substitution subtable, ordered by Coverage index

A LigatureSet2 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 LigatureSet2 table will define the "etc" ligature.

A LigatureSet2 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.

*LigatureSet2 table: All ligatures beginning with the same glyph*

Type	Name	Description
uint16	ligatureCount	Number of Ligature tables
Offset24	ligatureSetOffsets [ligatureCount]	Array of offsets to 24-bit Ligature tables. Offsets are from the beginning of the LigatureSet2 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 sequence 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 (24-bit): Glyph components for one ligature*

Type	Name	Description
uint24	ligatureGlyph	Glyph ID of ligature to substitute
uint16	componentCount	Number of components in the ligature
uint24	componentGlyphIDs [componentCount - 1]	Array of component glyph IDs – start with the second component, ordered in writing direction

At the end of 6.3.4.3.6, just before LookupType 6: Chaining contextual substitution subtable, insert the following two subsections:

#### **Context substitution Format 4: Simple Glyph Contexts**

Format 4 defines the context for a glyph substitution as a particular sequence of glyphs. Format 4 is based on Format 1, but uses 24-bit instead of 16-bit integers.

*ContextSubstFormat4 subtable:*

Type	Name	Description
uint16	substFormat	Format identifier: format = 4
Offset24	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint24	subRuleSetCount	Number of SubRuleSet2 tables – must equal glyphCount in Coverage table
Offset24	subRuleSetOffsets [subRuleSetCount]	Array of offsets to SubRuleSet2 tables. Offsets are from beginning of Substitution table, ordered by Coverage index

*SubRuleSet2 table: All contexts beginning with the same glyph*

Type	Name	Description
uint24	subRuleCount	Number of SubRule tables
Offset24	subRuleOffsets	Array of offsets to SubRule tables. Offsets are from

	[subRuleCount]	beginning of SubRuleSet table, ordered by preference
--	----------------	--

*SubRule2 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
uint24	inputSequence [glyphCount - 1]	Array of input glyph IDs – start with second glyph
SubstLookupRecord	substLookupRecords [substitutionCount]	Array of SubstLookupRecords, in design order

### Context substitution Format 5: Class-based Glyph Contexts

Format 5 is based on Context substitution format 2, which is a more flexible format than Format 1. In Format 5, 24-bit numbers are used instead of 16-bit numbers, for glyph IDs and offsets, where needed.

*ContextSubstFormat5 subtable:*

Type	Name	Description
uint16	substFormat	Format identifier: format = 5
Offset24	coverageOffset	Offset to Coverage table, from beginning of substitution table
Offset24	classDefOffset	Offset to glyph ClassDef table, from beginning of substitution table
uint24	subClassSetCount	Number of SubClassSet2 tables
Offset24	subClassSetOffsets [subClassSetCount]	Array of offsets to SubClassSet2 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 SubClassSet2 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.

*SubClassSet2 subtable*

Type	Name	Description
uint16	subClassRuleCount	Number of <b>SubClassRule2</b> tables
<b>Offset24</b>	subClassRuleOffsets [subClassRuleCount]	Array of offsets to <b>SubClassRule2</b> tables. Offsets are from beginning of <b>SubClassSet2</b> , ordered by preference

*SubClassRule2 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
<b>uint24</b>	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

At the end of 6.3.4.3.7 LookupType 6: Chaining contextual substitution subtable, just before 6.3.4.3.8 LookupType 7: Extension substitution, insert the following two subsections:

**Chaining context substitution Format 4: Simple chaining context glyph substitutions**

The Format 4 chaining context substitution subtable is based on Format 1, but uses 24-bit numbers for glyphIDs and to avoid overflow.

*ChainContextSubstFormat4 subtable:*

Type	Name	Description
uint16	substFormat	Format identifier: format = 4
<b>Offset24</b>	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint16	chainSubRuleSetCount	Number of <b>ChainSubRuleSet2</b> tables – must equal glyphCount in Coverage table

<b>Offset24</b>	chainSubRuleSetOffsets [chainSubRuleSetCount]	Array of offsets to <b>ChainSubRuleSet2</b> tables. Offsets are from beginning of substitution table, ordered by Coverage index
-----------------	--	---

A **ChainSubRuleSet2** table consists of an array of offsets to **ChainSubRule2** tables (chainSubRuleOffsets), ordered by preference, and a count of the **ChainSubRule2** tables defined in the set (chainSubRuleCount).

In Format 4, suitable for use when the GLYPF table is used, the offsets are 24-bit:

*ChainSubRuleSet2 table: All contexts beginning with the same glyph*

Type	Name	Description
uint16	chainSubRuleCount	Number of <b>ChainSubRule2</b> tables
<b>Offset24</b>	chainSubRuleOffsets [chainSubRuleCount]	Array of offsets to <b>ChainSubRule2</b> tables. Offsets are from beginning of <b>ChainSubRuleSet2</b> table, ordered by preference

*ChainSubRule2 table: One simple context definition*

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)
<b>uint24</b>	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.
<b>uint24</b>	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)
<b>uint24</b>	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 5: Class-based Glyph Contexts

Format 5, a more flexible format than Format 4, describes class-based context substitution. It is identical to Format 2, except for supporting 24-bit offsets.

*ChainContextSubstFormat5 subtable:*

Type	Name	Description
uint16	substFormat	Format identifier: format = 5
Offset24	coverageOffset	Offset to Coverage table, from beginning of substitution table
Offset24	backtrackClassDef	Offset to glyph ClassDef table containing backtrack sequence data, from beginning of substitution table
Offset24	inputClassDef	Offset to glyph ClassDef table containing input sequence data, from beginning of substitution table
Offset24	lookaheadClassDef	Offset to glyph ClassDef table containing lookahead sequence data, from beginning of substitution table
uint16	chainSubClassSetCount	Number of ChainSubClassSet2 tables
Offset24	chainSubClassSetOffsets [chainSubClassSetCount]	Array of offsets to ChainSubClassSet2 tables. Offsets are from beginning of substitution table, ordered by input class (may be NULL)

*ChainSubClassSet2 subtable*

Type	Name	Description
uint16	chainSubClassRuleCount	Number of ChainSubClassRule2 tables
Offset24	chainSubClassRuleOffsets [chainSubClassRuleCount]	Array of offsets to ChainSubClassRule2 tables. Offsets are from beginning of ChainSubClassSet2, ordered by preference

*ChainSubClassRule2 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).
uint24	backtrackSequence	Array of backtracking classes – to be



	[BacktrackGlyphCount]	matched before the input sequence.
uint16	inputGlyphCount	Total number of classes in the input sequence – includes the first class.
uint24	inputSequence [inputGlyphCount - 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).
uint24	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.

NOTE There is no Format 6, which might be a version of Format 3 but with 24-bit numbers. This is because a Format 3 subtable encodes only a single rule, and is therefore not likely to overflow.

Just before 6.3.4.4 GSUB subtable examples, add,

### Reverse chaining contextual single substitution Format 2: Coverage-based contexts

Format 2 is the same as Format 1, but supports 24-bit glyph Ids and offsets.

*ReverseChainSingleSubstFormat2 subtable:*

Type	Name	Description
uint16	substFormat	Format identifier; format = 2
Offset24	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint16	backtrackGlyphCount	Number of glyphs in the backtracking sequence.
Offset24	backtrackCoverageOffsets [backtrackGlyphCount]	Array of offsets to coverage tables in backtracking sequence, in glyph sequence order.
uint16	lookaheadGlyphCount	Number of glyphs in lookahead sequence.
Offset24	lookaheadCoverageOffsets [lookaheadGlyphCount]	Array of offsets to coverage tables in lookahead sequence, in glyph sequence

		order.
uint24	glyphCount	Number of glyph IDs in the substituteGlyphIDs array.
uint24	substituteGlyphIDs[glyphCount]	Array of substitute glyph IDs – ordered by Coverage index.