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 66260

Milford, Ontario, Canada – January 2024

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

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

Contents

Table of Contents

Introduction

This PDF document was produced on 14th January 2024.

Changes since the 9th January meeting:

Added *BaseCoordFormat4* to support pointing to GLYF for a reference glyph for determining a baseline (mostly for the sake of completeness)

Removed DMAP

Minor offset32 and typo fixes

Type definitions moved to coorrect places and integrated

Added notes that hhea/vhea HHEA/CHEA use the same variation tags

Added note about CFF2 to sbix

Added new fixed point formats (6.10, 4.12) to type table

Changes since the October 2023 meeting:

Changes and new sections are marked in turquoise.

- **Variable composites:** We propose a new mechanism to express the way parts of glyphs interact with one another and are transformed as variable-font axis values are changed. The method proposed, using a VARC table, is applicable to glyphs in CFF2 as well as glyf or GLYF tables.

  The mechanism includes new multi-byte compressed numbers. Preliminary measurements suggest that the VARC approach achieves substantial savings in file size.

  Variable Composites were thus also removed from GLYF.
- A new 24-bit MAXP table is introduced; fonts with both maxp and MAXP are “hybrid”
- The CUBIC flag is allowed on both on-curve and off-curve points
- An optional DMAP table was introduced for more efficient sharing in Collections
- COLR without CPAL now implies a monochrome font that can use COLR
- PaintGlyph2 supports 24-bit glyph IDs; Format 33 was added for this.
- New coverage tables are included, with formats 3 and 4.
- Offset sizes of top-level tables were changed to 32-bit
- New minor versions of the TTC header (signed and unsigned) are introduced

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 GLYF 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 'GLYF' 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 GLYF 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 GLYF 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 15 is added, with 24-bit glyph IDs
DMAP	New	Overrides cmap, e.g. in a collection

GDEF	changed	Updated to avoid overflow, and for 24-bit Glyph IDs
GLYF	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
LOCA	new	Augmented version of loca
maxp	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.
MAXP	New	Version of MAXP to refer to GLYF instead of glyf.
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. However, see below for a new MAXP table.

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. *Note*: The separate GLYF table does not use this number; the number of entries in the GLYF table is determined by the size in bytes of the LOCA table , taking into account the value of indexToLocFormat, or from the 'MAXP' table.
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.

MAXP—Maximum Profile, for 24-bit fonts

This table mirrors the ‘maxp’ table, but for fonts in which GlyphID values are represented as 24-bit integers; the older ‘maxp’ table assumes 16-bit values. Fonts containing both ‘MAXP’ and ‘maxp’ tables are said to be *hybrid*. They will work as 16-bit fonts in older software, and newer software will use the newer tables, typically ‘GLYF’ instead of, or alongside, ‘glyf’. However, older software will not of course be able to access more than 65535 glyphs in a font, will not be able to use the newer features allowed in glyph definitions found in the ‘GLYF’ table (such as cubic curves), and likely will not be able to access new features such as AVAR 2, so the user experience may be limited, as is also the case with software that does not support features such as swash (SWSH) or small caps (SMCP).

As with ‘maxp’ the values in the table are intended to establish memory requirements for the font. However, implementers are warned that the values may be inaccurate or maliciously incorrect.

Fonts with CFF or CFF2 outlines shall use Version 0.5 of this table, specifying only the numGlyphs field. Fonts with TrueType outlines must use Version 1.0 of this table, where all data is required.

Software supporting MAXP shall ignore maxp where both tables are present.

Version 0.5

Type	Name	Description
Version16Dot16	version	0x00005000 for version 0.5
uint24	numGlyphs	The number of glyphs in the font.

Version 1.0

Type	Name	Description
Version16Dot16	version	0x00010000 for version 1.0.
uint24	numGlyphs	The number of glyphs in the font. This is the number of glyphs in the 'GLYF' table, if any. *Note*: The number of entries in the GLYF table is determined by the size in bytes of the LOCA table, taking into account the value of indexToLocFormat. The value here may be useful for initial memory allocation, but cannot be relied upon.
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, or just before 5.3 to preserve section numbering]

[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 [35].

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 [35]. 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 65335, 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 'GLYF' table does not include any overall table header or records providing offsets to glyph data blocks. Rather, the 'LOCA' table provides an array of offsets, indexed by glyph IDs, which provide the location of each glyph data block within the 'GLYF' table.

The size of each glyph data block is inferred from the difference between two consecutive offsets in the 'LOCA' table (with one extra offset provided to give the size of the last glyph data block). As a result of the 'LOCA' format, glyph data blocks within the 'GLYF' table must be in glyph ID order.

Each glyph description uses one of two 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.

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 shall be used for composite glyphs.
int16	xMin	Minimum x for coordinate data.
int16	yMin	Minimum y for coordinate data.
int16	xMax	Maximum x for coordinate data.
int16	yMax	Maximum y for coordinate data.

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 'GLYF' table. For vertical layout, top side bearings are provided in the 'vmtx' table, and bottom side bearings are inferred. The rasterizer will generate a representation of side bearings in the form of “phantom” points, which are added as four additional points at the end of the glyph description and which can be referenced and manipulated by glyph instructions. See 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[*variable*]	Contour point y-coordinates. See below for details regarding the number of coordinate array elements. Coordinate for the first point is relative to (0,0); others are relative to previous point.

NOTE 1 In the 'glyf' table, the position of a point is not stored in absolute terms but as a vector relative to the previous point. The delta-x and delta-y vectors represent these (often small) changes in position. Coordinate values are in font design units, as defined by the unitsPerEm field in the 'head' table. Note that smaller unitsPerEm values will make it more likely that delta-x and delta-y values can fit in a smaller representation (8-bit rather than 16-bit), though with a trade-off in the level or precision that can be used for describing an outline.

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

In logical terms, there is one flag byte element, one x-coordinate, and one y-coordinate for each point. 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 [7] 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 'GLYF' table, not the 'glyf' table. In the 'glyf' 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 is allowed on both on-curve and off-curve points. This is because some hinting instructions can flip points between on-curve and off-curve. This specification does not define a meaning for the CUBIC flag on an on-curve point, but it is not an error.
- The number of consecutive off-curve points (with wraparound) shall be even, both before and after running the hinting program.

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, depending on the GID_IS_24_BIT flag
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 arg1and2; /* (arg1 << 8) | arg2 */
   }
   if ( flags & WE_HAVE_A_SCALE ) {
         F2DOT14  scale;     /* Format 2.14 */
   } else if ( flags & WE_HAVE_AN_X_AND_Y_SCALE ) {
         F2DOT14  xscale;    /* Format 2.14 */
         F2DOT14  yscale;    /* Format 2.14 */
   } else if ( flags & WE_HAVE_A_TWO_BY_TWO ) {
         F2DOT14  xscale;    /* Format 2.14 */
         F2DOT14  scale01;   /* Format 2.14 */
         F2DOT14  scale10;   /* Format 2.14 */
         F2DOT14  yscale;    /* Format 2.14 */
   }
} while ( flags & MORE_COMPONENTS )
if (flags & WE_HAVE_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 [7] 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) = (argument1, 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 incoporated 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.

Comparing CFF2 with CFF and glyf [5.3.3.13]

The tables intended for storing monochrome glyph outlines in OFF fonts are the 'GLYF' 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 'GLYF' table supports a mixture of quadratic and cubic Bézier curves.

CFF2 and CFF also use a different conceptual model for "hints" than the 'GLYF' 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', 'GLYF' 'glyf' tables. Note that some of these differences might not be exposed in high-level font editing software or in runtime programming interfaces.

Consideration	**GLYF**	**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 'GLYF' 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 ‘GLYF’ 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 ‘GLYF’.

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	offsets[*numGlyphs + 1*]	The local offset divided by 2 is stored.

Long format

Type	Name	Description
Offset32	offsets[*numGlyphs + 1*]	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:

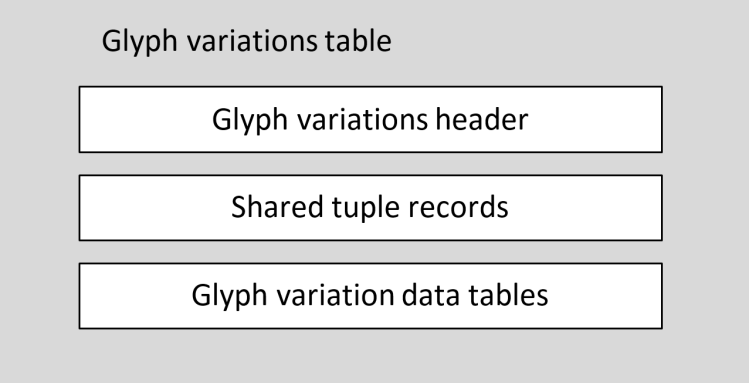


Figure 7.8: High-level organization of 'gvar' table

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:

‘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 GLYF 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.

NOTE The only difference between GVAR and gvar is that glyphCount is a uint32 in GVAR, not a uint16.

[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 15 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 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 15 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.

An optional 'DMAP' table, if present, shall take priority over 'cmap'—that is, the character-to-glyph lookup shall first look in the 'DMAP' table (including any variation-selectors). If a match is not found, 'cmap' is to be consulted. The 'DMAP' table is identical in structure to the 'cmap' table.

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 15: 24-bit CMAP table

Subtable format 15 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 'GLYF' table.

COLR—Color Table [5.6.11]

We introduce a new subtable format, 33 (PaintGlyph2) with 24-bit glyph identifiers. These are each given their own section, included here below. They are also mentioned in various other sections, as

follows.

We do *not* introduce a new PaintColrGlyph2, because BaseGlyphPaintRecord is limited to 16-bit glyph identifiers, and changing that implies a COLRv2, which would be a separate proposal. For now, then, fonts are limited to 65535 colour glyphs, although these can use a total of more than 65535 components (24-bits).

At the end of the introduction, just before 5.6.11.1 Graphic Compositions, change the last paragraph from:

The COLR table is used in combination with the CPAL table (5.7.12): all color values are specified as entries in color palettes defined in the CPAL table. If the COLR table is present in a font but no CPAL table exists, then the COLR table is ignored.

to

The COLR table may be used in combination with the CPAL table (5.7.12): all color values are specified as entries in color palettes defined in the CPAL table. If the COLR table is present in a font but no CPAL table exists, then the font is monochrome and no additional colors beyond the application foreground (0xFFFF) are available..

In 6.11.1[5.5.11.1] **Graphic compositions**, change the bullet list and the rest of the subsection to:

- A PaintColrLayers table provides a layering structure used for creating a color glyph from layered elements. A PaintColrLayers table can be used at the root of the graph, providing a base layering structure for the entire color glyph definition. A PaintColrLayers table can also be nested within the graph, providing a set of layers to define some graphic sub-component within the color glyph.

- The PaintSolid, PaintVarSolid, PaintLinearGradient, PaintVarLinearGradient, PaintRadialGradient, PaintVarRadialGradient, PaintSweepGradient, and PaintVarSweepGradient tables provide basic fills, using color entries from the CPAL table.

- The PaintGlyph and PaintGlyph2 tables provide glyph outlines as the basic shapes.

- The PaintTransform and PaintVarTransform tables are used to apply an affine transformation matrix to a sub-graph of paint tables, and the graphic operations they represent. Several Paint formats are also provided for specific transformation types: translate, scale, rotate, or skew, with additional variants of these formats for variations and other options.

- The PaintComposite table supports alternate compositing and blending modes for two sub-graphs.

- The PaintColrGlyph table allows a color glyph definition, referenced by a base glyph ID, to be re-used as a sub-graph within multiple color glyphs.

NOTE Some paint formats come in *Paint** and *PaintVar** pairs. In these cases, the latter format supports variations in variable fonts, while the former provides a more compact representation for the same graphic capability but without variation capability.

NOTE The PaintGlyph2 table supports 24-bit glyph identifiers; PaintGlyph is limited to 16-bit values. In this description, for readability, the term PaintGlyph is used to mean either a 16-bit or a 24-bit table. The definitions of the PaintGlyph and PaintGlyph2 tables use the terms precisely.

In a simple color glyph description, a PaintGlyph table might be linked to a PaintSolid table, for example, representing a glyph outline filled using a basic solid color fill. But the PaintGlyph table could instead be linked to a much more complex sub-graph of Paint tables, representing a shape that gets filled using the more-complex set of operations described by the sub-graph of Paint tables.

The graphic capabilities are described in more detail in 5.7.11.1.1 – 5.7.11.1.9. The formats used for each are specified in 5.7.11.2.

In section 5.6.11.1.3[5.5.11.1.3] **Filling Shapes,** we change only the first paragraph. The "of course" serves as a reminder that the new PaintGlyph2 table can be used wherever the examples use PaintGlyph:

All basic shapes used in a color glyph are obtained from glyph outlines, referenced using a glyph ID. In a color glyph description, a PaintGlyph table (or PaintGlyph2 table, of course) is used to represent a basic shape.

In 5.6.11.1.7.2 [5.5.11.1.7.2] **Re-use by referencing shared subtables**, change the first bullet point from

- PaintGlyph

to

- PaintGlyph, PaintGlyph2

Replace 5.5.11.1.8.2 **Metrics and boudedness of color glyphs using version 1 formats** with the following text:

For color glyphs using version 1 formats, the advance width of the base glyph shall be used as the advance width for the color glyph. If the font has vertical metrics, the advance height and vertical Y origin of the base glyph shall be used for the color glyph. The advance width and height of glyphs referenced by PaintGlyph and PaintGlyph2 tables are not required to be the same as that of the base glyph and are ignored.

A valid color glyph definition shall define a bounded region—that is, it shall paint within a region for which a finite bounding box could be defined. A clip box can be specified to set overall bounds for a color glyph (see below). Otherwise, boundedness is determined by the graph of paint tables that describe the color glyph content. The different paint formats have different boundedness characteristics:

- PaintGlyph and PaintGlyph2 are inherently bounded.

- PaintSolid, PaintVarSolid, PaintLinearGradient, PaintVarLinearGradient, PaintRadialGradient, PaintVarRadialGradient, PaintSweepGradient, and PaintVarSweepGradient are inherently unbounded.

- PaintColrLayers is bounded *if and only if* all referenced sub-graphs are bounded.

- PaintColrGlyph is bounded *if and only if* the color glyph definition for the referenced base glyph ID is bounded.

- Paint formats for transformations (PaintTransform, PaintVarTransform, PaintTranslate, PaintScale, etc.) are bounded *if and only if* the referenced sub-graph is bounded.

- PaintComposite is either bounded or unbounded, according to the composite mode used and the boundedness of the referenced sub-graphs. See 5.7.11.2.6.13 for details.

A ClipBox table (5.7.11.2.4) may be associated with a color glyph to define overall bounds for the color glyph. The clip box may vary in a variable font. If a clip box is provided for a color glyph, the color glyph is bounded, and no inspection of the Paint graph is required to determine boundedness. If no clip box is defined for a color glyph, however, applications shall confirm that the color glyph definition is bounded, and shall not render the color glyph if the defining graph is not bounded.

NOTE 1 If present, the clip box for a color glyph can be used to allocate a drawing surface without needing to traverse the graph of the color glyph definition.

NOTE 2 If no ClipBox table is present but a bounding box is required by the implementation, it can be computed for a given color glyph by traversing the graph of Paint tables that defines that color glyph.

To ensure that rendering implementations do not clip any part of a color glyph, the clip box needs to be large enough to encompass the entire color glyph composition. In a variable font, glyph outlines can vary, but transformations in a color glyph description can also vary, affecting the portions of the design grid to be painted. For example, a filled rectangle that is wide but not tall for one variation instance can be variably rotated to be tall but not wide for other instances. The clip box either should be large enough to encompass the color glyph for all instances, or should itself vary such that each instance of the clip box encompasses the instance color glyph.

In 5.6.11.2.6.6 / 5.5.11.2.6.6, **Format 10: PaintGlyph**, after the "For information about applying a fill to a shape", add a new note:

NOTE For use with 24-bit glyphID values, see Format 33: PaintGlyph2.

At the end of COLR, just before 5.6.11.3 COLR version 1 rendering algorithm, and just after the bullet list, add the definition for PaintGlyph2:

Format 33: PaintGlyph2 [5.6.11.2.6.14]

Format 33 is used to specify a glyph outline to use as a shape to be filled or, equivalently, a clip region. The outline sets a clip region that constrains the content of a separate paint subtable and the sub-graph linked from that subtable. The difference between Format 10 and Format 33 is that Format 33 uses 24-

bit glyphID values instead of 16-bit values.

PaintGlyph2 table (format 33):

Type	Name	Description
uint8	format	Set to 33.
Offset24	paintOffset	Offset to a Paint table.
uint24	glyphID	Glyph ID for the source outline.

The glyphID value shall be a valid glyph with outline data in the 'GLYF', 'glyf' (5.3.4), 'CFF' (5.4.2) or CFF2 (5.4.3) table. Only that outline data is used. In particular, if this glyph ID has a description in the COLR table (glyphID appears in a COLR BaseGlyph record or the BaseGlyphList), that COLR data is not relevant for purposes of the PaintGlyph2 table.

The glyphID value shall be a glyphID found in a BaseGlyphPaintRecord within the BaseGlyphList. The BaseGlyphPaintRecord provides an offset to a paint table; that paint table and the graph linked from it are incorporated as a child sub-graph of the PaintColrGlyph table within the current color glyph definition.

[5.6.11.3] COLR version 1 rendering algorithm

In the sample code, add PaintGlyph2 in one place:

```
if format 10, 33: // PaintGlyph, PaintGlyph2
```

DMAP—Delta map table [removed from this proposal]

[deleted from this proposal]

BASE table structure

In 5.3.1 Base Table Structure, in the paragraph just before BaseCoord Format 1,

Change from:

Three formats available for BaseCoord table data define single X or Y coordinate values in design units. Two of the formats also support fine adjustments to the X or Y values based on a contour point or a Device table. In a variable font, the third format uses a VariationIndex table (a variant of a Device table), as needed, to reference variation data for adjustment of the X or Y values for the current variation instance.

To:

Four formats available for BaseCoord table data define single X or Y coordinate values in design units. The first two formats also support fine adjustments to the X or Y values based on a contour point or a

Device table. In a variable font, the third format uses a VariationIndex table (a variant of a Device table), as needed, to reference variation data for adjustment of the X or Y values for the current variation instance. The fourth format is the same as the second, supplying a glyph index and a contour point, but uses a 24-bit glyph-ID.

Just before 6.3.1.5 BASE table examples, add

BaseCoordFormat4 table: Design units plus contour point (24-bit glyph ID)

Type	Name	Description
uint16	baseCoordFormat	Format identifier – format = 2
int16	coordinate	X or Y value, in design units
uint24	referenceGlyph	Glyph ID of control glyph
uint16	baseCoordPoint	Index of contour point on the reference glyph

The fourth BaseCoord format (BaseCoordFormat4) is the same as BaseCoordFormat2, but uses a 24-bit glyph ID, for example for use with the GLYF table rather than the 16-bit glyf table.

HHEA—Horizontal header [5.1.4]

This table is intended for use with the 'GLYF' 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(aw - (lsb + xMax - 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'

NOTE: The ‘HHEA’ and ‘hhea’ tables use the same variation tags

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

If the 'GLYF' table is present, any 'glyf' table shall be ignored, along with 'hmtx', and HTMLX shall be used. If the 'GLYF' table is not present , the 'HMTX' table shall be ignored, and 'htmx' shall be used.

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

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

In a font with CFF version 1 outline data, the 'CFF ' table does include advance widths. These values are used by PostScript processors, but are not used in OFF layout. In an OFF context, 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 GLYF. 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 'GLYF' 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 'GLYF' table. The advance width ("aw") and left side bearing ("lsb") can be derived from the glyph "phantom points", which are computed by the TrueType rasterizer; or they can be obtained from the 'HMTX' table.

In a font with CFF or CFF2 outlines, xMin (= left side bearing) and xMax values can be obtained from the CFF / CFF2 rasterizer. From those values, the right side bearing ("rsb") is calculated as follows:

rsb = aw - (lsb + xMax – xMin)

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

pp1 = xMin - lsb
pp2 = pp1 + aw

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).

@@above paragraph to be used in introduction of a joint section for both tables.

Note: The 'VHEA' and 'VMTX' tables, like the 'HHEA' and 'HMTX' counterparts, are used with the 'GLYF' table. If a 'GLYF' 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

Version 1.0 Type	Name	Description
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(tsb + (yMax-yMin))
int16	caretSlopeRise	The value of the caretSlopeRise field divided by the value of the caretSlopeRun Field determines the slope of the caret. A value of 0 for the rise and a value of 1 for the run specifies a horizontal caret. A value of 1 for the rise and a value of 0 for the run specifies a vertical caret. Intermediate values are desirable for fonts whose glyphs are

		oblique or italic. For a vertical font, a horizontal caret is best.
int16	caretSlopeRun	See the caretSlopeRise field. Value=1 for nonslanted vertical fonts.
int16	caretOffset	The amount by which the highlight on a slanted glyph needs to be shifted away from the glyph in order to produce the best appearance. Set value equal to 0 for nonslanted fonts.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	metricDataFormat	Set to 0.
uint32	numOf LongVerMetrics	Number of advance heights in the vertical metrics table.

Version 1.1 of the vertical header table format is the same except for numOfLongVerMetrics using 32-bits:

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	minTop SideBearing	The minimum top sidebearing measurement found in the font, in font design units. This value must be consistent with the entries in the vertical metrics table.
int16	minBottom SideBearing	The minimum bottom sidebearing measurement found in the font, in font design units. This value must be consistent with the entries in the vertical metrics table.
int16	yMaxExtent	Defined as max(tsb + (yMax-yMin))
int16	caretSlopeRise	The value of the caretSlopeRise field divided by the value of the caretSlopeRun Field determines the slope of the caret. A value of 0 for the rise and a value of 1 for the run specifies a horizontal caret. A value of 1 for the rise and a value of 0 for the run specifies a vertical caret. Intermediate values are desirable for fonts whose glyphs are oblique or italic. For a vertical font, a horizontal caret is best.
int16	caretSlopeRun	See the caretSlopeRise field. Value=1 for nonslanted vertical fonts.
int16	caretOffset	The amount by which the highlight on a slanted glyph needs to be shifted away from the glyph in order to produce the best appearance. Set value equal to 0 for nonslanted fonts.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	reserved	Set to 0.
int16	metricDataFormat	Set to 0.
uint32	numOf	Number of advance heights in the vertical

	LongVerMetrics	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.

NOTE: As with the 'HHEA' and 'hhea' tables, the 'VHEA' and 'vhea' tables use the same variation tags

Vertical Header Table Example

Offset/ length	Value	Name	Comment
0/4	0x00011000	version	Version number of the vertical header table, in fixed-point format, is 1.1
4/2	1024	vertTypoAscender	Half the em-square height.
6/2	-1024	vertTypoDescender	Minus half the em-square height.
8/2	0	vertTypoLineGap	Typographic line gap is 0 font design units.
10/2	2079	advanceHeightMax	The maximum advance height measurement found in the font is 2079 font design units.
12/2	-342	minTopSideBearing	The minimum top sidebearing measurement found in the font is -342 font design units.
14/2	-333	minBottomSideBearing	The minimum bottom sidebearing measurement found in the font is -333 font design units.
16/2	2036	yMaxExtent	max (tsb + (yMax - yMin)) = 2036.

18/2	0	caretSlopeRise	The caret slope rise of 0 and a caret slope run of 1 indicate a horizontal caret for a vertical font.
20/2	1	caretSlopeRun	The caret slope rise of 0 and a caret slope run of 1 indicate a horizontal caret for a vertical font.
22/2	0	caretOffset	Value set to 0 for nonslanted fonts.
24/4	0	reserved	Set to 0.
26/2	0	reserved	Set to 0.
28/2	0	reserved	Set to 0.
30/2	0	reserved	Set to 0.
32/2	0	metricDataFormat	Set to 0.
34/3	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.

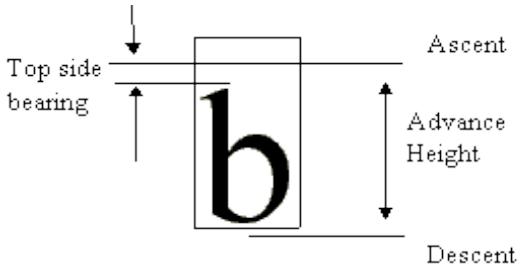


Figure 5.11 – Vertical Metrics

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').

Fonts using the upper-case named 'GLYF' 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 ('vkrn') could modify the vertical advances in a manner similar to kerning in horizontal mode.

Vertical Metrics Table Format

The overall structure of the vertical metrics table consists of two arrays shown below: the vMetrics array followed by an array of top side bearings. The top side bearing is measured relative to the top of the origin of glyphs, for vertical composition of ideographic glyphs.

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

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

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

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

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

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

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

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

sidebearing array is given below.

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

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 otherise.
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', ‘CFF2’ 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.

NOTE: The number of glyphs referenced by the table is the number of glyphs in the font. This also applies in practice to non-OpenFontFormat tables ‘kerx’ and ‘morx’.

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 ‘GLYF’ 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.

ClassDef and Coverage

We add formats 3 and 4 to Coverage [6.2.6] and to ClassDev [6.2.7] to support 24-bit glyphs.

Coverage table [6.2.6]

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

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

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

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

A Coverage table defines a unique index value (Coverage Index) for each covered glyph. The Coverage Indexes are sequential, from 0 to the number of covered glyphs minus 1. This unique value specifies the position of the covered glyph in the Coverage table. The client uses the Coverage Index to look up values in the subtable for each glyph.

Coverage Formats 1 and 2 are defined for 16-bit glyph identifiers, and Formats 3 and 4 are defined for 24-bit glyph identifiers.

Coverage Format 1

Coverage Format 1 consists of a format code (CoverageFormat) and a count of covered glyphs (GlyphCount), followed by an array of glyph indices (GlyphArray). The glyph indices must be in numerical order for binary searching of the list. When a glyph is found in the Coverage table, its position in the GlyphArray determines the Coverage Index that is returned-the first glyph has a Coverage Index = 0, and the last glyph has a Coverage Index = GlyphCount -1.

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

Coverage Format1 table: Individual glyph indices

Type	Name	Description
uint16	coverageFormat	Format identifier – format = 1
uint16	glyphCount	Number of glyphs in the glyph array
uint16	glyphArray[glyphCount]	Array of glyph IDs – in numerical order

Coverage Format 2

Format 2 consists of a format code (coverageFormat) and a count of glyph index ranges (rangeCount), followed by an array of records (rangeRecords). Each RangeRecord consists of a start glyph index (startGlyphID), an end glyph index (endGlyphID), and the Coverage Index associated with the range's Start glyph. Ranges shall be in glyph ID order, and they must be distinct, with no overlapping.

The Coverage Indexes for the first range begin with zero (0), and increase sequentially to (endGlyphId - startGlyphId). For each successive range, the starting Coverage Index is one greater than the ending Coverage Index of the preceding range. Thus, startCoverageIndex for each non-initial range must equal the length of the preceding range (endGlyphID - startGlyphID + 1) added to the startGlyphIndex of the preceding range. This allows for a quick calculation of the Coverage Index for any glyph in any range using the formula: Coverage Index (glyphID) = startCoverageIndex + glyphID - startGlyphID.

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

CoverageFormat2 table: Range of glyphs

Type	Name	Description
uint16	coverageFormat	Format identifier – format = 2
uint16	rangeCount	Number of RangeRecords
RangeRecord	rangeRecords [rangeCount]	Array of glyph ranges – ordered by startGlyphID

RangeRecord

Type	Name	Description
uint16	startGlyphID	First glyph ID in the range

uint16	endGlyphID	Last glyph ID in the range
uint16	startCoverageIndex	Coverage Index of first glyph ID in range

Coverage Format 3

Coverage Format 3 is the same as Format 1, except that it uses 24-bit GlyphID values.

CoverageFormat3 table: Individual glyph indices

Type	Name	Description
uint16	coverageFormat	Format identifier – format = 3
uint16	glyphCount	Number of glyphs in the glyph array
uint24	glyphArray[glyphCount]	Array of 24-bit glyph IDs – in numerical order

Coverage Format 4

Format 4 is the same as Format 2, but uses 24-bit GlyphID values in the RangeRecord subtables.

CoverageFormat4 table: Range of glyphs

Type	Name	Description
uint16	coverageFormat	Format identifier – format = 4
uint16	rangeCount	Number of RangeRecord2 records
RangeRecord2	rangeRecords [rangeCount]	Array of glyph ranges – ordered by startGlyphID

RangeRecord2

Type	Name	Description
uint24	startGlyphID	First glyph ID in the range
uint24	endGlyphID	Last glyph ID in the range
uint24	startCoverageIndex	Coverage Index of first glyph ID in range

Class definition table

In OFF Layout, index values identify glyphs. For efficiency and ease of representation, a font developer

can group glyph indices to form glyph classes. Class assignments vary in meaning from one lookup subtable to another. For example, in the GSUB and GPOS tables, classes are used to describe glyph contexts. GDEF tables also use the idea of glyph classes.

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

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

The ClassDef table can have one of four formats: one that assigns a range of consecutive glyph indices to different classes, or one that puts groups of consecutive glyph indices into the same class, each in both 16-bit and 24-bit versions. Formats 1 and 2 are 16-bit, and Formats 3 and 4 are 24-bit.

Class Definition Table Format 1

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

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

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

ClassDefFormat1 table: Class array

Type	Name	Description
uint16	classFormat	Format identifier-format = 1
uint16	startGlyphID	First glyph ID of the classValueArray
uint16	glyphCount	Size of the classValueArray
uint16	classValueArray[glyphCount]	Array of Class Values – one per glyph ID

Class Definition Table Format 2

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

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

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

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

ClassDefFormat2 table: Class ranges

Type	Name	Description
uint16	classFormat	Format identifier – format = 2
uint16	classRangeCount	Number of ClassRangeRecords
ClassRangeRecord	classRangeRecords [classRangeCount]	Array of ClassRangeRecords – ordered by startGlyphID

ClassRangeRecord

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

Class Definition Table Format 3

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

ClassDefFormat3 table: Class array

Type	Name	Description
uint16	classFormat	Format identifier - format = 3
uint24	startGlyphID	First glyph ID of the classValueArray
uint16	glyphCount	Size of the classValueArray
uint24	classValueArray[glyphCount]	Array of Class Values – one per glyph ID

Class Definition Table Format 4

Format 4 is the same as Format 2, but uses 24-bit glyph identifiers.

ClassDefFormat4 table: Class ranges

Type	Name	Description
uint16	classFormat	Format identifier – format = 4

uint16	classRangeCount	Number of ClassRangeRecords
ClassRangeRecord2	classRangeRecords [classRangeCount]	Array of ClassRangeRecords – ordered by startGlyphID

ClassRangeRecord2

Type	Name	Description
uint24	startGlyphID	First glyph ID in the range
uint24	endGlyphID	Last glyph ID in the range
uint16	class	Applied to all glyphs in the range

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, =1
uint16	minorVersion	Minor version of the GPOS table, = 2
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.

In version 1.2, the offsets whose names end in 2, such as featureListOffset2, 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.

[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 6.3.3.3.2 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
Offset32	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
Offset32	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 6.3.3.3.2 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 allows for larger values:

PairPosFormat3 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 3
Offset32	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
Offset32	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)
Offset32	classDef1Offset	Offset to ClassDef table, from beginning of PairPos subtable – for the first glyph of the pair
Offset32	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
Offset32	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 table

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 6.3.3.3.4 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
Offset32	markCoverageOffset	Offset to MarkCoverage table, from beginning of MarkBasePos subtable
Offset32	baseCoverageOffset	Offset to BaseCoverage table, from beginning of MarkBasePos subtable
uint16	markClassCount	Number of classes defined for marks
Offset32	markArrayOffset	Offset to MarkArray2 table, from the beginning of the MarkBasePos subtable
Offset32	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 table

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 table

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 6.3.3.3.6 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
Offset32	markCoverageOffset	Offset to markCoverage table, from the beginning of the MarkLigPos subtable
Offset32	ligatureCoverageOffset	Offset to the ligatureCoverage table, from the beginning of the MarkLigPos subtable
uint16	markClassCount	Number of defined mark classes
Offset32	markArrayOffset	Offset to the MarkArray table, from the beginning of the MarkLigPos subtable
Offset32	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 MarkMarkPosFormat2 subtable is based on Format 1, but allows for larger offsets:

MarkMarkPosFormat2 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 2
Offset32	mark1CoverageOffset	Offset to Combining Mark Coverage table, from beginning of MarkMarkPos subtable
Offset32	mark2CoverageOffset	Offset to Base Mark Coverage table, from beginning of MarkMarkPos subtable
uint16	markClassCount	Number of Combining Mark classes defined
Offset32	mark1ArrayOffset	Offset to MarkArray2 table for Mark1, from the beginning of the MarkMarkPos subtable
Offset32	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.

NOTE: The 2 in MarkArray2 is for version 2 of the MarkArray table; the first 2 in Mark2Array2 denotes the storage of Mark2, and the trailing 2 denotes version 2 of the table. The MarkArray table itself is defined in subsection *6.3.3.4. Shared tables: Value record, Anchor table and MarkArray table*.

Mark2Array2 table

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

	[mark2Count]	

The individual Mark2Record2 entries in Format 2 use a 24-bit offset to avoid potential overflow:

Mark2Record2 table

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 6.3.3.3.7 Lookup Type 7, Contextual positioning subtables, just before 6.3.3.3.8 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 offsets. 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
Offset32	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:

PosRule2 Table

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 larger offsets to avoid possible overflow.

ContextPosFormat5 Subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 5
Offset32	coverageOffset	Offset to Coverage table, from beginning of ContextPos subtable
Offset32	classDefOffset	Offset to ClassDef table, from beginning of ContextPos subtable
uint24	posClassSetCount	Number of PosClassSet2 tables
Offset24	posClassSetOffsets [posClassSetCount]	Array of offsets to PosClassSet2 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.

ContextPosFormat6 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 allows for larger values:

ChainContextPosFormat4 subtable

Type	Name	Description
uint16	posFormat	Format identifier: format = 4
Offset32	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

ChainPosRuleSet2 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

ChainPosRule2 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	Array of lookahead glyph IDs

	[lookAheadGlyphCount]	
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
Offset32	coverageOffset	Offset to Coverage table, from beginning of ChainContextPos subtable
Offset32	backtrackClassDefOffset	Offset to ClassDef table containing backtrack sequence context, from beginning of ChainContextPos subtable
Offset32	inputClassDefOffset	Offset to ClassDef table containing input sequence context, from beginning of ChainContextPos subtable
Offset32	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 ChainPosClassSet2 table. Consequently, the ChainPosClassSet2 table identifies the class of a context's first component.

ChainPosClassSet2 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:

ChainPosClassRule2 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 featureList2, 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, Multiple substitution subtable, 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
Offset32	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

Offset32	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 used.

MultipleSubstFormat2 subtable:

Type	Name	Description
uint16	substFormat	Format identifier: format = 2
Offset32	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 [sequenceCount]	Array of offsets to 24-bit Sequence2 tables. Offsets are from 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 Alternate substitution subtable 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 (alternateSetOffets). It is identical to the Format1 subtable, except for using 24-bit numbers.

AlternateSubstFormat2 subtable:

Type	Name	Description
uint16	substFormat	Format identifier: format = 2
Offset32	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint24	alternateSetCount	Number of AlternateSet2 tables
Offset32	alternateSetOffsets [alternateSetCount]	Array of offsets to AlternateSet2 tables. Offsets are from beginning of substitution table, ordered by Coverage index

AlternateSet2 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, in LigatureSet table fix LigatureSet to have ligatureOffsets and not ligatureSetOffsets

add then 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

Offset32	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	ligatureOffsets [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 sequnce index = 1, componentGlyphIDs array index = 0) because the first component glyph is specified in the Coverage table.

NOTE The componentGlyphIDs array lists glyph IDs according to the writing direction – that is, the logical order – of the text. For text written right to left, the right-most glyph will be first. Conversely, for text written left to right, the left-most glyph will be first.

Example 6 at the end of this clause shows how to replace a string of glyphs with a single ligature.

Ligature table (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 6.3.4.3.7 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
Offset32	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 [subRuleCount]	Array of offsets to SubRule tables. Offsets are from 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	Array of input glyph IDs – start with

	[glyphCount - 1]	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
Offset32	coverageOffset	Offset to Coverage table, from beginning of substitution table
Offset32	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 SubClassRule2 tables
Offset24	subClassRuleOffsets [subClassRuleCount]	Array of offsets to SubClassRule2 tables. Offsets are from beginning of SubClassSet2, 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
uint24	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
Offset32	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint16	chainSubRuleSetCount	Number of ChainSubRuleSet2 tables – must equal glyphCount in Coverage table
Offset24	chainSubRuleSetOffsets [chainSubRuleSetCount]	Array of offsets to ChainSubRuleSet2 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 GLYF table is used, the offsets are 24-bit:

ChainSubRuleSet2 table: All contexts beginning with the same glyph

Type	Name	Description
uint16	chainSubRuleCount	Number of ChainSubRule2 tables
Offset24	chainSubRuleOffsets [chainSubRuleCount]	Array of offsets to ChainSubRule2 tables. Offsets are from beginning of ChainSubRuleSet2 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)
uint24	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.
uint24	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)
uint24	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
Offset32	coverageOffset	Offset to Coverage table, from beginning of

		substitution table
Offset32	backtrackClassDef	Offset to glyph ClassDef table containing backtrack sequence data, from beginning of substitution table
Offset32	inputClassDef	Offset to glyph ClassDef table containing input sequence data, from beginning of substitution table
Offset32	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 [BacktrackGlyphCount]	Array of backtracking classes – to be matched before the input sequence.
uint16	inputGlyphCount	Total number of classes in the input sequence – includes the first class.
uint24	inputSequence iInputGlyphCount - 1]	Array of classes to be matched with the input glyph sequence – beginning with second glyph position.
uint16	lookaheadGlyphCount	Total number of glyphs in the lookahead sequence (number of glyphs to be matched after the input

		sequence).
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
Offset32	coverageOffset	Offset to Coverage table, from beginning of substitution table
uint24	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.

Near the end of **The Font Collection file structure** [4.6.2] we add a paragraph:

The Font Collection file structure [4.6.2]

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

The TTC file shall contain a complete table directory for each font resource. The same TableDirectory format is used for each font resource in a collection file as is used in a non-collection file. The table offsets in all table directories within a TTC file are measured from the beginning of the TTC file.

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

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

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

Each font in the TTC may contain its own DMAP table. In addition, a font using CMAP may support characters and GlyphIDs also defined by another font; in the case of overlapping coverage the order of precedence is first DMAP, then CMAP, and then finally cmap.

NOTE When building a collection file from separate font files, close attention needs to be paid to the issue of glyph renumbering in a font and the side effects that can result in the 'cmap' table and elsewhere. The fonts to be merged also need to have compatible TrueType instructions; that is, their preprograms, function definitions, and control values cannot conflict.

Collection files containing TrueType glyph outlines should use the filename suffix .TTC. Collection files containing CFF or CFF2 outlines should use the file extension .OTC.

In 4.6.3 TTC Header, we add new versions:

TTC header [4.6.3]

There are four versions of the TTC header: Version 1.0 has been used for TTC files without digital signatures. Version 2.0 can be used for TTC files with *or* without digital signatures — if there's no signature, then the last three fields of the version 2.0 header are left null. Versions 1.1 and 1.2 mirror versions 1 and 2 but are used for supporting collections containing fonts with 24-bit glyph identifiers. Software that does not support versions 1.1 and 2.1 of the TTC header table will continue to function using a version 1 or 2 header, if present, and will use the 16-bit tables pointed to from that header. Newer software will make use of the 24-bit tables. Software that supports *tableDirectoryOffsets2* shall use it instead of *tableDirectoryOffsets* when both are present.

Tables pointed to in collections using header versions 1.1 or 2.1 can *overlap*, so that, for example, much of the GPOS table could be shared between 16-bit and 24-bit fonts in the collection.

If a digital signature is used, the DSIG table for the file shall be located at the end of the TTC file, following any other font tables. Signatures in a TTC file are expected to be Format 1 signatures.

The purpose of the TTC header table is to locate the different table directories within a TTC file. The TTC header is located at the beginning of the TTC file (offset = 0). It consists of an identification tag, a version number, a count of the number of OFF fonts in the file, and an array of offsets to each table directory.

TTCHeader Version 1.0:

Type	Name	Description
Tag	ttcTag	Font Collection ID string: 'ttcf' (used for fonts with CFF or CFF2 outlines, as well as TrueType outlines)
uint16	majorVersion	Major version of the TTCHeader, = 1.
uint16	minorVersion	Minor version of the TTCHeader, = 0.
uint32	numFonts	Number of fonts in TTC
Offset32	tableDirectoryOffsets [numFonts]	Array of offsets to the TableDirectory for each font from the beginning of the file

TTCHeader Version 1.1:

Type	Name	Description
Tag	ttcTag	Font Collection ID string: 'ttcf' (used for fonts with CFF or CFF2 outlines, as well as TrueType outlines)
uint16	majorVersion	Major version of the TTCHeader, = 1.
uint16	minorVersion	Minor version of the TTCHeader, = 1.
uint32	numFonts	Number of fonts in TTC
Offset32	tableDirectoryOffsets [numFonts]	Array of offsets to the TableDirectory for each font from the beginning of the file
uint32	numFonts2	Number of 24-bit fonts in TTC
Offset32	tableDirectoryOffsets2 [numFonts2]	Array of offsets to the TableDirectory for each 24-bit font from the beginning of the file

TTCHeader Version 2.0:

Type	Name	Description
Tag	ttcTag	Font Collection ID string: 'ttcf'
uint16	majorVersion	Major version of the TTCHeader, = 2.
uint16	minorVersion	Minor version of the TTCHeader, = 0.
uint32	numFonts	Number of fonts in TTC
Offset32	tableDirectoryOffsets [numFonts]	Array of offsets to the TableDirectory for each font from the beginning of the file
uint32	dsigTag	Tag indicating that a DSIG table exists, 0x44534947 ('DSIG') (null if no signature)
uint32	dsigLength	The length (in bytes) of the DSIG table (null if no signature)
uint32	dsigOffset	The offset (in bytes) of the DSIG table from the beginning of the TTC file (null if no signature)

TTCHeader Version 2.1:

Type	Name	Description
Tag	ttcTag	Font Collection ID string: 'ttcf'
uint16	majorVersion	Major version of the TTCHeader, = 2
uint16	minorVersion	Minor version of the TTCHeader, = 1.
uint32	numFonts	Number of fonts in TTC
Offset32	tableDirectoryOffsets [numFonts]	Array of offsets to the TableDirectory for each font from the beginning of the file
uint32	dsigTag	Tag indicating that a DSIG table exists, 0x44534947 ('DSIG') (null if no signature)
uint32	dsigLength	The length (in bytes) of the DSIG table (null if no signature)
uint32	dsigOffset	The offset (in bytes) of the DSIG table from the beginning of the TTC file (null if no signature)
uint32	NumFonts2	Number of 24-bit fonts in TTC
Offset32	tableDirectoryOffsets2 [numFonts2]	Array of offsets to the TableDirectory for each 24-bit font from the beginning of the file

VARC—Variable Composite Glyph Descriptions

Introduction (not for the spec):

In the ISO OpenFont Format, composite glyphs are made by combining other glyphs. This might be as simple as a precombined accented letter such as é or š, or it could be a Chinese Han (ideogram) made by combining brush strokes, or it could be an emoji or icon such as a smiling face combined with a circle.

Glyphs constructed by combining other glyphs are referred to as composite glyphs.

In a Variable Font, it might be necessary to move or transform the composite glyph in complex ways: parts of a composite glyph might need to move further apart from one another as weight increases, or change relative position as italic angle changes.

The VARC table stores information about Variable Composite glyphs, giving the font designer control over the results of altering variable-font axis values. The control possible is similar to that in color fonts using COLR v1.

An earlier proposal included these new mechanisms inside the individual glyph definitions, but that meant they were specific to glyf-table glyphs, and were not available to CFF2-based fonts. After discussion, and a counter-proposal from Skef Iterum at Adobe Systems, a sample implementation of an external table proved more effective.

Data Types

Update Section 4.3 Data Types, as follows:

Data types

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

Data Type	Description
uint8	8-bit unsigned integer.
int8	8-bit signed integer.
uint16	16-bit unsigned integer.
int16	16-bit signed integer.
uint24	24-bit unsigned integer.
uint32	32-bit unsigned integer.
int32	32-bit signed integer.
uint32var	Encodes a 32-bit integer (uint32) in a variable number of bytes.
Fixed	32-bit signed fixed-point number (16.16)

FWORD	int16 that describes a quantity in font design units.
UFWORD	uint16 that describes a quantity in font design units.
F2DOT14	16-bit signed fixed number with the low 14 bits of fraction (2.14).
F4DOT12	16-bit signed fixed number with the low 12 bits of fraction (4.12).
F6DOT10	16-bit signed fixed number with the low 10 bits of fraction (6.10).
LONGDATETIME	Date and time represented in number of seconds since 12:00 midnight, January 1, 1904, UTC. The value is represented as a signed 64-bit integer.
Tag	Array of four uint8s (length = 32 bits) used to identify a table, design-variation axis, script, language system, feature, or baseline.
TupleValues	A version of Packed Deltas supporting 32-bit values
Offset8	8-bit offset to a table, same as uint8, NULL offset = 0x00
Offset16	Short offset to a table, same as uint16, NULL offset = 0x0000
Offset24	24-bit offset to a table, same as uint24, NULL offset = 0x000000
Offset32	Long offset to a table, same as uint32, NULL offset = 0x00000000
Version16Dot16	Packed 32-bit value with major and minor version numbers. (See 4.4.)

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

Decimal Value	Hex Value	Integer	Fraction
1.999939	0x7fff	1	16383/16384
1.75	0x7000	1	12288/16384
0.000061	0x0001	0	1/16384
0.0	0x0000	0	0/16384
-0.000061	0xffff	-1	16383/16384
-2.0	0x8000	-2	0/16384

The **F4DOT12** and **F6DOT10** formats work the same way as F2DOT14 but have a larger integer and less precision in the unsigned fraction part.

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

characters, padded with trailing spaces (byte value 0x20). A space character shall not be followed by a non-space character.

The **uint32var** format is an efficient and compact encoding of uint32 values, using from one to five bytes to store a single value.

Values are decoded as follows:

If the current byte's most significant bit (0x80) is zero, return the value.

Otherwise, the most significant bits are taken to indicate the number of bytes to follow, as indicated in the following table; the values are stored with the first byte being the most significant.

Bit				**Meaning**
0x80 (1000 0000)	0x40 (0100 0000)	0x20 (0010 0000)	0x10 (0001 0000)	
zero	(any value)	(any value)	(any value)	All 7 remaining bits are the value
1	0	(any value)	(any value)	One byte follows; the low 6 bits of the first are shifted left 8 bits (multiplied by 256) and the singly following byte is added.
1	1	0	(any value)	Two bytes follow
1	1	1	0	Three bytes follow
1	1	1	1	Four bytes follow

The following pseudo-code can be adapted to read and write these variable-byte numbers. Note that although they were inspired by UTF-8 encoding, they are not the same, and achieve greater compression in the font encoding context.

```
def _readuint32var(data, i):
    """Read a variable-length number from data starting at index i.

    Return the number and the next index.
    """

    b0 = data[i]
    if b0 < 0x80:
        return b0, i + 1
    elif b0 < 0xC0:
        return (b0 - 0x80) << 8 | data[i + 1], i + 2
    elif b0 < 0xE0:
        return (b0 - 0xC0) << 16 | data[i + 1] << 8 | data[i + 2], i + 3
```

```
    elif b0 < 0xF0:
        return (b0 - 0xE0) << 24 | data[i + 1] << 16 | data[i + 2] << 8 | data[
            i + 3
        ], i + 4
    else:
        return (b0 - 0xF0) << 32 | data[i + 1] << 24 | data[i + 2] << 16 | data[
            i + 3
        ] << 8 | data[i + 4], i + 5
```

```
def _writeuint32var(v):
    """Write a variable-length number.

    Return the data.
    """
    if v < 0x80:
        return struct.pack(">B", v)
    elif v < 0x4000:
        return struct.pack(">H", (v | 0x8000))
    elif v < 0x200000:
        return struct.pack(">L", (v | 0xC00000))[1:]
    elif v < 0x10000000:
        return struct.pack(">L", (v | 0xE0000000))
    else:
        return struct.pack(">B", 0xF0) + struct.pack(">L", v)
```

The **TupleValues** format is essentially the same as @@link-to Packed Deltas from the TupleVariationStore, but modified to allow storing 32-bit values: if DELTAS_ARE_ZERO and DELTAS_ARE_WORDS are both set in the control byte, then the following values are interpreted as 32-bit.

NOTE Those two flags, corresponding to the topmost two bits in the control byte, could not otherwise both be set.

TupleValues are used in two ways:

1. When the number of values to be decoded is known in advance, decoding stops when all the values are decoded.

2. When the number of values to be decoded is not known in advance, but the number of encoded bytes is known, then values are decoded until all the encoded bytes have been processed.

Within this specification, many structures are defined in terms of the data types listed above. Structures are characterized as either *records* or *tables*. The distinction between records and tables is based on these general criteria:

— Tables are referenced by offsets. If a table contains an offset to a sub-structure, the offset is normally from the start of that table.

— Records occur sequentially within a parent structure, either within a sequence of table fields or within an array of records of a given type. If a record contains an offset to a sub-structure, that structure is logically a subtable of the record's parent table and the offset is normally from the start of the parent table.

In some cases, fields for subtable offsets are documented as permitting NULL values when the given subtable is optional. For example, in the 'BASE' table header, the horizAxisOffset and vertAxisOffset fields may be NULL, meaning that either subtable (or both) is optional. A NULL subtable offset always indicates that the given subtable is not present. Applications shall never interpret a NULL offset value as the offset to subtable data. For cases in which subtable offset fields are not documented as permitting NULL values, font compilers shall include a subtable of the indicated format, even if it is a header stub without further data (for example, a coverage table with no glyph IDs). Applications parsing font data should, however, anticipate non-conformant font data that has a NULL subtable offset where only a non-NULL value is expected.

Replace section 6.2.2 as follows:

OFF layout and Font variations [6.2.2]

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

When different variation instances are selected, the design and metrics of individual glyphs changes. This can impact font-unit values given in GPOS, BASE, JSTF or GDEF tables, such as the X and Y coordinates of an attachment anchor position. The font-unit values given in these tables apply to the default instance of a variable font. If adjustments are needed for different variation instances, this is done using variation data with processes similar to those used for glyph outlines and other font data, as described in subclause 7.1. The variation data for GPOS, JSTF or GDEF values is contained in an *ItemVariationStore* table which, in turn, is contained within the GDEF table; variation data for BASE values is contained in an ItemVariationStore table within the BASE table itself. The format of the ItemVariationStore is described in detail in the subclause 7.2. For font-unit values within the GPOS, BASE, JSTF or GDEF tables that require variation, references to specific variation data within the ItemVariationStore are provided in *VariationIndex tables*, described below.

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

Where composite glyphs are constructed by statically transforming references to other glyphs, the VARC table allows for referring to other glyphs at different variation locations. Moreover, this location, and the component transforms, can vary. The 'VARC' table uses a *MultiItemVariationStore* subtable which is defined in [section 7..2 Font Variations Common Table Formats].

Add a new subsection 7.2.3.6 just before 7.2.4 Design-variation axis tag registry

MultiItemVariationStore [7.2.3.6]

This structure is a hybrid between *ItemVariationStore* and *TupleVariationStore* that is designed for more space-efficient storage of variations of tuples of numbers. It is used in the VARC table for variable composite glyphs.

Like *ItemVariationStore*, entries are addressed using a 32-bit *VarIdx*, with the top 16 bits called the "outer" index, and the lower 16 bits called the "inner" index.

Whereas the *ItemVariationStore* stores deltas for a single scalar value for each *VarIdx*, the *MultiItemVariationStore* stores deltas for a tuple for each *VarIdx*. Compared to *ItemVariationStore*, the *MultiItemVariationStore* uses a sparse encoding of the active axes for each region, which is more efficient in fonts with many axes.

Compared to *TupleVariationStore*, the *MultiItemVariationStore* is optimized for smaller tuples and allows tuple-sharing, which is important for its efficiency over the *TupleVariationStore*. It also avoids some of the limitations of *TupleVariationStore*, such as the total size of an entry being limited to 64KBytes.

The following structures form the *MultiItemVariationStore*. Its processing is fairly similar to that of the *ItemVariationStore*, except that the deltas encoded for each entry consist of multiple numbers per region. The *TupleValues* for each entry is the concatenation of the tuple deltas for each region.

MultiItemVariationStore

Type	Name	Description
uint16	format	Set to 1
Offset32	VariationRegionListOffset	Offset to array of SparseVariationRegionList, measured from the start of the MultiItemVariationStore
uint16	itemVariationDataCount	Number of ItemVariationDataOffsets records
Offset32	itemVariationDataOffsets[itemVariationDataCount]	Offset to array of ItemVariationDataOffsets records, measured from the start of the MultiItemVariationStore

SparseVariationRegionList

Type	Name	Description
uint16	regionCount	Number of regions
Offset32	variationRegionOffsets[reg	Offset to array of variationRegionOffsets records, measured from the start of the

	ionCount];	SparseVariationRegionList

SparseRegionAxisCoordinates

Type	Name	Description
uint16	axisIndex	
F2DOT14	startCoord	
F2DOT14	peakCoord	
F2DOT14	endCoord	

MultiItemVariationData

Type	Name	Description
uint16	Format	Set to 1
uint16	regionIndexCount	
uint16	regionIndexes[regionIndexCount]	
CFF2Index	deltaSets	The CFF2Index here stores TupleValues

The deltaSets in a *MultiItemVariationData* table each store the delta-set for a single tuple, addressed by the "inner" index of the VarIdx (see @@*MultiItemVariationStore*), whereas a *MultiItemVariationData* table itself represents the data for all values sharing the same "outer" index.

The TupleValues for each entry are the concatenation of the tuple deltas for each region. The length of the tuple is calculated by dividing the number of entries in the TupleValues structure by the number of regions.

the CFF2Index format used in VARC (and hence in *MultiItemVariationData* subtables) is a variable-sized CFF2-style Index structure, as defined in [@@CFFF2]. CFF2Index is used to store a list of variable-sized data, such as glyph records, in a space-efficient manner.

NOTE: The count field of the CFF2 Index structure is a 32-bit value (uint32), unlike the corresponding count field in the CFF table.

At the end of the Variable Fonts section, just before **8 Recommendations for OFF fonts,** add VARC as follows:

VARC—Variable Composite Glyph Descriptions [7.3.9]

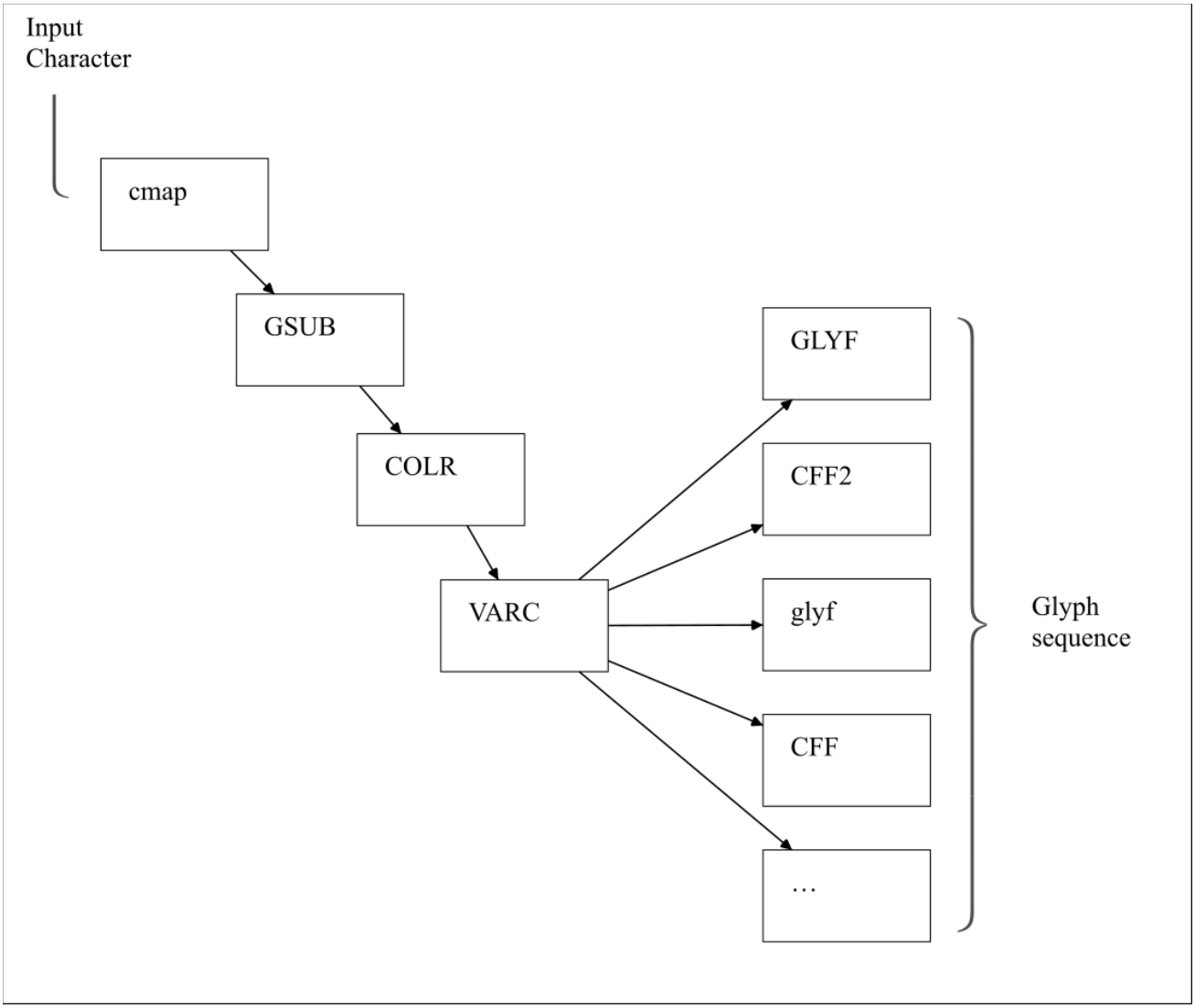
In the ISO OpenFont Format, composite glyphs are made by combining other glyphs. This might be as simple as a precombined accented letter such as é or š, or it could be a Chinese Han (ideogram) made by combining brush strokes, or it could be an emoji or icon such as a smiling face combined with a circle.

Glyphs constructed by combining other glyphs are referred to as composite glyphs.

In a Variable Font, it might be necessary to move or transform the composite glyph in complex ways: parts of a composite glyph might need to move further apart from one another as weight increases, or change relative position as italic angle changes.

The optional VARC table stores information about Variable Composite glyphs, giving the font designer control over the results of altering variable-font axis values. The control possible is similar to that in color fonts using COLR v1.

The following figure illustrates where the VARC table fits in the glyph selection process:



Variable Composite Glyphs: processing with VARC in relation to other tables

The optional ‘VARC’ table contains definitions of variable-composite glyphs. It is used in conjunction with ‘glyf’, ‘GLYF’, ‘gvar’, ‘GVAR’ and ‘CFF2’ tables. Variable composite glyphs are made up of one or more *variable components*, each of which resolves to a glyph that can be transformed and potentially given different font variation axis values before being combined to make up the variable composite glyph.

The ‘VARC’ table is designed to be very space-efficient, so that fonts making extensive use of its mechanism can sometimes be stored in considerably less file space than without it. To achieve these savings, several data structures specific to this table are used. The data structures are defined in [@@link to 4,3 Data Structures].

VARC table header

Type	Name	Description
uint16	majorVersion	Table version. Set to 1.
uint16	minorVersion	Set to zero.

Offset32	coverage	Offset to Coverage table
Offset32	varStore	Offset to MultiItemVariationStore table
Offset32	axisIndicesList	Offset to CFF2Index containing TupleValues
Offset32	glyphRecords	Offset to VarCompositeGlyphs

The *coverage* table contains a *glyphRecord* for each glyph for which this 'VARC' table contains a Variable-Composite record. The glyphRecords shall be arranged by coverage index.

The *varStore* subtable stores the variations of axis values and transform values that are references in the individual Variable Component records.

The *axisIndicesList* field is a list of tuples, each of which encodes an (index, axis) pair; the individual VariableComponent records refer to one of these TupleValues using the index into the list.

Variable Composite Description (VarCompositeGlyph)

A Variable Composite record is a concatenation of one or more Variable Component records.

Individual Variable Component records have varying sizes. When decoding a *VarCompositeGlyph*, the decoder stops when it has processed all of the bytes for that glyph.

VarCompositeGlyph

Type	Name	Description
VarComponent[]	components	An array of Variable Component Records

Variable Component Record

A Variable Component record encodes one component's glyph index, variations location, and transformation in a variable-sized and efficient manner.

type	name	Description
uint32var	flags	See below.
GlyphID16 or GlyphID24	gid	This is a GlyphID16 if GID_IS_24BIT bit of flags is clear, else GlyphID24.
uint32var	axisIndicesIndex	Optional, only present if HAVE_AXES bit of flags is set.
TupleValues	axisValues	The axis value for each axis, variable sized.

uint32var	axisValuesVarIndex	Optional, only present if AXIS_VALUES_HAVE_VARIATION bit of flags is set.
uint32var	transformVarIndex	Optional, only present if TRANSFORM_HAS_VARIATION bit of flags is set.
FWORD	TranslateX	Optional, only present if HAVE_TRANSLATE_X bit of flags is set.
FWORD	TranslateY	Optional, only present if HAVE_TRANSLATE_Y bit of flags is set.
F4DOT12	Rotation	Optional, only present if HAVE_ROTATION bit of flags is set. Counter-clockwise.
F6DOT10	ScaleX	Optional, only present if HAVE_SCALE_X bit of flags is set.
F6DOT10	ScaleY	Optional, only present if HAVE_SCALE_Y bit of flags is set.
F4DOT12	SkewX	Optional, only present if HAVE_SKEW_X bit of flags is set. Counter-clockwise.
F4DOT12	SkewY	Optional, only present if HAVE_SKEW_Y bit of flags is set. Counter-clockwise.
FWORD	TCenterX	Optional, only present if HAVE_TCENTER_X bit of flags is set.
FWORD	TCenterY	Optional, only present if HAVE_TCENTER_Y bit of flags is set.

Variable Component Flags

bit number	name
0	RESET_UNSPECIFIED_AXES
1	HAVE_AXES
2	AXIS_VALUES_HAVE_VARIATION
3	TRANSFORM_HAS_VARIATION
4	HAVE_TRANSLATE_X
5	HAVE_TRANSLATE_Y

6	HAVE_ROTATION
7	USE_MY_METRICS
8	HAVE_SCALE_X
9	HAVE_SCALE_Y
10	HAVE_TCENTER_X
11	HAVE_TCENTER_Y
12	GID_IS_24BIT
13	HAVE_SKEW_X
14	HAVE_SKEW_Y
15-31	Reserved. Set to 0

NOTE The flags are arranged in decreasing order of expected usage frequency, in order to minimize the storage bytes of the flags field.

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

The rotation and skew parameters are in angles as multiples of Pi.

The following Python example illustrations construction of a transformation matrix:

```
# 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)
```

Processing

The component glyphs to be loaded shall use the coordinate values specified (with any variations applied if present).

For any unspecified axis, the value used depends on flag RESET_UNSPECIFIED_AXES:

If RESET_UNSPECIFIED_AXES is set, then the normalized value zero shall be used.

If RESET_UNSPECIFIED_AXES is clear, then the axis values from the current glyph being processed (which itself might recursively come from the font or its own parent glyphs) shall be used.

For 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 the RESET_UNSPECIFIED_AXES flag bit is set, then the value of wght=0 (default) will be used. If on the other hand that flag bit is clear, wght=.5 (from current glyph) will be used.

The component axis values and transform components can vary. These variations are stored in the MultiItemVariationStore data-structure. The variations for location are referred to by the axisValuesVarIndex member of a component if any, and variations for transform are referred to by the transformVarIndex if any. For transform variations, only those fields specified and as such encoded as per the flags have variations.

Each component is recursively loaded from the VARC table, if it is present in this table; if it is not found in VARC, it is loaded from 'GLYF'/'GVAR', 'glyf'/'gvar', or 'CFF2' as appropriate. An exception to the recursive rule is that if a component refers to the glyphID of the current glyph, instead of recursing (which would result in infinite recursion), the glyph outline for the component shall be loaded directly instead, as if the glyphID had no VARC entry.

NOTE While it is the customarily implemented 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 variable-composite glyph.

NOTE A static (non-variable) font that uses the VARC table would not have fvar / avar tables but would have the GVAR or gvar table in a TrueType-flavored font, or CFF2 variations in a CFF-flavored font. This is because the components themselves store their variables in the classic way. The exception to this situation is a font with VARC table that does NOT vary the component locations and transforms, and does not encode any axis values for the components either. In practice, such a font would just use the VARC table in the manner of classic components in a glyf table.

[end]