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 YYYYY

Geneva, Switzerland – September 2023

Title: Changes to support larger character sets and 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)

Introduction

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. 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 65534 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.

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

Name	Status	Notes
cmap	extended	Format 16 is added, with 24-bit glyph IDs
GDEF	changed	Updated to avoid overflow, and for 24-bit Glyph IDs
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
kerx	changed	Updated to allow more entries
LOCA	new	Augmented version of loca
maxp	unchanged	Previous proposal required numGlyphs to be set to 63355 for a font with more than that number of glyphs.
morx	changed	Updated to allow more entries
sbix	changed	Updated to allow more entries
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

[5.1.6] maxp—Maximum profile

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 rely on 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.
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 / glyf—Glyph data [5.2.4]

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.

In order to be able to produce any rendered glyphs or paths, a font must contain at least one of a CFF or CFF table, a 'glyf' table, or a 'GLYF' table.

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 number of entries of the 'GLYF' table is equal to the number of entries in the 'LOCA' table.

The number of entries in the 'glyf' table is equal to the number of entries in the 'loca' table.

Glyph headers

Each glyph description begins with a header:

Glyph Header

Type	Name	Description
int16	numberOfContours	If the number of contours is greater than or equal to zero, this is a simple glyph. If negative, this is a composite glyph—the value -1 should be used for composite glyphs, and -2 for variable composite glyphs.
int16	xMin	Minimum x for coordinate data.
int16	yMin	Minimum y for coordinate data.
int16	xMax	Maximum x for coordinate data.
int16	yMax	Maximum y for coordinate data.

Note: Variable Composite Glyphs are allowed only in the 'GLYF' table, not in 'glyf', for backwards compatibility with older software.

Simple glyph description [5.2.4.1.1]

Simple Glyph table

Add the following flag to the Simple Glyph Description section's *Simple Glyph Flags*:

Mask	Name	Description
0x80	`CUBIC`	Bit 7: Off-curve point belongs to a cubic-Bezier segment

Just before the next section (Composite glyph description), add:

The CUBIC flag shall be used only in the 'GLYF' table, and not the 'glyf' table.

If the CUBIC flag is non-zero, the corresponding off-curve point belongs to a Cubic Bézier path segment.

There are several restrictions on how the CUBIC flag can be used. If any of the conditions below are not met, the behavior is undefined.

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

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 is considered the first control-point of a cubic Bézier curve, and implied on-curve points are inserted between the every second point and the next one as usual.

@@Liam note: maybe we want to say if the 0x80 flag is set, there's another byte of flags, and first of these is cubic vs quadratic. That way more flags can still be added in the future. An alternative might be to add a header to GLYF with a format version.

Variable Composite Description

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

Variable Component Record

type	name	notes
uint16	flags	see below
uint8	numAxes	Number of axes to follow
GlyphID16 or GlyphID24	gid	This is a GlyphID16 if bit 12 of `flags` is clear, else GlyphID24
uint8 or uint16	axisIndices[numAxes]	This is a uint16 if bit 1 of `flags` is set, else a uint8
F2DOT14	axisValues[numAxes]	The axis value for each axis
FWORD	TranslateX	Optional, only present if bit 3 of `flags` is set
FWORD	TranslateY	Optional, only present if bit 4 of `flags` is set
F4DOT12	Rotation	Optional, only present if bit 5 of `flags` is set
F6DOT10	ScaleX	Optional, only present if bit 6 of `flags` is set
F6DOT10	ScaleY	Optional, only present if bit 7 of `flags` is set
F4DOT12	SkewX	Optional, only present if bit 8 of `flags` is set
F4DOT12	SkewY	Optional, only present if bit 9 of `flags` is set
FWORD	TCenterX	Optional, only present if bit 10 of `flags` is set
FWORD	TCenterY	Optional, only present if bit 11 of `flags` is set

Variable Component Transformation

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

Transformation fields:

name	default value
TranslateX	0
TranslateY	0
Rotation	0
ScaleX	1
ScaleY	1
SkewX	0

name	default value
SkewY	0
TCenterX	0
TCenterY	0

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

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

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

Variable Component Flags

bit number	meaning
0	Use my metrics
1	axis indices are shorts (clear = bytes, set = shorts)
2	if ScaleY is missing: take value from ScaleX
3	have TranslateX
4	have TranslateY
5	have Rotation
6	have ScaleX
7	have ScaleY
8	have SkewX
9	have SkewY
10	have TCenterX
11	have TCenterY

bit number	meaning
12	gid is 24 bit
13	axis indices have variation
14	reset unspecified axes
15	(reserved, set to 0)

Processing

Variations of composite glyphs shall be processed as follows:

For each composite glyph, a vector of coordinate points is prepared, and its variation applied from the 'gvar' table. The coordinate points for the composite glyph are a concatenation of those for each variable component in order. For the purposes of 'GVAR' delta IUP calculations, each point is considered to be in its own contour.

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

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

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

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

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

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

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

be used.

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

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

A.a. Comparing CFF2 with CFF and glyf [5.3.3.13]

The formats 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' and '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	using subroutines	

LOCA/loca—Index to location

The ‘LOCA’ table is same as ‘loca’ 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, minus 1
- 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, minus 1.

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.

Composite glyphs

[5.3.4.1.2] Composite Glyph description [second table]

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.
0x200	GID_IS_24_BIT	Set to allow encoding 24bit glyph indices in composite glyphs.
0xE010	RESERVED	Bits 4, 14 and 15 are reserved: set to 0.

[7.3.4] gvar and GVAR tables

A.a. Introduction to changes:

Currently the gvar table encodes a uint16 glyphCount that must match "the number of glyphs stored elsewhere in the font." That field is then used in a subsequent array:

Type	Name	Description
Offset16 or Offset32	glyphVariationDataOffsets[glyphCount+1]	Offsets from the start of the GlyphVariationData array to each GlyphVariationData table.

We deprecate the glyphCount field, and make the array size match the number of glyphs stored elsewhere in the font instead.

In any case, the glyphCount in gvar shall be calculated ignoring any ‘GLYF’ table, if present.

Note—This allows us to expand gvar without needing to bump to table format 2, which we like to reserve for more extensive table overhaul.

A.b. Changes

[7.3.4.1.1] 'GVAR' table

The GVAR table has the same structure as the ‘gvar’ table, except that the glyphCount member in the GVAR heads is uint32 instead of uint16 in gvar and refers to the number of glyphs in the GLYF table for which variations are defined.

uint32	glyphCount	The number of glyphs in the GLYF table.

[5.1.2] cmap table

The cmap formats 12 and 13 subtables already support 32-bit glyph indices.

A new sub-table format 16 will be added which is similar to format 14 but uses 24-bit glyph indices:

```
struct UVSMapping24 {
  uint24 unicodeValue;
  uint24 glyphID;
};
```

The rest of format 16 subtable is similar to format 14.

A.a. Changes

A.b. Table overview

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

Regardless of the encoding scheme, character codes that do not correspond to any glyph in the font should be mapped to glyph index 0. The glyph at this location must be a special glyph representing a missing character, commonly known as .notdef.

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

Other enhancements in Unicode led to the addition of other subtable formats. Subtable format 13 allows for an efficient mapping of many characters to a single glyph; this is useful for “last-resort” fonts that

provide fallback rendering for all possible Unicode characters with a distinct fallback glyph for different Unicode ranges. Subtable format 14 provides a unified mechanism for supporting Unicode variation sequences. Subtable format 16 was added to allow more than 65535 glyphs in a font; it is similar to format 14 but uses 24-bit glyph indices.

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

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

[…]

[5.1.2.5.11] Format 16: U24-bit CMAP table

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

UVSMapping Record:

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

Hhea / vhea / hmtx / vmtx tables

HHEA and VHEA are the same as hhea and vhea respectively, but with 32-bit numberOf{H,V}metrics values.

The HMTX / VMTX tables have the exact same structure as the hmtx / vmtx tables, just respecting the numberOf{H,V}metrics from HHEA / VHEA instead of hhea / vhea.

HVAR / VVAR tables

No changes needed.

VORG table

@@todo need new version

COLR table

No changes needed. COLORv2 may introduce changes

`sbix` table

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.

`GDEF` / `GSUB` / `GPOS` tables

A.a. `GDEF` version 2

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

```
struct GDEFVersion2 {
  Version version; // 0x00020000
  Offset24To<ClassDef> glyphClassDef;
  Offset24To<AttachList> attachList;
  Offset24To<LigCaretList> ligCaretList;

  Offset24To<ClassDef> markAttachClassDef;
  Offset24To<MarkGlyphSets> markGlyphSets;
  Offset32To<ItemVariationStore> varStore;
};
```

Note that the varStore offset is 32bit, for compatibility with 0x00010003 version.

A.b. `GSUB`

The following sections when combined, fully cover the GSUB table extension:

- GSUB / GPOS version 2
- Coverage / ClassDef formats 3 & 4
- GSUB SingleSubst formats 3 & 4
- GSUB MultipleSubst / AlternateSubst format 2
- GSUB LigatureSubst format 2
- GSUB / GPOS (Chain)Context format 4 & 5

A.c. `GPOS`

The following sections when combined, fully cover the GSUB table extension:

- GSUB / GPOS version 2
- Coverage / ClassDef formats 3 & 4
- GPOS PairPos formats 3 & 4
- GPOS MarkBasePos / MarkLigPos / MarkMarkPos format 2
- GSUB / GPOS (Chain)Context format 4 & 5

A.d. GSUB / GPOS version 2

The main GSUB / GPOS structs currently result in an offset-overflow with more than ~3k lookups. They are augmented with a version 2 that alleviates this problem:

```
struct GSUBGPOSVersion2 {
  Version version; // 0x00020000
  Offset24To<ScriptList> scripts;
  Offset24To<FeatureList> features;
  Offset24To<LookupList24> lookups;
  Offset32To<FeatureVariations> featureVars;
};
```

Note that the last item is a 32bit offset, for compatibility with version 0x00010001 tables.

LookupList24 is a List16 of Offset24 to Lookup structures:

```
using LookupList24 = List16OfOffse24tTo<Lookup>;
```

A.e. `Coverage` / `ClassDef` formats 3 & 4

Coverage and ClassDef tables are augmented with formats 3 and 4 to allow for gid24, which parallel formats 1 & 2 respectively:

```
struct CoverageFormat3 {
  uint16 format; // == 3
  Array16Of<GlyphID24> glyphs;
};

struct CoverageFormat4 {
  uint16 format; // == 4
  Array16Of<Range24Record> ranges;
};

struct ClassDefFormat3 {
  uint16 format; // == 3
  GlyphID24 startGlyphID;
  Array24Of<uint16> classes;
};

struct ClassDefFormat4 {
  uint16 format; // == 4
  Array24Of<Range24Record> ranges;
};

struct Range24Record {
  GlyphID24 startGlyphID;
  GlyphID24 endGlyphID;
  uint16 value;
};
```

A.f. GSUB `SingleSubst` formats 3 & 4

Clarify that in format 1, delta addition math only affects the lower 16bits of the gid. Format 3 delta addition math is module 2^24.

Two new formats, 3 & 4, are introduced, that parallel formats 1 & 2 respectively:

```
struct SingleSubstFormat3 {
```

```
  uint16 format; // == 3
  Offset24To<Coverage> coverage;
  int24 deltaGlyphID;
};

struct SingleSubstFormat4 {
  uint16 format; // == 4
  Offset24To<Coverage> coverage;
  Array16Of<GlyphID24> substitutes;
};
```

A.g. GSUB `MultipleSubst` / `AlternateSubst` format 2

Format 2 is introduced to enable gid24:

```
typedef Array16Of<GlyphID24> Sequence24;

struct MultipleSubstFormat2 {
  uint16 format; // == 2
  Offset24To<Coverage> coverage;
  Array16Of<Offset24To<Sequence24>> sequences;
};

typedef Array16Of<GlyphID24> AlternateSet24;

struct AlternateSubstFormat2 {
  uint16 format; // == 2
  Offset24To<Coverage> coverage;
  Array16Of<Offset24To<AlternateSet24>> alternateSets;
};
```

A.h. GSUB LigatureSubst format 2

Format 2 is introduced to enable `gid24`:

```
struct LigatureSubstFormat2 {
  uint16 format; == 2
  Offset24To<Coverage> coverage;
  Array16Of<Offset24To<LigatureSet24>> ligatureSets;
};

struct LigatureSet24 {
  Array16Of<Offset16To<Ligature24>> ligatures;
};

struct Ligature24 {
  GlyphID24 ligatureGlyph;
  uint16 componentCount;
  GlyphID24 componentGlyphIDs[componentCount – 1];
};
```

A.i. GPOS PairPos formats 3 & 4

Two new formats, 3 & 4, are introduced that parallel formats 1 & 2.

Format 4 is only introduced to alleviate offset-overflow issues and is not otherwise needed for gid24 support.

```
struct PosFormat3 {
```

```
  uint16 format; == 3
  Offset24To<Coverage> coverage;
  uint16 valueFormat1;
  uint16 valueFormat2;
  Array16Of<Offset24To<PairSet24>> pairSets;
};

struct PairSet24 {
  uint24 pairValueCount;
  PairValueRecord24 pairValueRecords[pairValueCount];
};

struct PairValueRecord24 {
  GlyphID24 secondGlyph;
  ValueRecord valueRecord1;
  ValueRecord valueRecord2;
};

struct PosFormat4 {
  uint16 format; == 4
  Offset24To<Coverage> coverage;
  uint16 valueFormat1;
  uint16 valueFormat2;
  Offset24To<ClassDef> classDef1;
  Offset24To<ClassDef> classDef2;
  uint16 class1Count;
  uint16 class2Count;
  Class1Record class1Records[class1Count];
};
```

A.j. GPOS MarkBasePos / MarkLigPos / MarkMarkPos format 2

Format 2 is introduce just to alleviate offset-overflow issues at the top-level structure. All downstream structures are reused:

```
The Coverage table parts are covered in #30.

Add one new format of each, just to upgrade offsets of the top-level subtable to
24bit. All downstream structs are reused and not expanded.

struct MarkBasePosFormat2 {

  uint16 format; // == 2

  Offset24To<Coverage> markCoverage;

  Offset24To<Coverage> baseCoverage;

  uint16 markClassCount;

  Offset24To<MarkArray> markArray;

  Offset24To<BaseArray> baseArray;

};



struct MarkLigaturePosFormat2 {

  uint16 format; // == 2
```

```
  Offset24To<Coverage> markCoverage;
  Offset24To<Coverage> ligatureCoverage;
  uint16 markClassCount;
  Offset24To<MarkArray> markArray;
  Offset24To<LigatureArray> ligatureArray;
};

struct MarkMarkPosFormat2 {
  uint16 format; // == 2
  Offset24To<Coverage> mark1Coverage;
  Offset24To<Coverage> mark2Coverage;
  uint16 markClassCount;
  Offset24To<MarkArray> mark1Array;
  Offset24To<Mark2Array> mark2Array;
};
```

A.k. GSUB / GPOS (Chain)Context format 4 & 5

Add Context and ChainContext format 4 that parallels format 1 for gid24:

```
struct ContextFormat4 {
  uint16 format; == 4
  Offset24To<Coverage> coverage;
  Array16Of<Offset24To<GlyphRuleSet24>> ruleSets;
};

struct GlyphRuleSet24 {
  Array16Of<Offset16To<GlyphRule24>> rules;
};

struct GlyphRule24 {
  uint16 glyphCount;
  GlyphID24 glyphs[inputGlyphCount - 1];
  uint16 seqLookupCount;
```

```
  SequenceLookupRecord seqLookupRecords[seqLookupCount];
};
struct ChainContextFormat4 {
  uint16 format; == 4
  Offset24To<Coverage> coverage;
  Array16Of<Offset24To<ChainGlyphRuleSet24>> ruleSets;
};

struct ChainGlyphRuleSet24 {
  Array16Of<Offset16To<ChainGlyphRule24>> rules;
};

struct ChainGlyphRule24 {
  uint16 backtrackGlyphCount;
  GlyphID24 backtrackGlyphs[backtrackGlyphCount];
  uint16 inputGlyphCount;
  GlyphID24 inputGlyphs[inputGlyphCount - 1];
  uint16 lookaheadGlyphCount;
  GlyphID24 lookaheadGlyphs[lookaheadGlyphCount];
  uint16 seqLookupCount;
  SequenceLookupRecord seqLookupRecords[seqLookupCount];
};
```

Add Context and ChainContext format 5 that parallels format 2 for offset-overflow alleviation:

```
struct ContextFormat5 {
  uint16 format; == 5
  Offset24To<Coverage> coverage;
  Offset24To<ClassDef> classDef;
  Array16Of<Offset24To<ClassRuleSet>> ruleSets;
};
struct ChainContextFormat5 {
  uint16 format; == 5
```

```
  Offset24To<Coverage> coverage;

  Offset24To<ClassDef> backtrackClassDef;

  Offset24To<ClassDef> inputClassDef;

  Offset24To<ClassDef> lookaheadClassDef;

  Array16Of<Offset24To<ClassRuleSet>> ruleSets;

};
```

The RuleSet and ChainRuleSet are *not* extended, because they are class-based, not glyph-based, so no extension is necessary.

Format 3 (Coverage-based format) is *not* extended, because it only encodes one rule, so overflows are unlikely.