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 62937
Antalya, TR – April 2023

Title: Proposed improvements and new functionality for the Open Font Format standard for global font support.

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

1. Contents

Table of Contents

2. Introduction

The changes proposed in this document improve support for languages requiring large numbers of glyphs; enable more space-efficient glyph representations; improve variable font functionality; enable greater leveraging of composite glyphs.

3. The `glyf` table: Support Cubic Bézier Curves

The glyf data table (`glyf`) stores the glyph outline data. The version number for this table is stored in the head table's `glyphDataFormat` field. Before this change, the only defined version of the `glyf` table was version 0.

We add glyph version 1.

In `glyf` table format 0, glyph outlines use quadratic Bezier curve segments.

In `glyf` table format 1, glyph outlines can mix quadratic and cubic Bezier curve segments.

Note: since format 1 extends format 0, any format 0 `glyf` table is also a valid Format 1 `glyf` table.

3.1. Specification Change

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

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) *must* be even.

All the off-curve points between two on-curve points (with wrap-around) *must* either have the CUBIC flag clear, or have the CUBIC flag set.

The CUBIC flag *must* only be used on off-curve points. It is *reserved* and *must* be set to zero for on-curve points.

3.2. Processing

Every successive two off-curve points that have the CUBIC bit set define a cubic Bezier segment. Within any consecutive set of cubic off-curve points within a contour (with wrap-around), an implied on-curve point is inserted 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-Bezier, and implied on-curve points are inserted between the every second point and the next one as usual.

As in the glyf version 0 table, an implied on-curve point is inserted between any two neighboring quadratic off-curve points.

4. The glyf table: Variable Composites / Components

In glyf table format 0, there exist two types of glyphs: Simple glyphs which have a numberOfContours of >= 0, and Composite glyphs that have a numberOfContours of < 0, with a recommended value of -1.

In glyf table format 1, the old-style Simple and Composite glyphs exist as well. The old-style Composite glyphs shall always use a numberOfContours of -1. A new

composite glyph type named `VariableComposite` is introduced which uses `numberOfContours` of -2. A `VariableComposite` glyph has the ability to refer to *variable components*.

4.1. 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.

4.2. 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

4.3. 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)
```

4.3.1. Variable Component Flags

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

4.4. Processing

Variations of composite glyphs are processed this way: 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 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: (TranslateX,TranslateY), (Rotation,0), (ScaleX,ScaleY), (SkewX,SkewY), (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 or 4 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. For example, if only flag 3 is set (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. 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. For example, if the font variations have wght=.25 (normalized), and 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 is set, then the value of wght=0 (default) will be used. If flag bit 14 is clear, wght=.5 (from current glyph) will be used.

Note: While it is the (undocumented?) 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 / avar tables but *would* have the 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.

5. Increased capacity: support more than 65535 glyphs in a font.

This change extends the number of glyphs in the font from 65,535 (16 bits) to 16,777,216 (24 bits).

The number of glyphs in the font is encoded in the maxp table and is limited to 65535. In this proposal, the number of glyphs in the font will be calculated based on the length of the loca table. Moreover, tables that encode glyph indices as two-byte numbers are extended with versions of structures that encode 24-bit glyph indices.

While extending tables to encode 24-bit glyph indices, many tables are also extended to use 24-bit offsets instead of 16-bit offsets to allow for larger tables necessary for the larger number of glyphs.

5.1. Per-table changes

5.1.1. `maxp` table

The `numGlyphs` field of the `maxp` table for a font with more than 65,535 glyphs *should* be set to 65,535.

5.1.2. `loca` / `glyf` tables

The `loca` / `glyf` tables are not required to have the same number of glyphs as specified in the `maxp` table. The length of the `loca` table now determines the number of glyphs in the font, which can be larger than maxp.`numGlyphs`.

For a Format 0 `loca` table (`head.indexToLocFormat = 0`), the number of glyphs in the font is the length of the table divided by 2; for a Format 1 `loca` table (`head.indexToLocFormat = 1`), the number of glyphs in the font is the length of the table divided by 4.

5.1.2.1. Composite glyphs

Add a new flag to allow encoding 24bit glyph indices in composite glyphs:

```
enum ComponentGlyphFlags
    ARG_1_AND_2_ARE_WORDS       = 0x0001,
    ARGS_ARE_XY_VALUES          = 0x0002,
    ROUND_XY_TO_GRID            = 0x0004,
    WE_HAVE_A_SCALE             = 0x0008,
    MORE_COMPONENTS             = 0x0020,
    WE_HAVE_AN_X_AND_Y_SCALE    = 0x0040,
    WE_HAVE_A_TWO_BY_TWO        = 0x0080,
    WE_HAVE_INSTRUCTIONS        = 0x0100,
    USE_MY_METRICS              = 0x0200,
    OVERLAP_COMPOUND            = 0x0400,
    SCALED_COMPONENT_OFFSET     = 0x0800,
    UNSCALED_COMPONENT_OFFSET   = 0x1000,
    GID_IS_24BIT                = 0x2000
};
```

5.1.3. `gvar` table

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	glyphVariationDataOffsets[glyph	Offsets from the start of the

Type	Name	Description
Offset32	Count+1]	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.

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

5.1.4. `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.

5.1.5. `hmtx` / `vmtx` tables

Below we discuss `hmtx`. The case of `vmtx` is similar.

Currently the `hmtx` table length is determined by `hhea`/`maxp` in the following way:

- `hhea.metricDataFormat` is required to be 0 for the current format;
- `hmtx` is required to be `2 * hhea.numberOfHMetrics + 2 * maxp.numGlyphs` bytes long.

We describe how we to relax the second requirement, to allow encoding advance width of arbitrary number of glyphs in the `hmtx` table, regardless of the value of `maxp.numGlyphs`.

This is how an advance width is assigned to glyph indices beyond `maxp.numGlyphs`:

Let B be the excess bytes at the end of the `hmtx` table beyond the `2 * hhea.numberOfHMetrics + 2 * maxp.numGlyphs` bytes.

- If the length of B is odd, ignore the last byte of B.

- If B is empty, use the last advanceWidth in the `hmtx` table for all extra glyphs.
- Treat B as an array of `uint16` advance-width numbers of glyph indices starting at `maxp.numGlyphs`. For any glyph index that is in range in the font but out of range in this array, use the last item of the array.

No `lsb` (leftside-bearing) is encoded for glyphs beyond `maxp.numGlyphs`.

5.1.5.1. `lsb` offsetting of glyphs

It is an undocumented behavior of OpenType that glyphs from the `glyf` table are shifted by `lsb - xMin` when rasterized. Since our extension to the `hmtx` table does *not* encode `lsb` for new gids, no shifting is done for such glyphs. In other words, the `lsb` of such glyphs is assumed to be the same as `xMin`.

5.1.5.2. `tsb` and vertical glyphs

The vertical origin of glyphs in the `glyf` table is computed based on `tsb` encoded in the `vmtx` table. Since our extension to the `vmtx` table does not encode `tsb` for new glyphs, the `VORG` table *must* be used to encode vertical glyph origins instead.

5.1.6. `HVAR` / `VVAR` tables

No changes needed.

5.1.7. `VORG` table

Currently the length of the `VORG` table is determined from its major/minor version and the subsequent content. The only currently defined version 1.0 can only encode 16bit glyph indices. Here is how we expand the table's definition to encode glyph vertical origin for glyph indices beyond `maxp.numGlyphs`:

Derive the length of the table from the version number and the table struct, then derive the excess bytes, then follow the same algorithm for deriving advance-width in the hmtx/vmtx expansion and use those numbers as vertical origin instead.

5.1.8. `COLR` table

No changes needed.

5.1.9. `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`.

5.1.10. GDEF / GSUB / GPOS tables

5.1.10.1. 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.

5.1.10.2. 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

5.1.10.3. 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

5.1.10.4. 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>;
```

5.1.10.5. `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;
};
```

5.1.10.6. `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;
};
```

5.1.10.7. `GSUB` `MultipleSubst` / `AlternateSubst` format 2

Format 2 is introduced to enable gid24:

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

typedef Array16Of<GlyphID24> Sequence24;

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

typedef Array16Of<GlyphID24> AlternateSet24;
```

5.1.10.8. `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];
};
```

5.1.10.9. `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];
};
```

5.1.10.10. 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;
};
```

5.1.10.11. 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.

6. The `avar` Table: Axis Variations Table Version 2

The axis variations table (`avar`) is an optional table used in variable fonts. Version 1 of `avar` modifies aspects of how a design varies for different instances along a particular design-variation axis. It does this by piecewise linear remapping on a per-axis basis, with certain restrictions. See the OpenType specification for details.

Version 2, as proposed here, enables flexible axis remapping where each design-variation axis is modified according to the coordinates of multiple design-variation axes. In order to combine the effects of multiple input values, `avar` version 2 uses the OpenType variation mechanism itself to determine interpolated delta values and add them to design-variation axis coordinates. The final interpolated axis coordinates are used in all subsequent variation operations.

The efficiency of `avar` version 2 enables fonts that:

- are significantly smaller;
- are easier for users to control;
- simplify source maintenance by avoiding redundant data.

Use cases include:

- warped variable font designspaces to reflect a typeface designer's intention accurately;
- parametric fonts with intuitive control methods and a much reduced data footprint;
- offering simpler methods of control for specialized variable fonts.

6.1. The avar version 2.0 header format

Type	Name	Description
uint16	majorVersion	Major version number of the axis variations table — set to 2.
uint16	minorVersion	Minor version number of the axis variations table — set to 0.
uint16	<reserved>	Permanently reserved; set to zero.
uint16	axisSegmentMapCount	The number of *axisSegmentMap*s for this font. If this is not 0, it must be the same as *axisCount* in the 'fvar' table.
SegmentMaps	axisSegmentMaps[axisSegmentMapCount]	The segment maps array — one segment map for each axis, in the order of axes specified in the 'fvar' table.
Offset32To<DeltaSetIndexMap>	axisIndexMapOffset	Offset from beginning of the table to axisIndexMap.
Offset32To<ItemVariationStore>	varStoreOffset	Offset from beginning of the table to varStore.

The table format for avar version 2 is the same as avar version 1, appended with offsets to two extra structures, axisIndexMap and varStore, and with the data for those structures.

axisIndexMap is a *DeltaSetIndexMap* structure that maps the axis indices implied in fvar to indices used in varStore. The outer index identifies an *ItemVariationData* structure in varStore. The inner index identifies a *deltaSet* within an *ItemVariationData*.

varStore is an *ItemVariationStore* structure that points to a *VariationRegions* array and a list of *ItemVariationData* structures. Each *ItemVariationData* specifies a subset of *VariationRegions* and an array of *deltaSets*. Each *deltaSet* specifies a delta value for each region.

Delta values are typically in (but not limited to) the range [-1.0, 1.0].

Delta values are stored as if they were signed integers by multiplying their true value by 16384. Thus 1.0 is stored as 16384; -1.0 is stored as -16384.

The *DeltaSetIndexMap* and *ItemVariationStore* formats are given in OpenType Font Variations Common Table Formats.

6.2. Processing

Processing of axis values in an avar version 2 table happens in 3 stages for a given

instance:

1. Initial normalization to convert user coordinates of each axis to initial normalized coordinates in the range [-1.0, 1.0].
2. Remap initial normalized coordinates of each axis via the *axisSegmentMap*s of `avar` version 1, providing intermediate coordinates also in the range [-1.0, 1.0].
3. Calculate interpolated deltas for each axis via `avar` version 2 and add them to intermediate coordinates, providing final coordinates that are clamped to the range [-1,1].

In more detail, step 3 proceeds as follows. Considering the coordinates provided by step 2, a scalar is determined for each *variationRegion* by the usual variation interpolation algorithm. Then, the *deltaSet*s of each *ItemVariationData* are processed such that each delta value is multiplied by its associated scalar. Summing those products gives an interpolated delta to add to a particular axis coordinate, the axis index being defined in `axisIndexMap`. After the summing operations for all *ItemVariationData* structures is complete, axis values are clamped to to the range [-1.0, 1.0], giving final axis coordinates.

The final axis coordinates obtained by step 3 are subsequently used in the standard variation process described in Algorithm for Interpolation of Instance Values, applying to all `gvar` data and all `ItemVariationStore` data elsewhere in the font.

The following algorithm implements step 3 above, producing the final normalized axis coordinates:

```
// let coords be the vector of current normalized coordinates.

std::vector<int> out;
for (unsigned i = 0; i < coords.size(); i++)
{
  int v = coords[i];
  uint32_t varidx = i;

  if (axisIdxMap != 0)
    varidx = (this+axisIdxMap).map(varidx);

  float delta = 0;

  if (varStore != 0)
    delta = (this+varStore).get_delta (varidx, coords);

  v += std::roundf (delta);
  v = std::clamp (v, -(1<<14), +(1<<14));

  out.push_back (v);
}
for (unsigned i = 0; i < coords.size(); i++)
  coords[i] = out[i];
```

6.3. Construction

The way `ItemVariationStores` are built is typically by using a variation modeler, that takes a series of *master* values at certain locations in the design-space, and produces `VariationRegions` and `deltaSets` to be stored in one or more `ItemVariationData` structures. This usage is identical, except that delta values apply to normalized axis coordinates rather than distances measured in font units. To build the `avar` version 2 mapping tables, the designer will need to produce a mapping of input axis locations and their respective output axis locations. This data then will constitute the set of *masters* to be fed to the variation modeler and populate the `ItemVariationStore` that will go into the `avar` version 2 table. The variation index for each axis will be stored in `axisIndexMap`.

6.4. Use cases

6.4.1. Designspace warping

In the OpenType variations specification, registered axes offer a standard way to offer users control of a variable font on reasonably well defined scales. For example, users learn that the Regular version of a font has Weight=400, Width=100, while the Bold has wght=700, wdth=100. A typical 2-axis variable font with `wght` and `wdth` axes might have the following nine Named Instances:

	wdth=75	**wdth=100**	**wdth=125**
wght=300	Light Condensed	Light	Light Extended
wght=400	Condensed	Regular	Extended
wght=700	Bold Condensed	Bold	Bold Extended

However, in fonts with more than one design axis, this approach lacks the flexibility of older methods of interpolation, where the type designer used a font design application to specify freely the axis values for, say, the Bold Condensed. Notably, the `wght` coordinate of Bold Condensed did not need to match the `wght` coordinate of Bold, nor did the `wdth` coordinate of Bold Condensed need to match the `wdth` coordinate of Condensed. A Bold Condensed with `wght`,`wdth` of (677,81) rather than (700,75) was perfectly reasonable. Our grid of instances in such as case would look something like this:

	Condensed	**Regular**	**Expanded**
Light	300,75	300,100	300,115
Regular	400,75	400,100	400,120
Bold	677,81	695,100	700,125

While with OpenType 1.8 it is possible to set up a designspace like this, it has the significant drawback that many applications and systems expect all Bold weights to be at `wght`=700 and all Condensed weights to be at `wdth`=75, whatever the values of other axes. In practice, many OpenType 1.8 fonts encode numerous additional masters that ensure the design is as intended at all designspace locations. This not only increases the data footprint significantly, but also adds to the maintenance burden of the font.

Furthermore, there are many cases where a type designer does not intend to offer instances in certain regions of the design space. For example, a Black Condensed instance is often difficult to design, and many type families omit it. Desired behaviour, in case a user requests `wght`=900, `wdth`=75, may be for the font to provide the same instance for the Black Condensed as it does for the Bold Condensed.

By applying deltas to axis values anywhere in the designspace, the `avar` table version 2 “warps” the designspace and so resolves the issues described.

As with variation deltas in general, we also benefit from interpolation when axis values are influenced by a delta but are not exactly at the location of the delta. Each delta value is encoded as F2DOT14. Its interpolated value is determined by the standard OpenType variations algorithm, then added to the value obtained from the standard normalization process. Thus, assuming the Bold Condensed instance at (700,75) has normalized coordinates (1,-1) and assuming the designspace location (677,81) has normalized coordinates ((677-400)/(700-400), -1 + (81-75)/(100-75)) = (0.9233,-0.76), then the `ItemVariationStore` needs a delta set with peaks at (-1,1), thus with region ((-1,-1,0),(0,1,1)) and the following delta values:

- `wght` -0.0767
- `wdth` +0.24

In practice there may be other delta sets required at the partial default locations (-1,0) and (0,1), which then influence the delta values required at (-1,1), in line with normal variations math.

The result of implementing designspace warping this way is that we preserve the original axis values where applications and systems expect them, namely all Bold instances at `wght`=700 and all Condensed weights at `wdth`=75, while at the same time minimizing the data footprint.

6.4.2. Duplication of axis values for non-linear interpolation

Certain variable fonts encode delta sets in such a way that, when multiple axes are synchronized, outline points move in curves rather than in straight lines. This

technique has been referred to as HOI (higher order interpolation). The details of the non-linear encoding need not concern us here, but a practical drawback under OpenType 1.8 is that it requires users to set multiple design axes to identical values in order to obtain valid instances. When axes are not synchronized, the resulting glyphs are severely distorted and useless. Synchronization using JavaScript has been proposed as a solution. A clever exploit of the CFF2 format, allowing a single axis to perform all the necessary non-linear adjustment of outline points, has also been demonstrated (but this method cannot be adapted to adjust delta values in an *ItemVariationStore* non-linearly).

The `avar` version 2 table provides a solution to the synchonization problem, such that a user adjusts one “primary” axis and that value is cloned to other subordinate axes. Ideally the subordinate axes have the Hidden flag set in `fvar` to discourage manual operation. For a font with 3 axes requiring identical values, if the first axis is considered primary, then the other two can encode delta sets in `avar`’s *ItemVariationStore* to clone the value of the first axis. Assuming the subordinate axes have the same minimum, default and maximum values as the primary axis, each subordinate axis requires either one or two delta sets to clone the negative and positive regions of the normalized value of the primary axis:

- If the axes use only the normalized region [0,1] then one delta set per subordinate axis is needed, referring to the region (0,1,1) of the primary axis and having the delta value of 1. This is the common case.

- If the axes use only the normalized region [-1,0] then one delta set per subordinate axis is needed, referring to the region (-1,-1,0) of the primary axis and having the delta value of -1.

- If the axes use both the normalized regions [-1,0] and [0,1] then two delta sets per subordinate axis are needed, one referring to the region (0,1,1) of the primary axis and having the delta value of 1, the other referring to the region (-1,-1,0) of the primary axis and having the delta value of -1.

6.4.3. Simplified controls for parametric fonts without redundant data

Parametric fonts, explored by Donald Knuth in Metafont and by others, offer fine-tuned control of specific font-wide aspects of their design. Such aspects typically include the thickness of vertical strokes, thickness of horizontal strokes, the overall width of the font, and direct control over vertical zones including cap-height, ascender height, x-height and descender height.

In an OpenType variable font these parameters can be encoded as design-variation

axes. The Type Network Variations Proposal[1] of 2017 provides the basis for parametric OpenType fonts Amstelvar, Roboto Flex and others, demonstrating the potential for usable parametric fonts with large character sets. That flexibility comes from a designspace of many axes – sometimes 10 or more – which raises issues of control. Faced with 10 sliders, users are unlikely to be able to find the instance they want and are likely to get "lost" in a unusable region of the designspace.

In OpenType 1.8, the solution to getting "lost" is to include "blended" axes that combine the deltas of parametric axes in such a way that they function in exactly the same way as they would in a non-parametric font. The blended axes are exposed using the registered axes of `wght`, `wdth`, `opsz`, etc., or well specified custom axes, and they are encoded in the standard OpenType 1.8 manner using `gvar` and related data. The major problem with such fonts is their large data footprint, as the blended axes effectively replicate much of what the parametric axes do. For a font containing both parametric and blended axes, the problem is particularly acute, so the parametric axes may be omitted to save space — thereby removing core functionality. In such cases the parametric nature of the font is thus reduced to a convenience of sources.

The `avar` table version 2 provides a method to deploy a purely parametric font with user-friendly axes, while avoiding the data bloat of blended axes. The user axes, for example `wght` and `wdth`, on their own do nothing (i.e. they have no `gvar` or similar data), but they control the parametric axes by means of `avar`. The blending of parametric values into user-facing values effectively happens live in the font engine. The parametric axes are expected to have the Hidden flag set in `fvar` to discourage (but not block) manual operation.

6.5. Inverse avar processing

Normally it is not possible for an application to obtain normalized axis values directly, these remaining private to the font engine. In fonts with an `avar` version 2 table, however, it can be useful to know the axis values that would invoke a given instance if the font engine did not support `avar` version 2. Use cases include:

- informing users of the effective settings of parametric axes;
- providing a polyfill for `avar` version 2.

The solution is for the application itself to implement the `avar` algorithms, both version 1 and version 2. This requires not only access to the binary `avar` table, but also knowledge of the font's axis extents as defined in the `fvar` table. Once an application has a complete set of final normalized values as defined above, the inverse of the `avar` version 1 algorithm can be applied to obtain user coordinates for

1 https://variationsguide.typenetwork.com/

all axes. (Care must be taken to avoid a divide-by-zero error in implementing an inverse `avar` version 1 mapping, in the case where consecutive `toCoordinate` values are identical.)

Almost all fonts can provide a valid set of user axis coordinates for a given set of final normalized axis coordinates in this way. Exceptions are those fonts that encode deltas in a region that is not accessible without `avar` version 2, namely those fonts where the `fvar` definition of an axis normally precludes an active negative region or a active positive region by having its default equal to its minimum or its maximum respectively. Such cases are expected to be rare.

6.6. Recommendations and notes

6.6.1. Hidden axes

Since many `avar` version 2 fonts have axes not not intended for manual adjustment, it is recommended that such axes set the “hidden” flag in `VariationAxisRecord` of the `fvar` table.

6.6.2. Efficient axisIdxMap and ItemVariationData construction

In order to reduce proliferation of zero deltas, it is recommended to store in `axisIndexMap` only those axes that are remapped by `avar` version 2. For the same reason, if there are multiple different types of axis mapping affecting different sets of axes, consider using multiple `ItemVariationData` structures.

6.6.3. Other notes

- The set of axes involved in adjusting the coordinate for a given axis may include the axis itself.

- It is expected that implementations that handle only `avar` version 1 will ignore the entire table by rejecting the `majorVersion` value.