Approx. Page 26:

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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bits | | | | 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 single following byte is added. |
| 1 | 1 | 0 | (any value) | Two bytes follow; the low 5 bits of the first are shifted left by 16 bits and added, then the following byte is shifted left by 8 bits and added (bitwise or), and then the second following byte is or’d as-is |
| 1 | 1 | 1 | 0 | Three bytes follow; the low 4 bits of the first become the most significant bits of the new value, as above |
| 1 | 1 | 1 | 1 | Four bytes follow; the low 4 bits of the first become the most significant bits of the new value, as above |

Page 41 (approx) maxp

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 numGlyphs in 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. |

Page 140 or so, GLYF—Glyph data

#### Table structure

The GLYF 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].

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. The 'glyf' table, however, can contain no more than 65535 entries, and some flags and features specified in this subclause are not supported.

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 ‘GLYF’ table present.

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.

Several new tables have been defined to support larger glyph repertoire. Most of these new tables have table tags specified using capital letters that are matching their already existing counterparts – a set of tables with tags defined using lowercase letters. When these new tables are present, the GLYF-aware software shall ignore their lowercase tag counterparts.

**Liam suggests:**

Several tables, such as LOCA, exist to support fonts that can contain more than 65535 glyphs. Most of these tables have names in upper case, and replace a similar table whose name is in lower case. When such tables are present in a font with upper-case names, GLYF-aware software shall ignore any corresponding lower-case named tables that may also be present.

On page 150, 5.2.9 LOCA—Index to location

The 'LOCA' table format is similar to 'loca' (see 5.2.5), except that the entries in the 'loca' table point into the 'glyf' table while the entries in the LOCA table point into the GLYF table. If both 'LOCA' and 'loca' tables are present, and if the font engine software is capable of processing new GLYF and LOCA tables – the 'loca' table shall be ignored. The 'loca' table is used by older software that cannot process ‘LOCA’.

Liam: I don’t think it hurts to state it—the idea was to try and help tool writers understand what’s going on. But also i’m Ok with it going away.

(still on page 150)

The LOCA table is an array of n offsets, where n is the number of glyphs in the font plus one. The number of entries in LOCA is determined by dividing the length of the LOCA table (specified by the corresponding TableRecord in Table Directory, see 4.5.2) by two or by four, depending on the format determined by the *indexToLocFormat* field in the 'head' table (5.1.3):

* For *indexToLocFormat = 0*, the number of entries is in the LOCA table is determined by the length of the table divided by 2.
* For *indexToLocFormat = 1*, the number of entries is in the LOCA table is determined by the length of the table divided by 4.

NOTE 1 The format of both LOCA and 'loca' is determined by the same value of *indexToLocFormat* in the font header. If both tables are present, they must be encoded in the same format.

NOTE 2 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].

Offsets shall 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 shall 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].

Page 599 (numbered 579)

Format 5, a more flexible format than Format 4, describes class-based chaining context substitution. It is identical to Format 2, except that it was modified to support extended glyph repertoire.

Liam: should read, it was modified to support an extended glyph repertoire.

Page 827

#### MultiItem Variation Store

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 64K bytes.

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 table

|  |  |  |
| --- | --- | --- |
| Type | Name | Description |
| uint16 | format | Set to 1. |
| Offset32 | sparseVariationRegionListOffset | Offset to the SparseVariationRegionList, from the start of the MultiItemVariationStore |
| uint16 | multiItemVariationDataCount | The number of MultiItemVariationData tables. |
| Offset32 | multiItemVariationDataOffsets [multiItemVariationDataCount] | Array of offsets to MultiItemVariationData tables, from the start of the MultiItemVariationStore. |

SparseVariationRegionList

|  |  |  |
| --- | --- | --- |
| Type | Name | Description |
| uint16 | regionCount | The number of regions. |
| Offset32 | variationRegionOffsets[regionCount] | Array ofoffsets to SparseVariationRegion records, measured from the start of the SparseVariationRegionList. |

SparseVariationRegion

|  |  |  |
| --- | --- | --- |
| Type | Name | Description |
| uint16 | regionAxisCount | The number of axes for the region. |
| Offset32 | sparseRegionAxisCoordinates[regionAxisCount] | Array of offsets to SparseRegionAxisCoordinates records, measured from the start of the SparseVariationRegion. |

SparseRegionAxisCoordinates

|  |  |  |
| --- | --- | --- |
| Type | Name | Description |
| uint16 | axisIndex | Zero-base index of the axis this record applies to. |
| F2DOT14 | startCoord | The region start coordinate value for the axis |
| F2DOT14 | peakCoord | The region peak coordinate value for the axis |
| F2DOT14 | endCoord | The region end coordinate value for the axis. |

MultiItemVariationData table

|  |  |  |
| --- | --- | --- |
| Type | Name | Description |
| uint16 | format | Set to 1. |
| uint16 | regionIndexCount | The number of variation regions defined in this subtable. |
| uint16 | regionIndices [regionIndexCount] | Array of indices into ... Array of indices into the sparse variation region list for the regions referenced by this item variation data table. |
| Offset32 | deltaSets | Offset to CFF2-style INDEX array of TupleValues, measured from the start of the MultiItemVariationData table. |

The deltaSets field in a MultiItemVariationData table is a CFF2-style INDEX structure, which stores the delta-sets for a single tuple within each of its slots, addressed by the “inner” index of the VarIdx (see MultiItemVariationStore above), whereas a MultiItemVariationData table itself represents the data for all values sharing the same “outer” index. The tuple data encoded within each entry of the deltaSets INDEX object are each encoded as a TupleValues structure.

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

The CFF2 INDEX format used in VARC (and hence in MultiItemVariationData subtables) is a variable-sized INDEX structure, as defined for the CFF2 table in section 5.3.3.5.2 (INDEX Data). A CFF2-style INDEX is used to store a list of variable-sized data, such as glyph records, in a space-efficient manner.

NOTE This is the INDEX structure defined for CFF2, and not the INDEX structure defined for CFF. The count field of the CFF2 Index structure is a 32-bit value (uint32), unlike the corresponding count field in the CFF table.

Page 893 (numbered 873) 7.3.9.1 Glyph variations table format

The GVAR glyph variation table duplicates the functionality of the 'gvar' table (7.3.4) enabling support for extended glyph repertoire defined by the GLYF table (5.2.8). As its predecessor, 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.

Liam suggests:

The GVAR glyph variation table duplicates the functionality of the 'gvar' table (7.3.4) enabling support for extended glyph repertoire defined by the GLYF table (5.2.8). As with ‘gvar’, the glyph variations table has a header followed by GlyphVariationData subtables for each glyph that describe the ways that each glyph is transformed across the font’s variation space.

'GVAR' header

The glyph variations table header format is as follows:

‘GVAR’ header

|  |  |  |
| --- | --- | --- |
| Type | Name | Description |
| uint16 | majorVersion | Major version number of the glyph variations table – set to 1. |
| uint16 | minorVersion | Minor version number of the glyph variations table – set to 0. |
| uint16 | axisCount | The number of variation axes for this font. If the 'fvar' table is present, axisCount shall 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. |
| uint24 | glyphCount | The number of glyphs in the font, equal to numGlyphs from MAXP. |
| 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 major difference between GVAR and 'gvar' tables is that glyphCount is a uint32 in GVAR, allowing for more than 64K of glyphs to be stored in a font.

(section starting on page 896, numbered 876)

#### Processing the ‘GVAR’ table

This note refers to hmtx and should refer to HMTX, or to HTMX or html data.

NOTE When a composite glyph has a component with the USE\_MY\_METRICS flag set, the ‘hmtx’ and 'HVAR' data for the composite glyph are used in the same manner in which the ‘hmtx’ data would be used for a non-variable font. The ‘hmtx’ and 'HVAR' data should be set to appropriate values for the composite glyph, though the hinted phantom point positions may not exactly match the linearly-scaled metrics obtained from the ‘hmtx’ and 'HVAR' data.