WOFF Ultra Condensed file format

Background

This document describes the wire format for a new compressed font file format. The primary goal is to achieve significantly better compression than gzip (which is the compression standardized in WOFF), while exactly preserving the semantics of WOFF and the underlying TrueType files.

Additional goals include keeping the format simple, quick to compute, and modest in memory requirements. All these help with end-to-end performance, especially on mobile devices.

The format is based on WOFF, but departs from it when there are clear performance benefits.

Many of the ideas (especially the preprocessing of the “glyf” table) are based on MicroType Express, which was documented in a submission to the W3C [MTX].

The working name for this format is Webfonts Ultra Condensed, and is subject to change as the project matures.

Overall File Structure

The structure of a WOFF Ultra Condensed file is as follows:

WebfontsHeader
TableDirectory
Streams (containing Font Tables)
ExtendedMetadata
PrivateData

Webfonts Header

This is identical to the WOFF spec in syntax and meaning, with the exception that the signature is 0x774F4632 ‘wOF2’.

Table Directory

The table directory is an array of table directory entries. Its offset is constant because it immediately follows the header, which is of constant size. Its size is calculated by multiplying numTables in the header times the size of a single entry.

UInt32	tag	4-byte sfnt table identifier
UInt32	flags	flags, as interpreted below
UInt32	compLength	Length of the compressed data, or 0 if not new stream
UInt32	transformLength	Length of the uncompressed data
UInt32	origLength	Length of the final uncompressed table, excluding padding

The flags are interpreted as follows:

bits 0..3	compressionType	Compression type, see below
bit 4	continueStream	1 if there isn’t a new compressed stream

bit 5 applyTransform 1 if there is an additional transform to be applied

Note that offsets are not encoded explicitly, as the compressed streams are stored in sequence without padding. Thus, the offset of a stream is equal to the sum of the sizes of the Webfonts Header and the Table Directory, plus the sum of all compLength fields for preceding streams.

Unlike WOFF, the sfnt tags are not required to be sorted in ascending order. Rather, they reflect the physical order of tables within the font. The decoding process should sort the sfnt table of the reconstructed font in order of ascending sfnt tag.

Continue streams

WOFF requires that each compressed table be stored in its own separate stream. A goal of this format is to offer more flexibility. The same mode as WOFF is allowed, as is the mode of combining all tables into a single compressed stream, or a mixture. The mechanism is “continue streams.”

When multiple tables are combined into a single stream, it is coded as follows:

The table directory entry for the first table has the continueStream flag set to 0. The compLength is the length of the compressed stream containing all tables.

The table directory entry for successive tables all have the continueStream flag set to 1, and a compLength of 0.

Three values are specified for compressionType:

0	None	No compression, output is identical to input
1	Zlib	ZLIB Compressed Data Format, see [ZLIB]
2	Lzma	LZMA, see [LZMA]

Values out of this range are invalid.

[Musing: assuming a typical font has 16 tables, this is 320 bytes for the entries, plus 44 for the WOFF2 header, or 364 bytes. For a tiny subset (one containing only a few glyphs), that’s nontrivial overhead. One possibility to consider is a more compact representation of the headers.]

Table-specific transforms

In some cases, the TrueType tables contain significant redundancy. For these, this file format defines transforms that strip out the redundancy, and then allow the table to be reconstituted. The main table is ‘glyf’, and is fully documented, but applying transforms to other tables is under consideration as well.

When the glyf table is transformed, both the ‘glyf’ and ‘loca’ tables should be listed in the directory with the applyTransform flag set. The transformSize of the loca table should be zero - the data in the transformed ‘glyf’ table will be used to reconstruct both glyf and loca tables.

The ‘glyf’ table

When the applyTransform flag is set on the ‘glyf’ table, the contents of the glyf table are transformed using an algorithm designed to optimize the compressibility of the resulting stream. This transform is based on the one in [MTX], but has been updated to achieve even more performance.

The transformed glyf table consists of seven substreams. There is a header consisting of a version, the number of glyphs, and the sizes of each of the substreams, then the data of the substreams follow.

Fixed version // table version number = 0x00000000
USHORT numGlyphs
USHORT indexFormat
ULONG nContourStreamSize
ULONG nPointsStreamSize
ULONG flagStreamSize
ULONG glyphStreamSize
ULONG compositeStreamSize
ULONG bboxStreamSize
ULONG instructionStreamSize

The individual glyphs are interleaved across all these streams. Thus, each stream contains some number of bytes for glyph 0, followed by some number of bytes for glyph 1, etc. The reconstruction process is defined in terms of reading from the various streams.

Glyph reconstruction process

1. Read a SHORT from the nContourStream. This is numberOfContours for the glyph. As defined by the TrueType standard, if it is -1, the glyph is composite. If it is zero, then the glyph is empty and no further processing is required. If it is greater than one, then it is a simple glyph and the value is the number of contours.

[Change being considered: store the 255USHORT - as defined by 6.1.1 of the MTX spec of (numberOfContours+1). In this encoding, the number of contours will almost always be a single byte]

For a simple glyph, the process continues as follows:

2. Read numberOfContours 255USHORT values from the nPointsStream. Each of these is the number of points of that contour. The endPtsOfContours[] array can be reconstructed as the cumulative sum of the number of points, less one. The total number of points in the glyph (nPoints) is the sum of all these values.

3. Read nPoints BYTE values from flagStream. The interpretation of these values is as described as section 5.11 of the MTX spec. Each such flag byte corresponds to one point in the

glyph. Each such flag byte also indicates a count of 1 to 4 triplet bytes, as indicated in the table in section 5.11. For completeness, the rule is that a value of (flag&0x7f) in the range 0 to 83 inclusive indicates 1 triplet byte, in the range 84 to 119 inclusive indicates 2 triplet bytes, in the range 120 to 123 inclusive indicates 3 triplet bytes, and 124 to 127 inclusive indicates 4 triplet bytes.

4. For each point, read the indicated number of triplet bytes from glyphStream. The interpretation of these bytes is as described in section 5.11 of the MTX spec, and the result of decoding them is a (delta) X and Y coordinate for each point.

5. Read one 255USHORT value from glyphStream, which is instructionLength, the number of instruction bytes.

6. Read instructionLength bytes from instructionStream, which are to be stored into “instructions” in the reconstructed glyph.

[Change being considered: separate push sequence from main instruction stream. This probably a good idea, but not implemented by the current C++ reconstruction code]

For a composite glyph (ie numberOfContours = -1), the reconstruction process is as follows:

2a. Read a USHORT from compositeStream. This is interpreted as a component flag word as in the TrueType spec. Based on the actual flag values, there are between 4 and 14 additional argument bytes (glyph index, arg1, arg2, and optional scale or affine matrix).

3a. Read the number of argument bytes as determined in step 2a from compositeStream. If the flag word has FLAG_MORE_COMPONENTS set (1 << 5), then go back to step 2a.

4a. (The last flag word read does not have FLAG_MORE_COMPONENTS set). If any of the flag words had FLAG_WE_HAVE_INSTRUCTIONS set (1 << 8), then read the instruction stream as described by steps 5 and 6 above (in the description of simple glyph processing).

Glyph size issues

One invariant guaranteed by the producer of the file format is that the origLength in the header entry for the ‘glyf’ table is big enough to accommodate the reconstructed glyf table. Since there are multiple valid reconstructions of a glyph, this section describes the nominal size. In practice, nearly all source fonts will be the same size (or possibly larger), but since that’s not guaranteed it’s probably best for the encoder to specify an origLength based on the nominal size, rather than actual size in the source font.

To compute the nominal size, assume:

- The flags for each point are set for minimal coordinate data (ie same as last when the

delta is zero, and signed byte when the absolute value is less than 256)

- When successive flags have the same value, repeat is used
- Glyph data is padded to 4 bytes

The origLength for the glyf table must be at least large enough to accommodate this. If not, the font may be rejected.

Bounding boxes

Each glyph can optionally have an explicitly specified ounding box. Since a bounding box takes 8 bytes, it should be omitted where possible. However, reconstructing a bounding box in the case of arbitrary glyph transformations is non-trivial. Further, it is possible for the source font to have a bounding box inconsistent with the actual glyph data. In this case, it the role of the compression algorithm to represent the font as accurately as possible, inconsistencies or no, so that the use of the font does not vary at all dependent on whether compression was applied.

The bounding box data stream is defined as follows. First, there is a bitmask, consisting of 4 * ((nGlyphs + 31) / 32) bytes. Bits are packed big-end-first. Thus, glyph 0 is represented as a value of 0x80 in the first byte, glyph 1 as 0x40, and glyph 8 as a value of 0x80 in the second byte. A bit of 1 indicates that the bbox is present, and a bit of 0 indicates its absence.

For each glyph with a 1 bit set, the bounding box is represented as 4 SHORT values, xMin, yMin, xMax, yMax, and the interpretation of these values is the same as in the header of an individual glyph.

[In the present implementation, bounding boxes must be specified for all composite glyphs. Under consideration is reconstructing bounding boxes for composite glyphs with offset-only transforms. For this reason, the bbox reconstruction is described (and implemented, in the reference code), as a separate pass rather than part of the stream processing for glyphs.]

A bounding box present for a glyph with zero contours is an error.

The ‘loca’ table

The origLength for the ‘loca’ table should be large enough to hold the reconstructed loca table. If the indexFormat is short, this means 2 * (numGlyphs + 1). If long, 4 * (numGlyphs + 1). The origLength in the ‘maxp’ table must match that in the transformed ‘glyf’ table. (Rationale for duplicating the value: the transformed table should contain sufficient information for reconstructing the final table. Otherwise, it’s much harder to do incremental processing as

[Change being considered: always use long index format, which would simplify the consistency constraints somewhat]

Design for compressed header format:

The header format above, particularly the table headers, is stylistically very similar to WOFF and SFNT. It is possible to reduce this by an estimated 200 bytes per font, which could be significant

for very small fonts (for example, fonts that contain a small subset of glyphs). Rather than a fixed 20 bytes per header, the new design uses a byte-packed compressed format.

The number of bytes in the table headers depends on the exact contents. However, it can be bounded, which facilitates an implementation which reads blocks into fixed-size buffers.

The format is as follows. The TableDirectory consists of numTables table headers, concatenated.

The first byte consists of flags and the table identifier. Bits 6-7 of the byte identify the compression type, as above (0 = none, 1 = zlib, 2 = lzma), but with a special value of 3 indicating that the table has a continue stream. Bit 5 identifies whether the table is transformed.

Bits 0-4 identify the table. The most common table types (as identified in the OpenType spec) are represented. Any table not appearing in this list is represented with a value of 31, followed by four bytes containing the tag.

0 cmap
1 head
2 hhea
3 hmtx
4 maxp
5 name
6 OS/2
7 post
8 cvt
9 fpgm
10 glyf
11 loca
12 prep
13 CFF
14 VORG
15 EBDT
16 EBLC
17 gasp
18 hdmx
19 kern
20 LTSH
21 PCLT
22 VDMX
23 vhea
24 vmtx
25 BASE
26 GDEF
27 GPOS
28 GSUB
29 [reserved - invalid]
30 [reserved - invalid]
31 arbitrary tag follows

Following the flags and table id byte (and optional tag) are a succession of length values. Each

is base-128 encoded. [Examples: 0..127 are a single byte containing the length, 0x81 0x00 is 128, etc.] 32 bits of length can be encoded into a maximum of 5 bytes. The number of length values depends on the flags.

For a compression type of ‘none’ (ie bits 6-7 = 0), a single length value follows, which is the size of the table. For other compression types

References

[ZLIB] RFC 1950 ZLIB Compressed Data Format Specification. P. Deutsch, J-L. Gailly, Editors. Internet Engineeering Task Force, May 1996.

[LZMA]

TODO: which of these is best?

The .lzma File Format

The .xz File Format

[MTX] MicroType® Express (MTX) Font Format, Sarah Martin, Al Ristow, Steve Martin, Vladimir Levantovsky, and Christopher Chapman. W3C Member Submission 5 March 2008.