NEWTON FONT SPECIFICATIONS

Document Version 0.01
© Copyright Apple Computer, Inc. 1994
All Rights Reserved

This document contains the structure of Newton fonts. This document assumes you are familiar with the structures of TrueType fonts. This document is extremely sketchy - please let us know if you have any corrections, requests, additions, concerns.

0. RELEASE NOTES

Date	Version	Changes
02-Feb-94	0.01	Initial Release

1. OVERVIEW

This is an attempt to quickly document the structure of Newton fonts. It does not attempt to document in detail all aspects of a TrueType font; Rather it will contain many pointers to existing TrueType documentation which should be used along with this document as a reference.

Based on TrueType

Newton fonts are based on the TrueType font technology. It is important to understand that TrueType is not just outline fonts, but an entire scalable font architecture. It is possible to use many aspects of TrueType's font technology, without having to use actual outline fonts. Other aspects of this technology include the ability to support non-ASCII character encodings (such as Unicode), and the ability to easily handle large characters sets (such as Kanji).

Because the existing Newton software does not actually support outline fonts, we use a new extension to TrueType known generally as "sbit". This is a set of tables which have been defined as segments of a TrueType font. These tables allow bitmap data to be included in a TrueType font.

The original design goal for "sbit" fonts was to allow small hand-tuned point sizes to be included along with an outline font. For Newton, we have gone a step further and completely removed the outline data. Fonts built for Newton contain ONLY bitmap data.

For more detailed information on TrueType, and the tables used by Newton, see The TrueType Book, chapter 6 and the appendices. For more detailed information on the sbit data structures, see the latest sbit ERS (1.0.6 as of this writing). The final arbiter of current structures, of course, remains the latest header files.

Types of Newton Fonts

There are actually two types of Newton fonts. Most commonly known are the embedded bitmap formats used for screen and dot-matrix printing. Less known, but equally important, are "widths" fonts used for accurate PostScript printing. These fonts are specially constructed to contain only high-accuracy font widths data, and must match the metrics of the fonts in various PostScript printers.

2. REQUIRED TABLES IN NEWTON FONTS

The following lists describe the tables which must be present in each type of font. No attempt will be made to describe each table, except where Newton-specific information is required - see the TrueType book for exact information.

A Newton bitmap font will contain the following tables:

```
      Font Directory
cmap  Character Mapping
head  Font Header
hhea  Horizontal Metrics Header
hmtx  Horizontal Metrics
maxp  Maximum Profile
name  Naming Table
bloc  Bitmap Locations
bdat  Bitmap Data
```

A Newton widths font will contain the following tables:

```
      Font Directory
cmap  Character Mapping
head  Font Header
hhea  Horizontal Metrics Header
hmtx  Horizontal Metrics
maxp  Maximum Profile
name  Naming Table
hsty  Horizontal Style Info
```

Please refer to the TrueType or sbit documents for basic definitions of the above tables; The following descriptions contain only that information which is important for Newton fonts.

Font Directory

Used as described.

cmap Character Mapping

Newton only supports Unicode, so only formats 4 and 6 should be used. For all but the most specialized fonts, format 4 (permitting discontiguous ranges) will be the one to use. Newton's font system can use all permutations of the delta, rangeOffset, and glyphIndex arrays, so it is recommended that font tools construct the smallest possible cmap.

head Font Header

All fields are used as described. The standard value for unitsPerEm is 2048, but this may be altered as long as all other dependent values are changed in accordance. Be sure to set macStyle if this is a styled font. The smallest embedded bitmap size should be placed in lowestRecPPEM.

hhea Horizontal Metrics Header

Most of these fields are duplicated on a per-size basis in the bloc entries. The values here should be based on the equivalent hi-precision information in the hmtx. If you are generating this data from a mac font, much of this data will be in the FOND. These numbers must be very accurate if this is a widths font.

hmtx Horizontal Metrics

If no hires font metrics are available for a bitmap font, the font tool should generate this table by scaling the highest point size of the bitmaps embedded. Otherwise, use the highest resolution font metrics available (eg from the FOND).

maxp Maximum Profile

The only field which is important here is the numGlyphs field. However, if you are embedding outline data, then the other fields must of course be accurate.

name Naming Table

All fields are used as described.

bloc Bitmap Locations

All fields are used as described. Currently, only pure black and white (bitDepth = 0, colorRef = 0) fonts are used. Typically, index subtables will always be of type 3, but an extremely large font could use type 1. However, a better way to do this would be to generate multiple subranges where the strike data for each subrange is < 64k.

bdat Bitmap Data

As described in the documentation, the bdat area is very simple - it's just a single version number, followed by hundreds or thousands of individual bitmap records, each indexed by the various subtables in the bloc. Currently, only formats 1 and 6 are supported, with format 1 preferred (because it's smaller).

hsty Horizontal Style Info

This is an all-new table which is used by Newton width fonts. It is used to mimic the way in which PostScript will modify fonts when the requested style is not directly available.

The format of an hsty table is as follows:

```
typedef struct  {
  Fixed  version;              // 1.0
  short  extraPlain;
  short  extraBold;
  short  extraItalic;
  short  extraUnderline;
  short  extraOutline;
  short  extraShadow;
  short  extraCondensed;
  short  extraExtended;
  short  extraUnused;
} sfnt_HorizontalStyleInfo;
```

The simplest way to generate this structure is to copy the values directly from the FOND, except that they must be scaled along the way into the same units as used by the hmtx.

3. OTHER REQUIREMENTS

The following lists describe the tables which must be present in each type of font. It is strongly recommended that the glyph mapping requirements described in the appendices be followed. The only exception to this is the full list of glyph codes spelled out as being required for Mac Roman fonts - they don't really matter (although a mac-oriented font tool will probably do them that way as a matter of course). However, the low numbered special glyphs should be as documented.

Summarizing the rules specified in each section:

Glyph Ordering

Glyph #	Glyph Image
0	missing character glyph
1	null glyph
2	carriage return glyph
3	space

All other glyph codes may be unique to the font.

Character to Glyph Mapping

Char #	Glyph #
0	0: missing character glyph
1	1: null glyph

```
1A                 0: missing character glyph
```

All others are abitrary.
The requirement of all mac codes to be implemented is not necessary.
The tables of missing, null, and space glyphs are recommended but not required.

Metrics

The requirements stated are required for Newton
The recommendations stated are recommended for Newton.

sbit Format

(sfnt bitmaps)
This chapter contains generic, platform independent TrueType font information.

bloc

The 'bloc' table starts out with a header; simply the version, numGlyphs, and numSizes. We refer to the bitmaps for a face at a given point size to be a strike. The 'sfnt' will typically have various strikes embedded in the 'bdat' table. For example, Hon Mincho may have 12 point, 18 point, and 24 point strikes embedded in the sfnt. Each strike will have a bitmapSizeTable associated with it.

```
typedef struct {
      Fixed             version;
      unsigned long  numSizes;
/*    bitmapSizeTable[numSizes]     */
} blocHeader;
#define BITMAP_HEADER_VERSION          0x20000
```

The numSizes in the blocHeader tell us the number of bitmapSizeSubTables that follow. Most of the information needed at transform time is in the sizeSubTable.

```
typedef struct {
    unsigned long  indexSubTableArrayOffset;     /*offset to corresponding index
                                             subtable array from beginning of bloc
                                             table*/
    unsigned long  indexTablesSize;          /* number bytes of corresponding index
                                             subtables and array*/
    unsigned long  numberOfIndexSubTables; /* an index subtable for each range or
                                             format change*/
    unsigned long  colorRef;                 /*Set to 0, ignore for now*/
    sbitLineMetrics                          hori;  /* line metrics*/
    sbitLineMetrics                          vert;  /*line metrics*/
    unsigned short                           startGlyphIndex;   /*lowest glyph index
                                             for this size*/
    unsigned short                           endGlyphIndex;     /*highest glyph index
                                             for this size*/
    uChar          ppemX;                    /* target horizontal ppem*/
    uChar          ppemY;                    /* target vertical ppem*/
    uChar          bitDepth;                 /* bit depth of the strike */
    sChar          flags;                    /* see bitmapFlags, first two bits are
                                             for orientation, vertical or horizontal
                                             */
} bitmapSizeTable;
```

The indexSubTableArrayOffset is the offset from the beginning of the 'bloc' table to the indexSubTableArray. Each strike has one of these arrays to support various formats and discontiguous ranges of bitmaps. More on this array below.

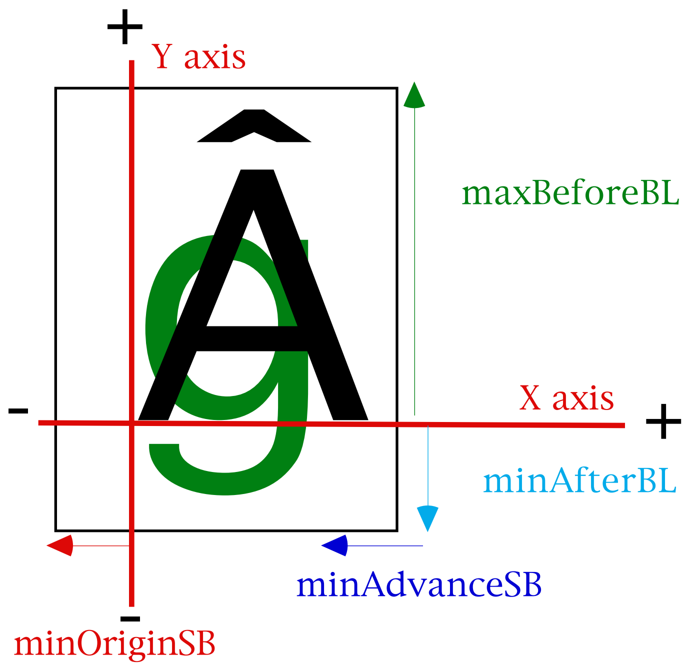
The indexTablesSize is the total number of bytes in the indexSubTableArray and the associated index tables.

The numberOfIndexSubTables is a count of the index tables for this strike.

The colorRef is put in for future enhancements that are not currently supported by the TrueType scaler. The bitDepth is what it says, the depth of this strike. This is useful for greyscale or color fonts. To maintain compatibility with scalers that will read sbit data the following table shows how the colorRef and bitDepth information will be assimilated.

bitDepth	colorRef	meaning
1	0	standard 1-bit font
n	0	multi-bit black-and-white font
n	x	color, grayscale, anti-aliased, etc.

The horizontal and vertical line metrics follow and as you will notice contain the ascender, descender, linegap, and advance information from previous formats. Additionally the caret information for Layout is included. The slope is to determine the angle at which the caret is to be drawn and the offset is how many pixels (+ or -) to move the caret. This is a signed char and not a fixed since we are dealing with integer metrics. The minOriginSB, minAdvanceSB , maxBeforeBL, and minAfterBL is described below. The main need for these numbers is for scalers that may need to pre-allocate memory and/or need more metric information to position glyphs. These numbers are not used by the sbit scaler on the macintosh, but may be used on other platforms.



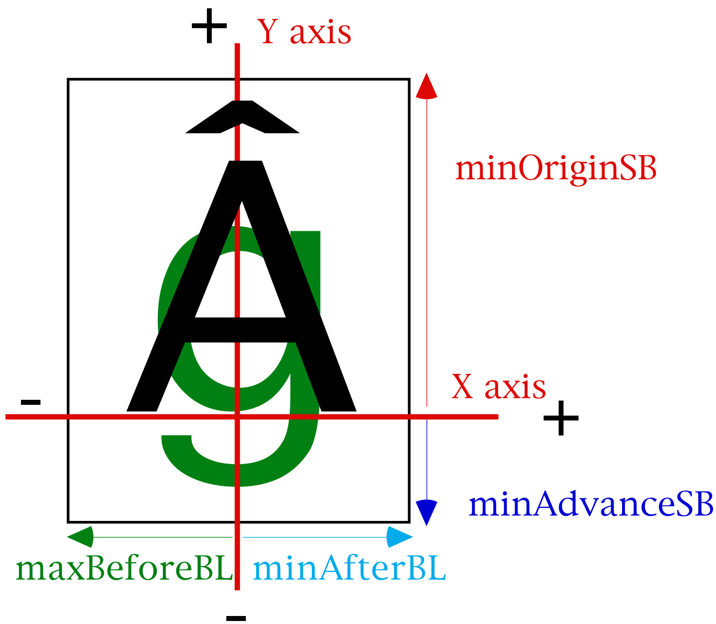
minOriginSB = Minimum of (horiBearingX)

minAdvanceSB = Minimum of (hori.advance - (horiBearingX + widt

maxBeforeBL = Maximum of (horiBearingY)

minAfterBL = Minimum of (horiBearingY - height)

Horizontal Text



minOriginSB = Minimum of (vertBearingY)

minAdvanceSB = Minimum of (vert.advance - (vertBearingY + height)

maxBeforeBL = Maximum of (vertBearingX)

minAfterBL = Minimum of (vertBearingX - width)

Vertical Text

All of the metrics are one byte in length. This should be long enough for our use of bitmaps.
The startGlyphIndex is the index of the first glyph included after the missing glyph. If you are only embedding kanji glyphs, the first index may be 700.

The endGlyphIndex is the index of the last glyph for which a bitmap has been embedded. This is an outer search range will be used by some scalers.

The ppem is the resolution information and for us will be the point size of the strike at 72 dpi. Basically the size given in the 'fond' or 'NFNT'.

Sbit only fonts must have every glyph in every strike embedded. This is only required when no outlines are present.

```
typedef struct {
      sChar              ascender;
      sChar              descender;
      uChar              widthMax;
      sChar              caretSlopeNumerator;
      sChar              caretSlopeDenominator;
      sChar              caretOffset;
      sChar              minOriginSB;
      sChar              minAdvanceSB;
      sChar              maxBeforeBL;
      sChar              minAfterBL;
      sChar              pad1;
      sChar              pad2;
} sbitLineMetrics;
```

```
typedef struct {
      unsigned short  firstGlyphIndex;
      unsigned short  lastGlyphIndex;
      unsigned long    additionalOffsetToIndexSubtable;        /* add to
                                    indexSubTableArrayOffset to get offset from
                                    beginning of bloc*/
} indexSubTableArray;
```

This array contains the glyph ranges and offsets to indices within the 'bloc'. After determining the strike to use the scaler goes to this array to search for the index that contains the glyph number. When the index is found it will be an offset to an indexSubTable. The indexSubTableArray is located from indexSubTableArrayOffset in the corresponding bitmapSizeSubTable. The first indexSubTableArray is after the last bitmapSizeSubTable entry. Then the indexSubTables for the strike follow. The next indexSubTableArray(if more than one strike) and its indexSubTables are next. The 'bloc' continues with this array and indices for each strike (See the following diagram for clarification).

Finally we get to the indexSubTable. Currently there are two formats of indices, one for proportional data and one for monospaced data. Whether the data is fixed size or variable size determines which index to use. Note: the format of the 'bdat' data can be in any format. It is not determined by the index format. The index tells us where the information for a particular glyph starts in the 'bdat' and how large the data is. The size of the data for the glyph is the difference of offsetArray[glyphIndex+1] - offsetArray[glyphIndex].
The indexSubTable1 is the proportional format index. This will be used for most roman fonts and the first 500 or so of the CJK fonts. The imageFormat tells us what kind of data is in the 'bdat', more on this in the 'bdat' section.

```
typedef struct {
      unsigned short  indexFormat;
      unsigned short  imageFormat;
      unsigned long  imageDataOffset;        /*offset to corresponding image data
                                    from beginning of bdat table*/
```

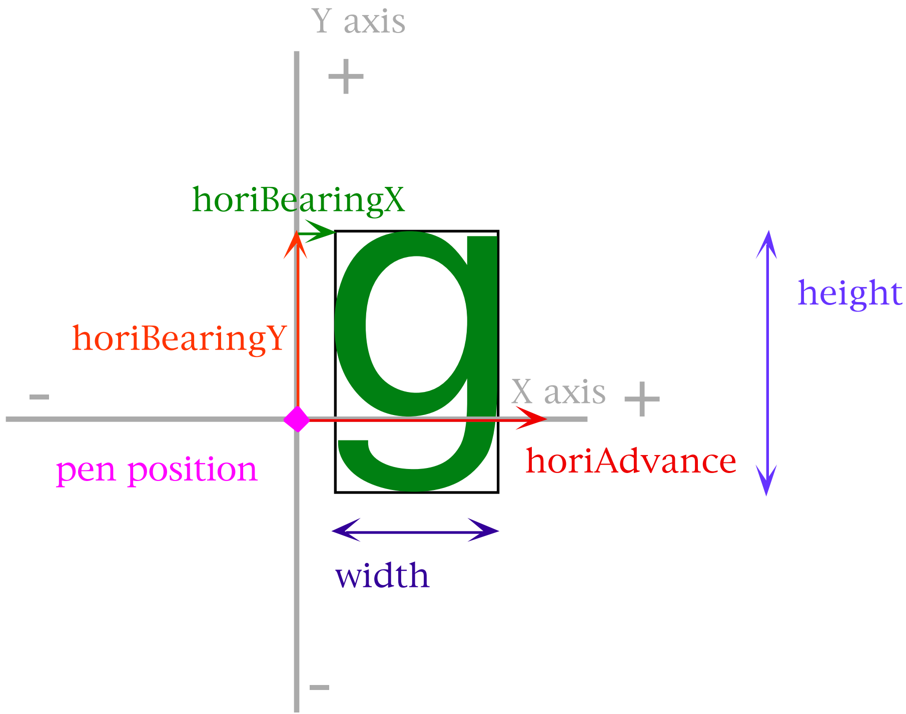
```
} indexSubHeader;

      /* format 1 has variable length images of the same format*/
typedef struct {
      indexSubHeader  header;
      unsigned long  offsetArray[anyNumber]; /*offsetArrary[glyphIndex] +
                                    imageDataOffset = startOfBitDataForGlyph*/
                                    /*sizeOfArray = lastGlyph - firstGlyph + 1*/
} indexSubTable1;
```

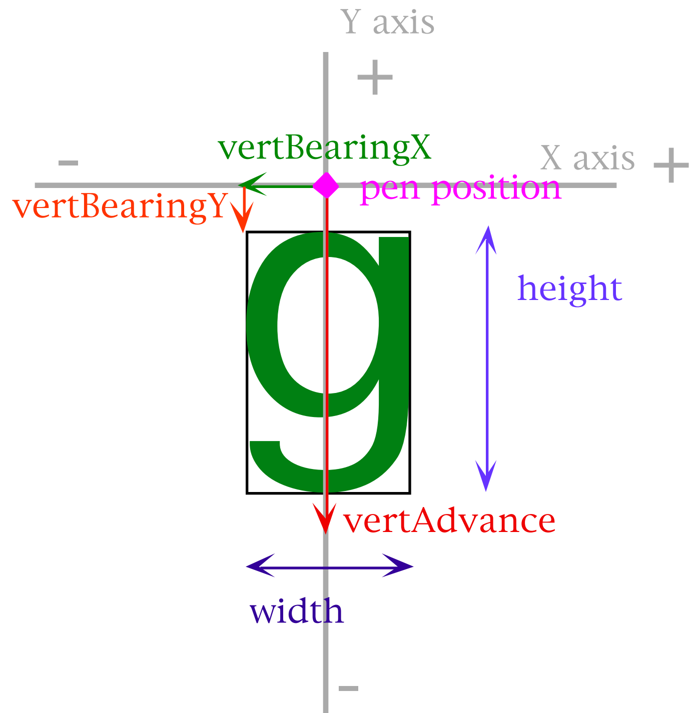
The indexSubTable3 is also a proportional format index. It is just like format1 except the offsets are shorts instead of longs.
NOTE: THIS MUST BE PADDED TO A LONG BOUNDARY!

```
      /* format 3 has variable length images of the same format with word
                                    offsets*/
typedef struct {
      indexSubHeader  header;
      unsigned short  offsetArray[anyNumber]; /*offsetArrary[glyphIndex] +
                                    imageDataOffset = startOfBitDataForGlyph*/
                                    /*sizeOfArray = lastGlyph - firstGlyph + 1 +
                                    padding if needed*/
} indexSubTable3;
```

For monospaced fonts the metrics are the same and only the bitmap changes. For those we use indexSubTable2. The metrics are not in the 'bdat' but are here in the subTable. Only the data for the bitmap is put in the 'bdat' for these glyphs. The data can be in various formats; byte aligned, bit aligned, compressed, etc. The size of all the bitmap data needed to be read in from the 'bdat' is the imageSize. The imageDataOffset is the offset into the 'bdat' for the first of this range of glyphs.



Horizontal Text



Vertical Text

```
/* format 2 all glyphs have images of the same length and of the same format*/
      typedef struct {
                        uChar       height;
                        uChar       width;
                        sChar       horiBearingX;
                        sChar       horiBearingY;
                        uChar       horiAdvance;
                        sChar       vertBearingX;
                        sChar       vertBearingY;
                        uChar       vertAdvance;
      } bigGlyphMetrics;

      typedef struct {
                        uChar       height;
                        uChar       width;
                        sChar       bearingX;
                        sChar       bearingY;
                        uChar       advance;
      } smallGlyphMetrics;

typedef struct {
      indexSubHeader  header;
      unsigned long     imageSize;  /* all the glyphs are of the same size.  May
                                    be compressed, bit-aligned, or byte-aligned*/
      bigGlyphMetrics bigMetrics;   /*all glyphs have the same metrics*/
} indexSubTable2;
```

The metric information for index type 2 glyphs is of type bigGlyphMetrics. The pictures above show the meaning of the metrics. Possible values for the 'g' glyph above could be: horiBearingX = 2, horiBearingY = 11, horiAdvance = 12, vertBearingX = -3; vertBearingY = -2, vertAdvance = 20(Note: although the advance is in the -y direction, it is still unsigned since it is a distance and not a direction, the direction of vertical text is assumed top to bottom), height = 16, width = 9. Once again we are dealing with integer metrics. If you use smallGlyphMetrics, you must set the bit in the bitmapSizeTable.flags to the inform the scaler as to the orientation of the metrics(using bitmapFlags).

```
typedef enum {
      flgHorizontal = 0x01,
      flgVertical = 0x02
} bitmapFlags;
```

The scaler makes up fake metrics for the orientation that is not included.

'bdat'

The 'bdat' table is a very simple structure. A version followed by data. The data can be in various formats. Some of the formats contain metric information and other formats contain only the image. Some of the data may be compressed and other data may not. The 'bloc' contains the information to locate and understand how to interpret the data contained in the 'bdat' table.

```
typedef enum {
      ndxProportional = 1,
      ndxMono,
      ndxProportionalSmall
} bitmapIndexFormats;

typedef enum{
      proportionalFormat1 = 1,            /*small metrics and data, byte-aligned*/
      proportionalFormat2,                      /*small metrics and data, bit-
                                    aligned*/
      proportionalCompressedFormat3,/*NOT USED*//*metric info followed by
                                    compressed data*/
      monoCompressedFormat4,              /*just compressed data, metrics are in
                                    bloc*/
      monoFormat5 = 5,                    /*just bit aligned data, metrics are in
                                    bloc*/
      proportionalByteFormat6,            /*metrics & byte-aligned image*/
      proportionalBitFormat7              /*metrics & bit-aligned image*/
} bitmapDataFormats;

typedef struct {
      smallGlyphMetrics smallMetrics;
      /*      bitmap image data */
} glyphBitmap_1, glyphBitmap_2;
```

Note that format 1 and 2 are the same except the bitmap data. Type 1 is byte aligns and type 2 is bit aligned.

The data format similar to fbit7 that is used with monospaced glyphs is not included in the header. This is a real simple format of bit-aligned bitmaps, padded to a byte boundary. All the metric information is in the indexSubTable2.

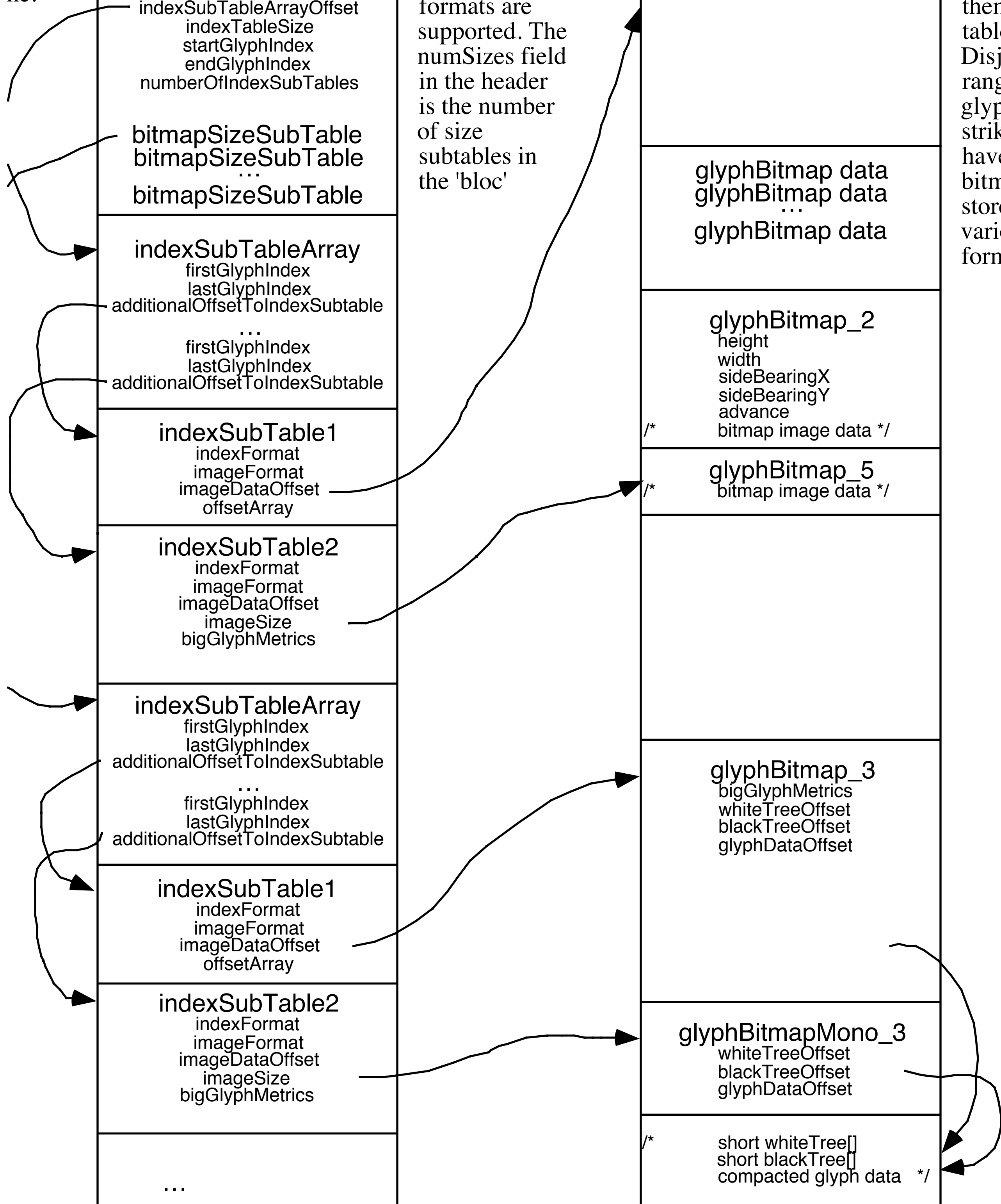
Data format 4 is similar to the fbit3 compressed format. The format contains metric information and offsets to the binary trees and glyph data.

```
/* for monospaced bitmaps the metric information is in the 'bloc' indexSubTable2*/
/* for monospaced bitmaps the metric information is in the 'bloc' indexSubTable2*/
typedef struct {
      unsigned long     whiteTreeOffset;        /*offset from 'height'*/
      unsigned long     blackTreeOffset;        /*offset from 'height'*/
      unsigned long     glyphDataOffset;        /*offset from 'height'*/
} glyphBitmap_4;
```

Format 5 is similar to format 2 and 7 except that no metric information is included, just the bit-packed bits. (There has been no request for a byte-aligned format of this type)

```
typedef struct {
      bigGlyphMetrics bigMetrics;
/*    bitmap image data */
} glyphBitmap_6, glyphBitmap_7;
```

Note that format 6 and 7 are the same except the bitmap data. Type 6 is byte aligned and type 7 is bit aligned.



The 'bloc' table provides information on how to locate the bitmaps for a particular glyph. Discontiguous ranges and multiple formats are supported. The numSizes field in the header is the number of size subtables in the 'bloc'

This is a sample of the types of data that can be in the 'bdat' table. For each one listed there is a variable number of them in the table. Disjoint ranges of glyphs for a strike can have their bitmaps stored in various formats.