# GNU Unifont

Generated by Doxygen 1.9.3

# Chapter 1

# GNU Unifont

## 1.1 GNU Unifont C Utilities

This documentation covers C utility programs for creating GNU Unifont glyphs and fonts.

## 1.2 LICENSE



## 1.3 Introduction

Unifont is the creation of Roman Czyborra, who created Perl utilities for generating a dual-width Bitmap Distribution Format (BDF) font 16 pixels tall, unifont.bdf, from an input file named unifont.hex. The unifont.hex file contained two fields separated by a colon: a Unicode code point as four hexadecimal digits, and a hexadecimal string of 32 or 64 characters representing the glyph bitmap pattern. Roman also wrote other Perl scripts for manipulating unifont.hex files.

Jungshik Shin wrote a Perl script, johab2ucs2, to convert Hangul syllable glyph elements into Hangul Johab-encoded fonts. These glyph elements are compatible with Jaekyung ”Jake” Song's Hanterm terminal emulator. Paul Hardy modified johab2ucs2 and drew Hangul Syllables Unicode elements for compatibility with this Johab encoding and with Hanterm. These new glyphs were created to avoid licensing issues with the Hangul Syllables glyphs that were in the original unifont.hex file.

Over time, Unifont was extended to allow correct positioning of combining marks in a TrueType font, coverage beyond Unicode Plane 0, and the addition of Under-ConScript Unicode Registry (UCSUR) glyphs. There is also partial support for experimental quadruple-width glyphs.

Paul Hardy wrote the first pair of C programs, unihex2bmp.c and unibmp2hex.c, to facilitate editing the bitmaps at their real aspect ratio. These programs allow conversion between the Unifont .hex format and a Windows Bitmap or Wireless Bitmap file for editing with a graphics editor. This was followed by make files, other C programs, Perl scripts, and shell scripts.

Luis Alejandro González Miranda wrote scripts for converting unifont.hex into a TrueType font using FontForge.

Andrew Miller wrote additional Perl programs for directly rendering unifont.hex files, for converting unifont.hex to and from Portable Network Graphics (PNG) files for editing based upon Paul Hardy's BMP conversion programs, and also wrote other Perl scripts.

David Corbett wrote a Perl script to rotate glyphs in a unifont.hex file and an awk script to substitute new glyphs for old glyphs of the same Unicode code point in a unifont.hex file.

何志翔 (He Zhixiang) wrote a program to convert Unifont files into OpenType fonts, hex2otf.c.

## 1.4 The C Programs

This documentation only covers C programs and their header files. These programs are typically longer than the Unifont package's Perl scripts, which being much smaller are easier to understand. The C programs are, in alphabetical order:

| Program | Description |
|---|---|
| hex2otf.c | Convert a GNU Unifont .hex file to an OpenType font |
| unibdf2hex.c | Convert a BDF file into a unifont.hex file |

| Program | Description |
|---|---|
| unibmp2hex.c | Turn a .bmp or .wbmp glyph matrix into a GNU Unifont hex glyph set of 256 characters |
| unibmpbump.c | Adjust a Microsoft bitmap (.bmp) file that was created by unihex2png but converted to .bmp |
| unicoverage.c | Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file |

| Program | Description |
|---|---|
| unidup.c | Check for duplicate code points in sorted unifont.↩ hex file |
| unifont1per.c | Read a Unifont .hex file from standard input and produce one glyph per .bmp bitmap file as output |
| unifontpic.c | See the "Big Picture"↩ : the entire Unifont in one BMP bitmap |

| Program | Description |
|---|---|
| unigencircles.c | Superimpose dashed combining circles on combining glyphs |
| unigenwidth.c | IEEE 1003.1-2008 setup to calculate wchar_t string widths |
| unihex2bmp.c | Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing |
| unihexgen.c | Generate a series of glyphs containing hexadecimal code points |

| Program | Description |
| --- | --- |
| unipagecount | Count the number of glyphs defined in each page of 256 code points |

## 1.5 Perl Scripts

The very first program written for Unifont conversion was Roman Czyborra's hexdraw Perl script. That one script would convert a unifont.hex file into a text file with 16 lines per glyph (one for each glyph row) followed by a blank line aftr each glyph. That allowed editing unifont.hex glyphs with a text-based editor.

Combined with Roman's hex2bdf Perl script to convert a unifont.hex file into a BDF font, these two scripts formed a complete package for editing Unifont and generating the resulting BDF fonts.

There was no combining mark support initially, and the original unifont.hex file included combining circles with combining mark glyphs.

The list below gives a brief description of these and the other Perl scripts that are in the Unifont package src subdirectory.

| Perl Script | Description |
| --- | --- |
| bdfimplode | Convert a BDF font into GNU Unifont .hex format |

| Perl Script | Description |
| --- | --- |
| hex2bdf | Convert a GNU Unifont .hex file into a BDF font |
| hex2sfd | Convert a GNU Unifont .hex file into a FontForge .sfd format |
| hexbraille | Algorithmically generate the Unicode Braille range (U+28xx) |
| hexdraw | Convert a GNU Unifont .hex file to and from an ASCII text file |

| Perl Script | Description |
|---|---|
| hexkinya | Create the Private Use Area Kinya syllables |
| hexmerge | Merge two or more GNU Unifont .hex font files into one |
| johab2ucs2 | Convert a Johab BDF font into GNU Unifont Hangul Syllables |
| unifont-viewer | View a .hex font file with a graphical user interface |

| Perl Script | Description |
|---|---|
| unifontchojung | Extract Hangul syllables that have no final consonant |
| unifontksx | Extract Hangul syllables that comprise KS X 1001:1992 |
| unihex2png | GNU Unifont .hex file to Portable Network Graphics converter |
| unihexfill | Generate range of Unifont 4- or 6-digit hexadecimal glyp |

| Perl Script | Description |
| --- | --- |
| unihexrotate | Rotate Unifont hex glyphs in quarter turn increments |
| unipng2hex | Portable Network Graphics to GNU Unifont .hex file converter |

Chapter 2

# Data Structure Index

## 2.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 3

# File Index

## 3.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 4

# Data Structure Documentation

## 4.1 Buffer Struct Reference

Generic data structure for a linked list of buffer elements.

### Data Fields

- size_t capacity
- byte * begin
- byte * next
- byte * end

### 4.1.1 Detailed Description

Generic data structure for a linked list of buffer elements.

A buffer can act as a vector (when filled with 'store*' functions), or a temporary output area (when filled with 'cache*' functions). The 'store*' functions use native endian. The 'cache*' functions use big endian or other formats in OpenType. Beware of memory alignment.

Definition at line 133 of file hex2otf.c.

### 4.1.2 Field Documentation

#### 4.1.2.1 begin

byte* Buffer::begin

Definition at line 136 of file hex2otf.c.

#### 4.1.2.2 capacity

size_t Buffer::capacity

Definition at line 135 of file hex2otf.c.

#### 4.1.2.3 end

byte * Buffer::end

Definition at line 136 of file hex2otf.c.

#### 4.1.2.4 next

byte * Buffer::next

Definition at line 136 of file hex2otf.c.

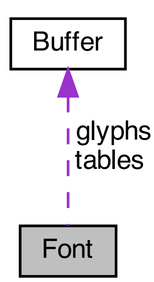
The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.2 Font Struct Reference

Data structure to hold information for one font.

Collaboration diagram for Font:



## Data Fields

- Buffer * tables
- Buffer * glyphs
- uint_fast32_t glyphCount
- pixels_t maxWidth

### 4.2.1 Detailed Description

Data structure to hold information for one font.

Definition at line 628 of file hex2otf.c.

### 4.2.2 Field Documentation

#### 4.2.2.1 glyphCount

uint_fast32_t Font::glyphCount

Definition at line 632 of file hex2otf.c.

#### 4.2.2.2 glyphs

Buffer* Font::glyphs

Definition at line 631 of file hex2otf.c.

#### 4.2.2.3 maxWidth

pixels_t Font::maxWidth

Definition at line 633 of file hex2otf.c.

#### 4.2.2.4 tables

Buffer* Font::tables

Definition at line 630 of file hex2otf.c.

The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.3 Glyph Struct Reference

Data structure to hold data for one bitmap glyph.

### Data Fields

- uint_least32_t codePoint
  undefined for glyph 0
- byte bitmap [GLYPH_MAX_BYTE_COUNT]
  hexadecimal bitmap character array
- uint_least8_t byteCount
  length of bitmap data
- bool combining
  whether this is a combining glyph
- pixels_t pos
- pixels_t lsb
  left side bearing (x position of leftmost contour point)

### 4.3.1 Detailed Description

Data structure to hold data for one bitmap glyph.

This data structure holds data to represent one Unifont bitmap glyph: Unicode code point, number of bytes in its bitmap array, whether or not it is a combining character, and an offset from the glyph origin to the start of the bitmap.

Definition at line 614 of file hex2otf.c.

### 4.3.2 Field Documentation

#### 4.3.2.1 bitmap

byte Glyph::bitmap[GLYPH_MAX_BYTE_COUNT]

hexadecimal bitmap character array

Definition at line 617 of file hex2otf.c.

#### 4.3.2.2 byteCount

uint_least8_t Glyph::byteCount

length of bitmap data

Definition at line 618 of file hex2otf.c.

#### 4.3.2.3 codePoint

uint_least32_t Glyph::codePoint

undefined for glyph 0

Definition at line 616 of file hex2otf.c.

#### 4.3.2.4 combining

bool Glyph::combining

whether this is a combining glyph

Definition at line 619 of file hex2otf.c.

#### 4.3.2.5 lsb

pixels_t Glyph::lsb

left side bearing (x position of leftmost contour point)

Definition at line 622 of file hex2otf.c.

#### 4.3.2.6 pos

pixels_t Glyph::pos

number of pixels the glyph should be moved to the right (negative number means moving to the left)

Definition at line 620 of file hex2otf.c.

The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.4 NamePair Struct Reference

Data structure for a font ID number and name character string.

#include <hex2otf.h>

### Data Fields

- int id
- const char * str

### 4.4.1 Detailed Description

Data structure for a font ID number and name character string.

Definition at line 77 of file hex2otf.h.

### 4.4.2 Field Documentation

#### 4.4.2.1 id

int NamePair::id

Definition at line 79 of file hex2otf.h.

#### 4.4.2.2 str

const char∗ NamePair::str

Definition at line 80 of file hex2otf.h.

The documentation for this struct was generated from the following file:

- src/hex2otf.h

## 4.5 Options Struct Reference

Data structure to hold options for OpenType font output.

### Data Fields

- bool truetype
- bool blankOutline
- bool bitmap
- bool gpos
- bool gsub
- int cff
- const char ∗ hex
- const char ∗ pos
- const char ∗ out
- NameStrings nameStrings

### 4.5.1 Detailed Description

Data structure to hold options for OpenType font output.

This data structure holds the status of options that can be specified as command line arguments for creating the output OpenType font file.

Definition at line 2453 of file hex2otf.c.

### 4.5.2 Field Documentation

#### 4.5.2.1 bitmap

bool Options::bitmap

Definition at line 2455 of file hex2otf.c.

#### 4.5.2.2 blankOutline

bool Options::blankOutline

Definition at line 2455 of file hex2otf.c.

#### 4.5.2.3 cff

int Options::cff

Definition at line 2456 of file hex2otf.c.

#### 4.5.2.4 gpos

bool Options::gpos

Definition at line 2455 of file hex2otf.c.

#### 4.5.2.5 gsub

bool Options::gsub

Definition at line 2455 of file hex2otf.c.

#### 4.5.2.6 hex

const char* Options::hex

Definition at line 2457 of file hex2otf.c.

#### 4.5.2.7 nameStrings

NameStrings Options::nameStrings

Definition at line 2458 of file hex2otf.c.

#### 4.5.2.8 out

const char * Options::out

Definition at line 2457 of file hex2otf.c.

#### 4.5.2.9 pos

const char * Options::pos

Definition at line 2457 of file hex2otf.c.

#### 4.5.2.10 truetype

bool Options::truetype

Definition at line 2455 of file hex2otf.c.

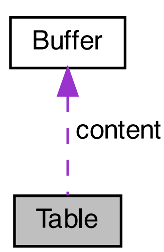
The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.6 Table Struct Reference

Data structure for an OpenType table.

Collaboration diagram for Table:



## Data Fields

- uint_fast32_t tag
- Buffer * content

### 4.6.1 Detailed Description

Data structure for an OpenType table.

This data structure contains a table tag and a pointer to the start of the buffer that holds data for this OpenType table.

For information on the OpenType tables and their structure, see https://docs.microsoft.com/en-us/typography/opentype/spec/otff#font-tables.

Definition at line 645 of file hex2otf.c.

### 4.6.2 Field Documentation

#### 4.6.2.1 content

Buffer* Table::content

Definition at line 648 of file hex2otf.c.

#### 4.6.2.2 tag

uint_fast32_t Table::tag

Definition at line 647 of file hex2otf.c.

The documentation for this struct was generated from the following file:

- src/hex2otf.c

## 4.7 TableRecord Struct Reference

Data structure for data associated with one OpenType table.

## Data Fields

- uint_least32_t tag
- uint_least32_t offset
- uint_least32_t length
- uint_least32_t checksum

### 4.7.1 Detailed Description

Data structure for data associated with one OpenType table.

This data structure contains an OpenType table's tag, start within an OpenType font file, length in bytes, and checksum at the end of the table.

Definition at line 747 of file hex2otf.c.

### 4.7.2 Field Documentation

#### 4.7.2.1 checksum

uint_least32_t TableRecord::checksum

Definition at line 749 of file hex2otf.c.

#### 4.7.2.2 length

uint_least32_t TableRecord::length

Definition at line 749 of file hex2otf.c.

#### 4.7.2.3 offset

uint_least32_t TableRecord::offset

Definition at line 749 of file hex2otf.c.

#### 4.7.2.4 tag

uint_least32_t TableRecord::tag

Definition at line 749 of file hex2otf.c.

The documentation for this struct was generated from the following file:

- src/hex2otf.c

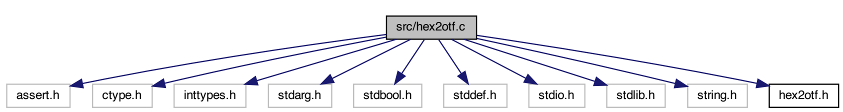
# Chapter 5

# File Documentation

## 5.1 src/hex2otf.c File Reference

hex2otf - Convert GNU Unifont .hex file to OpenType font

#include <assert.h>
#include <ctype.h>
#include <inttypes.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hex2otf.h"
Include dependency graph for hex2otf.c:




### Data Structures

- struct Buffer

    Generic data structure for a linked list of buffer elements.
- struct Glyph

    Data structure to hold data for one bitmap glyph.
- struct Font

    Data structure to hold information for one font.
- struct Table

Data structure for an OpenType table.

- struct TableRecord

  Data structure for data associated with one OpenType table.
- struct Options

  Data structure to hold options for OpenType font output.

## Macros

- #define VERSION ”1.0.1”

  Program version, for ”--version” option.
- #define U16MAX 0xffff

  Maximum UTF-16 code point value.
- #define U32MAX 0xffffffff

  Maximum UTF-32 code point value.
- #define PRI_CP ”U+%.4”PRIXFAST32

  Format string to print Unicode code point.
- #define static_assert(a, b) (assert(a))

  If ”a” is true, return string ”b”.
- #define BX(shift, x) ((uintmax_t)(!!(x)) << (shift))

  Truncate & shift word.
- #define B0(shift) BX((shift), 0)

  Clear a given bit in a word.
- #define B1(shift) BX((shift), 1)

  Set a given bit in a word.
- #define GLYPH_MAX_WIDTH 16

  Maximum glyph width, in pixels.
- #define GLYPH_HEIGHT 16

  Maximum glyph height, in pixels.
- #define GLYPH_MAX_BYTE_COUNT (GLYPH_HEIGHT * GLYPH_MAX_WIDTH / 8)

  Number of bytes to represent one bitmap glyph as a binary array.
- #define DESCENDER 2

  Count of pixels below baseline.
- #define ASCENDER (GLYPH_HEIGHT - DESCENDER)

  Count of pixels above baseline.
- #define FUPEM 64

  Font units per em.
- #define MAX_GLYPHS 65536

  An OpenType font has at most 65536 glyphs.
- #define MAX_NAME_IDS 256

  Name IDs 0-255 are used for standard names.
- #define FU(x) ((x) * FUPEM / GLYPH_HEIGHT)

  Convert pixels to font units.
- #define PW(x) ((x) / (GLYPH_HEIGHT / 8))

  Convert glyph byte count to pixel width.
- #define defineStore(name, type)

  Temporary define to look up an element in an array of given type.
- #define addByte(shift)
- #define getRowBit(rows, x, y) ((rows)[(y)] & x0 >> (x))
- #define flipRowBit(rows, x, y) ((rows)[(y)] ^= x0 >> (x))
- #define stringCount (sizeof strings / sizeof *strings)
- #define cacheCFF32(buf, x) (cacheU8 ((buf), 29), cacheU32 ((buf), (x)))

## Typedefs

- typedef unsigned char byte

  Definition of "byte" type as an unsigned char.
- typedef int_least8_t pixels_t

  This type must be able to represent max(GLYPH_MAX_WIDTH, GLYPH_HEIGHT).
- typedef struct Buffer Buffer

  Generic data structure for a linked list of buffer elements.
- typedef const char * NameStrings[MAX_NAME_IDS]

  Array of OpenType names indexed directly by Name IDs.
- typedef struct Glyph Glyph

  Data structure to hold data for one bitmap glyph.
- typedef struct Font Font

  Data structure to hold information for one font.
- typedef struct Table Table

  Data structure for an OpenType table.
- typedef struct Options Options

  Data structure to hold options for OpenType font output.

## Enumerations

- enum LocaFormat { LOCA_OFFSET16 = 0 , LOCA_OFFSET32 = 1 }

  Index to Location ("loca") offset information.
- enum ContourOp { OP_CLOSE , OP_POINT }

  Specify the current contour drawing operation.
- enum FillSide { FILL_LEFT , FILL_RIGHT }

  Fill to the left side (CFF) or right side (TrueType) of a contour.

## Functions

- void fail (const char *reason,...)

  Print an error message on stderr, then exit.
- void initBuffers (size_t count)

  Initialize an array of buffer pointers to all zeroes.
- void cleanBuffers ()

  Free all allocated buffer pointers.
- Buffer * newBuffer (size_t initialCapacity)

  Create a new buffer.
- void ensureBuffer (Buffer *buf, size_t needed)

  Ensure that the buffer has at least the specified minimum size.
- void freeBuffer (Buffer *buf)

  Free the memory previously allocated for a buffer.
- defineStore (storeU8, uint_least8_t)
- void cacheU8 (Buffer *buf, uint_fast8_t value)

  Append one unsigned byte to the end of a byte array.
- void cacheU16 (Buffer *buf, uint_fast16_t value)

Append two unsigned bytes to the end of a byte array.

- void cacheU32 (Buffer *buf, uint_fast32_t value)

  Append four unsigned bytes to the end of a byte array.
- void cacheCFFOperand (Buffer *buf, int_fast32_t value)

  Cache charstring number encoding in a CFF buffer.
- void cacheZeros (Buffer *buf, size_t count)

  Append 1 to 4 bytes of zeroes to a buffer, for padding.
- void cacheBytes (Buffer *restrict buf, const void *restrict src, size_t count)

  Append a string of bytes to a buffer.
- void cacheBuffer (Buffer *restrict bufDest, const Buffer *restrict bufSrc)

  Append bytes of a table to a byte buffer.
- void writeBytes (const byte bytes[ ], size_t count, FILE *file)

  Write an array of bytes to an output file.
- void writeU16 (uint_fast16_t value, FILE *file)

  Write an unsigned 16-bit value to an output file.
- void writeU32 (uint_fast32_t value, FILE *file)

  Write an unsigned 32-bit value to an output file.
- void addTable (Font *font, const char tag[static 4], Buffer *content)

  Add a TrueType or OpenType table to the font.
- void organizeTables (Font *font, bool isCFF)

  Sort tables according to OpenType recommendations.
- int byTableTag (const void *a, const void *b)

  Compare tables by 4-byte unsigned table tag value.
- void writeFont (Font *font, bool isCFF, const char *fileName)

  Write OpenType font to output file.
- bool readCodePoint (uint_fast32_t *codePoint, const char *fileName, FILE *file)

  Read up to 6 hexadecimal digits and a colon from file.
- void readGlyphs (Font *font, const char *fileName)

  Read glyph definitions from a Unifont .hex format file.
- int byCodePoint (const void *a, const void *b)

  Compare two Unicode code points to determine which is greater.
- void positionGlyphs (Font *font, const char *fileName, pixels_t *xMin)

  Position a glyph within a 16-by-16 pixel bounding box.
- void sortGlyphs (Font *font)

  Sort the glyphs in a font by Unicode code point.
- void buildOutline (Buffer *result, const byte bitmap[ ], const size_t byteCount, const enum FillSide fillSide)

  Build a glyph outline.
- void prepareOffsets (size_t *sizes)

  Prepare 32-bit glyph offsets in a font table.
- Buffer * prepareStringIndex (const NameStrings names)

  Prepare a font name string index.
- void fillCFF (Font *font, int version, const NameStrings names)

  Add a CFF table to a font.
- void fillTrueType (Font *font, enum LocaFormat *format, uint_fast16_t *maxPoints, uint_fast16_t *maxContours)

  Add a TrueType table to a font.

- void fillBlankOutline (Font *font)
  Create a dummy blank outline in a font table.
- void fillBitmap (Font *font)
  Fill OpenType bitmap data and location tables.
- void fillHeadTable (Font *font, enum LocaFormat locaFormat, pixels_t xMin)
  Fill a "head" font table.
- void fillHheaTable (Font *font, pixels_t xMin)
  Fill a "hhea" font table.
- void fillMaxpTable (Font *font, bool isCFF, uint_fast16_t maxPoints, uint_fast16_t maxContours)
  Fill a "maxp" font table.
- void fillOS2Table (Font *font)
  Fill an "OS/2" font table.
- void fillHmtxTable (Font *font)
  Fill an "hmtx" font table.
- void fillCmapTable (Font *font)
  Fill a "cmap" font table.
- void fillPostTable (Font *font)
  Fill a "post" font table.
- void fillGposTable (Font *font)
  Fill a "GPOS" font table.
- void fillGsubTable (Font *font)
  Fill a "GSUB" font table.
- void cacheStringAsUTF16BE (Buffer *buf, const char *str)
  Cache a string as a big-ending UTF-16 surrogate pair.
- void fillNameTable (Font *font, NameStrings nameStrings)
  Fill a "name" font table.
- void printVersion ()
  Print program version string on stdout.
- void printHelp ()
  Print help message to stdout and then exit.
- const char * matchToken (const char *operand, const char *key, char delimiter)
  Match a command line option with its key for enabling.
- Options parseOptions (char *const argv[const])
  Parse command line options.
- int main (int argc, char *argv[])
  The main function.

## Variables

- Buffer * allBuffers
  Initial allocation of empty array of buffer pointers.
- size_t bufferCount
  Number of buffers in a Buffer * array.
- size_t nextBufferIndex
  Index number to tail element of Buffer * array.

### 5.1.1 Detailed Description

hex2otf - Convert GNU Unifont .hex file to OpenType font

This program reads a Unifont .hex format file and a file containing combining mark offset information, and produces an OpenType font file.

Copyright



Author

何志翔 (He Zhixiang)

Definition in file hex2otf.c.

### 5.1.2 Macro Definition Documentation

#### 5.1.2.1 addByte

#define addByte(
shift )

Value:

```
if (p == end) \
    break; \
record->checksum += (uint_fast32_t)*p++ « (shift);
```

#### 5.1.2.2 ASCENDER

#define ASCENDER (GLYPH_HEIGHT - DESCENDER)

Count of pixels above baseline.

Definition at line 79 of file hex2otf.c.

#### 5.1.2.3 B0

#define B0(
shift ) BX((shift), 0)

Clear a given bit in a word.

Definition at line 66 of file hex2otf.c.

#### 5.1.2.4 B1

#define B1(
shift ) BX((shift), 1)

Set a given bit in a word.

Definition at line 67 of file hex2otf.c.

#### 5.1.2.5 BX

#define BX(
shift,
x ) ((uintmax_t)(!!(x)) << (shift))

Truncate & shift word.

Definition at line 65 of file hex2otf.c.

#### 5.1.2.6 defineStore

#define defineStore(
name,
type )

Value:
```
void name (Buffer *buf, type value) \
{ \
    type *slot = getBufferSlot (buf, sizeof value); \
    *slot = value; \
}
```

Temporary define to look up an element in an array of given type.

This defintion is used to create lookup functions to return a given element in unsigned arrays of size 8, 16, and 32 bytes, and in an array of pixels.

Definition at line 350 of file hex2otf.c.

#### 5.1.2.7 DESCENDER

#define DESCENDER 2

Count of pixels below baseline.

Definition at line 76 of file hex2otf.c.

#### 5.1.2.8 FU

#define FU(
x ) ((x) * FUPEM / GLYPH_HEIGHT)

Convert pixels to font units.

Definition at line 91 of file hex2otf.c.

#### 5.1.2.9 FUPEM

#define FUPEM 64

Font units per em.

Definition at line 82 of file hex2otf.c.

#### 5.1.2.10 GLYPH_HEIGHT

#define GLYPH_HEIGHT 16

Maximum glyph height, in pixels.

Definition at line 70 of file hex2otf.c.

#### 5.1.2.11 GLYPH_MAX_BYTE_COUNT

#define GLYPH_MAX_BYTE_COUNT (GLYPH_HEIGHT * GLYPH_MAX_WIDTH / 8)

Number of bytes to represent one bitmap glyph as a binary array.

Definition at line 73 of file hex2otf.c.

#### 5.1.2.12 GLYPH_MAX_WIDTH

```
#define GLYPH_MAX_WIDTH 16
```

Maximum glyph width, in pixels.

Definition at line 69 of file hex2otf.c.

#### 5.1.2.13 MAX_GLYPHS

```
#define MAX_GLYPHS 65536
```

An OpenType font has at most 65536 glyphs.

Definition at line 85 of file hex2otf.c.

#### 5.1.2.14 MAX_NAME_IDS

```
#define MAX_NAME_IDS 256
```

Name IDs 0-255 are used for standard names.

Definition at line 88 of file hex2otf.c.

#### 5.1.2.15 PRI_CP

```
#define PRI_CP "U+%.4"PRIXFAST32
```

Format string to print Unicode code point.

Definition at line 58 of file hex2otf.c.

#### 5.1.2.16 PW

```
#define PW(
        x ) ((x) / (GLYPH_HEIGHT / 8))
```

Convert glyph byte count to pixel width.

Definition at line 94 of file hex2otf.c.

#### 5.1.2.17 static_assert

```
#define static_assert(
            a,
            b ) (assert(a))
```

If "a" is true, return string "b".

Definition at line 61 of file hex2otf.c.

#### 5.1.2.18 U16MAX

```
#define U16MAX 0xffff
```

Maximum UTF-16 code point value.

Definition at line 55 of file hex2otf.c.

#### 5.1.2.19 U32MAX

```
#define U32MAX 0xffffffff
```

Maximum UTF-32 code point value.

Definition at line 56 of file hex2otf.c.

#### 5.1.2.20 VERSION

```
#define VERSION "1.0.1"
```

Program version, for "--version" option.

Definition at line 51 of file hex2otf.c.

### 5.1.3 Typedef Documentation

#### 5.1.3.1 Buffer

typedef struct Buffer Buffer

Generic data structure for a linked list of buffer elements.

A buffer can act as a vector (when filled with 'store*' functions), or a temporary output area (when filled with 'cache*' functions). The 'store*' functions use native endian. The 'cache*' functions use big endian or other formats in OpenType. Beware of memory alignment.

#### 5.1.3.2 byte

typedef unsigned char byte

Definition of "byte" type as an unsigned char.

Definition at line 97 of file hex2otf.c.

#### 5.1.3.3 Glyph

typedef struct Glyph Glyph

Data structure to hold data for one bitmap glyph.

This data structure holds data to represent one Unifont bitmap glyph: Unicode code point, number of bytes in its bitmap array, whether or not it is a combining character, and an offset from the glyph origin to the start of the bitmap.

#### 5.1.3.4 NameStrings

typedef const char* NameStrings[MAX_NAME_IDS]

Array of OpenType names indexed directly by Name IDs.

Definition at line 604 of file hex2otf.c.

#### 5.1.3.5 Options

typedef struct Options Options

Data structure to hold options for OpenType font output.

This data structure holds the status of options that can be specified as command line arguments for creating the output OpenType font file.

#### 5.1.3.6 pixels_t

typedef int_least8_t pixels_t

This type must be able to represent max(GLYPH_MAX_WIDTH, GLYPH_HEIGHT).

Definition at line 100 of file hex2otf.c.

#### 5.1.3.7 Table

typedef struct Table Table

Data structure for an OpenType table.

This data structure contains a table tag and a pointer to the start of the buffer that holds data for this OpenType table.

For information on the OpenType tables and their structure, see https://docs.microsoft.com/en-us/typography/opentype/spec/otff#font-tables.

### 5.1.4 Enumeration Type Documentation

#### 5.1.4.1 ContourOp

enum ContourOp

Specify the current contour drawing operation.

Enumerator

| | |
|---|---|
| OP_CLOSE | Close the current contour path that was being drawn. |
| OP_POINT | Add one more (x,y) point to the contor being drawn. |

Definition at line 1136 of file hex2otf.c.

```
01136              {
01137     OP_CLOSE,    ///< Close the current contour path that was being drawn.
01138     OP_POINT     ///< Add one more (x,y) point to the contor being drawn.
01139 };
```

#### 5.1.4.2 FillSide

enum FillSide

Fill to the left side (CFF) or right side (TrueType) of a contour.

Enumerator

| | |
|---|---|
| FILL_LEFT | Draw outline counter-clockwise (CFF, PostScript). |
| FILL_RIGHT | Draw outline clockwise (TrueType). |

Definition at line 1144 of file hex2otf.c.

```
01144              {
01145     FILL_LEFT,    ///< Draw outline counter-clockwise (CFF, PostScript).
01146     FILL_RIGHT    ///< Draw outline clockwise (TrueType).
01147 };
```

#### 5.1.4.3 LocaFormat

enum LocaFormat

Index to Location ("loca") offset information.

This enumerated type encodes the type of offset to locations in a table. It denotes Offset16 (16-bit) and Offset32 (32-bit) offset types.

Enumerator

| | |
|---|---|
| LOCA_OFFSET16 | Offset to location is a 16-bit Offset16 value. |
| LOCA_OFFSET32 | Offset to location is a 32-bit Offset32 value. |

Definition at line 658 of file hex2otf.c.

```
00658         {
00659     LOCA_OFFSET16 = 0,    ///< Offset to location is a 16-bit Offset16 value
00660     LOCA_OFFSET32 = 1     ///< Offset to location is a 32-bit Offset32 value
00661 };
```

### 5.1.5 Function Documentation

#### 5.1.5.1 addTable()

```
void addTable (
        Font * font,
        const char tag[static 4],
        Buffer * content )
```

Add a TrueType or OpenType table to the font.

This function adds a TrueType or OpenType table to a font. The 4-byte table tag is passed as an unsigned 32-bit integer in big-endian format.

Parameters

| | | |
|---|---|---|
| in,out | font | The font to which a font table will be added. |

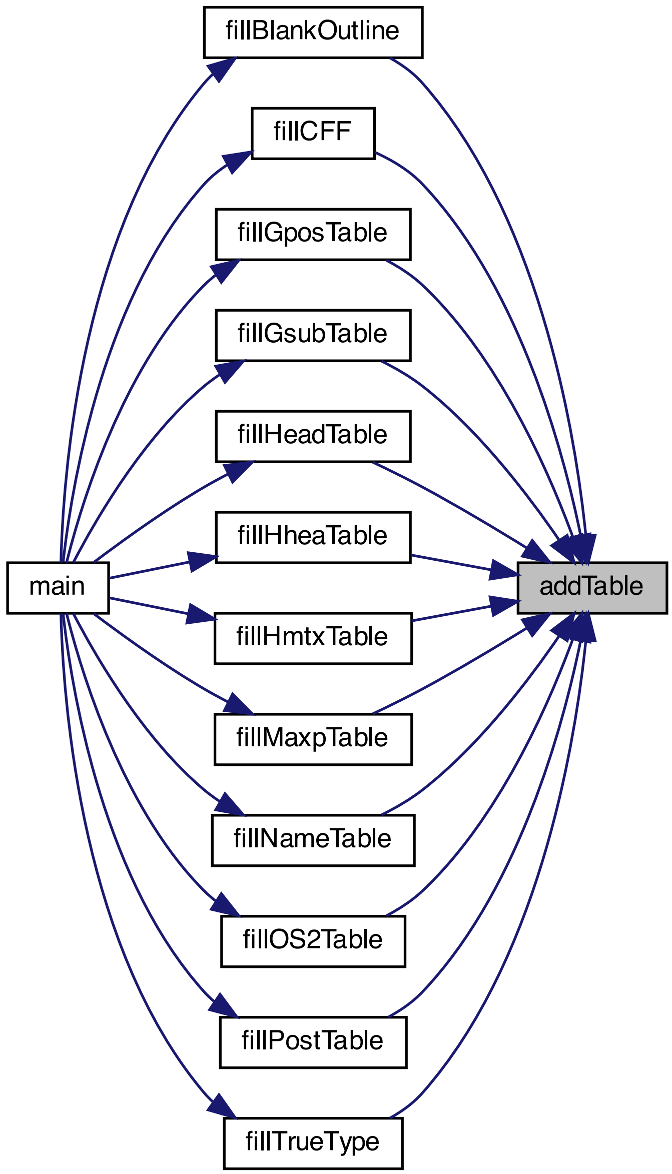
Parameters

| | | |
|---|---|---|
| in | tag | The 4-byte table name. |
| in | content | The table bytes to add, of type Buffer *. |

Definition at line 694 of file hex2otf.c.

```
{
    Table *table = getBufferSlot (font->tables, sizeof (Table));
    table->tag = tagAsU32 (tag);
    table->content = content;
}
```

Here is the caller graph for this function:




#### 5.1.5.2 buildOutline()

void buildOutline (
    Buffer ∗ result,

const byte bitmap[],
const size_t byteCount,
const enum FillSide fillSide )

Build a glyph outline.

This function builds a glyph outline from a Unifont glyph bitmap.

Parameters

| | | |
|---|---|---|
| out | result | The resulting glyph outline. |
| in | bitmap | A bitmap array. |
| in | byteCount | the number of bytes in the input bitmap array. |
| in | fillSide | Enumerated indicator to fill left or right side. |

Get the value of a given bit that is in a given row.

Invert the value of a given bit that is in a given row.

Definition at line 1160 of file hex2otf.c.

```
01162 {
01163     enum Direction {RIGHT, LEFT, DOWN, UP}; // order is significant
01164
01165     // respective coordinate deltas
01166     const pixels_t dx[] = {1, -1, 0, 0}, dy[] = {0, 0, -1, 1};
01167
01168     assert (byteCount % GLYPH_HEIGHT == 0);
01169     const uint_fast8_t bytesPerRow = byteCount / GLYPH_HEIGHT;
01170     const pixels_t glyphWidth = bytesPerRow * 8;
01171     assert (glyphWidth <= GLYPH_MAX_WIDTH);
01172
01173     #if GLYPH_MAX_WIDTH < 32
01174         typedef uint_fast32_t row_t;
01175     #elif GLYPH_MAX_WIDTH < 64
01176         typedef uint_fast64_t row_t;
01177     #else
01178         #error GLYPH_MAX_WIDTH is too large.
01179     #endif
01180
```

```
row_t pixels[GLYPH_HEIGHT + 2] = {0};
for (pixels_t row = GLYPH_HEIGHT; row > 0; row--)
    for (pixels_t b = 0; b < bytesPerRow; b++)
        pixels[row] = pixels[row] « 8 | *bitmap++;
typedef row_t graph_t[GLYPH_HEIGHT + 1];
graph_t vectors[4];
const row_t *lower = pixels, *upper = pixels + 1;
for (pixels_t row = 0; row <= GLYPH_HEIGHT; row++)
{
    const row_t m = (fillSide == FILL_RIGHT) - 1;
    vectors[RIGHT][row] = (m ^ (*lower « 1)) & (~m ^ (*upper « 1));
    vectors[LEFT ][row] = (m ^ (*upper     )) & (~m ^ (*lower     ));
    vectors[DOWN ][row] = (m ^ (*lower     )) & (~m ^ (*lower « 1));
    vectors[UP   ][row] = (m ^ (*upper « 1)) & (~m ^ (*upper     ));
    lower++;
    upper++;
}
graph_t selection = {0};
const row_t x0 = (row_t)1 « glyphWidth;

/// Get the value of a given bit that is in a given row.
#define getRowBit(rows, x, y)  ((rows)[(y)] &  x0 » (x))

/// Invert the value of a given bit that is in a given row.
#define flipRowBit(rows, x, y) ((rows)[(y)] ^= x0 » (x))

for (pixels_t y = GLYPH_HEIGHT; y >= 0; y--)
{
    for (pixels_t x = 0; x <= glyphWidth; x++)
    {
        assert (!getRowBit (vectors[LEFT], x, y));
        assert (!getRowBit (vectors[UP], x, y));
        enum Direction initial;

        if (getRowBit (vectors[RIGHT], x, y))
            initial = RIGHT;
        else if (getRowBit (vectors[DOWN], x, y))
            initial = DOWN;
        else
            continue;

        static_assert ((GLYPH_MAX_WIDTH + 1) * (GLYPH_HEIGHT + 1) * 2 <=
            U16MAX, ”potential overflow”);

        uint_fast16_t lastPointCount = 0;
        for (bool converged = false;;)
        {
            uint_fast16_t pointCount = 0;
            enum Direction heading = initial;
            for (pixels_t tx = x, ty = y;;)
            {
                if (converged)
                {
                    storePixels (result, OP_POINT);
                    storePixels (result, tx);
                    storePixels (result, ty);
                }
                do
                {
                    if (converged)
                        flipRowBit (vectors[heading], tx, ty);
                    tx += dx[heading];
                    ty += dy[heading];
                } while (getRowBit (vectors[heading], tx, ty));
                if (tx == x && ty == y)
                    break;
                static_assert ((UP ^ DOWN) == 1 && (LEFT ^ RIGHT) == 1,
                    ”wrong enums”);
                heading = (heading & 2) ^ 2;
                heading |= !!getRowBit (selection, tx, ty);
                heading ^= !getRowBit (vectors[heading], tx, ty);
                assert (getRowBit (vectors[heading], tx, ty));
                flipRowBit (selection, tx, ty);
                pointCount++;
            }
            if (converged)
                break;
            converged = pointCount == lastPointCount;
            lastPointCount = pointCount;
        }

```

```
            storePixels (result, OP_CLOSE);
        }
    }
    #undef getRowBit
    #undef flipRowBit
}
```

#### 5.1.5.3 byCodePoint()

```
int byCodePoint (
        const void * a,
        const void * b )
```

Compare two Unicode code points to determine which is greater.

This function compares the Unicode code points contained within two Glyph data structures. The function returns 1 if the first code point is greater, and -1 if the second is greater.

Parameters

| | | |
|---|---|---|
| in | a | A Glyph data structure containing the first code point. |
| in | b | A Glyph data structure containing the second code point. |

Returns

1 if the code point a is greater, -1 if less, 0 if equal.

Definition at line 1040 of file hex2otf.c.

```
{
    const Glyph *const ga = a, *const gb = b;
    int gt = ga->codePoint > gb->codePoint;
    int lt = ga->codePoint < gb->codePoint;
    return gt - lt;
}
```

#### 5.1.5.4 byTableTag()

int byTableTag (
const void ∗ a,
const void ∗ b )

Compare tables by 4-byte unsigned table tag value.

This function takes two pointers to a TableRecord data structure and extracts the four-byte tag structure element for each. The two 32-bit numbers are then compared. If the first tag is greater than the first, then gt = 1 and lt = 0, and so 1 - 0 = 1 is returned. If the first is less than the second, then gt = 0 and lt = 1, and so 0 - 1 = -1 is returned.

Parameters

| | | |
|---|---|---|
| in | a | Pointer to the first TableRecord structure. |
| in | b | Pointer to the second TableRecord structure. |

Returns

1 if the tag in "a" is greater, -1 if less, 0 if equal.

Definition at line 767 of file hex2otf.c.

```
{
    const struct TableRecord *const ra = a, *const rb = b;
    int gt = ra->tag > rb->tag;
    int lt = ra->tag < rb->tag;
    return gt - lt;
}
```

#### 5.1.5.5 cacheBuffer()

void cacheBuffer (
Buffer ∗restrict bufDest,
const Buffer ∗restrict bufSrc )

Append bytes of a table to a byte buffer.

Parameters

| | | |
|---|---|---|
| in,out | bufDest | The buffer to which the new bytes are appended. |
| in | bufSrc | The bytes to append to the buffer array. |

Definition at line 523 of file hex2otf.c.

```
{
    size_t length = countBufferedBytes (bufSrc);
    ensureBuffer (bufDest, length);
    memcpy (bufDest->next, bufSrc->begin, length);
    bufDest->next += length;
}
```

#### 5.1.5.6 cacheBytes()

void cacheBytes (
Buffer ∗restrict buf,
const void ∗restrict src,
size_t count )

Append a string of bytes to a buffer.

This function appends an array of 1 to 4 bytes to the end of a buffer.

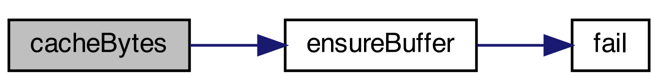
Parameters

| | | |
|---|---|---|
| in,out | buf | The buffer to which the bytes are appended. |
| in | src | The array of bytes to append to the buffer. |
| in | count | The number of bytes containing zeroes to append. |

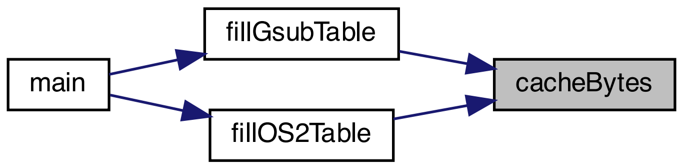
Definition at line 509 of file hex2otf.c.

```
{
    ensureBuffer (buf, count);
    memcpy (buf->next, src, count);
    buf->next += count;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.7 cacheCFFOperand()

```
void cacheCFFOperand (
        Buffer * buf,
        int_fast32_t value )
```

Cache charstring number encoding in a CFF buffer.

This function caches two's complement 8-, 16-, and 32-bit words as per Adobe's Type 2 Charstring encoding for operands. These operands are used in Compact Font Format data structures.

Byte values can have offsets, for which this function compensates, optionally followed by additional bytes:

| Byte Range | Offset | Bytes | Adjusted Range |
|---|---|---|---|
| 0 to 11 | 0 | 1 | 0 to 11 (operators) |
| 12 | 0 | 2 | Next byte is 8-bit op code |
| 13 to 18 | 0 | 1 | 13 to 18 (operators) |
| 19 to 20 | 0 | 2+ | hintmask and cntrmask operators |
| 21 to 27 | 0 | 1 | 21 to 27 (operators) |
| 28 | 0 | 3 | 16-bit 2's complement number |
| 29 to 31 | 0 | 1 | 29 to 31 (operators) |
| 32 to 246 | -139 | 1 | -107 to +107 |
| 247 to 250 | +108 | 2 | +108 to +1131 |
| 251 to 254 | -108 | 2 | -108 to -1131 |
| 255 | 0 | 5 | 16-bit integer and 16-bit fraction |

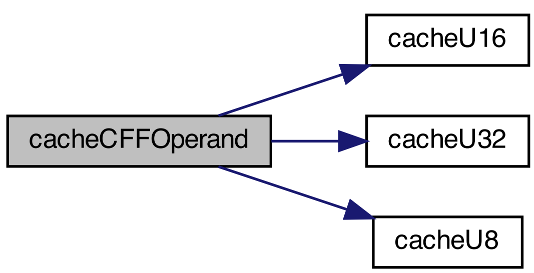
Parameters

| | | |
|---|---|---|
| in,out | buf | The buffer to which the operand value is appended. |
| in | value | The operand value. |

Definition at line 460 of file hex2otf.c.

```
{
    if (-107 <= value && value <= 107)
        cacheU8 (buf, value + 139);
    else if (108 <= value && value <= 1131)
    {
        cacheU8 (buf, (value - 108) / 256 + 247);
        cacheU8 (buf, (value - 108) % 256);
    }
    else if (-32768 <= value && value <= 32767)
    {
        cacheU8 (buf, 28);
        cacheU16 (buf, value);
    }
    else if (-2147483647 <= value && value <= 2147483647)
    {
        cacheU8 (buf, 29);
        cacheU32 (buf, value);
    }
    else
        assert (false); // other encodings are not used and omitted
    static_assert (GLYPH_MAX_WIDTH <= 107, "More encodings are needed.");
}
```

Here is the call graph for this function:



### 5.1.5.8 cacheStringAsUTF16BE()

void cacheStringAsUTF16BE (
    Buffer ∗ buf,
    const char ∗ str )

Cache a string as a big-ending UTF-16 surrogate pair.

This function encodes a UTF-8 string as a big-endian UTF-16 surrogate pair.

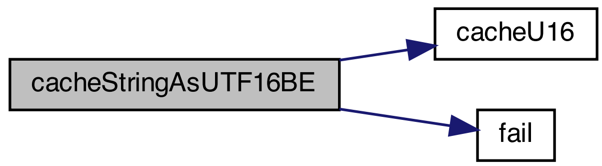
Parameters

| | | |
|---|---|---|
| in,out | buf | Pointer to a Buffer struct to update. |
| in | str | The character array to encode. |

Definition at line 2316 of file hex2otf.c.

```
{
    for (const char *p = str; *p; p++)
    {
        byte c = *p;
        if (c < 0x80)
        {
            cacheU16 (buf, c);
            continue;
        }
        int length = 1;
        byte mask = 0x40;
        for (; c & mask; mask >>= 1)
            length++;
        if (length == 1 || length > 4)
            fail ("Ill-formed UTF-8 sequence.");
        uint_fast32_t codePoint = c & (mask - 1);
        for (int i = 1; i < length; i++)
        {
            c = *++p;
            if ((c & 0xc0) != 0x80) // NUL checked here
                fail ("Ill-formed UTF-8 sequence.");
            codePoint = (codePoint << 6) | (c & 0x3f);
        }
        const int lowerBits = length==2 ? 7 : length==3 ? 11 : 16;
        if (codePoint >> lowerBits == 0)
            fail ("Ill-formed UTF-8 sequence."); // sequence should be shorter
        if (codePoint >= 0xd800 && codePoint <= 0xdfff)
            fail ("Ill-formed UTF-8 sequence.");
        if (codePoint > 0x10ffff)
            fail ("Ill-formed UTF-8 sequence.");
        if (codePoint > 0xffff)
        {
            cacheU16 (buf, 0xd800 | (codePoint - 0x10000) >> 10);
            cacheU16 (buf, 0xdc00 | (codePoint & 0x3ff));
        }
        else
            cacheU16 (buf, codePoint);
    }
}
```

Here is the call graph for this function:




#### 5.1.5.9 cacheU16()

void cacheU16 (
Buffer ∗ buf,
uint_fast16_t value )

Append two unsigned bytes to the end of a byte array.

This function adds two bytes to the end of a byte array. The buffer is updated to account for the newly-added bytes.

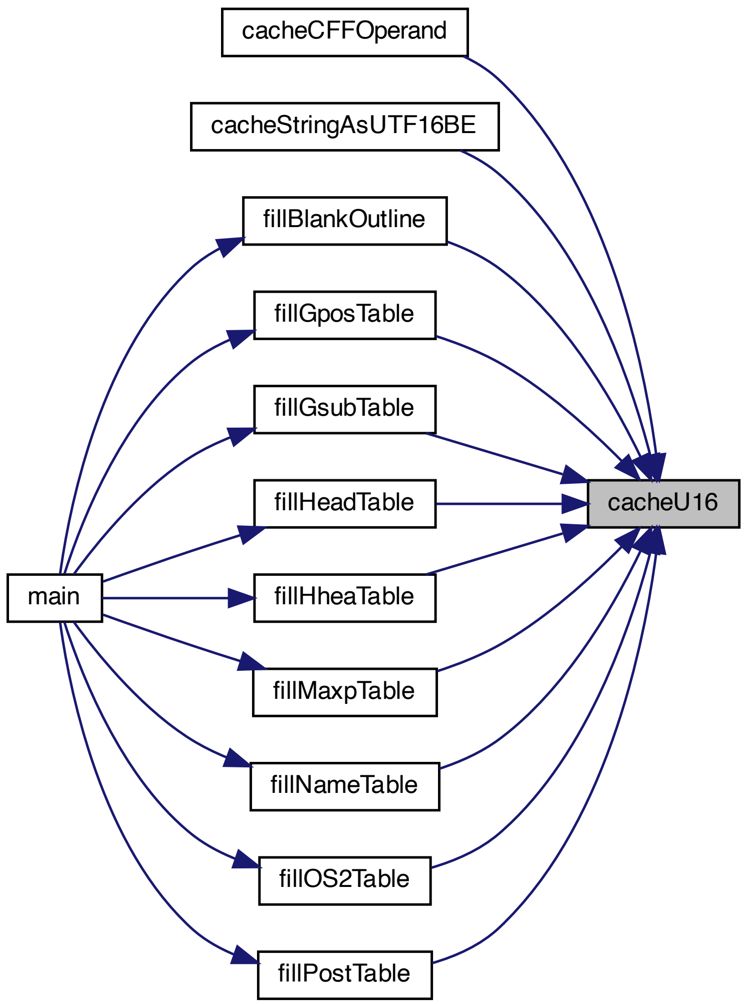
Parameters

| | | |
|---|---|---|
| in,out | buf | The array of bytes to which to append two new bytes. |
| in | value | The 16-bit unsigned value to append to the buf array. |

Definition at line 412 of file hex2otf.c.

```
00413 {
00414     cacheU (buf, value, 2);
00415 }
```

Here is the caller graph for this function:




#### 5.1.5.10 cacheU32()

void cacheU32 (
    Buffer ∗ buf,
    uint_fast32_t value )

Append four unsigned bytes to the end of a byte array.

This function adds four bytes to the end of a byte array. The buffer is updated to account for the newly-added bytes.

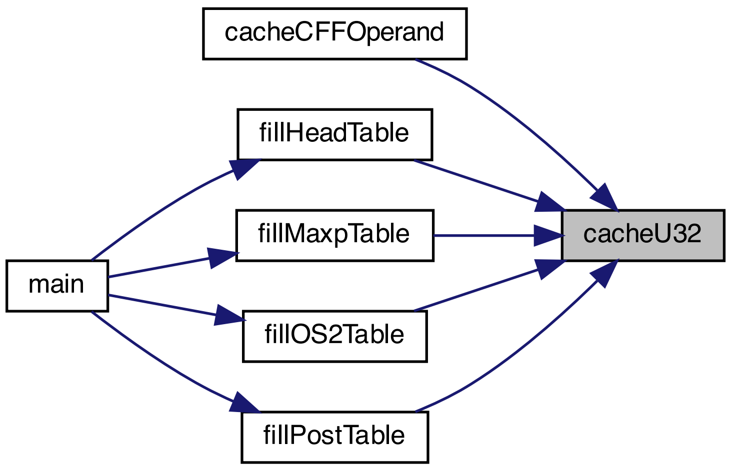
Parameters

| | | |
|---|---|---|
| in,out | buf | The array of bytes to which to append four new bytes. |
| in | value | The 32-bit unsigned value to append to the buf array. |

Definition at line 427 of file hex2otf.c.

```
00428 {
00429     cacheU (buf, value, 4);
00430 }
```

Here is the caller graph for this function:




#### 5.1.5.11 cacheU8()

void cacheU8 (
Buffer ∗ buf,
uint_fast8_t value )

Append one unsigned byte to the end of a byte array.

This function adds one byte to the end of a byte array. The buffer is updated to account for the newly-added byte.

Parameters

| | | |
|---|---|---|
| in,out | buf | The array of bytes to which to append a new byte. |

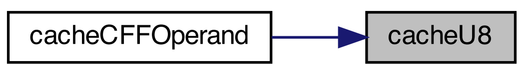
Parameters

| | | |
|---|---|---|
| in | value | The 8-bit unsigned value to append to the buf array. |

Definition at line 397 of file hex2otf.c.

```
00398 {
00399     storeU8 (buf, value & 0xff);
00400 }
```

Here is the caller graph for this function:




#### 5.1.5.12 cacheZeros()

void cacheZeros (
Buffer ∗ buf,
size_t count )

Append 1 to 4 bytes of zeroes to a buffer, for padding.

Parameters

| | | |
|---|---|---|
| in,out | buf | The buffer to which the operand value is appended. |

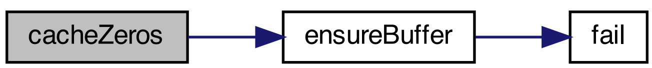
Parameters

| | | |
|---|---|---|
| in | count | The number of bytes containing zeroes to append. |

Definition at line 491 of file hex2otf.c.

```
00492 {
00493     ensureBuffer (buf, count);
00494     memset (buf->next, 0, count);
00495     buf->next += count;
00496 }
```

Here is the call graph for this function:




Here is the caller graph for this function:


#### 5.1.5.13 cleanBuffers()

void cleanBuffers ( )

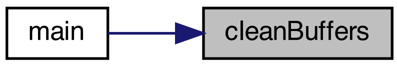
Free all allocated buffer pointers.

This function frees all buffer pointers previously allocated in the initBuffers function.

Definition at line 170 of file hex2otf.c.

```
{
    for (size_t i = 0; i < bufferCount; i++)
        if (allBuffers[i].capacity)
            free (allBuffers[i].begin);
    free (allBuffers);
    bufferCount = 0;
}
```

Here is the caller graph for this function:




#### 5.1.5.14 defineStore()

defineStore (
storeU8 ,
uint_least8_t )

Definition at line 356 of file hex2otf.c.

```
{
    assert (1 <= bytes && bytes <= 4);
    ensureBuffer (buf, bytes);
    switch (bytes)
    {
        case 4: *buf->next++ = value » 24 & 0xff; // fall through
        case 3: *buf->next++ = value » 16 & 0xff; // fall through
        case 2: *buf->next++ = value » 8  & 0xff; // fall through
        case 1: *buf->next++ = value      & 0xff;
    }
}
```

#### 5.1.5.15 ensureBuffer()

void ensureBuffer (
Buffer ∗ buf,
size_t needed )

Ensure that the buffer has at least the specified minimum size.

This function takes a buffer array of type Buffer and the necessary minimum number of elements as inputs, and attempts to increase the size of the buffer if it must be larger.

If the buffer is too small and cannot be resized, the program will terminate with an error message and an exit status of EXIT_FAILURE.

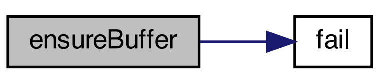
Parameters

| | | |
|---|---|---|
| in,out | buf | The buffer to check. |
| in | needed | The required minimum number of elements in the buffer. |

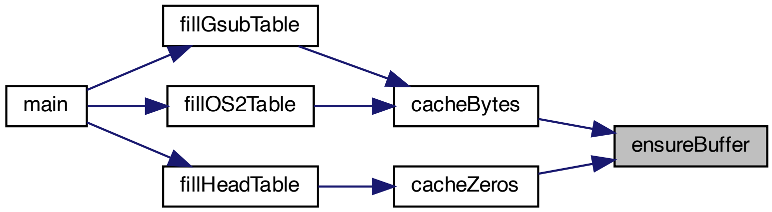
Definition at line 239 of file hex2otf.c.

```
{
    if (buf->end - buf->next >= needed)
        return;
    ptrdiff_t occupied = buf->next - buf->begin;
    size_t required = occupied + needed;
    if (required < needed) // overflow
        fail ("Cannot allocate %zu + %zu bytes of memory.", occupied, needed);
    if (required > SIZE_MAX / 2)
        buf->capacity = required;
    else while (buf->capacity < required)
        buf->capacity *= 2;
    void *extended = realloc (buf->begin, buf->capacity);
    if (!extended)
        fail ("Failed to allocate %zu bytes of memory.", buf->capacity);
    buf->begin = extended;
    buf->next = buf->begin + occupied;
    buf->end = buf->begin + buf->capacity;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.16 fail()

```
void fail (
        const char * reason,
         ... )
```

Print an error message on stderr, then exit.

This function prints the provided error string and optional following arguments to stderr, and then exits with a status of EXIT_FAILURE.

Parameters

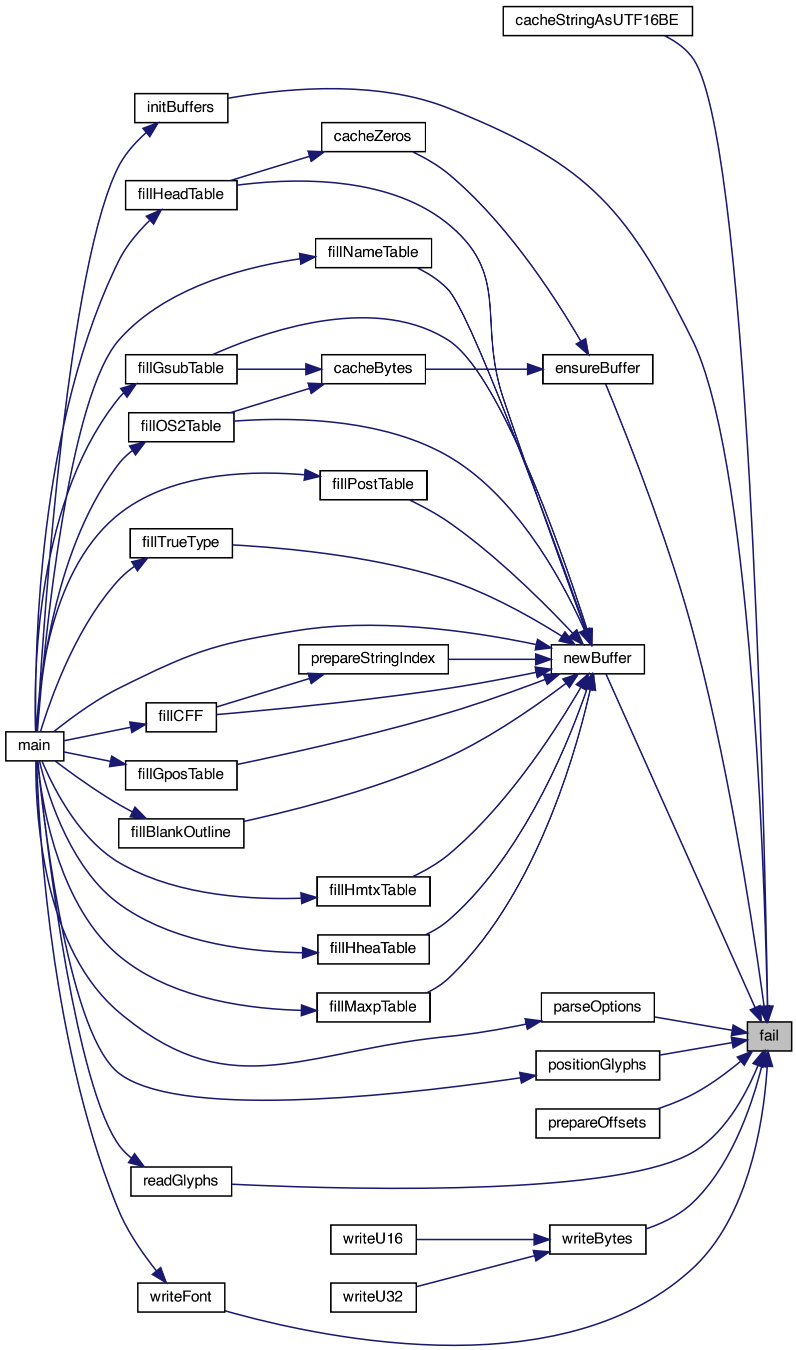
| | | |
|---|---|---|
| in | reason | The output string to describe the error. |
| in | ... | Optional following arguments to output. |

Definition at line 113 of file hex2otf.c.

```
00114 {
00115     fputs ("ERROR: ", stderr);
```

```
00116    va_list args;
00117    va_start (args, reason);
00118    vfprintf (stderr, reason, args);
00119    va_end (args);
00120    putc ('\n', stderr);
00121    exit (EXIT_FAILURE);
00122 }
```

Here is the caller graph for this function:



#### 5.1.5.17 fillBitmap()

void fillBitmap (
Font ∗ font )

Fill OpenType bitmap data and location tables.

This function fills an Embedded Bitmap Data (EBDT) Table and an Embedded Bitmap Location (EBLC) Table with glyph bitmap information. These tables enable embedding bitmaps in OpenType fonts. No Embedded Bitmap Scaling (EBSC) table is used for the bitmap glyphs, only EBDT and EBLC.

Parameters

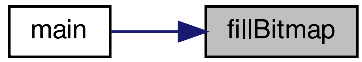
| | | |
|---|---|---|
| in,out | font | Pointer to a Font struct in which to add bitmaps. |

Definition at line 1728 of file hex2otf.c.

```
{
    const Glyph *const glyphs = getBufferHead (font->glyphs);
    const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
    size_t bitmapsSize = 0;
    for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
        bitmapsSize += glyph->byteCount;
    Buffer *ebdt = newBuffer (4 + bitmapsSize);
    addTable (font, "EBDT", ebdt);
    cacheU16 (ebdt, 2); // majorVersion
    cacheU16 (ebdt, 0); // minorVersion
    uint_fast8_t byteCount = 0; // unequal to any glyph
    pixels_t pos = 0;
    bool combining = false;
    Buffer *rangeHeads = newBuffer (32);
    Buffer *offsets = newBuffer (64);
    for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
    {
        if (glyph->byteCount != byteCount || glyph->pos != pos ||
            glyph->combining != combining)
        {
            storeU16 (rangeHeads, glyph - glyphs);
            storeU32 (offsets, countBufferedBytes (ebdt));
            byteCount = glyph->byteCount;
            pos = glyph->pos;
            combining = glyph->combining;
        }
        cacheBytes (ebdt, glyph->bitmap, byteCount);
    }
    const uint_least16_t *ranges = getBufferHead (rangeHeads);
    const uint_least16_t *rangesEnd = getBufferTail (rangeHeads);
    uint_fast32_t rangeCount = rangesEnd - ranges;
    storeU16 (rangeHeads, font->glyphCount);
    Buffer *eblc = newBuffer (4096);
    addTable (font, "EBLC", eblc);
    cacheU16 (eblc, 2); // majorVersion
    cacheU16 (eblc, 0); // minorVersion
    cacheU32 (eblc, 1); // numSizes
    { // bitmapSizes[0]
        cacheU32 (eblc, 56); // indexSubTableArrayOffset
        cacheU32 (eblc, (8 + 20) * rangeCount); // indexTablesSize
```

```
        cacheU32 (eblc, rangeCount); // numberOfIndexSubTables
        cacheU32 (eblc, 0); // colorRef
        { // hori
            cacheU8 (eblc, ASCENDER); // ascender
            cacheU8 (eblc, -DESCENDER); // descender
            cacheU8 (eblc, font->maxWidth); // widthMax
            cacheU8 (eblc, 1); // caretSlopeNumerator
            cacheU8 (eblc, 0); // caretSlopeDenominator
            cacheU8 (eblc, 0); // caretOffset
            cacheU8 (eblc, 0); // minOriginSB
            cacheU8 (eblc, 0); // minAdvanceSB
            cacheU8 (eblc, ASCENDER); // maxBeforeBL
            cacheU8 (eblc, -DESCENDER); // minAfterBL
            cacheU8 (eblc, 0); // pad1
            cacheU8 (eblc, 0); // pad2
        }
        { // vert
            cacheU8 (eblc, ASCENDER); // ascender
            cacheU8 (eblc, -DESCENDER); // descender
            cacheU8 (eblc, font->maxWidth); // widthMax
            cacheU8 (eblc, 1); // caretSlopeNumerator
            cacheU8 (eblc, 0); // caretSlopeDenominator
            cacheU8 (eblc, 0); // caretOffset
            cacheU8 (eblc, 0); // minOriginSB
            cacheU8 (eblc, 0); // minAdvanceSB
            cacheU8 (eblc, ASCENDER); // maxBeforeBL
            cacheU8 (eblc, -DESCENDER); // minAfterBL
            cacheU8 (eblc, 0); // pad1
            cacheU8 (eblc, 0); // pad2
        }
        cacheU16 (eblc, 0); // startGlyphIndex
        cacheU16 (eblc, font->glyphCount - 1); // endGlyphIndex
        cacheU8 (eblc, 16); // ppemX
        cacheU8 (eblc, 16); // ppemY
        cacheU8 (eblc, 1); // bitDepth
        cacheU8 (eblc, 1); // flags = Horizontal
    }
    { // IndexSubTableArray
        uint_fast32_t offset = rangeCount * 8;
        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
        {
            cacheU16 (eblc, *p); // firstGlyphIndex
            cacheU16 (eblc, p[1] - 1); // lastGlyphIndex
            cacheU32 (eblc, offset); // additionalOffsetToIndexSubtable
            offset += 20;
        }
    }
    { // IndexSubTables
        const uint_least32_t *offset = getBufferHead (offsets);
        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
        {
            const Glyph *glyph = &glyphs[*p];
            cacheU16 (eblc, 2); // indexFormat
            cacheU16 (eblc, 5); // imageFormat
            cacheU32 (eblc, *offset++); // imageDataOffset
            cacheU32 (eblc, glyph->byteCount); // imageSize
            { // bigMetrics
                cacheU8 (eblc, GLYPH_HEIGHT); // height
                const uint_fast8_t width = PW (glyph->byteCount);
                cacheU8 (eblc, width); // width
                cacheU8 (eblc, glyph->pos); // horiBearingX
                cacheU8 (eblc, ASCENDER); // horiBearingY
                cacheU8 (eblc, glyph->combining ? 0 : width); // horiAdvance
                cacheU8 (eblc, 0); // vertBearingX
                cacheU8 (eblc, 0); // vertBearingY
                cacheU8 (eblc, GLYPH_HEIGHT); // vertAdvance
            }
        }
    }
    freeBuffer (rangeHeads);
    freeBuffer (offsets);
}
```

Here is the caller graph for this function:




### 5.1.5.18 fillBlankOutline()

void fillBlankOutline (
Font ∗ font )

Create a dummy blank outline in a font table.

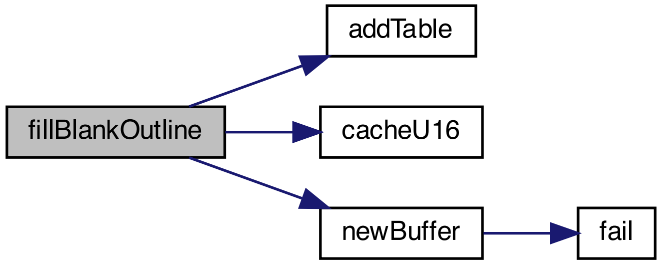
Parameters

| | | |
|---|---|---|
| in,out | font | Pointer to a Font struct to insert a blank outline. |

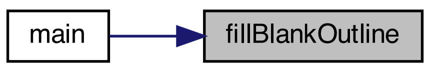
Definition at line 1697 of file hex2otf.c.

```
{
    Buffer *glyf = newBuffer (12);
    addTable (font, "glyf", glyf);
    // Empty table is not allowed, but an empty outline for glyph 0 suffices.
    cacheU16 (glyf, 0); // numberOfContours
    cacheU16 (glyf, FU (0)); // xMin
    cacheU16 (glyf, FU (0)); // yMin
    cacheU16 (glyf, FU (0)); // xMax
    cacheU16 (glyf, FU (0)); // yMax
    cacheU16 (glyf, 0); // instructionLength
    Buffer *loca = newBuffer (2 * (font->glyphCount + 1));
    addTable (font, "loca", loca);
    cacheU16 (loca, 0); // offsets[0]
    assert (countBufferedBytes (glyf) % 2 == 0);
    for (uint_fast32_t i = 1; i <= font->glyphCount; i++)
        cacheU16 (loca, countBufferedBytes (glyf) / 2); // offsets[i]
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.19 fillCFF()

void fillCFF (
    Font ∗ font,
    int version,
    const NameStrings names )

Add a CFF table to a font.

Parameters

| | | |
|---|---|---|
| in,out | font | Pointer to a Font struct to contain the CFF table. |
| in | version | Version of CFF table, with value 1 or 2. |
| in | names | List of NameStrings. |

Use fixed width integer for variables to simplify offset calculation.

Definition at line 1329 of file hex2otf.c.

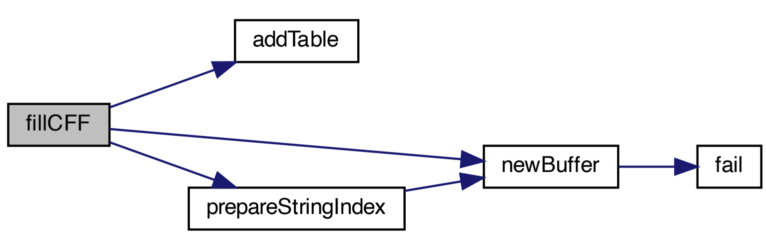
```
{
    // HACK: For convenience, CFF data structures are hard coded.
    assert (0 < version && version <= 2);
    Buffer *cff = newBuffer (65536);
    addTable (font, version == 1 ? "CFF " : "CFF2", cff);

    /// Use fixed width integer for variables to simplify offset calculation.
    #define cacheCFF32(buf, x) (cacheU8 ((buf), 29), cacheU32 ((buf), (x)))

    // In Unifont, 16px glyphs are more common. This is used by CFF1 only.
    const pixels_t defaultWidth = 16, nominalWidth = 8;
    if (version == 1)
    {
        Buffer *strings = prepareStringIndex (names);
        size_t stringsSize = countBufferedBytes (strings);
        const char *cffName = names[6];
        assert (cffName);
        size_t nameLength = strlen (cffName);
        size_t namesSize = nameLength + 5;
        // These sizes must be updated together with the data below.
        size_t offsets[] = {4, namesSize, 45, stringsSize, 2, 5, 8, 32, 4, 0};
        prepareOffsets (offsets);
        { // Header
            cacheU8 (cff, 1); // major
            cacheU8 (cff, 0); // minor
            cacheU8 (cff, 4); // hdrSize
            cacheU8 (cff, 1); // offSize
        }
        assert (countBufferedBytes (cff) == offsets[0]);
        { // Name INDEX (should not be used by OpenType readers)
            cacheU16 (cff, 1); // count
            cacheU8 (cff, 1); // offSize
            cacheU8 (cff, 1); // offset[0]
            if (nameLength + 1 > 255) // must be too long; spec limit is 63
                fail ("PostScript name is too long.");
            cacheU8 (cff, nameLength + 1); // offset[1]
            cacheBytes (cff, cffName, nameLength);
        }
        assert (countBufferedBytes (cff) == offsets[1]);
        { // Top DICT INDEX
            cacheU16 (cff, 1); // count
```

```
            cacheU8 (cff, 1); // offSize
            cacheU8 (cff, 1); // offset[0]
            cacheU8 (cff, 41); // offset[1]
            cacheCFFOperand (cff, 391); // "Adobe"
            cacheCFFOperand (cff, 392); // "Identity"
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, (byte[]){12, 30}, 2); // ROS
            cacheCFF32 (cff, font->glyphCount);
            cacheBytes (cff, (byte[]){12, 34}, 2); // CIDCount
            cacheCFF32 (cff, offsets[6]);
            cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
            cacheCFF32 (cff, offsets[5]);
            cacheBytes (cff, (byte[]){12, 37}, 2); // FDSelect
            cacheCFF32 (cff, offsets[4]);
            cacheU8 (cff, 15); // charset
            cacheCFF32 (cff, offsets[8]);
            cacheU8 (cff, 17); // CharStrings
        }
        assert (countBufferedBytes (cff) == offsets[2]);
        { // String INDEX
            cacheBuffer (cff, strings);
            freeBuffer (strings);
        }
        assert (countBufferedBytes (cff) == offsets[3]);
        cacheU16 (cff, 0); // Global Subr INDEX
        assert (countBufferedBytes (cff) == offsets[4]);
        { // Charsets
            cacheU8 (cff, 2); // format
            { // Range2[0]
                cacheU16 (cff, 1); // first
                cacheU16 (cff, font->glyphCount - 2); // nLeft
            }
        }
        assert (countBufferedBytes (cff) == offsets[5]);
        { // FDSelect
            cacheU8 (cff, 3); // format
            cacheU16 (cff, 1); // nRanges
            cacheU16 (cff, 0); // first
            cacheU8 (cff, 0); // fd
            cacheU16 (cff, font->glyphCount); // sentinel
        }
        assert (countBufferedBytes (cff) == offsets[6]);
        { // FDArray
            cacheU16 (cff, 1); // count
            cacheU8 (cff, 1); // offSize
            cacheU8 (cff, 1); // offset[0]
            cacheU8 (cff, 28); // offset[1]
            cacheCFFOperand (cff, 393);
            cacheBytes (cff, (byte[]){12, 38}, 2); // FontName
            // Windows requires FontMatrix in Font DICT.
            const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
            cacheCFFOperand (cff, offsets[8] - offsets[7]); // size
            cacheCFF32 (cff, offsets[7]); // offset
            cacheU8 (cff, 18); // Private
        }
        assert (countBufferedBytes (cff) == offsets[7]);
        { // Private
            cacheCFFOperand (cff, FU (defaultWidth));
            cacheU8 (cff, 20); // defaultWidthX
            cacheCFFOperand (cff, FU (nominalWidth));
            cacheU8 (cff, 21); // nominalWidthX
        }
        assert (countBufferedBytes (cff) == offsets[8]);
    }
    else
    {
        assert (version == 2);
        // These sizes must be updated together with the data below.
        size_t offsets[] = {5, 21, 4, 10, 0};
        prepareOffsets (offsets);
        { // Header
            cacheU8 (cff, 2); // majorVersion
            cacheU8 (cff, 0); // minorVersion
            cacheU8 (cff, 5); // headerSize
```

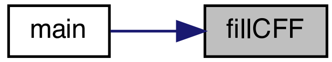
```
            cacheU16 (cff, offsets[1] - offsets[0]); // topDictLength
        }
        assert (countBufferedBytes (cff) == offsets[0]);
        { // Top DICT
            const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
            cacheCFFOperand (cff, offsets[2]);
            cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
            cacheCFFOperand (cff, offsets[3]);
            cacheU8 (cff, 17); // CharStrings
        }
        assert (countBufferedBytes (cff) == offsets[1]);
        cacheU32 (cff, 0); // Global Subr INDEX
        assert (countBufferedBytes (cff) == offsets[2]);
        { // Font DICT INDEX
            cacheU32 (cff, 1); // count
            cacheU8 (cff, 1); // offSize
            cacheU8 (cff, 1); // offset[0]
            cacheU8 (cff, 4); // offset[1]
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheU8 (cff, 18); // Private
        }
        assert (countBufferedBytes (cff) == offsets[3]);
    }
    { // CharStrings INDEX
        Buffer *offsets = newBuffer (4096);
        Buffer *charstrings = newBuffer (4096);
        Buffer *outline = newBuffer (1024);
        const Glyph *glyph = getBufferHead (font->glyphs);
        const Glyph *const endGlyph = glyph + font->glyphCount;
        for (; glyph < endGlyph; glyph++)
        {
            // CFF offsets start at 1
            storeU32 (offsets, countBufferedBytes (charstrings) + 1);

            pixels_t rx = -glyph->pos;
            pixels_t ry = DESCENDER;
            resetBuffer (outline);
            buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_LEFT);
            enum CFFOp {rmoveto=21, hmoveto=22, vmoveto=4, hlineto=6,
                vlineto=7, endchar=14};
            enum CFFOp pendingOp = 0;
            const int STACK_LIMIT = version == 1 ? 48 : 513;
            int stackSize = 0;
            bool isDrawing = false;
            pixels_t width = glyph->combining ? 0 : PW (glyph->byteCount);
            if (version == 1 && width != defaultWidth)
            {
                cacheCFFOperand (charstrings, FU (width - nominalWidth));
                stackSize++;
            }
            for (const pixels_t *p = getBufferHead (outline),
                 *const end = getBufferTail (outline); p < end;)
            {
                int s = 0;
                const enum ContourOp op = *p++;
                if (op == OP_POINT)
                {
                    const pixels_t x = *p++, y = *p++;
                    if (x != rx)
                    {
                        cacheCFFOperand (charstrings, FU (x - rx));
                        rx = x;
                        stackSize++;
                        s |= 1;
                    }
                    if (y != ry)
                    {
                        cacheCFFOperand (charstrings, FU (y - ry));
                        ry = y;
                        stackSize++;
                        s |= 2;
                    }
                    assert (!(isDrawing && s == 3));
```

```
                }
                if (s)
                {
                    if (!isDrawing)
                    {
                        const enum CFFOp moves[] = {0, hmoveto, vmoveto,
                            rmoveto};
                        cacheU8 (charstrings, moves[s]);
                        stackSize = 0;
                    }
                    else if (!pendingOp)
                        pendingOp = (enum CFFOp[]){0, hlineto, vlineto}[s];
                }
                else if (!isDrawing)
                {
                    // only when the first point happens to be (0, 0)
                    cacheCFFOperand (charstrings, FU (0));
                    cacheU8 (charstrings, hmoveto);
                    stackSize = 0;
                }
                if (op == OP_CLOSE || stackSize >= STACK_LIMIT)
                {
                    assert (stackSize <= STACK_LIMIT);
                    cacheU8 (charstrings, pendingOp);
                    pendingOp = 0;
                    stackSize = 0;
                }
                isDrawing = op != OP_CLOSE;
            }
            if (version == 1)
                cacheU8 (charstrings, endchar);
        }
        size_t lastOffset = countBufferedBytes (charstrings) + 1;
        #if SIZE_MAX > U32MAX
            if (lastOffset > U32MAX)
                fail ("CFF data exceeded size limit.");
        #endif
        storeU32 (offsets, lastOffset);
        int offsetSize = 1 + (lastOffset > 0xff)
                           + (lastOffset > 0xffff)
                           + (lastOffset > 0xffffff);
        // count (must match 'numGlyphs' in 'maxp' table)
        cacheU (cff, font->glyphCount, version * 2);
        cacheU8 (cff, offsetSize); // offSize
        const uint_least32_t *p = getBufferHead (offsets);
        const uint_least32_t *const end = getBufferTail (offsets);
        for (; p < end; p++)
            cacheU (cff, *p, offsetSize); // offsets
        cacheBuffer (cff, charstrings); // data
        freeBuffer (offsets);
        freeBuffer (charstrings);
        freeBuffer (outline);
    }
    #undef cacheCFF32
}
```

Here is the call graph for this function:



Here is the caller graph for this function:




### 5.1.5.20 fillCmapTable()

void fillCmapTable (
    Font * font )

Fill a "cmap" font table.

The "cmap" table contains character to glyph index mapping information.

Parameters

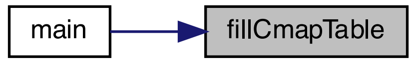
| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

Definition at line 2109 of file hex2otf.c.

```
02110 {
02111     Glyph *const glyphs = getBufferHead (font->glyphs);
02112     Buffer *rangeHeads = newBuffer (16);
02113     uint_fast32_t rangeCount = 0;
02114     uint_fast32_t bmpRangeCount = 1; // 1 for the last 0xffff-0xffff range
02115     glyphs[0].codePoint = glyphs[1].codePoint; // to start a range at glyph 1
02116     for (uint_fast16_t i = 1; i < font->glyphCount; i++)
02117     {
02118         if (glyphs[i].codePoint != glyphs[i - 1].codePoint + 1)
02119         {
02120             storeU16 (rangeHeads, i);
02121             rangeCount++;
02122             bmpRangeCount += glyphs[i].codePoint < 0xffff;
02123         }
02124     }
02125     Buffer *cmap = newBuffer (256);
02126     addTable (font, "cmap", cmap);
02127     // Format 4 table is always generated for compatibility.
02128     bool hasFormat12 = glyphs[font->glyphCount - 1].codePoint > 0xffff;
02129     cacheU16 (cmap, 0); // version
02130     cacheU16 (cmap, 1 + hasFormat12); // numTables
02131     { // encodingRecords[0]
02132         cacheU16 (cmap, 3); // platformID
```

```
        cacheU16 (cmap, 1); // encodingID
        cacheU32 (cmap, 12 + 8 * hasFormat12); // subtableOffset
    }
    if (hasFormat12) // encodingRecords[1]
    {
        cacheU16 (cmap, 3); // platformID
        cacheU16 (cmap, 10); // encodingID
        cacheU32 (cmap, 36 + 8 * bmpRangeCount); // subtableOffset
    }
    const uint_least16_t *ranges = getBufferHead (rangeHeads);
    const uint_least16_t *const rangesEnd = getBufferTail (rangeHeads);
    storeU16 (rangeHeads, font->glyphCount);
    { // format 4 table
        cacheU16 (cmap, 4); // format
        cacheU16 (cmap, 16 + 8 * bmpRangeCount); // length
        cacheU16 (cmap, 0); // language
        if (bmpRangeCount * 2 > U16MAX)
            fail (”Too many ranges in ’cmap’ table.”);
        cacheU16 (cmap, bmpRangeCount * 2); // segCountX2
        uint_fast16_t searchRange = 1, entrySelector = -1;
        while (searchRange <= bmpRangeCount)
        {
            searchRange «= 1;
            entrySelector++;
        }
        cacheU16 (cmap, searchRange); // searchRange
        cacheU16 (cmap, entrySelector); // entrySelector
        cacheU16 (cmap, bmpRangeCount * 2 - searchRange); // rangeShift
        { // endCode[]
            const uint_least16_t *p = ranges;
            for (p++; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
                cacheU16 (cmap, glyphs[*p - 1].codePoint);
            uint_fast32_t cp = glyphs[*p - 1].codePoint;
            if (cp > 0xfffe)
                cp = 0xfffe;
            cacheU16 (cmap, cp);
            cacheU16 (cmap, 0xffff);
        }
        cacheU16 (cmap, 0); // reservedPad
        { // startCode[]
            for (uint_fast32_t i = 0; i < bmpRangeCount - 1; i++)
                cacheU16 (cmap, glyphs[ranges[i]].codePoint);
            cacheU16 (cmap, 0xffff);
        }
        { // idDelta[]
            const uint_least16_t *p = ranges;
            for (; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
                cacheU16 (cmap, *p - glyphs[*p].codePoint);
            uint_fast16_t delta = 1;
            if (p < rangesEnd && *p == 0xffff)
                delta = *p - glyphs[*p].codePoint;
            cacheU16 (cmap, delta);
        }
        { // idRangeOffsets[]
            for (uint_least16_t i = 0; i < bmpRangeCount; i++)
                cacheU16 (cmap, 0);
        }
    }
    if (hasFormat12) // format 12 table
    {
        cacheU16 (cmap, 12); // format
        cacheU16 (cmap, 0); // reserved
        cacheU32 (cmap, 16 + 12 * rangeCount); // length
        cacheU32 (cmap, 0); // language
        cacheU32 (cmap, rangeCount); // numGroups

        // groups[]
        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
        {
            cacheU32 (cmap, glyphs[*p].codePoint); // startCharCode
            cacheU32 (cmap, glyphs[p[1] - 1].codePoint); // endCharCode
            cacheU32 (cmap, *p); // startGlyphID
        }
    }
    freeBuffer (rangeHeads);
}
```

Here is the caller graph for this function:




#### 5.1.5.21 fillGposTable()

void fillGposTable (
    Font * font )

Fill a "GPOS" font table.

The "GPOS" table contains information for glyph positioning.

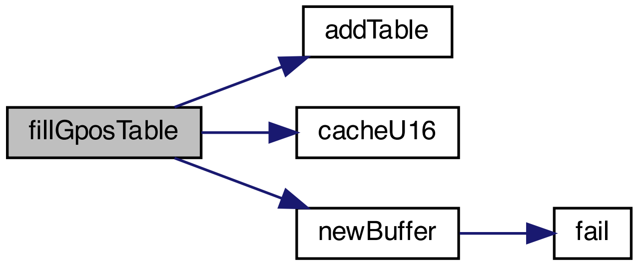
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

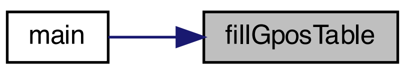
Definition at line 2241 of file hex2otf.c.

```
{
    Buffer *gpos = newBuffer (16);
    addTable (font, "GPOS", gpos);
    cacheU16 (gpos, 1); // majorVersion
    cacheU16 (gpos, 0); // minorVersion
    cacheU16 (gpos, 10); // scriptListOffset
    cacheU16 (gpos, 12); // featureListOffset
    cacheU16 (gpos, 14); // lookupListOffset
    { // ScriptList table
        cacheU16 (gpos, 0); // scriptCount
    }
    { // Feature List table
        cacheU16 (gpos, 0); // featureCount
    }
    { // Lookup List Table
        cacheU16 (gpos, 0); // lookupCount
    }
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.22 fillGsubTable()

void fillGsubTable (
Font * font )

Fill a "GSUB" font table.

The "GSUB" table contains information for glyph substitution.

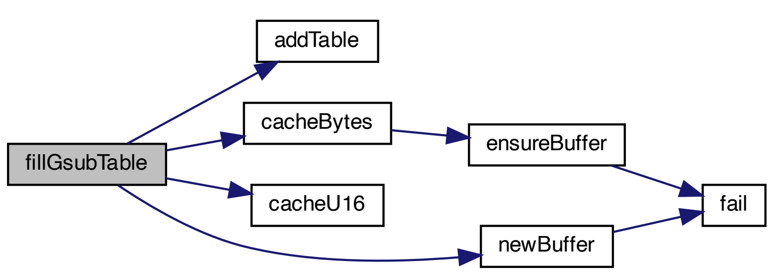
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

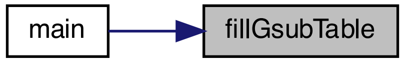
Definition at line 2269 of file hex2otf.c.

```
{
    Buffer *gsub = newBuffer (38);
    addTable (font, "GSUB", gsub);
    cacheU16 (gsub, 1); // majorVersion
    cacheU16 (gsub, 0); // minorVersion
    cacheU16 (gsub, 10); // scriptListOffset
    cacheU16 (gsub, 34); // featureListOffset
    cacheU16 (gsub, 36); // lookupListOffset
    { // ScriptList table
        cacheU16 (gsub, 2); // scriptCount
        { // scriptRecords[0]
            cacheBytes (gsub, "DFLT", 4); // scriptTag
            cacheU16 (gsub, 14); // scriptOffset
        }
        { // scriptRecords[1]
            cacheBytes (gsub, "thai", 4); // scriptTag
            cacheU16 (gsub, 14); // scriptOffset
        }
        { // Script table
            cacheU16 (gsub, 4); // defaultLangSysOffset
            cacheU16 (gsub, 0); // langSysCount
            { // Default Language System table
                cacheU16 (gsub, 0); // lookupOrderOffset
                cacheU16 (gsub, 0); // requiredFeatureIndex
                cacheU16 (gsub, 0); // featureIndexCount
            }
        }
    }
    { // Feature List table
        cacheU16 (gsub, 0); // featureCount
    }
    { // Lookup List Table
        cacheU16 (gsub, 0); // lookupCount
    }
}
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.23 fillHeadTable()

void fillHeadTable (
Font ∗ font,
enum LocaFormat locaFormat,
pixels_t xMin )

Fill a "head" font table.

The "head" table contains font header information common to the whole font.

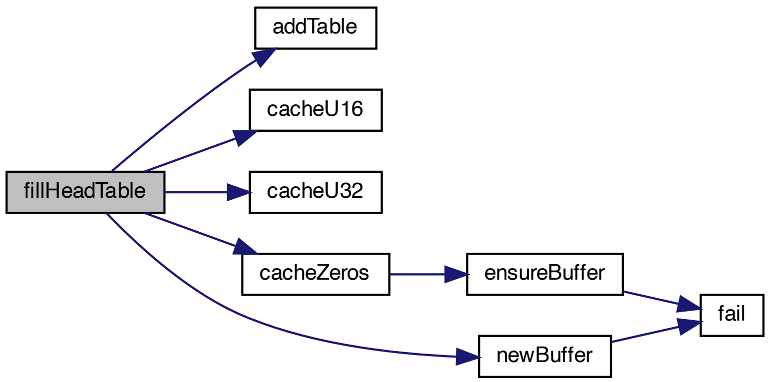
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |
| in | locaFormat | The "loca" offset index location table. |
| in | xMin | The minimum x-coordinate for a glyph. |

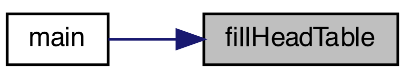
Definition at line 1853 of file hex2otf.c.

```
{
    Buffer *head = newBuffer (56);
    addTable (font, "head", head);
    cacheU16 (head, 1); // majorVersion
    cacheU16 (head, 0); // minorVersion
    cacheZeros (head, 4); // fontRevision (unused)
    // The 'checksumAdjustment' field is a checksum of the entire file.
    // It is later calculated and written directly in the 'writeFont' function.
    cacheU32 (head, 0); // checksumAdjustment (placeholder)
    cacheU32 (head, 0x5f0f3cf5); // magicNumber
    const uint_fast16_t flags =
        + B1 ( 0) // baseline at y=0
        + B1 ( 1) // LSB at x=0 (doubtful; probably should be LSB=xMin)
        + B0 ( 2) // instructions may depend on point size
        + B0 ( 3) // force internal ppem to integers
        + B0 ( 4) // instructions may alter advance width
        + B0 ( 5) // not used in OpenType
        + B0 ( 6) // not used in OpenType
        + B0 ( 7) // not used in OpenType
        + B0 ( 8) // not used in OpenType
        + B0 ( 9) // not used in OpenType
        + B0 (10) // not used in OpenType
        + B0 (11) // font transformed
        + B0 (12) // font converted
        + B0 (13) // font optimized for ClearType
        + B0 (14) // last resort font
        + B0 (15) // reserved
    ;
    cacheU16 (head, flags); // flags
    cacheU16 (head, FUPEM); // unitsPerEm
    cacheZeros (head, 8); // created (unused)
    cacheZeros (head, 8); // modified (unused)
    cacheU16 (head, FU (xMin)); // xMin
    cacheU16 (head, FU (-DESCENDER)); // yMin
    cacheU16 (head, FU (font->maxWidth)); // xMax
    cacheU16 (head, FU (ASCENDER)); // yMax
    // macStyle (must agree with 'fsSelection' in 'OS/2' table)
    const uint_fast16_t macStyle =
        + B0 (0) // bold
        + B0 (1) // italic
        + B0 (2) // underline
        + B0 (3) // outline
        + B0 (4) // shadow
        + B0 (5) // condensed
        + B0 (6) // extended
        //    7-15  reserved
    ;
    cacheU16 (head, macStyle);
    cacheU16 (head, GLYPH_HEIGHT); // lowestRecPPEM
    cacheU16 (head, 2); // fontDirectionHint
    cacheU16 (head, locaFormat); // indexToLocFormat
    cacheU16 (head, 0); // glyphDataFormat
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.24 fillHheaTable()

void fillHheaTable (
Font ∗ font,
pixels_t xMin )

Fill a "hhea" font table.

The "hhea" table contains horizontal header information, for example left and right side bearings.

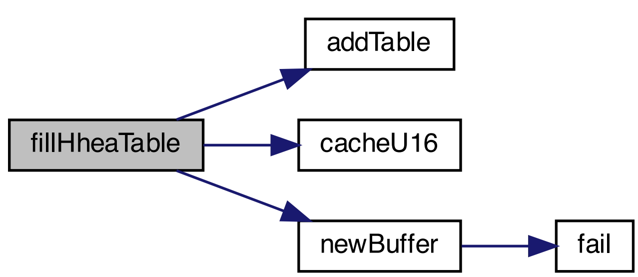
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |
| in | xMin | The minimum x-coordinate for a glyph. |

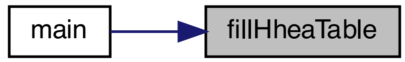
Definition at line 1918 of file hex2otf.c.

```
{
    Buffer *hhea = newBuffer (36);
    addTable (font, "hhea", hhea);
    cacheU16 (hhea, 1); // majorVersion
    cacheU16 (hhea, 0); // minorVersion
    cacheU16 (hhea, FU (ASCENDER)); // ascender
    cacheU16 (hhea, FU (-DESCENDER)); // descender
    cacheU16 (hhea, FU (0)); // lineGap
    cacheU16 (hhea, FU (font->maxWidth)); // advanceWidthMax
    cacheU16 (hhea, FU (xMin)); // minLeftSideBearing
    cacheU16 (hhea, FU (0)); // minRightSideBearing (unused)
    cacheU16 (hhea, FU (font->maxWidth)); // xMaxExtent
    cacheU16 (hhea, 1); // caretSlopeRise
    cacheU16 (hhea, 0); // caretSlopeRun
    cacheU16 (hhea, 0); // caretOffset
    cacheU16 (hhea, 0); // reserved
    cacheU16 (hhea, 0); // reserved
    cacheU16 (hhea, 0); // reserved
    cacheU16 (hhea, 0); // reserved
    cacheU16 (hhea, 0); // metricDataFormat
    cacheU16 (hhea, font->glyphCount); // numberOfHMetrics
}
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.25 fillHmtxTable()

void fillHmtxTable (
Font ∗ font )

Fill an "hmtx" font table.

The "hmtx" table contains horizontal metrics information.

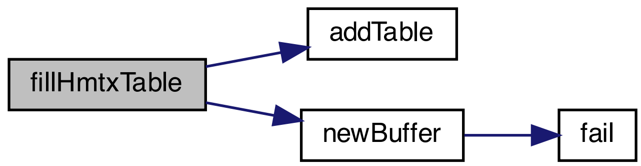
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

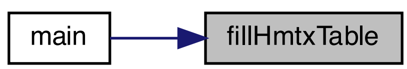
Definition at line 2087 of file hex2otf.c.

```
{
    Buffer *hmtx = newBuffer (4 * font->glyphCount);
    addTable (font, "hmtx", hmtx);
    const Glyph *const glyphs = getBufferHead (font->glyphs);
    const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
    for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
    {
        int_fast16_t aw = glyph->combining ? 0 : PW (glyph->byteCount);
        cacheU16 (hmtx, FU (aw)); // advanceWidth
        cacheU16 (hmtx, FU (glyph->lsb)); // lsb
    }
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.26 fillMaxpTable()

void fillMaxpTable (
Font ∗ font,
bool isCFF,
uint_fast16_t maxPoints,
uint_fast16_t maxContours )

Fill a "maxp" font table.

The "maxp" table contains maximum profile information, such as the memory required to contain the font.

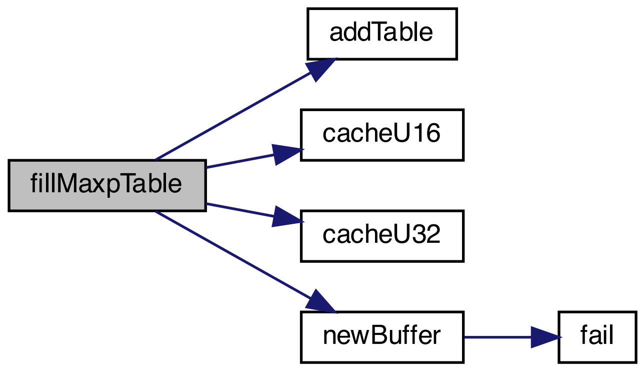
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |
| in | isCFF | true if a CFF font is included, false otherwise. |
| in | maxPoints | Maximum points in a non-composite glyph. |
| in | maxContours | Maximum contours in a non-composite glyph. |

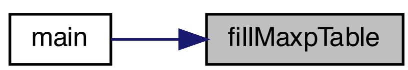
Definition at line 1954 of file hex2otf.c.

```
{
    Buffer *maxp = newBuffer (32);
    addTable (font, "maxp", maxp);
    cacheU32 (maxp, isCFF ? 0x00005000 : 0x00010000); // version
    cacheU16 (maxp, font->glyphCount); // numGlyphs
    if (isCFF)
        return;
    cacheU16 (maxp, maxPoints); // maxPoints
    cacheU16 (maxp, maxContours); // maxContours
    cacheU16 (maxp, 0); // maxCompositePoints
    cacheU16 (maxp, 0); // maxCompositeContours
    cacheU16 (maxp, 0); // maxZones
    cacheU16 (maxp, 0); // maxTwilightPoints
    cacheU16 (maxp, 0); // maxStorage
    cacheU16 (maxp, 0); // maxFunctionDefs
    cacheU16 (maxp, 0); // maxInstructionDefs
    cacheU16 (maxp, 0); // maxStackElements
    cacheU16 (maxp, 0); // maxSizeOfInstructions
    cacheU16 (maxp, 0); // maxComponentElements
    cacheU16 (maxp, 0); // maxComponentDepth
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.27 fillNameTable()

void fillNameTable (
Font ∗ font,
NameStrings nameStrings )

Fill a "name" font table.

The "name" table contains name information, for example for Name IDs.

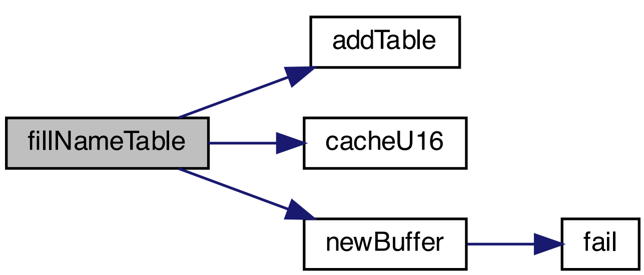
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |
| in | names | List of NameStrings. |

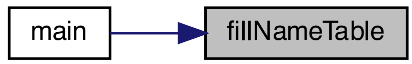
Definition at line 2366 of file hex2otf.c.

```
{
    Buffer *name = newBuffer (2048);
    addTable (font, "name", name);
    size_t nameStringCount = 0;
    for (size_t i = 0; i < MAX_NAME_IDS; i++)
        nameStringCount += !!nameStrings[i];
    cacheU16 (name, 0); // version
    cacheU16 (name, nameStringCount); // count
    cacheU16 (name, 2 * 3 + 12 * nameStringCount); // storageOffset
    Buffer *stringData = newBuffer (1024);
    // nameRecord[]
    for (size_t i = 0; i < MAX_NAME_IDS; i++)
    {
        if (!nameStrings[i])
            continue;
        size_t offset = countBufferedBytes (stringData);
        cacheStringAsUTF16BE (stringData, nameStrings[i]);
        size_t length = countBufferedBytes (stringData) - offset;
        if (offset > U16MAX || length > U16MAX)
            fail ("Name strings are too long.");
        // Platform ID 0 (Unicode) is not well supported.
        // ID 3 (Windows) seems to be the best for compatibility.
        cacheU16 (name, 3); // platformID = Windows
        cacheU16 (name, 1); // encodingID = Unicode BMP
        cacheU16 (name, 0x0409); // languageID = en-US
        cacheU16 (name, i); // nameID
        cacheU16 (name, length); // length
        cacheU16 (name, offset); // stringOffset
    }
    cacheBuffer (name, stringData);
    freeBuffer (stringData);
}
```

Here is the call graph for this function:



Here is the caller graph for this function:




### 5.1.5.28 fillOS2Table()

void fillOS2Table (
Font ∗ font )

Fill an "OS/2" font table.

The "OS/2" table contains OS/2 and Windows font metrics information.

Parameters

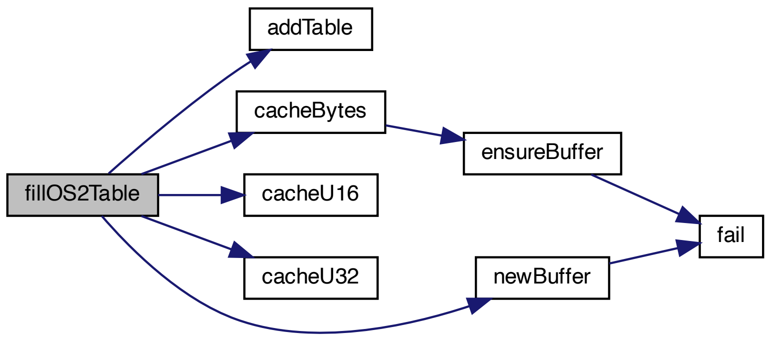
| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

Definition at line 1986 of file hex2otf.c.

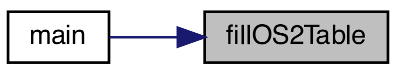
```
01987 {
01988     Buffer *os2 = newBuffer (100);
01989     addTable (font, "OS/2", os2);
01990     cacheU16 (os2, 5); // version
01991     // HACK: Average glyph width is not actually calculated.
01992     cacheU16 (os2, FU (font->maxWidth)); // xAvgCharWidth
01993     cacheU16 (os2, 400); // usWeightClass = Normal
01994     cacheU16 (os2, 5); // usWidthClass = Medium
01995     const uint_fast16_t typeFlags =
01996         + B0 (0) // reserved
01997         // usage permissions, one of:
01998             // Default: Installable embedding
01999             + B0 (1) // Restricted License embedding
02000             + B0 (2) // Preview & Print embedding
02001             + B0 (3) // Editable embedding
02002         //    4-7   reserved
02003         + B0 (8) // no subsetting
02004         + B0 (9) // bitmap embedding only
02005         //    10-15 reserved
02006     ;
02007     cacheU16 (os2, typeFlags); // fsType
02008     cacheU16 (os2, FU (5)); // ySubscriptXSize
02009     cacheU16 (os2, FU (7)); // ySubscriptYSize
```

```
    cacheU16 (os2, FU (0)); // ySubscriptXOffset
    cacheU16 (os2, FU (1)); // ySubscriptYOffset
    cacheU16 (os2, FU (5)); // ySuperscriptXSize
    cacheU16 (os2, FU (7)); // ySuperscriptYSize
    cacheU16 (os2, FU (0)); // ySuperscriptXOffset
    cacheU16 (os2, FU (4)); // ySuperscriptYOffset
    cacheU16 (os2, FU (1)); // yStrikeoutSize
    cacheU16 (os2, FU (5)); // yStrikeoutPosition
    cacheU16 (os2, 0x080a); // sFamilyClass = Sans Serif, Matrix
    const byte panose[] =
    {
        2, // Family Kind = Latin Text
        11, // Serif Style = Normal Sans
        4, // Weight = Thin
        // Windows would render all glyphs to the same width,
        // if 'Proportion' is set to 'Monospaced' (as Unifont should be).
        // 'Condensed' is the best alternative according to metrics.
        6, // Proportion = Condensed
        2, // Contrast = None
        2, // Stroke = No Variation
        2, // Arm Style = Straight Arms
        8, // Letterform = Normal/Square
        2, // Midline = Standard/Trimmed
        4, // X-height = Constant/Large
    };
    cacheBytes (os2, panose, sizeof panose); // panose
    // HACK: All defined Unicode ranges are marked functional for convenience.
    cacheU32 (os2, 0xffffffff); // ulUnicodeRange1
    cacheU32 (os2, 0xffffffff); // ulUnicodeRange2
    cacheU32 (os2, 0xffffffff); // ulUnicodeRange3
    cacheU32 (os2, 0x0effffff); // ulUnicodeRange4
    cacheBytes (os2, "GNU ", 4); // achVendID
    // fsSelection (must agree with 'macStyle' in 'head' table)
    const uint_fast16_t selection =
        + B0 (0) // italic
        + B0 (1) // underscored
        + B0 (2) // negative
        + B0 (3) // outlined
        + B0 (4) // strikeout
        + B0 (5) // bold
        + B1 (6) // regular
        + B1 (7) // use sTypo* metrics in this table
        + B1 (8) // font name conforms to WWS model
        + B0 (9) // oblique
        //    10-15 reserved
    ;
    cacheU16 (os2, selection);
    const Glyph *glyphs = getBufferHead (font->glyphs);
    uint_fast32_t first = glyphs[1].codePoint;
    uint_fast32_t last = glyphs[font->glyphCount - 1].codePoint;
    cacheU16 (os2, first < U16MAX ? first : U16MAX); // usFirstCharIndex
    cacheU16 (os2, last  < U16MAX ? last  : U16MAX); // usLastCharIndex
    cacheU16 (os2, FU (ASCENDER)); // sTypoAscender
    cacheU16 (os2, FU (-DESCENDER)); // sTypoDescender
    cacheU16 (os2, FU (0)); // sTypoLineGap
    cacheU16 (os2, FU (ASCENDER)); // usWinAscent
    cacheU16 (os2, FU (DESCENDER)); // usWinDescent
    // HACK: All reasonable code pages are marked functional for convenience.
    cacheU32 (os2, 0x603f01ff); // ulCodePageRange1
    cacheU32 (os2, 0xffff0000); // ulCodePageRange2
    cacheU16 (os2, FU (8)); // sxHeight
    cacheU16 (os2, FU (10)); // sCapHeight
    cacheU16 (os2, 0); // usDefaultChar
    cacheU16 (os2, 0x20); // usBreakChar
    cacheU16 (os2, 0); // usMaxContext
    cacheU16 (os2, 0); // usLowerOpticalPointSize
    cacheU16 (os2, 0xffff); // usUpperOpticalPointSize
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.29 fillPostTable()

void fillPostTable (
Font ∗ font )

Fill a ”post” font table.

The ”post” table contains information for PostScript printers.

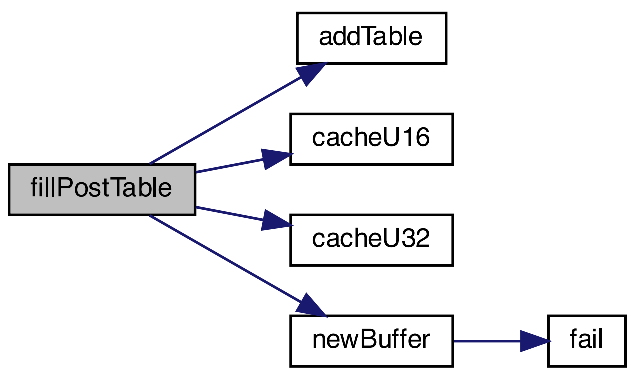
Parameters

| | | |
|---|---|---|
| in,out | font | The Font struct to which to add the table. |

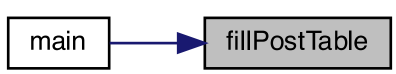
Definition at line 2218 of file hex2otf.c.

```
{
    Buffer *post = newBuffer (32);
    addTable (font, "post", post);
    cacheU32 (post, 0x00030000); // version = 3.0
    cacheU32 (post, 0); // italicAngle
    cacheU16 (post, 0); // underlinePosition
    cacheU16 (post, 1); // underlineThickness
    cacheU32 (post, 1); // isFixedPitch
    cacheU32 (post, 0); // minMemType42
    cacheU32 (post, 0); // maxMemType42
    cacheU32 (post, 0); // minMemType1
    cacheU32 (post, 0); // maxMemType1
}
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.30 fillTrueType()

void fillTrueType (
    Font ∗ font,
    enum LocaFormat ∗ format,
    uint_fast16_t ∗ maxPoints,
    uint_fast16_t ∗ maxContours )

Add a TrueType table to a font.

Parameters

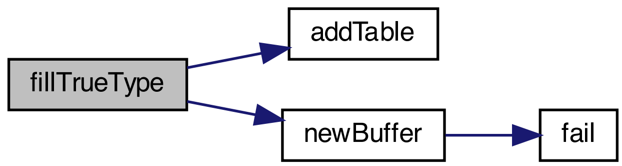
| | | |
|---|---|---|
| in,out | font | Pointer to a Font struct to contain the TrueType table. |
| in | format | The TrueType "loca" table format, Offset16 or Offset32. |
| in | names | List of NameStrings. |

Definition at line 1597 of file hex2otf.c.

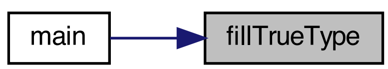
```
{
    Buffer *glyf = newBuffer (65536);
    addTable (font, "glyf", glyf);
    Buffer *loca = newBuffer (4 * (font->glyphCount + 1));
    addTable (font, "loca", loca);
    *format = LOCA_OFFSET32;
    Buffer *endPoints = newBuffer (256);
    Buffer *flags = newBuffer (256);
    Buffer *xs = newBuffer (256);
    Buffer *ys = newBuffer (256);
    Buffer *outline = newBuffer (1024);
    Glyph *const glyphs = getBufferHead (font->glyphs);
    const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
    for (Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
    {
```

```
        cacheU32 (loca, countBufferedBytes (glyf));
        pixels_t rx = -glyph->pos;
        pixels_t ry = DESCENDER;
        pixels_t xMin = GLYPH_MAX_WIDTH, xMax = 0;
        pixels_t yMin = ASCENDER, yMax = -DESCENDER;
        resetBuffer (endPoints);
        resetBuffer (flags);
        resetBuffer (xs);
        resetBuffer (ys);
        resetBuffer (outline);
        buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_RIGHT);
        uint_fast32_t pointCount = 0, contourCount = 0;
        for (const pixels_t *p = getBufferHead (outline),
            *const end = getBufferTail (outline); p < end;)
        {
            const enum ContourOp op = *p++;
            if (op == OP_CLOSE)
            {
                contourCount++;
                assert (contourCount <= U16MAX);
                cacheU16 (endPoints, pointCount - 1);
                continue;
            }
            assert (op == OP_POINT);
            pointCount++;
            assert (pointCount <= U16MAX);
            const pixels_t x = *p++, y = *p++;
            uint_fast8_t pointFlags =
                + B1 (0) // point is on curve
                + BX (1, x != rx) // x coordinate is 1 byte instead of 2
                + BX (2, y != ry) // y coordinate is 1 byte instead of 2
                + B0 (3) // repeat
                + BX (4, x >= rx) // when x is 1 byte: x is positive;
                                  // when x is 2 bytes: x unchanged and omitted
                + BX (5, y >= ry) // when y is 1 byte: y is positive;
                                  // when y is 2 bytes: y unchanged and omitted
                + B1 (6) // contours may overlap
                + B0 (7) // reserved
            ;
            cacheU8 (flags, pointFlags);
            if (x != rx)
                cacheU8 (xs, FU (x > rx ? x - rx : rx - x));
            if (y != ry)
                cacheU8 (ys, FU (y > ry ? y - ry : ry - y));
            if (x < xMin) xMin = x;
            if (y < yMin) yMin = y;
            if (x > xMax) xMax = x;
            if (y > yMax) yMax = y;
            rx = x;
            ry = y;
        }
        if (contourCount == 0)
            continue; // blank glyph is indicated by the 'loca' table
        glyph->lsb = glyph->pos + xMin;
        cacheU16 (glyf, contourCount); // numberOfContours
        cacheU16 (glyf, FU (glyph->pos + xMin)); // xMin
        cacheU16 (glyf, FU (yMin)); // yMin
        cacheU16 (glyf, FU (glyph->pos + xMax)); // xMax
        cacheU16 (glyf, FU (yMax)); // yMax
        cacheBuffer (glyf, endPoints); // endPtsOfContours[]
        cacheU16 (glyf, 0); // instructionLength
        cacheBuffer (glyf, flags); // flags[]
        cacheBuffer (glyf, xs); // xCoordinates[]
        cacheBuffer (glyf, ys); // yCoordinates[]
        if (pointCount > *maxPoints)
            *maxPoints = pointCount;
        if (contourCount > *maxContours)
            *maxContours = contourCount;
    }
    cacheU32 (loca, countBufferedBytes (glyf));
    freeBuffer (endPoints);
    freeBuffer (flags);
    freeBuffer (xs);
    freeBuffer (ys);
    freeBuffer (outline);
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.31 freeBuffer()

void freeBuffer (
Buffer ∗ buf )

Free the memory previously allocated for a buffer.

This function frees the memory allocated to an array of type Buffer ∗.

Parameters

| | | |
|---|---|---|
| in | buf | The pointer to an array of type Buffer ∗. |

Definition at line 337 of file hex2otf.c.

```
00338 {
00339     free (buf->begin);
00340     buf->capacity = 0;
00341 }
```

#### 5.1.5.32 initBuffers()

void initBuffers (
size_t count )

Initialize an array of buffer pointers to all zeroes.

This function initializes the "allBuffers" array of buffer pointers to all zeroes.

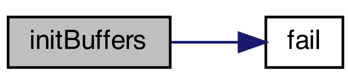
Parameters

| | | |
|---|---|---|
| in | count | The number of buffer array pointers to allocate. |

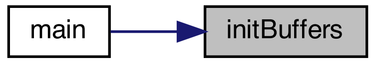
Definition at line 152 of file hex2otf.c.

```
00153 {
00154     assert (count > 0);
00155     assert (bufferCount == 0); // uninitialized
00156     allBuffers = calloc (count, sizeof *allBuffers);
00157     if (!allBuffers)
00158         fail ("Failed to initialize buffers.");
00159     bufferCount = count;
00160     nextBufferIndex = 0;
00161 }
```

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.5.33 main()

int main (
    int argc,
    char * argv[] )

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The number of command-line arguments. |
| in | argv | The array of command-line arguments. |

Returns

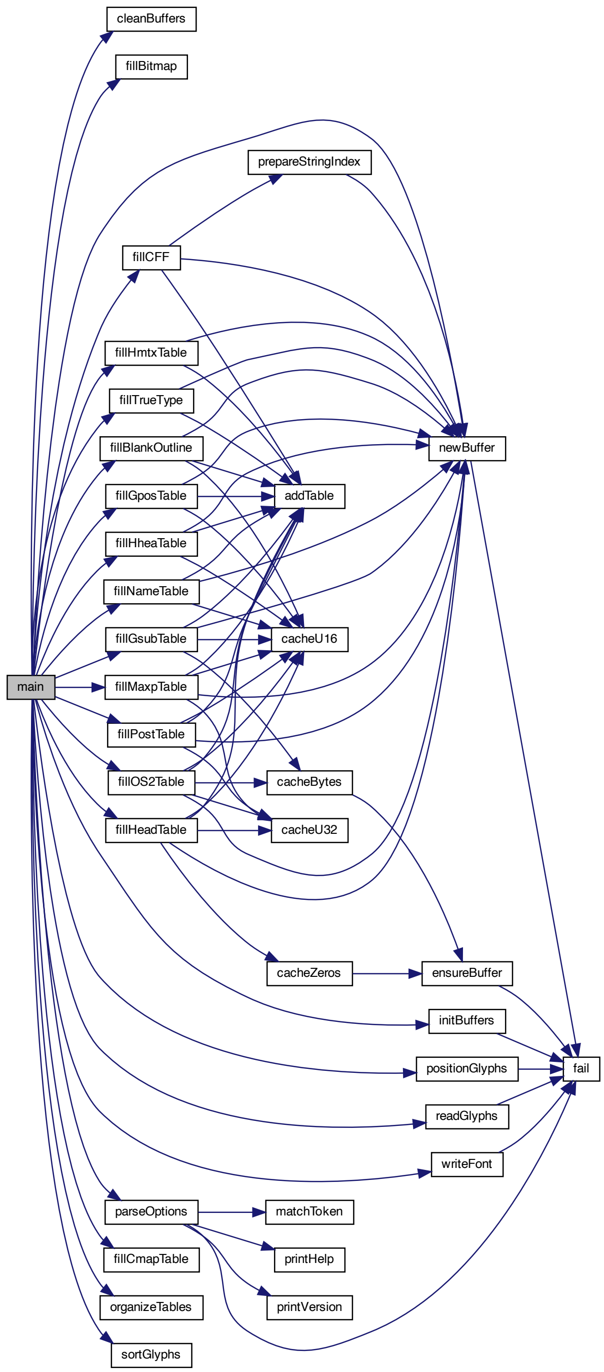
EXIT_FAILURE upon fatal error, EXIT_SUCCESS otherwise.

Definition at line 2603 of file hex2otf.c.

```
02604 {
02605     initBuffers (16);
02606     atexit (cleanBuffers);
02607     Options opt = parseOptions (argv);
02608     Font font;
02609     font.tables = newBuffer (sizeof (Table) * 16);
02610     font.glyphs = newBuffer (sizeof (Glyph) * MAX_GLYPHS);
02611     readGlyphs (&font, opt.hex);
```

```
    sortGlyphs (&font);
    enum LocaFormat loca = LOCA_OFFSET16;
    uint_fast16_t maxPoints = 0, maxContours = 0;
    pixels_t xMin = 0;
    if (opt.pos)
        positionGlyphs (&font, opt.pos, &xMin);
    if (opt.gpos)
        fillGposTable (&font);
    if (opt.gsub)
        fillGsubTable (&font);
    if (opt.cff)
        fillCFF (&font, opt.cff, opt.nameStrings);
    if (opt.truetype)
        fillTrueType (&font, &loca, &maxPoints, &maxContours);
    if (opt.blankOutline)
        fillBlankOutline (&font);
    if (opt.bitmap)
        fillBitmap (&font);
    fillHeadTable (&font, loca, xMin);
    fillHheaTable (&font, xMin);
    fillMaxpTable (&font, opt.cff, maxPoints, maxContours);
    fillOS2Table (&font);
    fillNameTable (&font, opt.nameStrings);
    fillHmtxTable (&font);
    fillCmapTable (&font);
    fillPostTable (&font);
    organizeTables (&font, opt.cff);
    writeFont (&font, opt.cff, opt.out);
    return EXIT_SUCCESS;
}
```

Here is the call graph for this function:



#### 5.1.5.34 matchToken()

const char ∗ matchToken (
    const char ∗ operand,
    const char ∗ key,
    char delimiter )

Match a command line option with its key for enabling.

**Parameters**

| | | |
|---|---|---|
| in | operand | A pointer to the specified operand. |
| in | key | Pointer to the option structure. |
| in | delimeter | The delimiter to end searching. |

**Returns**

Pointer to the first character of the desired option.

Definition at line 2470 of file hex2otf.c.

```
02471 {
02472     while (*key)
02473         if (*operand++ != *key++)
02474             return NULL;
02475     if (!*operand || *operand++ == delimiter)
02476         return operand;
02477     return NULL;
02478 }
```

Here is the caller graph for this function:

#### 5.1.5.35 newBuffer()

Buffer * newBuffer (
    size_t initialCapacity )

Create a new buffer.

This function creates a new buffer array of type Buffer, with an initial size of initialCapacity elements.

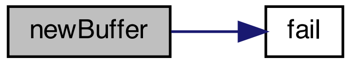
Parameters

| | | |
|---|---|---|
| in | initialCapacity | The initial number of elements in the buffer. |

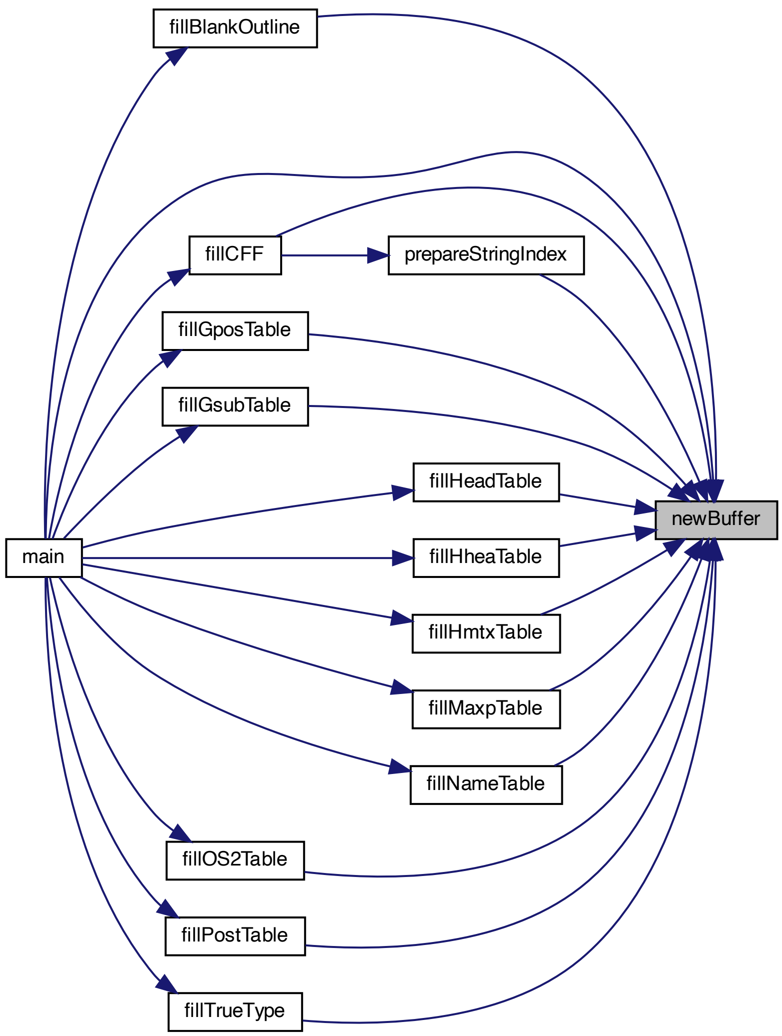
Definition at line 188 of file hex2otf.c.

```
{
    assert (initialCapacity > 0);
    Buffer *buf = NULL;
    size_t sentinel = nextBufferIndex;
    do
    {
        if (nextBufferIndex == bufferCount)
            nextBufferIndex = 0;
        if (allBuffers[nextBufferIndex].capacity == 0)
        {
            buf = &allBuffers[nextBufferIndex++];
            break;
        }
    } while (++nextBufferIndex != sentinel);
    if (!buf) // no existing buffer available
    {
        size_t newSize = sizeof (Buffer) * bufferCount * 2;
        void *extended = realloc (allBuffers, newSize);
        if (!extended)
            fail ("Failed to create new buffers.");
        allBuffers = extended;
        memset (allBuffers + bufferCount, 0, sizeof (Buffer) * bufferCount);
        buf = &allBuffers[bufferCount];
        nextBufferIndex = bufferCount + 1;
        bufferCount *= 2;
    }
    buf->begin = malloc (initialCapacity);
    if (!buf->begin)
        fail ("Failed to allocate %zu bytes of memory.", initialCapacity);
    buf->capacity = initialCapacity;
    buf->next = buf->begin;
    buf->end = buf->begin + initialCapacity;
    return buf;
}
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.36 organizeTables()

void organizeTables (
    Font * font,
    bool isCFF )

Sort tables according to OpenType recommendations.

The various tables in a font are sorted in an order recommended for TrueType font files.

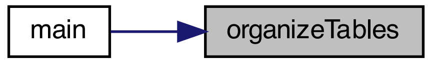
Parameters

| | | |
|---|---|---|
| in,out | font | The font in which to sort tables. |
| in | isCFF | True iff Compact Font Format (CFF) is being used. |

Definition at line 711 of file hex2otf.c.

```
{
    const char *const cffOrder[] = {"head","hhea","maxp","OS/2","name",
        "cmap","post","CFF ",NULL};
    const char *const truetypeOrder[] = {"head","hhea","maxp","OS/2",
        "hmtx","LTSH","VDMX","hdmx","cmap","fpgm","prep","cvt ","loca",
        "glyf","kern","name","post","gasp","PCLT","DSIG",NULL};
    const char *const *const order = isCFF ? cffOrder : truetypeOrder;
    Table *unordered = getBufferHead (font->tables);
    const Table *const tablesEnd = getBufferTail (font->tables);
    for (const char *const *p = order; *p; p++)
    {
        uint_fast32_t tag = tagAsU32 (*p);
        for (Table *t = unordered; t < tablesEnd; t++)
        {
            if (t->tag != tag)
                continue;
            if (t != unordered)
            {
                Table temp = *unordered;
                *unordered = *t;
                *t = temp;
            }
            unordered++;
            break;
        }
    }
}
```

Here is the caller graph for this function:




#### 5.1.5.37 parseOptions()

Options parseOptions (
char ∗const argv[const ] )

Parse command line options.

| Option | Data Type | Description |
|---|---|---|
| truetype | bool | Generate TrueType outlines |
| blankOutline | bool | Generate blank outlines |
| bitmap | bool | Generate embedded bitmap |
| gpos | bool | Generate a dummy GPOS table |
| gsub | bool | Generate a dummy GSUB table |
| cff | int | Generate CFF 1 or CFF 2 outlines |
| hex | const char * | Name of Unifont .hex file |
| pos | const char * | Name of Unifont combining data file |
| out | const char * | Name of output font file |
| nameStrings | NameStrings | Array of TrueType font Name IDs |

Parameters

| | | |
|---|---|---|
| in | argv | Pointer to array of command line options. |

Returns

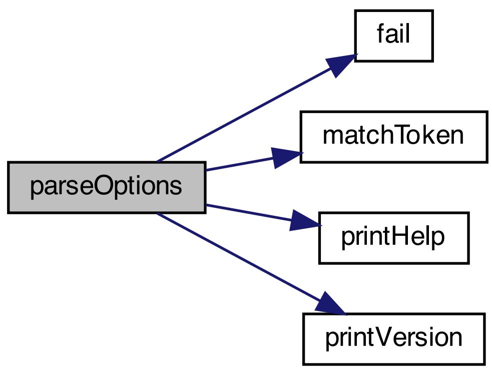
Data structure to hold requested command line options.

Definition at line 2500 of file hex2otf.c.

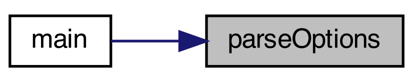
```
{
    Options opt = {0}; // all options default to 0, false and NULL
    const char *format = NULL;
    struct StringArg
    {
        const char *const key;
        const char **const value;
    } strArgs[] =
    {
        {"hex", &opt.hex},
        {"pos", &opt.pos},
        {"out", &opt.out},
        {"format", &format},
        {NULL, NULL} // sentinel
    };
    for (char *const *argp = argv + 1; *argp; argp++)
    {
        const char *const arg = *argp;
        struct StringArg *p;
        const char *value = NULL;
        if (strcmp (arg, "--help") == 0)
            printHelp ();
        if (strcmp (arg, "--version") == 0)
            printVersion ();
        for (p = strArgs; p->key; p++)
            if ((value = matchToken (arg, p->key, '=')))
                break;
        if (p->key)
        {
            if (!*value)
                fail ("Empty argument: '%s'.", p->key);
            if (*p->value)
                fail ("Duplicate argument: '%s'.", p->key);
            *p->value = value;
        }
        else // shall be a name string
        {
            char *endptr;
            unsigned long id = strtoul (arg, &endptr, 10);
            if (endptr == arg || id >= MAX_NAME_IDS || *endptr != '=')
                fail ("Invalid argument: '%s'.", arg);
            endptr++; // skip '='
            if (opt.nameStrings[id])
                fail ("Duplicate name ID: %lu.", id);
            opt.nameStrings[id] = endptr;
        }
    }
    if (!opt.hex)
        fail ("Hex file is not specified.");
    if (opt.pos && opt.pos[0] == '\0')
        opt.pos = NULL; // Position file is optional. Empty path means none.
    if (!opt.out)
        fail ("Output file is not specified.");
    if (!format)
        fail ("Format is not specified.");
    for (const NamePair *p = defaultNames; p->str; p++)
        if (!opt.nameStrings[p->id])
            opt.nameStrings[p->id] = p->str;
    bool cff = false, cff2 = false;
    struct Symbol
    {
        const char *const key;
        bool *const found;
    } symbols[] =
    {
        {"cff", &cff},
        {"cff2", &cff2},
        {"truetype", &opt.truetype},
        {"blank", &opt.blankOutline},
        {"bitmap", &opt.bitmap},
        {"gpos", &opt.gpos},
        {"gsub", &opt.gsub},
        {NULL, NULL} // sentinel
    };
    while (*format)
    {
        const struct Symbol *p;
        const char *next = NULL;
        for (p = symbols; p->key; p++)
            if ((next = matchToken (format, p->key, ',')))
                break;
```

```
        if (!p->key)
            fail ("Invalid format.");
        *p->found = true;
        format = next;
    }
    if (cff + cff2 + opt.truetype + opt.blankOutline > 1)
        fail ("At most one outline format can be accepted.");
    if (!(cff || cff2 || opt.truetype || opt.bitmap))
        fail ("Invalid format.");
    opt.cff = cff + cff2 * 2;
    return opt;
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.38 positionGlyphs()

void positionGlyphs (
    Font * font,

const char * fileName,
pixels_t * xMin )

Position a glyph within a 16-by-16 pixel bounding box.

Position a glyph within the 16-by-16 pixel drawing area and note whether or not the glyph is a combining character.

N.B.: Glyphs must be sorted by code point before calling this function.

Parameters

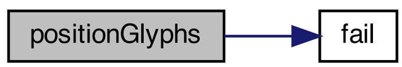
| | | |
|---|---|---|
| in,out | font | Font data structure pointer to store glyphs. |
| in | fileName | Name of glyph file to read. |
| in | xMin | Minimum x-axis value (for left side bearing). |

Definition at line 1061 of file hex2otf.c.

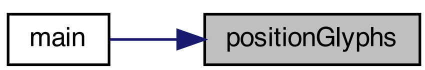
```
01062 {
01063     *xMin = 0;
01064     FILE *file = fopen (fileName, "r");
01065     if (!file)
01066         fail ("Failed to open file '%s'.", fileName);
01067     Glyph *glyphs = getBufferHead (font->glyphs);
01068     const Glyph *const endGlyph = glyphs + font->glyphCount;
01069     Glyph *nextGlyph = &glyphs[1]; // predict and avoid search
01070     for (;;)
01071     {
01072         uint_fast32_t codePoint;
01073         if (readCodePoint (&codePoint, fileName, file))
01074             break;
01075         Glyph *glyph = nextGlyph;
01076         if (glyph == endGlyph || glyph->codePoint != codePoint)
01077         {
01078             // Prediction failed. Search.
01079             const Glyph key = { .codePoint = codePoint };
01080             glyph = bsearch (&key, glyphs + 1, font->glyphCount - 1,
01081                 sizeof key, byCodePoint);
01082             if (!glyph)
01083                 fail ("Glyph "PRI_CP" is positioned but not defined.",
01084                     codePoint);
01085         }
01086         nextGlyph = glyph + 1;
01087         char s[8];
01088         if (!fgets (s, sizeof s, file))
```

```
            fail ("%s: Read error.", fileName);
        char *end;
        const long value = strtol (s, &end, 10);
        if (*end != '\n' && *end != '\0')
            fail ("Position of glyph "PRI_CP" is invalid.", codePoint);
        // Currently no glyph is moved to the right,
        // so positive position is considered out of range.
        // If this limit is to be lifted,
        // 'xMax' of bounding box in 'head' table shall also be updated.
        if (value < -GLYPH_MAX_WIDTH || value > 0)
            fail ("Position of glyph "PRI_CP" is out of range.", codePoint);
        glyph->combining = true;
        glyph->pos = value;
        glyph->lsb = value; // updated during outline generation
        if (value < *xMin)
            *xMin = value;
    }
    fclose (file);
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.39 prepareOffsets()

void prepareOffsets (
size_t ∗ sizes )

Prepare 32-bit glyph offsets in a font table.

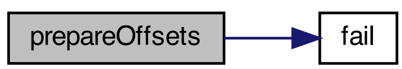
Parameters

| | | |
|---|---|---|
| in | sizes | Array of glyph sizes, for offset calculations. |

Definition at line 1275 of file hex2otf.c.

```
01276 {
01277     size_t *p = sizes;
01278     for (size_t *i = sizes + 1; *i; i++)
01279         *i += *p++;
01280     if (*p > 2147483647U) // offset not representable
01281         fail ("CFF table is too large.");
01282 }
```

Here is the call graph for this function:




### 5.1.5.40 prepareStringIndex()

Buffer ∗ prepareStringIndex (
    const NameStrings names )

Prepare a font name string index.

Parameters

| | | |
|---|---|---|
| in | names | List of name strings. |

Returns

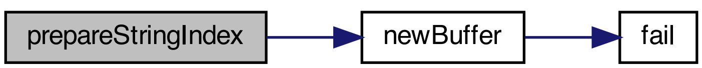
Pointer to a Buffer struct containing the string names.

Get the number of elements in array char ∗strings[].

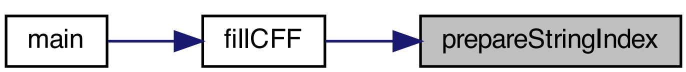
Definition at line 1291 of file hex2otf.c.

```
{
    Buffer *buf = newBuffer (256);
    assert (names[6]);
    const char *strings[] = {"Adobe", "Identity", names[6]};
    /// Get the number of elements in array char *strings[].
    #define stringCount (sizeof strings / sizeof *strings)
    static_assert (stringCount <= U16MAX, "too many strings");
    size_t offset = 1;
    size_t lengths[stringCount];
    for (size_t i = 0; i < stringCount; i++)
    {
        assert (strings[i]);
        lengths[i] = strlen (strings[i]);
        offset += lengths[i];
    }
    int offsetSize = 1 + (offset > 0xff)
                       + (offset > 0xffff)
                       + (offset > 0xffffff);
    cacheU16 (buf, stringCount); // count
    cacheU8 (buf, offsetSize); // offSize
    cacheU (buf, offset = 1, offsetSize); // offset[0]
    for (size_t i = 0; i < stringCount; i++)
        cacheU (buf, offset += lengths[i], offsetSize); // offset[i + 1]
    for (size_t i = 0; i < stringCount; i++)
        cacheBytes (buf, strings[i], lengths[i]);
    #undef stringCount
    return buf;
}
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.41 printHelp()

void printHelp ( )

Print help message to stdout and then exit.

Print help message if invoked with the "--help" option, and then exit successfully.

Definition at line 2426 of file hex2otf.c.

```
02426            {
02427     printf ("Synopsis: hex2otf <options>:\n\n");
02428     printf ("    hex=<filename>        Specify Unifont .hex input file.\n");
02429     printf ("    pos=<filename>        Specify combining file. (Optional)\n");
02430     printf ("    out=<filename>        Specify output font file.\n");
02431     printf ("    format=<f1>,<f2>,...  Specify font format(s); values:\n");
02432     printf ("                            cff\n");
02433     printf ("                            cff2\n");
02434     printf ("                            truetype\n");
02435     printf ("                            blank\n");
02436     printf ("                            bitmap\n");
02437     printf ("                            gpos\n");
02438     printf ("                            gsub\n");
02439     printf ("\nExample:\n\n");
02440     printf ("    hex2otf hex=Myfont.hex out=Myfont.otf format=cff\n\n");
02441     printf ("For more information, consult the hex2otf(1) man page.\n\n");
02442
02443     exit (EXIT_SUCCESS);
02444 }
```

Here is the caller graph for this function:


#### 5.1.5.42 printVersion()

void printVersion ( )

Print program version string on stdout.

Print program version if invoked with the "--version" option, and then exit successfully.

Definition at line 2407 of file hex2otf.c.

```
02407              {
02408     printf ("hex2otf (GNU Unifont) %s\n", VERSION);
02409     printf ("Copyright \u00A9 2022 \u4F55\u5FD7\u7FD4 (He Zhixiang)\n");
02410     printf ("License GPLv2+: GNU GPL version 2 or later\n");
02411     printf ("<https://gnu.org/licenses/gpl.html>\n");
02412     printf ("This is free software: you are free to change and\n");
02413     printf ("redistribute it.  There is NO WARRANTY, to the extent\n");
02414     printf ("permitted by law.\n");
```

```
02415
02416     exit (EXIT_SUCCESS);
02417 }
```

Here is the caller graph for this function:


#### 5.1.5.43 readCodePoint()

```
bool readCodePoint (
        uint_fast32_t * codePoint,
        const char * fileName,
        FILE * file )
```

Read up to 6 hexadecimal digits and a colon from file.

This function reads up to 6 hexadecimal digits followed by a colon from a file.

If the end of the file is reached, the function returns true. The file name is provided to include in an error message if the end of file was reached unexpectedly.

Parameters

| | | |
|---|---|---|
| out | codePoint | The Unicode code point. |
| in | fileName | The name of the input file. |
| in | file | Pointer to the input file stream. |

Returns

true if at end of file, false otherwise.

Definition at line 919 of file hex2otf.c.

```
{
    *codePoint = 0;
    uint_fast8_t digitCount = 0;
    for (;;)
    {
        int c = getc (file);
        if (isxdigit (c) && ++digitCount <= 6)
        {
            *codePoint = (*codePoint « 4) | nibbleValue (c);
            continue;
        }
        if (c == ':' && digitCount > 0)
            return false;
        if (c == EOF)
        {
            if (digitCount == 0)
                return true;
            if (feof (file))
                fail ("%s: Unexpected end of file.", fileName);
            else
                fail ("%s: Read error.", fileName);
        }
        fail ("%s: Unexpected character: %#.2x.", fileName, (unsigned)c);
    }
}
```

#### 5.1.5.44 readGlyphs()

```
void readGlyphs (
        Font * font,
        const char * fileName )
```

Read glyph definitions from a Unifont .hex format file.

This function reads in the glyph bitmaps contained in a Unifont .hex format file. These input files contain one glyph bitmap per line. Each line is of the form

<hexadecimal code point> ':' <hexadecimal bitmap sequence>

The code point field typically consists of 4 hexadecimal digits for a code point in Unicode Plane 0, and 6 hexadecimal digits for code points above Plane 0. The hexadecimal bitmap sequence is 32 hexadecimal digits long for a glyph that is 8 pixels wide by 16 pixels high, and 64 hexadecimal digits long for a glyph that is 16 pixels wide by 16 pixels high.

Parameters

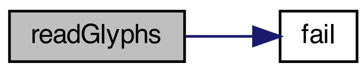
| | | |
|---|---|---|
| in,out | font | The font data structure to update with new glyphs. |
| in | fileName | The name of the Unifont .hex format input file. |

Definition at line 966 of file hex2otf.c.

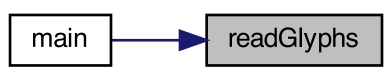
```
{
    FILE *file = fopen (fileName, "r");
    if (!file)
        fail ("Failed to open file '%s'.", fileName);
    uint_fast32_t glyphCount = 1; // for glyph 0
    uint_fast8_t maxByteCount = 0;
    { // Hard code the .notdef glyph.
        const byte bitmap[] = "\0\0\0~fZZzvv~vv~\0\0"; // same as U+FFFD
        const size_t byteCount = sizeof bitmap - 1;
        assert (byteCount <= GLYPH_MAX_BYTE_COUNT);
        assert (byteCount % GLYPH_HEIGHT == 0);
        Glyph *notdef = getBufferSlot (font->glyphs, sizeof (Glyph));
        memcpy (notdef->bitmap, bitmap, byteCount);
        notdef->byteCount = maxByteCount = byteCount;
        notdef->combining = false;
        notdef->pos = 0;
        notdef->lsb = 0;
    }
    for (;;)
    {
        uint_fast32_t codePoint;
        if (readCodePoint (&codePoint, fileName, file))
            break;
        if (++glyphCount > MAX_GLYPHS)
            fail ("OpenType does not support more than %lu glyphs.",
                MAX_GLYPHS);
        Glyph *glyph = getBufferSlot (font->glyphs, sizeof (Glyph));
        glyph->codePoint = codePoint;
        glyph->byteCount = 0;
        glyph->combining = false;
        glyph->pos = 0;
        glyph->lsb = 0;
        for (byte *p = glyph->bitmap;; p++)
        {
            int h, l;
            if (isxdigit (h = getc (file)) && isxdigit (l = getc (file)))
            {
                if (++glyph->byteCount > GLYPH_MAX_BYTE_COUNT)
                    fail ("Hex stream of "PRI_CP" is too long.", codePoint);
                *p = nibbleValue (h) << 4 | nibbleValue (l);
            }
            else if (h == '\n' || (h == EOF && feof (file)))
                break;
            else if (ferror (file))
```

```
                fail ("%s: Read error.", fileName);
            else
                fail ("Hex stream of "PRI_CP" is invalid.", codePoint);
        }
        if (glyph->byteCount % GLYPH_HEIGHT != 0)
            fail ("Hex length of "PRI_CP" is indivisible by glyph height %d.",
                codePoint, GLYPH_HEIGHT);
        if (glyph->byteCount > maxByteCount)
            maxByteCount = glyph->byteCount;
    }
    if (glyphCount == 1)
        fail ("No glyph is specified.");
    font->glyphCount = glyphCount;
    font->maxWidth = PW (maxByteCount);
    fclose (file);
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.45 sortGlyphs()

void sortGlyphs (
    Font ∗ font )

Sort the glyphs in a font by Unicode code point.

This function reads in an array of glyphs and sorts them by Unicode code point. If a duplicate code point is encountered, that will result in a fatal error with an error message to stderr.

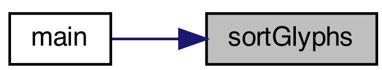
Parameters

| | | |
|---|---|---|
| in,out | font | Pointer to a Font structure with glyphs to sort. |

Definition at line 1119 of file hex2otf.c.

```
{
    Glyph *glyphs = getBufferHead (font->glyphs);
    const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
    glyphs++; // glyph 0 does not need sorting
    qsort (glyphs, glyphsEnd - glyphs, sizeof *glyphs, byCodePoint);
    for (const Glyph *glyph = glyphs; glyph < glyphsEnd - 1; glyph++)
    {
        if (glyph[0].codePoint == glyph[1].codePoint)
            fail ("Duplicate code point: "PRI_CP".", glyph[0].codePoint);
        assert (glyph[0].codePoint < glyph[1].codePoint);
    }
}
```

Here is the caller graph for this function:




### 5.1.5.46 writeBytes()

void writeBytes (
    const byte bytes[ ],
    size_t count,
    FILE * file )

Write an array of bytes to an output file.

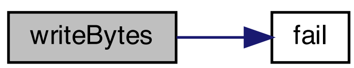
Parameters

| | | |
|---|---|---|
| in | bytes | An array of unsigned bytes to write. |
| in | file | The file pointer for writing, of type FILE *. |

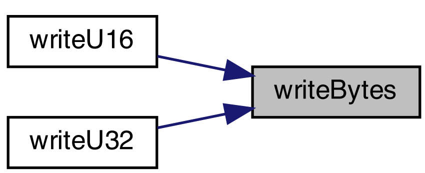
Definition at line 538 of file hex2otf.c.

```
00539 {
00540     if (fwrite (bytes, count, 1, file) != 1 && count != 0)
00541         fail ("Failed to write %zu bytes to output file.", count);
00542 }
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.1.5.47 writeFont()

void writeFont (
    Font ∗ font,
    bool isCFF,
    const char ∗ fileName )

Write OpenType font to output file.

This function writes the constructed OpenType font to the output file named "filename".

Parameters

| | | |
|---|---|---|
| in | font | Pointer to the font, of type Font ∗. |
| in | isCFF | Boolean indicating whether the font has CFF data. |
| in | filename | The name of the font file to create. |

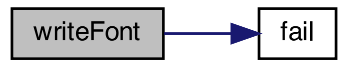
Add a byte shifted by 24, 16, 8, or 0 bits.

Definition at line 786 of file hex2otf.c.

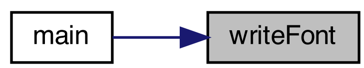
```
{
    FILE *file = fopen (fileName, "wb");
    if (!file)
        fail ("Failed to open file '%s'.", fileName);
    const Table *const tables = getBufferHead (font->tables);
    const Table *const tablesEnd = getBufferTail (font->tables);
    size_t tableCount = tablesEnd - tables;
    assert (0 < tableCount && tableCount <= U16MAX);
    size_t offset = 12 + 16 * tableCount;
    uint_fast32_t totalChecksum = 0;
    Buffer *tableRecords =
        newBuffer (sizeof (struct TableRecord) * tableCount);
    for (size_t i = 0; i < tableCount; i++)
    {
        struct TableRecord *record =
            getBufferSlot (tableRecords, sizeof *record);
        record->tag = tables[i].tag;
        size_t length = countBufferedBytes (tables[i].content);
        #if SIZE_MAX > U32MAX
```

```
            if (offset > U32MAX)
                fail (”Table offset exceeded 4 GiB.”);
            if (length > U32MAX)
                fail (”Table size exceeded 4 GiB.”);
        #endif
        record->length = length;
        record->checksum = 0;
        const byte *p = getBufferHead (tables[i].content);
        const byte *const end = getBufferTail (tables[i].content);

        /// Add a byte shifted by 24, 16, 8, or 0 bits.
        #define addByte(shift) \
            if (p == end) \
                break; \
            record->checksum += (uint_fast32_t)*p++ « (shift);

        for (;;)
        {
            addByte (24)
            addByte (16)
            addByte (8)
            addByte (0)
        }
        #undef addByte
        cacheZeros (tables[i].content, (~length + 1U) & 3U);
        record->offset = offset;
        offset += countBufferedBytes (tables[i].content);
        totalChecksum += record->checksum;
    }
    struct TableRecord *records = getBufferHead (tableRecords);
    qsort (records, tableCount, sizeof *records, byTableTag);
    // Offset Table
    uint_fast32_t sfntVersion = isCFF ? 0x4f54544f : 0x00010000;
    writeU32 (sfntVersion, file); // sfntVersion
    totalChecksum += sfntVersion;
    uint_fast16_t entrySelector = 0;
    for (size_t k = tableCount; k != 1; k »= 1)
        entrySelector++;
    uint_fast16_t searchRange = 1 « (entrySelector + 4);
    uint_fast16_t rangeShift = (tableCount - (1 « entrySelector)) « 4;
    writeU16 (tableCount, file); // numTables
    writeU16 (searchRange, file); // searchRange
    writeU16 (entrySelector, file); // entrySelector
    writeU16 (rangeShift, file); // rangeShift
    totalChecksum += (uint_fast32_t)tableCount « 16;
    totalChecksum += searchRange;
    totalChecksum += (uint_fast32_t)entrySelector « 16;
    totalChecksum += rangeShift;
    // Table Records (always sorted by table tags)
    for (size_t i = 0; i < tableCount; i++)
    {
        // Table Record
        writeU32 (records[i].tag, file); // tableTag
        writeU32 (records[i].checksum, file); // checkSum
        writeU32 (records[i].offset, file); // offset
        writeU32 (records[i].length, file); // length
        totalChecksum += records[i].tag;
        totalChecksum += records[i].checksum;
        totalChecksum += records[i].offset;
        totalChecksum += records[i].length;
    }
    freeBuffer (tableRecords);
    for (const Table *table = tables; table < tablesEnd; table++)
    {
        if (table->tag == 0x68656164) // 'head' table
        {
            byte *begin = getBufferHead (table->content);
            byte *end = getBufferTail (table->content);
            writeBytes (begin, 8, file);
            writeU32 (0xb1b0afbaU - totalChecksum, file); // checksumAdjustment
            writeBytes (begin + 12, end - (begin + 12), file);
            continue;
        }
        writeBuffer (table->content, file);
    }
    fclose (file);
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.1.5.48 writeU16()

```
void writeU16 (
        uint_fast16_t value,
        FILE * file )
```

Write an unsigned 16-bit value to an output file.

This function writes a 16-bit unsigned value in big-endian order to an output file specified with a file pointer.

Parameters

| | | |
|---|---|---|
| in | value | The 16-bit value to write. |
| in | file | The file pointer for writing, of type FILE *. |

Definition at line 554 of file hex2otf.c.

```
{
    byte bytes[] =
    {
        (value » 8) & 0xff,
        (value     ) & 0xff,
    };
    writeBytes (bytes, sizeof bytes, file);
}
```

Here is the call graph for this function:


#### 5.1.5.49 writeU32()

```
void writeU32 (
        uint_fast32_t value,
        FILE * file )
```

Write an unsigned 32-bit value to an output file.

This function writes a 32-bit unsigned value in big-endian order to an output file specified with a file pointer.

Parameters

| | | |
|---|---|---|
| in | value | The 32-bit value to write. |
| in | file | The file pointer for writing, of type FILE *. |

Definition at line 574 of file hex2otf.c.

```
{
```

```
    byte bytes[] =
    {
        (value » 24) & 0xff,
        (value » 16) & 0xff,
        (value »  8) & 0xff,
        (value      ) & 0xff,
    };
    writeBytes (bytes, sizeof bytes, file);
}
```

Here is the call graph for this function:


### 5.1.6 Variable Documentation

#### 5.1.6.1 allBuffers

Buffer∗ allBuffers

Initial allocation of empty array of buffer pointers.

Definition at line 139 of file hex2otf.c.

#### 5.1.6.2 bufferCount

size_t bufferCount

Number of buffers in a Buffer ∗ array.

Definition at line 140 of file hex2otf.c.

#### 5.1.6.3 nextBufferIndex

size_t nextBufferIndex

Index number to tail element of Buffer ∗ array.

Definition at line 141 of file hex2otf.c.

## 5.2 hex2otf.c

Go to the documentation of this file.

```
/**
   @file hex2otf.c

   @brief hex2otf - Convert GNU Unifont .hex file to OpenType font

   This program reads a Unifont .hex format file and a file containing
   combining mark offset information, and produces an OpenType font file.

   @copyright Copyright © 2022 何志翔 (He Zhixiang)

   @author 何志翔 (He Zhixiang)
*/

/*
   LICENSE:

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License
   as published by the Free Software Foundation; either version 2
   of the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
   02110-1301, USA.

   NOTE: It is a violation of the license terms of this software
   to delete or override license and copyright information contained
   in the hex2otf.h file if creating a font derived from Unifont glyphs.
   Fonts derived from Unifont can add names to the copyright notice
   for creators of new or modified glyphs.
*/

#include <assert.h>
#include <ctype.h>
#include <inttypes.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include ”hex2otf.h”

#define VERSION ”1.0.1”  ///< Program version, for ”--version” option.

// This program assumes the execution character set is compatible with ASCII.

#define U16MAX 0xffff       ///< Maximum UTF-16 code point value.
#define U32MAX 0xffffffff   ///< Maximum UTF-32 code point value.

#define PRI_CP ”U+%.4”PRIXFAST32 ///< Format string to print Unicode code point.

#ifndef static_assert
#define static_assert(a, b) (assert(a)) ///< If ”a” is true, return string ”b”.
#endif

// Set or clear a particular bit.
#define BX(shift, x) ((uintmax_t)(!!(x)) « (shift)) ///< Truncate & shift word.
#define B0(shift) BX((shift), 0)    ///< Clear a given bit in a word.
#define B1(shift) BX((shift), 1)    ///< Set   a given bit in a word.

#define GLYPH_MAX_WIDTH 16  ///< Maximum glyph width, in pixels.
#define GLYPH_HEIGHT 16     ///< Maximum glyph height, in pixels.

/// Number of bytes to represent one bitmap glyph as a binary array.
#define GLYPH_MAX_BYTE_COUNT (GLYPH_HEIGHT * GLYPH_MAX_WIDTH / 8)

/// Count of pixels below baseline.
#define DESCENDER 2
```

```c

/// Count of pixels above baseline.
#define ASCENDER (GLYPH_HEIGHT - DESCENDER)

/// Font units per em.
#define FUPEM 64

/// An OpenType font has at most 65536 glyphs.
#define MAX_GLYPHS 65536

/// Name IDs 0-255 are used for standard names.
#define MAX_NAME_IDS 256

/// Convert pixels to font units.
#define FU(x) ((x) * FUPEM / GLYPH_HEIGHT)

/// Convert glyph byte count to pixel width.
#define PW(x) ((x) / (GLYPH_HEIGHT / 8))

/// Definition of "byte" type as an unsigned char.
typedef unsigned char byte;

/// This type must be able to represent max(GLYPH_MAX_WIDTH, GLYPH_HEIGHT).
typedef int_least8_t pixels_t;

/**
    @brief Print an error message on stderr, then exit.

    This function prints the provided error string and optional
    following arguments to stderr, and then exits with a status
    of EXIT_FAILURE.

    @param[in] reason The output string to describe the error.
    @param[in] ... Optional following arguments to output.
*/
void
fail (const char *reason, ...)
{
    fputs ("ERROR: ", stderr);
    va_list args;
    va_start (args, reason);
    vfprintf (stderr, reason, args);
    va_end (args);
    putc ('\n', stderr);
    exit (EXIT_FAILURE);
}

/**
    @brief Generic data structure for a linked list of buffer elements.

    A buffer can act as a vector (when filled with 'store*' functions),
    or a temporary output area (when filled with 'cache*' functions).
    The 'store*' functions use native endian.
    The 'cache*' functions use big endian or other formats in OpenType.
    Beware of memory alignment.
*/
typedef struct Buffer
{
    size_t capacity; // = 0 iff this buffer is free
    byte *begin, *next, *end;
} Buffer;

Buffer *allBuffers; ///< Initial allocation of empty array of buffer pointers.
size_t bufferCount;  ///< Number of buffers in a Buffer * array.
size_t nextBufferIndex; ///< Index number to tail element of Buffer * array.

/**
    @brief Initialize an array of buffer pointers to all zeroes.

    This function initializes the "allBuffers" array of buffer
    pointers to all zeroes.

    @param[in] count The number of buffer array pointers to allocate.
*/
void
initBuffers (size_t count)
{
    assert (count > 0);
    assert (bufferCount == 0); // uninitialized
    allBuffers = calloc (count, sizeof *allBuffers);
    if (!allBuffers)
```

```
        fail ("Failed to initialize buffers.");
    bufferCount = count;
    nextBufferIndex = 0;
}

/**
    @brief Free all allocated buffer pointers.

    This function frees all buffer pointers previously allocated
    in the initBuffers function.
*/
void
cleanBuffers ()
{
    for (size_t i = 0; i < bufferCount; i++)
        if (allBuffers[i].capacity)
            free (allBuffers[i].begin);
    free (allBuffers);
    bufferCount = 0;
}

/**
    @brief Create a new buffer.

    This function creates a new buffer array of type Buffer,
    with an initial size of initialCapacity elements.

    @param[in] initialCapacity The initial number of elements in the buffer.
*/
Buffer *
newBuffer (size_t initialCapacity)
{
    assert (initialCapacity > 0);
    Buffer *buf = NULL;
    size_t sentinel = nextBufferIndex;
    do
    {
        if (nextBufferIndex == bufferCount)
            nextBufferIndex = 0;
        if (allBuffers[nextBufferIndex].capacity == 0)
        {
            buf = &allBuffers[nextBufferIndex++];
            break;
        }
    } while (++nextBufferIndex != sentinel);
    if (!buf) // no existing buffer available
    {
        size_t newSize = sizeof (Buffer) * bufferCount * 2;
        void *extended = realloc (allBuffers, newSize);
        if (!extended)
            fail ("Failed to create new buffers.");
        allBuffers = extended;
        memset (allBuffers + bufferCount, 0, sizeof (Buffer) * bufferCount);
        buf = &allBuffers[bufferCount];
        nextBufferIndex = bufferCount + 1;
        bufferCount *= 2;
    }
    buf->begin = malloc (initialCapacity);
    if (!buf->begin)
        fail ("Failed to allocate %zu bytes of memory.", initialCapacity);
    buf->capacity = initialCapacity;
    buf->next = buf->begin;
    buf->end = buf->begin + initialCapacity;
    return buf;
}

/**
    @brief Ensure that the buffer has at least the specified minimum size.

    This function takes a buffer array of type Buffer and the
    necessary minimum number of elements as inputs, and attempts
    to increase the size of the buffer if it must be larger.

    If the buffer is too small and cannot be resized, the program
    will terminate with an error message and an exit status of
    EXIT_FAILURE.

    @param[in,out] buf The buffer to check.
    @param[in] needed The required minimum number of elements in the buffer.
*/
void
```

```
ensureBuffer (Buffer *buf, size_t needed)
{
    if (buf->end - buf->next >= needed)
        return;
    ptrdiff_t occupied = buf->next - buf->begin;
    size_t required = occupied + needed;
    if (required < needed) // overflow
        fail ("Cannot allocate %zu + %zu bytes of memory.", occupied, needed);
    if (required > SIZE_MAX / 2)
        buf->capacity = required;
    else while (buf->capacity < required)
        buf->capacity *= 2;
    void *extended = realloc (buf->begin, buf->capacity);
    if (!extended)
        fail ("Failed to allocate %zu bytes of memory.", buf->capacity);
    buf->begin = extended;
    buf->next = buf->begin + occupied;
    buf->end = buf->begin + buf->capacity;
}

/**
    @brief Count the number of elements in a buffer.

    @param[in] buf The buffer to be examined.
    @return The number of elements in the buffer.
*/
static inline size_t
countBufferedBytes (const Buffer *buf)
{
    return buf->next - buf->begin;
}

/**
    @brief Get the start of the buffer array.

    @param[in] buf The buffer to be examined.
    @return A pointer of type Buffer * to the start of the buffer.
*/
static inline void *
getBufferHead (const Buffer *buf)
{
    return buf->begin;
}

/**
    @brief Get the end of the buffer array.

    @param[in] buf The buffer to be examined.
    @return A pointer of type Buffer * to the end of the buffer.
*/
static inline void *
getBufferTail (const Buffer *buf)
{
    return buf->next;
}

/**
    @brief Add a slot to the end of a buffer.

    This function ensures that the buffer can grow by one slot,
    and then returns a pointer to the new slot within the buffer.

    @param[in] buf The pointer to an array of type Buffer *.
    @param[in] slotSize The new slot number.
    @return A pointer to the new slot within the buffer.
*/
static inline void *
getBufferSlot (Buffer *buf, size_t slotSize)
{
    ensureBuffer (buf, slotSize);
    void *slot = buf->next;
    buf->next += slotSize;
    return slot;
}

/**
    @brief Reset a buffer pointer to the buffer's beginning.

    This function resets an array of type Buffer * to point
    its tail to the start of the array.

```

```
    @param[in] buf The pointer to an array of type Buffer *.
*/
static inline void
resetBuffer (Buffer *buf)
{
    buf->next = buf->begin;
}

/**
    @brief Free the memory previously allocated for a buffer.

    This function frees the memory allocated to an array
    of type Buffer *.

    @param[in] buf The pointer to an array of type Buffer *.
*/
void
freeBuffer (Buffer *buf)
{
    free (buf->begin);
    buf->capacity = 0;
}

/**
    @brief Temporary define to look up an element in an array of given type.

    This defintion is used to create lookup functions to return
    a given element in unsigned arrays of size 8, 16, and 32 bytes,
    and in an array of pixels.
*/
#define defineStore(name, type) \
void name (Buffer *buf, type value) \
{ \
    type *slot = getBufferSlot (buf, sizeof value); \
    *slot = value; \
}
defineStore (storeU8, uint_least8_t)
defineStore (storeU16, uint_least16_t)
defineStore (storeU32, uint_least32_t)
defineStore (storePixels, pixels_t)
#undef defineStore

/**
    @brief Cache bytes in a big-endian format.

    This function adds from 1, 2, 3, or 4 bytes to the end of
    a byte array in big-endian order.  The buffer is updated
    to account for the newly-added bytes.

    @param[in,out] buf The array of bytes to which to append new bytes.
    @param[in] value The bytes to add, passed as a 32-bit unsigned word.
    @param[in] bytes The number of bytes to append to the buffer.
*/
void
cacheU (Buffer *buf, uint_fast32_t value, int bytes)
{
    assert (1 <= bytes && bytes <= 4);
    ensureBuffer (buf, bytes);
    switch (bytes)
    {
        case 4: *buf->next++ = value » 24 & 0xff; // fall through
        case 3: *buf->next++ = value » 16 & 0xff; // fall through
        case 2: *buf->next++ = value » 8  & 0xff; // fall through
        case 1: *buf->next++ = value       & 0xff;
    }
}

/**
    @brief Append one unsigned byte to the end of a byte array.

    This function adds one byte to the end of a byte array.
    The buffer is updated to account for the newly-added byte.

    @param[in,out] buf The array of bytes to which to append a new byte.
    @param[in] value The 8-bit unsigned value to append to the buf array.
*/
void
cacheU8 (Buffer *buf, uint_fast8_t value)
{
    storeU8 (buf, value & 0xff);
}
```

```
/**
   @brief Append two unsigned bytes to the end of a byte array.

   This function adds two bytes to the end of a byte array.
   The buffer is updated to account for the newly-added bytes.

   @param[in,out] buf The array of bytes to which to append two new bytes.
   @param[in] value The 16-bit unsigned value to append to the buf array.
*/
void
cacheU16 (Buffer *buf, uint_fast16_t value)
{
   cacheU (buf, value, 2);
}

/**
   @brief Append four unsigned bytes to the end of a byte array.

   This function adds four bytes to the end of a byte array.
   The buffer is updated to account for the newly-added bytes.

   @param[in,out] buf The array of bytes to which to append four new bytes.
   @param[in] value The 32-bit unsigned value to append to the buf array.
*/
void
cacheU32 (Buffer *buf, uint_fast32_t value)
{
   cacheU (buf, value, 4);
}

/**
   @brief Cache charstring number encoding in a CFF buffer.

   This function caches two's complement 8-, 16-, and 32-bit
   words as per Adobe's Type 2 Charstring encoding for operands.
   These operands are used in Compact Font Format data structures.

   Byte values can have offsets, for which this function
   compensates, optionally followed by additional bytes:

      Byte Range   Offset   Bytes   Adjusted Range
      ----------   ------   -----   --------------
        0 to 11       0       1      0 to 11 (operators)
          12          0       2      Next byte is 8-bit op code
       13 to 18       0       1      13 to 18 (operators)
       19 to 20       0       2+     hintmask and cntrmask operators
       21 to 27       0       1      21 to 27 (operators)
          28          0       3      16-bit 2's complement number
       29 to 31       0       1      29 to 31 (operators)
       32 to 246   -139       1      -107 to +107
      247 to 250   +108       2      +108 to +1131
      251 to 254   -108       2      -108 to -1131
         255          0       5      16-bit integer and 16-bit fraction

   @param[in,out] buf The buffer to which the operand value is appended.
   @param[in] value The operand value.
*/
void
cacheCFFOperand (Buffer *buf, int_fast32_t value)
{
   if (-107 <= value && value <= 107)
      cacheU8 (buf, value + 139);
   else if (108 <= value && value <= 1131)
   {
      cacheU8 (buf, (value - 108) / 256 + 247);
      cacheU8 (buf, (value - 108) % 256);
   }
   else if (-32768 <= value && value <= 32767)
   {
      cacheU8 (buf, 28);
      cacheU16 (buf, value);
   }
   else if (-2147483647 <= value && value <= 2147483647)
   {
      cacheU8 (buf, 29);
      cacheU32 (buf, value);
   }
   else
      assert (false); // other encodings are not used and omitted
   static_assert (GLYPH_MAX_WIDTH <= 107, "More encodings are needed.");
```

```
}

/**
   @brief Append 1 to 4 bytes of zeroes to a buffer, for padding.

   @param[in,out] buf The buffer to which the operand value is appended.
   @param[in] count The number of bytes containing zeroes to append.
*/
void
cacheZeros (Buffer *buf, size_t count)
{
   ensureBuffer (buf, count);
   memset (buf->next, 0, count);
   buf->next += count;
}

/**
   @brief Append a string of bytes to a buffer.

   This function appends an array of 1 to 4 bytes to the end of
   a buffer.

   @param[in,out] buf The buffer to which the bytes are appended.
   @param[in] src The array of bytes to append to the buffer.
   @param[in] count The number of bytes containing zeroes to append.
*/
void
cacheBytes (Buffer *restrict buf, const void *restrict src, size_t count)
{
   ensureBuffer (buf, count);
   memcpy (buf->next, src, count);
   buf->next += count;
}

/**
   @brief Append bytes of a table to a byte buffer.

   @param[in,out] bufDest The buffer to which the new bytes are appended.
   @param[in] bufSrc The bytes to append to the buffer array.
*/
void
cacheBuffer (Buffer *restrict bufDest, const Buffer *restrict bufSrc)
{
   size_t length = countBufferedBytes (bufSrc);
   ensureBuffer (bufDest, length);
   memcpy (bufDest->next, bufSrc->begin, length);
   bufDest->next += length;
}

/**
   @brief Write an array of bytes to an output file.

   @param[in] bytes An array of unsigned bytes to write.
   @param[in] file The file pointer for writing, of type FILE *.
*/
void
writeBytes (const byte bytes[], size_t count, FILE *file)
{
   if (fwrite (bytes, count, 1, file) != 1 && count != 0)
      fail ("Failed to write %zu bytes to output file.", count);
}

/**
   @brief Write an unsigned 16-bit value to an output file.

   This function writes a 16-bit unsigned value in big-endian order
   to an output file specified with a file pointer.

   @param[in] value The 16-bit value to write.
   @param[in] file The file pointer for writing, of type FILE *.
*/
void
writeU16 (uint_fast16_t value, FILE *file)
{
   byte bytes[] =
   {
      (value » 8) & 0xff,
      (value     ) & 0xff,
   };
   writeBytes (bytes, sizeof bytes, file);
}
```

```c

/**
    @brief Write an unsigned 32-bit value to an output file.

    This function writes a 32-bit unsigned value in big-endian order
    to an output file specified with a file pointer.

    @param[in] value The 32-bit value to write.
    @param[in] file The file pointer for writing, of type FILE *.
*/
void
writeU32 (uint_fast32_t value, FILE *file)
{
    byte bytes[] =
    {
        (value >> 24) & 0xff,
        (value >> 16) & 0xff,
        (value >>  8) & 0xff,
        (value      ) & 0xff,
    };
    writeBytes (bytes, sizeof bytes, file);
}

/**
    @brief Write an entire buffer array of bytes to an output file.

    This function determines the size of a buffer of bytes and
    writes that number of bytes to an output file specified with
    a file pointer.  The number of bytes is determined from the
    length information stored as part of the Buffer * data structure.

    @param[in] buf An array containing unsigned bytes to write.
    @param[in] file The file pointer for writing, of type FILE *.
*/
static inline void
writeBuffer (const Buffer *buf, FILE *file)
{
    writeBytes (getBufferHead (buf), countBufferedBytes (buf), file);
}

/// Array of OpenType names indexed directly by Name IDs.
typedef const char *NameStrings[MAX_NAME_IDS];

/**
   @brief Data structure to hold data for one bitmap glyph.

   This data structure holds data to represent one Unifont bitmap
   glyph: Unicode code point, number of bytes in its bitmap array,
   whether or not it is a combining character, and an offset from
   the glyph origin to the start of the bitmap.
*/
typedef struct Glyph
{
    uint_least32_t codePoint; ///< undefined for glyph 0
    byte bitmap[GLYPH_MAX_BYTE_COUNT]; ///< hexadecimal bitmap character array
    uint_least8_t byteCount; ///< length of bitmap data
    bool combining; ///< whether this is a combining glyph
    pixels_t pos; ///< number of pixels the glyph should be moved to the right
                  ///< (negative number means moving to the left)
    pixels_t lsb; ///< left side bearing (x position of leftmost contour point)
} Glyph;

/**
   @brief Data structure to hold information for one font.
*/
typedef struct Font
{
    Buffer *tables;
    Buffer *glyphs;
    uint_fast32_t glyphCount;
    pixels_t maxWidth;
} Font;

/**
   @brief Data structure for an OpenType table.

   This data structure contains a table tag and a pointer to the
   start of the buffer that holds data for this OpenType table.

   For information on the OpenType tables and their structure, see
   https://docs.microsoft.com/en-us/typography/opentype/spec/otff#font-tables.
```

```
*/
typedef struct Table
{
    uint_fast32_t tag;
    Buffer *content;
} Table;

/**
    @brief Index to Location (”loca”) offset information.

    This enumerated type encodes the type of offset to locations
    in a table.  It denotes Offset16 (16-bit) and Offset32 (32-bit)
    offset types.
*/
enum LocaFormat {
    LOCA_OFFSET16 = 0,    ///< Offset to location is a 16-bit Offset16 value
    LOCA_OFFSET32 = 1     ///< Offset to location is a 32-bit Offset32 value
};

/**
    @brief Convert a 4-byte array to the machine’s native 32-bit endian order.

    This function takes an array of 4 bytes in big-endian order and
    converts it to a 32-bit word in the endian order of the native machine.

    @param[in] tag The array of 4 bytes in big-endian order.
    @return The 32-bit unsigned word in a machine’s native endian order.
*/
static inline uint_fast32_t tagAsU32 (const char tag[static 4])
{
    uint_fast32_t r = 0;
    r |= (tag[0] & 0xff) « 24;
    r |= (tag[1] & 0xff) « 16;
    r |= (tag[2] & 0xff) « 8;
    r |= (tag[3] & 0xff);
    return r;
}

/**
    @brief Add a TrueType or OpenType table to the font.

    This function adds a TrueType or OpenType table to a font.
    The 4-byte table tag is passed as an unsigned 32-bit integer
    in big-endian format.

    @param[in,out] font The font to which a font table will be added.
    @param[in] tag The 4-byte table name.
    @param[in] content The table bytes to add, of type Buffer *.
*/
void
addTable (Font *font, const char tag[static 4], Buffer *content)
{
    Table *table = getBufferSlot (font->tables, sizeof (Table));
    table->tag = tagAsU32 (tag);
    table->content = content;
}

/**
    @brief  Sort tables according to OpenType recommendations.

    The various tables in a font are sorted in an order recommended
    for TrueType font files.

    @param[in,out] font The font in which to sort tables.
    @param[in] isCFF True iff Compact Font Format (CFF) is being used.
*/
void
organizeTables (Font *font, bool isCFF)
{
    const char *const cffOrder[] = {”head”,”hhea”,”maxp”,”OS/2”,”name”,
        ”cmap”,”post”,”CFF ”,NULL};
    const char *const truetypeOrder[] = {”head”,”hhea”,”maxp”,”OS/2”,
        ”hmtx”,”LTSH”,”VDMX”,”hdmx”,”cmap”,”fpgm”,”prep”,”cvt ”,”loca”,
        ”glyf”,”kern”,”name”,”post”,”gasp”,”PCLT”,”DSIG”,NULL};
    const char *const *const order = isCFF ? cffOrder : truetypeOrder;
    Table *unordered = getBufferHead (font->tables);
    const Table *const tablesEnd = getBufferTail (font->tables);
    for (const char *const *p = order; *p; p++)
    {
        uint_fast32_t tag = tagAsU32 (*p);
        for (Table *t = unordered; t < tablesEnd; t++)
```

```
        {
            if (t->tag != tag)
                continue;
            if (t != unordered)
            {
                Table temp = *unordered;
                *unordered = *t;
                *t = temp;
            }
            unordered++;
            break;
        }
    }
}

/**
   @brief Data structure for data associated with one OpenType table.

   This data structure contains an OpenType table's tag, start within
   an OpenType font file, length in bytes, and checksum at the end of
   the table.
*/
struct TableRecord
{
    uint_least32_t tag, offset, length, checksum;
};

/**
    @brief Compare tables by 4-byte unsigned table tag value.

    This function takes two pointers to a TableRecord data structure
    and extracts the four-byte tag structure element for each.  The
    two 32-bit numbers are then compared.  If the first tag is greater
    than the first, then gt = 1 and lt = 0, and so 1 - 0 = 1 is
    returned.  If the first is less than the second, then gt = 0 and
    lt = 1, and so 0 - 1 = -1 is returned.

    @param[in] a Pointer to the first TableRecord structure.
    @param[in] b Pointer to the second TableRecord structure.
    @return 1 if the tag in "a" is greater, -1 if less, 0 if equal.
*/
int
byTableTag (const void *a, const void *b)
{
    const struct TableRecord *const ra = a, *const rb = b;
    int gt = ra->tag > rb->tag;
    int lt = ra->tag < rb->tag;
    return gt - lt;
}

/**
   @brief Write OpenType font to output file.

   This function writes the constructed OpenType font to the
   output file named "filename".

   @param[in] font Pointer to the font, of type Font *.
   @param[in] isCFF Boolean indicating whether the font has CFF data.
   @param[in] filename The name of the font file to create.
*/
void
writeFont (Font *font, bool isCFF, const char *fileName)
{
    FILE *file = fopen (fileName, "wb");
    if (!file)
        fail ("Failed to open file '%s'.", fileName);
    const Table *const tables = getBufferHead (font->tables);
    const Table *const tablesEnd = getBufferTail (font->tables);
    size_t tableCount = tablesEnd - tables;
    assert (0 < tableCount && tableCount <= U16MAX);
    size_t offset = 12 + 16 * tableCount;
    uint_fast32_t totalChecksum = 0;
    Buffer *tableRecords =
        newBuffer (sizeof (struct TableRecord) * tableCount);
    for (size_t i = 0; i < tableCount; i++)
    {
        struct TableRecord *record =
            getBufferSlot (tableRecords, sizeof *record);
        record->tag = tables[i].tag;
        size_t length = countBufferedBytes (tables[i].content);
        #if SIZE_MAX > U32MAX
```

```
            if (offset > U32MAX)
                fail ("Table offset exceeded 4 GiB.");
            if (length > U32MAX)
                fail ("Table size exceeded 4 GiB.");
        #endif
        record->length = length;
        record->checksum = 0;
        const byte *p = getBufferHead (tables[i].content);
        const byte *const end = getBufferTail (tables[i].content);

        /// Add a byte shifted by 24, 16, 8, or 0 bits.
        #define addByte(shift) \
            if (p == end) \
                break; \
            record->checksum += (uint_fast32_t)*p++ « (shift);

        for (;;)
        {
            addByte (24)
            addByte (16)
            addByte (8)
            addByte (0)
        }
        #undef addByte
        cacheZeros (tables[i].content, (~length + 1U) & 3U);
        record->offset = offset;
        offset += countBufferedBytes (tables[i].content);
        totalChecksum += record->checksum;
    }
    struct TableRecord *records = getBufferHead (tableRecords);
    qsort (records, tableCount, sizeof *records, byTableTag);
    // Offset Table
    uint_fast32_t sfntVersion = isCFF ? 0x4f54544f : 0x00010000;
    writeU32 (sfntVersion, file); // sfntVersion
    totalChecksum += sfntVersion;
    uint_fast16_t entrySelector = 0;
    for (size_t k = tableCount; k != 1; k »= 1)
        entrySelector++;
    uint_fast16_t searchRange = 1 « (entrySelector + 4);
    uint_fast16_t rangeShift = (tableCount - (1 « entrySelector)) « 4;
    writeU16 (tableCount, file); // numTables
    writeU16 (searchRange, file); // searchRange
    writeU16 (entrySelector, file); // entrySelector
    writeU16 (rangeShift, file); // rangeShift
    totalChecksum += (uint_fast32_t)tableCount « 16;
    totalChecksum += searchRange;
    totalChecksum += (uint_fast32_t)entrySelector « 16;
    totalChecksum += rangeShift;
    // Table Records (always sorted by table tags)
    for (size_t i = 0; i < tableCount; i++)
    {
        // Table Record
        writeU32 (records[i].tag, file); // tableTag
        writeU32 (records[i].checksum, file); // checkSum
        writeU32 (records[i].offset, file); // offset
        writeU32 (records[i].length, file); // length
        totalChecksum += records[i].tag;
        totalChecksum += records[i].checksum;
        totalChecksum += records[i].offset;
        totalChecksum += records[i].length;
    }
    freeBuffer (tableRecords);
    for (const Table *table = tables; table < tablesEnd; table++)
    {
        if (table->tag == 0x68656164) // 'head' table
        {
            byte *begin = getBufferHead (table->content);
            byte *end = getBufferTail (table->content);
            writeBytes (begin, 8, file);
            writeU32 (0xb1b0afbaU - totalChecksum, file); // checksumAdjustment
            writeBytes (begin + 12, end - (begin + 12), file);
            continue;
        }
        writeBuffer (table->content, file);
    }
    fclose (file);
}

/**
    @brief Convert a hexadecimal digit character to a 4-bit number.

```

```
    This function takes a character that contains one hexadecimal digit
    and returns the 4-bit value (as an unsigned 8-bit value) corresponding
    to the hexadecimal digit.

    @param[in] nibble The character containing one hexadecimal digit.
    @return The hexadecimal digit value, 0 through 15, inclusive.
*/
static inline byte
nibbleValue (char nibble)
{
    if (isdigit (nibble))
        return nibble - '0';
    nibble = toupper (nibble);
    return nibble - 'A' + 10;
}

/**
    @brief Read up to 6 hexadecimal digits and a colon from file.

    This function reads up to 6 hexadecimal digits followed by
    a colon from a file.

    If the end of the file is reached, the function returns true.
    The file name is provided to include in an error message if
    the end of file was reached unexpectedly.

    @param[out] codePoint The Unicode code point.
    @param[in] fileName The name of the input file.
    @param[in] file Pointer to the input file stream.
    @return true if at end of file, false otherwise.
*/
bool
readCodePoint (uint_fast32_t *codePoint, const char *fileName, FILE *file)
{
    *codePoint = 0;
    uint_fast8_t digitCount = 0;
    for (;;)
    {
        int c = getc (file);
        if (isxdigit (c) && ++digitCount <= 6)
        {
            *codePoint = (*codePoint << 4) | nibbleValue (c);
            continue;
        }
        if (c == ':' && digitCount > 0)
            return false;
        if (c == EOF)
        {
            if (digitCount == 0)
                return true;
            if (feof (file))
                fail ("%s: Unexpected end of file.", fileName);
            else
                fail ("%s: Read error.", fileName);
        }
        fail ("%s: Unexpected character: %#.2x.", fileName, (unsigned)c);
    }
}

/**
    @brief Read glyph definitions from a Unifont .hex format file.

    This function reads in the glyph bitmaps contained in a Unifont
    .hex format file.  These input files contain one glyph bitmap
    per line.  Each line is of the form

        <hexadecimal code point> ':' <hexadecimal bitmap sequence>

    The code point field typically consists of 4 hexadecimal digits
    for a code point in Unicode Plane 0, and 6 hexadecimal digits for
    code points above Plane 0.  The hexadecimal bitmap sequence is
    32 hexadecimal digits long for a glyph that is 8 pixels wide by
    16 pixels high, and 64 hexadecimal digits long for a glyph that
    is 16 pixels wide by 16 pixels high.

    @param[in,out] font The font data structure to update with new glyphs.
    @param[in] fileName The name of the Unifont .hex format input file.
*/
void
readGlyphs (Font *font, const char *fileName)
{
```

```c
    FILE *file = fopen (fileName, "r");
    if (!file)
        fail ("Failed to open file '%s'.", fileName);
    uint_fast32_t glyphCount = 1; // for glyph 0
    uint_fast8_t maxByteCount = 0;
    { // Hard code the .notdef glyph.
        const byte bitmap[] = "\0\0\0~fZZzvv~vv~\0\0"; // same as U+FFFD
        const size_t byteCount = sizeof bitmap - 1;
        assert (byteCount <= GLYPH_MAX_BYTE_COUNT);
        assert (byteCount % GLYPH_HEIGHT == 0);
        Glyph *notdef = getBufferSlot (font->glyphs, sizeof (Glyph));
        memcpy (notdef->bitmap, bitmap, byteCount);
        notdef->byteCount = maxByteCount = byteCount;
        notdef->combining = false;
        notdef->pos = 0;
        notdef->lsb = 0;
    }
    for (;;)
    {
        uint_fast32_t codePoint;
        if (readCodePoint (&codePoint, fileName, file))
            break;
        if (++glyphCount > MAX_GLYPHS)
            fail ("OpenType does not support more than %lu glyphs.",
                MAX_GLYPHS);
        Glyph *glyph = getBufferSlot (font->glyphs, sizeof (Glyph));
        glyph->codePoint = codePoint;
        glyph->byteCount = 0;
        glyph->combining = false;
        glyph->pos = 0;
        glyph->lsb = 0;
        for (byte *p = glyph->bitmap;; p++)
        {
            int h, l;
            if (isxdigit (h = getc (file)) && isxdigit (l = getc (file)))
            {
                if (++glyph->byteCount > GLYPH_MAX_BYTE_COUNT)
                    fail ("Hex stream of "PRI_CP" is too long.", codePoint);
                *p = nibbleValue (h) « 4 | nibbleValue (l);
            }
            else if (h == '\n' || (h == EOF && feof (file)))
                break;
            else if (ferror (file))
                fail ("%s: Read error.", fileName);
            else
                fail ("Hex stream of "PRI_CP" is invalid.", codePoint);
        }
        if (glyph->byteCount % GLYPH_HEIGHT != 0)
            fail ("Hex length of "PRI_CP" is indivisible by glyph height %d.",
                codePoint, GLYPH_HEIGHT);
        if (glyph->byteCount > maxByteCount)
            maxByteCount = glyph->byteCount;
    }
    if (glyphCount == 1)
        fail ("No glyph is specified.");
    font->glyphCount = glyphCount;
    font->maxWidth = PW (maxByteCount);
    fclose (file);
}

/**
    @brief Compare two Unicode code points to determine which is greater.

    This function compares the Unicode code points contained within
    two Glyph data structures.  The function returns 1 if the first
    code point is greater, and -1 if the second is greater.

    @param[in] a A Glyph data structure containing the first code point.
    @param[in] b A Glyph data structure containing the second code point.
    @return 1 if the code point a is greater, -1 if less, 0 if equal.
*/
int
byCodePoint (const void *a, const void *b)
{
    const Glyph *const ga = a, *const gb = b;
    int gt = ga->codePoint > gb->codePoint;
    int lt = ga->codePoint < gb->codePoint;
    return gt - lt;
}

/**
```

```
    @brief Position a glyph within a 16-by-16 pixel bounding box.

    Position a glyph within the 16-by-16 pixel drawing area and
    note whether or not the glyph is a combining character.

    N.B.: Glyphs must be sorted by code point before calling this function.

    @param[in,out] font Font data structure pointer to store glyphs.
    @param[in] fileName Name of glyph file to read.
    @param[in] xMin Minimum x-axis value (for left side bearing).
*/
void
positionGlyphs (Font *font, const char *fileName, pixels_t *xMin)
{
    *xMin = 0;
    FILE *file = fopen (fileName, "r");
    if (!file)
        fail ("Failed to open file '%s'.", fileName);
    Glyph *glyphs = getBufferHead (font->glyphs);
    const Glyph *const endGlyph = glyphs + font->glyphCount;
    Glyph *nextGlyph = &glyphs[1]; // predict and avoid search
    for (;;)
    {
        uint_fast32_t codePoint;
        if (readCodePoint (&codePoint, fileName, file))
            break;
        Glyph *glyph = nextGlyph;
        if (glyph == endGlyph || glyph->codePoint != codePoint)
        {
            // Prediction failed. Search.
            const Glyph key = { .codePoint = codePoint };
            glyph = bsearch (&key, glyphs + 1, font->glyphCount - 1,
                sizeof key, byCodePoint);
            if (!glyph)
                fail ("Glyph "PRI_CP" is positioned but not defined.",
                    codePoint);
        }
        nextGlyph = glyph + 1;
        char s[8];
        if (!fgets (s, sizeof s, file))
            fail ("%s: Read error.", fileName);
        char *end;
        const long value = strtol (s, &end, 10);
        if (*end != '\n' && *end != '\0')
            fail ("Position of glyph "PRI_CP" is invalid.", codePoint);
        // Currently no glyph is moved to the right,
        // so positive position is considered out of range.
        // If this limit is to be lifted,
        // 'xMax' of bounding box in 'head' table shall also be updated.
        if (value < -GLYPH_MAX_WIDTH || value > 0)
            fail ("Position of glyph "PRI_CP" is out of range.", codePoint);
        glyph->combining = true;
        glyph->pos = value;
        glyph->lsb = value; // updated during outline generation
        if (value < *xMin)
            *xMin = value;
    }
    fclose (file);
}

/**
    @brief Sort the glyphs in a font by Unicode code point.

    This function reads in an array of glyphs and sorts them
    by Unicode code point.  If a duplicate code point is encountered,
    that will result in a fatal error with an error message to stderr.

    @param[in,out] font Pointer to a Font structure with glyphs to sort.
*/
void
sortGlyphs (Font *font)
{
    Glyph *glyphs = getBufferHead (font->glyphs);
    const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
    glyphs++; // glyph 0 does not need sorting
    qsort (glyphs, glyphsEnd - glyphs, sizeof *glyphs, byCodePoint);
    for (const Glyph *glyph = glyphs; glyph < glyphsEnd - 1; glyph++)
    {
        if (glyph[0].codePoint == glyph[1].codePoint)
            fail ("Duplicate code point: "PRI_CP".", glyph[0].codePoint);
        assert (glyph[0].codePoint < glyph[1].codePoint);
```

```
    }
}

/**
    @brief Specify the current contour drawing operation.
*/
enum ContourOp {
    OP_CLOSE,    ///< Close the current contour path that was being drawn.
    OP_POINT     ///< Add one more (x,y) point to the contor being drawn.
};

/**
    @brief Fill to the left side (CFF) or right side (TrueType) of a contour.
*/
enum FillSide {
    FILL_LEFT,    ///< Draw outline counter-clockwise (CFF, PostScript).
    FILL_RIGHT    ///< Draw outline clockwise (TrueType).
};

/**
    @brief Build a glyph outline.

    This function builds a glyph outline from a Unifont glyph bitmap.

    @param[out] result The resulting glyph outline.
    @param[in] bitmap A bitmap array.
    @param[in] byteCount the number of bytes in the input bitmap array.
    @param[in] fillSide Enumerated indicator to fill left or right side.
*/
void
buildOutline (Buffer *result, const byte bitmap[], const size_t byteCount,
    const enum FillSide fillSide)
{
    enum Direction {RIGHT, LEFT, DOWN, UP}; // order is significant

    // respective coordinate deltas
    const pixels_t dx[] = {1, -1, 0, 0}, dy[] = {0, 0, -1, 1};

    assert (byteCount % GLYPH_HEIGHT == 0);
    const uint_fast8_t bytesPerRow = byteCount / GLYPH_HEIGHT;
    const pixels_t glyphWidth = bytesPerRow * 8;
    assert (glyphWidth <= GLYPH_MAX_WIDTH);

    #if GLYPH_MAX_WIDTH < 32
        typedef uint_fast32_t row_t;
    #elif GLYPH_MAX_WIDTH < 64
        typedef uint_fast64_t row_t;
    #else
        #error GLYPH_MAX_WIDTH is too large.
    #endif

    row_t pixels[GLYPH_HEIGHT + 2] = {0};
    for (pixels_t row = GLYPH_HEIGHT; row > 0; row--)
        for (pixels_t b = 0; b < bytesPerRow; b++)
            pixels[row] = pixels[row] « 8 | *bitmap++;
    typedef row_t graph_t[GLYPH_HEIGHT + 1];
    graph_t vectors[4];
    const row_t *lower = pixels, *upper = pixels + 1;
    for (pixels_t row = 0; row <= GLYPH_HEIGHT; row++)
    {
        const row_t m = (fillSide == FILL_RIGHT) - 1;
        vectors[RIGHT][row] = (m ^ (*lower « 1)) & (~m ^ (*upper « 1));
        vectors[LEFT ][row] = (m ^ (*upper     )) & (~m ^ (*lower     ));
        vectors[DOWN ][row] = (m ^ (*lower     )) & (~m ^ (*lower « 1));
        vectors[UP   ][row] = (m ^ (*upper « 1)) & (~m ^ (*upper     ));
        lower++;
        upper++;
    }
    graph_t selection = {0};
    const row_t x0 = (row_t)1 « glyphWidth;

    /// Get the value of a given bit that is in a given row.
    #define getRowBit(rows, x, y)  ((rows)[(y)] &  x0 » (x))

    /// Invert the value of a given bit that is in a given row.
    #define flipRowBit(rows, x, y) ((rows)[(y)] ^= x0 » (x))

    for (pixels_t y = GLYPH_HEIGHT; y >= 0; y--)
    {
        for (pixels_t x = 0; x <= glyphWidth; x++)
        {
```

```
            assert (!getRowBit (vectors[LEFT], x, y));
            assert (!getRowBit (vectors[UP], x, y));
            enum Direction initial;

            if (getRowBit (vectors[RIGHT], x, y))
                initial = RIGHT;
            else if (getRowBit (vectors[DOWN], x, y))
                initial = DOWN;
            else
                continue;

            static_assert ((GLYPH_MAX_WIDTH + 1) * (GLYPH_HEIGHT + 1) * 2 <=
                U16MAX, "potential overflow");

            uint_fast16_t lastPointCount = 0;
            for (bool converged = false;;)
            {
                uint_fast16_t pointCount = 0;
                enum Direction heading = initial;
                for (pixels_t tx = x, ty = y;;)
                {
                    if (converged)
                    {
                        storePixels (result, OP_POINT);
                        storePixels (result, tx);
                        storePixels (result, ty);
                    }
                    do
                    {
                        if (converged)
                            flipRowBit (vectors[heading], tx, ty);
                        tx += dx[heading];
                        ty += dy[heading];
                    } while (getRowBit (vectors[heading], tx, ty));
                    if (tx == x && ty == y)
                        break;
                    static_assert ((UP ^ DOWN) == 1 && (LEFT ^ RIGHT) == 1,
                        "wrong enums");
                    heading = (heading & 2) ^ 2;
                    heading |= !!getRowBit (selection, tx, ty);
                    heading ^= !getRowBit (vectors[heading], tx, ty);
                    assert (getRowBit (vectors[heading], tx, ty));
                    flipRowBit (selection, tx, ty);
                    pointCount++;
                }
                if (converged)
                    break;
                converged = pointCount == lastPointCount;
                lastPointCount = pointCount;
            }

            storePixels (result, OP_CLOSE);
        }
    }
    #undef getRowBit
    #undef flipRowBit
}

/**
    @brief Prepare 32-bit glyph offsets in a font table.

    @param[in] sizes Array of glyph sizes, for offset calculations.
*/
void
prepareOffsets (size_t *sizes)
{
    size_t *p = sizes;
    for (size_t *i = sizes + 1; *i; i++)
        *i += *p++;
    if (*p > 2147483647U) // offset not representable
        fail ("CFF table is too large.");
}

/**
    @brief Prepare a font name string index.

    @param[in] names List of name strings.
    @return Pointer to a Buffer struct containing the string names.
*/
Buffer *
prepareStringIndex (const NameStrings names)
```

```
{
    Buffer *buf = newBuffer (256);
    assert (names[6]);
    const char *strings[] = {"Adobe", "Identity", names[6]};
    /// Get the number of elements in array char *strings[].
    #define stringCount (sizeof strings / sizeof *strings)
    static_assert (stringCount <= U16MAX, "too many strings");
    size_t offset = 1;
    size_t lengths[stringCount];
    for (size_t i = 0; i < stringCount; i++)
    {
        assert (strings[i]);
        lengths[i] = strlen (strings[i]);
        offset += lengths[i];
    }
    int offsetSize = 1 + (offset > 0xff)
                       + (offset > 0xffff)
                       + (offset > 0xffffff);
    cacheU16 (buf, stringCount); // count
    cacheU8 (buf, offsetSize); // offSize
    cacheU (buf, offset = 1, offsetSize); // offset[0]
    for (size_t i = 0; i < stringCount; i++)
        cacheU (buf, offset += lengths[i], offsetSize); // offset[i + 1]
    for (size_t i = 0; i < stringCount; i++)
        cacheBytes (buf, strings[i], lengths[i]);
    #undef stringCount
    return buf;
}

/**
    @brief Add a CFF table to a font.

    @param[in,out] font Pointer to a Font struct to contain the CFF table.
    @param[in] version Version of CFF table, with value 1 or 2.
    @param[in] names List of NameStrings.
*/
void
fillCFF (Font *font, int version, const NameStrings names)
{
    // HACK: For convenience, CFF data structures are hard coded.
    assert (0 < version && version <= 2);
    Buffer *cff = newBuffer (65536);
    addTable (font, version == 1 ? "CFF " : "CFF2", cff);

    /// Use fixed width integer for variables to simplify offset calculation.
    #define cacheCFF32(buf, x) (cacheU8 ((buf), 29), cacheU32 ((buf), (x)))

    // In Unifont, 16px glyphs are more common. This is used by CFF1 only.
    const pixels_t defaultWidth = 16, nominalWidth = 8;
    if (version == 1)
    {
        Buffer *strings = prepareStringIndex (names);
        size_t stringsSize = countBufferedBytes (strings);
        const char *cffName = names[6];
        assert (cffName);
        size_t nameLength = strlen (cffName);
        size_t namesSize = nameLength + 5;
        // These sizes must be updated together with the data below.
        size_t offsets[] = {4, namesSize, 45, stringsSize, 2, 5, 8, 32, 4, 0};
        prepareOffsets (offsets);
        { // Header
            cacheU8 (cff, 1); // major
            cacheU8 (cff, 0); // minor
            cacheU8 (cff, 4); // hdrSize
            cacheU8 (cff, 1); // offSize
        }
        assert (countBufferedBytes (cff) == offsets[0]);
        { // Name INDEX (should not be used by OpenType readers)
            cacheU16 (cff, 1); // count
            cacheU8 (cff, 1); // offSize
            cacheU8 (cff, 1); // offset[0]
            if (nameLength + 1 > 255) // must be too long; spec limit is 63
                fail ("PostScript name is too long.");
            cacheU8 (cff, nameLength + 1); // offset[1]
            cacheBytes (cff, cffName, nameLength);
        }
        assert (countBufferedBytes (cff) == offsets[1]);
        { // Top DICT INDEX
            cacheU16 (cff, 1); // count
            cacheU8 (cff, 1); // offSize
            cacheU8 (cff, 1); // offset[0]
```

```
            cacheU8 (cff, 41); // offset[1]
            cacheCFFOperand (cff, 391); // "Adobe"
            cacheCFFOperand (cff, 392); // "Identity"
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, (byte[]){12, 30}, 2); // ROS
            cacheCFF32 (cff, font->glyphCount);
            cacheBytes (cff, (byte[]){12, 34}, 2); // CIDCount
            cacheCFF32 (cff, offsets[6]);
            cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
            cacheCFF32 (cff, offsets[5]);
            cacheBytes (cff, (byte[]){12, 37}, 2); // FDSelect
            cacheCFF32 (cff, offsets[4]);
            cacheU8 (cff, 15); // charset
            cacheCFF32 (cff, offsets[8]);
            cacheU8 (cff, 17); // CharStrings
        }
        assert (countBufferedBytes (cff) == offsets[2]);
        { // String INDEX
            cacheBuffer (cff, strings);
            freeBuffer (strings);
        }
        assert (countBufferedBytes (cff) == offsets[3]);
        cacheU16 (cff, 0); // Global Subr INDEX
        assert (countBufferedBytes (cff) == offsets[4]);
        { // Charsets
            cacheU8 (cff, 2); // format
            { // Range2[0]
                cacheU16 (cff, 1); // first
                cacheU16 (cff, font->glyphCount - 2); // nLeft
            }
        }
        assert (countBufferedBytes (cff) == offsets[5]);
        { // FDSelect
            cacheU8 (cff, 3); // format
            cacheU16 (cff, 1); // nRanges
            cacheU16 (cff, 0); // first
            cacheU8 (cff, 0); // fd
            cacheU16 (cff, font->glyphCount); // sentinel
        }
        assert (countBufferedBytes (cff) == offsets[6]);
        { // FDArray
            cacheU16 (cff, 1); // count
            cacheU8 (cff, 1); // offSize
            cacheU8 (cff, 1); // offset[0]
            cacheU8 (cff, 28); // offset[1]
            cacheCFFOperand (cff, 393);
            cacheBytes (cff, (byte[]){12, 38}, 2); // FontName
            // Windows requires FontMatrix in Font DICT.
            const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
            cacheCFFOperand (cff, offsets[8] - offsets[7]); // size
            cacheCFF32 (cff, offsets[7]); // offset
            cacheU8 (cff, 18); // Private
        }
        assert (countBufferedBytes (cff) == offsets[7]);
        { // Private
            cacheCFFOperand (cff, FU (defaultWidth));
            cacheU8 (cff, 20); // defaultWidthX
            cacheCFFOperand (cff, FU (nominalWidth));
            cacheU8 (cff, 21); // nominalWidthX
        }
        assert (countBufferedBytes (cff) == offsets[8]);
    }
    else
    {
        assert (version == 2);
        // These sizes must be updated together with the data below.
        size_t offsets[] = {5, 21, 4, 10, 0};
        prepareOffsets (offsets);
        { // Header
            cacheU8 (cff, 2); // majorVersion
            cacheU8 (cff, 0); // minorVersion
            cacheU8 (cff, 5); // headerSize
            cacheU16 (cff, offsets[1] - offsets[0]); // topDictLength
        }
```

```
        assert (countBufferedBytes (cff) == offsets[0]);
        { // Top DICT
            const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
            cacheCFFOperand (cff, offsets[2]);
            cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
            cacheCFFOperand (cff, offsets[3]);
            cacheU8 (cff, 17); // CharStrings
        }
        assert (countBufferedBytes (cff) == offsets[1]);
        cacheU32 (cff, 0); // Global Subr INDEX
        assert (countBufferedBytes (cff) == offsets[2]);
        { // Font DICT INDEX
            cacheU32 (cff, 1); // count
            cacheU8 (cff, 1); // offSize
            cacheU8 (cff, 1); // offset[0]
            cacheU8 (cff, 4); // offset[1]
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheU8 (cff, 18); // Private
        }
        assert (countBufferedBytes (cff) == offsets[3]);
    }
    { // CharStrings INDEX
        Buffer *offsets = newBuffer (4096);
        Buffer *charstrings = newBuffer (4096);
        Buffer *outline = newBuffer (1024);
        const Glyph *glyph = getBufferHead (font->glyphs);
        const Glyph *const endGlyph = glyph + font->glyphCount;
        for (; glyph < endGlyph; glyph++)
        {
            // CFF offsets start at 1
            storeU32 (offsets, countBufferedBytes (charstrings) + 1);

            pixels_t rx = -glyph->pos;
            pixels_t ry = DESCENDER;
            resetBuffer (outline);
            buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_LEFT);
            enum CFFOp {rmoveto=21, hmoveto=22, vmoveto=4, hlineto=6,
                vlineto=7, endchar=14};
            enum CFFOp pendingOp = 0;
            const int STACK_LIMIT = version == 1 ? 48 : 513;
            int stackSize = 0;
            bool isDrawing = false;
            pixels_t width = glyph->combining ? 0 : PW (glyph->byteCount);
            if (version == 1 && width != defaultWidth)
            {
                cacheCFFOperand (charstrings, FU (width - nominalWidth));
                stackSize++;
            }
            for (const pixels_t *p = getBufferHead (outline),
                 *const end = getBufferTail (outline); p < end;)
            {
                int s = 0;
                const enum ContourOp op = *p++;
                if (op == OP_POINT)
                {
                    const pixels_t x = *p++, y = *p++;
                    if (x != rx)
                    {
                        cacheCFFOperand (charstrings, FU (x - rx));
                        rx = x;
                        stackSize++;
                        s |= 1;
                    }
                    if (y != ry)
                    {
                        cacheCFFOperand (charstrings, FU (y - ry));
                        ry = y;
                        stackSize++;
                        s |= 2;
                    }
                    assert (!(isDrawing && s == 3));
                }
                if (s)
```

```
            {
                if (!isDrawing)
                {
                    const enum CFFOp moves[] = {0, hmoveto, vmoveto,
                        rmoveto};
                    cacheU8 (charstrings, moves[s]);
                    stackSize = 0;
                }
                else if (!pendingOp)
                    pendingOp = (enum CFFOp[]){0, hlineto, vlineto}[s];
            }
            else if (!isDrawing)
            {
                // only when the first point happens to be (0, 0)
                cacheCFFOperand (charstrings, FU (0));
                cacheU8 (charstrings, hmoveto);
                stackSize = 0;
            }
            if (op == OP_CLOSE || stackSize >= STACK_LIMIT)
            {
                assert (stackSize <= STACK_LIMIT);
                cacheU8 (charstrings, pendingOp);
                pendingOp = 0;
                stackSize = 0;
            }
            isDrawing = op != OP_CLOSE;
        }
        if (version == 1)
            cacheU8 (charstrings, endchar);
    }
    size_t lastOffset = countBufferedBytes (charstrings) + 1;
    #if SIZE_MAX > U32MAX
        if (lastOffset > U32MAX)
            fail ("CFF data exceeded size limit.");
    #endif
    storeU32 (offsets, lastOffset);
    int offsetSize = 1 + (lastOffset > 0xff)
                       + (lastOffset > 0xffff)
                       + (lastOffset > 0xffffff);
    // count (must match 'numGlyphs' in 'maxp' table)
    cacheU (cff, font->glyphCount, version * 2);
    cacheU8 (cff, offsetSize); // offSize
    const uint_least32_t *p = getBufferHead (offsets);
    const uint_least32_t *const end = getBufferTail (offsets);
    for (; p < end; p++)
        cacheU (cff, *p, offsetSize); // offsets
    cacheBuffer (cff, charstrings); // data
    freeBuffer (offsets);
    freeBuffer (charstrings);
    freeBuffer (outline);
  }
  #undef cacheCFF32
}

/**
  @brief Add a TrueType table to a font.

  @param[in,out] font Pointer to a Font struct to contain the TrueType table.
  @param[in] format The TrueType "loca" table format, Offset16 or Offset32.
  @param[in] names List of NameStrings.
*/
void
fillTrueType (Font *font, enum LocaFormat *format,
  uint_fast16_t *maxPoints, uint_fast16_t *maxContours)
{
  Buffer *glyf = newBuffer (65536);
  addTable (font, "glyf", glyf);
  Buffer *loca = newBuffer (4 * (font->glyphCount + 1));
  addTable (font, "loca", loca);
  *format = LOCA_OFFSET32;
  Buffer *endPoints = newBuffer (256);
  Buffer *flags = newBuffer (256);
  Buffer *xs = newBuffer (256);
  Buffer *ys = newBuffer (256);
  Buffer *outline = newBuffer (1024);
  Glyph *const glyphs = getBufferHead (font->glyphs);
  const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
  for (Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
  {
    cacheU32 (loca, countBufferedBytes (glyf));
    pixels_t rx = -glyph->pos;
```

```
        pixels_t ry = DESCENDER;
        pixels_t xMin = GLYPH_MAX_WIDTH, xMax = 0;
        pixels_t yMin = ASCENDER, yMax = -DESCENDER;
        resetBuffer (endPoints);
        resetBuffer (flags);
        resetBuffer (xs);
        resetBuffer (ys);
        resetBuffer (outline);
        buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_RIGHT);
        uint_fast32_t pointCount = 0, contourCount = 0;
        for (const pixels_t *p = getBufferHead (outline),
             *const end = getBufferTail (outline); p < end;)
        {
            const enum ContourOp op = *p++;
            if (op == OP_CLOSE)
            {
                contourCount++;
                assert (contourCount <= U16MAX);
                cacheU16 (endPoints, pointCount - 1);
                continue;
            }
            assert (op == OP_POINT);
            pointCount++;
            assert (pointCount <= U16MAX);
            const pixels_t x = *p++, y = *p++;
            uint_fast8_t pointFlags =
                + B1 (0) // point is on curve
                + BX (1, x != rx) // x coordinate is 1 byte instead of 2
                + BX (2, y != ry) // y coordinate is 1 byte instead of 2
                + B0 (3) // repeat
                + BX (4, x >= rx) // when x is 1 byte: x is positive;
                                  // when x is 2 bytes: x unchanged and omitted
                + BX (5, y >= ry) // when y is 1 byte: y is positive;
                                  // when y is 2 bytes: y unchanged and omitted
                + B1 (6) // contours may overlap
                + B0 (7) // reserved
            ;
            cacheU8 (flags, pointFlags);
            if (x != rx)
                cacheU8 (xs, FU (x > rx ? x - rx : rx - x));
            if (y != ry)
                cacheU8 (ys, FU (y > ry ? y - ry : ry - y));
            if (x < xMin) xMin = x;
            if (y < yMin) yMin = y;
            if (x > xMax) xMax = x;
            if (y > yMax) yMax = y;
            rx = x;
            ry = y;
        }
        if (contourCount == 0)
            continue; // blank glyph is indicated by the 'loca' table
        glyph->lsb = glyph->pos + xMin;
        cacheU16 (glyf, contourCount); // numberOfContours
        cacheU16 (glyf, FU (glyph->pos + xMin)); // xMin
        cacheU16 (glyf, FU (yMin)); // yMin
        cacheU16 (glyf, FU (glyph->pos + xMax)); // xMax
        cacheU16 (glyf, FU (yMax)); // yMax
        cacheBuffer (glyf, endPoints); // endPtsOfContours[]
        cacheU16 (glyf, 0); // instructionLength
        cacheBuffer (glyf, flags); // flags[]
        cacheBuffer (glyf, xs); // xCoordinates[]
        cacheBuffer (glyf, ys); // yCoordinates[]
        if (pointCount > *maxPoints)
            *maxPoints = pointCount;
        if (contourCount > *maxContours)
            *maxContours = contourCount;
    }
    cacheU32 (loca, countBufferedBytes (glyf));
    freeBuffer (endPoints);
    freeBuffer (flags);
    freeBuffer (xs);
    freeBuffer (ys);
    freeBuffer (outline);
}

/**
    @brief Create a dummy blank outline in a font table.

    @param[in,out] font Pointer to a Font struct to insert a blank outline.
*/
void
```

```
fillBlankOutline (Font *font)
{
    Buffer *glyf = newBuffer (12);
    addTable (font, "glyf", glyf);
    // Empty table is not allowed, but an empty outline for glyph 0 suffices.
    cacheU16 (glyf, 0); // numberOfContours
    cacheU16 (glyf, FU (0)); // xMin
    cacheU16 (glyf, FU (0)); // yMin
    cacheU16 (glyf, FU (0)); // xMax
    cacheU16 (glyf, FU (0)); // yMax
    cacheU16 (glyf, 0); // instructionLength
    Buffer *loca = newBuffer (2 * (font->glyphCount + 1));
    addTable (font, "loca", loca);
    cacheU16 (loca, 0); // offsets[0]
    assert (countBufferedBytes (glyf) % 2 == 0);
    for (uint_fast32_t i = 1; i <= font->glyphCount; i++)
        cacheU16 (loca, countBufferedBytes (glyf) / 2); // offsets[i]
}

/**
    @brief Fill OpenType bitmap data and location tables.

    This function fills an Embedded Bitmap Data (EBDT) Table
    and an Embedded Bitmap Location (EBLC) Table with glyph
    bitmap information.  These tables enable embedding bitmaps
    in OpenType fonts.  No Embedded Bitmap Scaling (EBSC) table
    is used for the bitmap glyphs, only EBDT and EBLC.

    @param[in,out] font Pointer to a Font struct in which to add bitmaps.
*/
void
fillBitmap (Font *font)
{
    const Glyph *const glyphs = getBufferHead (font->glyphs);
    const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
    size_t bitmapsSize = 0;
    for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
        bitmapsSize += glyph->byteCount;
    Buffer *ebdt = newBuffer (4 + bitmapsSize);
    addTable (font, "EBDT", ebdt);
    cacheU16 (ebdt, 2); // majorVersion
    cacheU16 (ebdt, 0); // minorVersion
    uint_fast8_t byteCount = 0; // unequal to any glyph
    pixels_t pos = 0;
    bool combining = false;
    Buffer *rangeHeads = newBuffer (32);
    Buffer *offsets = newBuffer (64);
    for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
    {
        if (glyph->byteCount != byteCount || glyph->pos != pos ||
            glyph->combining != combining)
        {
            storeU16 (rangeHeads, glyph - glyphs);
            storeU32 (offsets, countBufferedBytes (ebdt));
            byteCount = glyph->byteCount;
            pos = glyph->pos;
            combining = glyph->combining;
        }
        cacheBytes (ebdt, glyph->bitmap, byteCount);
    }
    const uint_least16_t *ranges = getBufferHead (rangeHeads);
    const uint_least16_t *rangesEnd = getBufferTail (rangeHeads);
    uint_fast32_t rangeCount = rangesEnd - ranges;
    storeU16 (rangeHeads, font->glyphCount);
    Buffer *eblc = newBuffer (4096);
    addTable (font, "EBLC", eblc);
    cacheU16 (eblc, 2); // majorVersion
    cacheU16 (eblc, 0); // minorVersion
    cacheU32 (eblc, 1); // numSizes
    { // bitmapSizes[0]
        cacheU32 (eblc, 56); // indexSubTableArrayOffset
        cacheU32 (eblc, (8 + 20) * rangeCount); // indexTablesSize
        cacheU32 (eblc, rangeCount); // numberOfIndexSubTables
        cacheU32 (eblc, 0); // colorRef
        { // hori
            cacheU8 (eblc, ASCENDER); // ascender
            cacheU8 (eblc, -DESCENDER); // descender
            cacheU8 (eblc, font->maxWidth); // widthMax
            cacheU8 (eblc, 1); // caretSlopeNumerator
            cacheU8 (eblc, 0); // caretSlopeDenominator
            cacheU8 (eblc, 0); // caretOffset
```

```
            cacheU8 (eblc, 0); // minOriginSB
            cacheU8 (eblc, 0); // minAdvanceSB
            cacheU8 (eblc, ASCENDER); // maxBeforeBL
            cacheU8 (eblc, -DESCENDER); // minAfterBL
            cacheU8 (eblc, 0); // pad1
            cacheU8 (eblc, 0); // pad2
        }
        { // vert
            cacheU8 (eblc, ASCENDER); // ascender
            cacheU8 (eblc, -DESCENDER); // descender
            cacheU8 (eblc, font->maxWidth); // widthMax
            cacheU8 (eblc, 1); // caretSlopeNumerator
            cacheU8 (eblc, 0); // caretSlopeDenominator
            cacheU8 (eblc, 0); // caretOffset
            cacheU8 (eblc, 0); // minOriginSB
            cacheU8 (eblc, 0); // minAdvanceSB
            cacheU8 (eblc, ASCENDER); // maxBeforeBL
            cacheU8 (eblc, -DESCENDER); // minAfterBL
            cacheU8 (eblc, 0); // pad1
            cacheU8 (eblc, 0); // pad2
        }
        cacheU16 (eblc, 0); // startGlyphIndex
        cacheU16 (eblc, font->glyphCount - 1); // endGlyphIndex
        cacheU8 (eblc, 16); // ppemX
        cacheU8 (eblc, 16); // ppemY
        cacheU8 (eblc, 1); // bitDepth
        cacheU8 (eblc, 1); // flags = Horizontal
    }
    { // IndexSubTableArray
        uint_fast32_t offset = rangeCount * 8;
        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
        {
            cacheU16 (eblc, *p); // firstGlyphIndex
            cacheU16 (eblc, p[1] - 1); // lastGlyphIndex
            cacheU32 (eblc, offset); // additionalOffsetToIndexSubtable
            offset += 20;
        }
    }
    { // IndexSubTables
        const uint_least32_t *offset = getBufferHead (offsets);
        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
        {
            const Glyph *glyph = &glyphs[*p];
            cacheU16 (eblc, 2); // indexFormat
            cacheU16 (eblc, 5); // imageFormat
            cacheU32 (eblc, *offset++); // imageDataOffset
            cacheU32 (eblc, glyph->byteCount); // imageSize
            { // bigMetrics
                cacheU8 (eblc, GLYPH_HEIGHT); // height
                const uint_fast8_t width = PW (glyph->byteCount);
                cacheU8 (eblc, width); // width
                cacheU8 (eblc, glyph->pos); // horiBearingX
                cacheU8 (eblc, ASCENDER); // horiBearingY
                cacheU8 (eblc, glyph->combining ? 0 : width); // horiAdvance
                cacheU8 (eblc, 0); // vertBearingX
                cacheU8 (eblc, 0); // vertBearingY
                cacheU8 (eblc, GLYPH_HEIGHT); // vertAdvance
            }
        }
    }
    freeBuffer (rangeHeads);
    freeBuffer (offsets);
}

/**
    @brief Fill a ”head” font table.

    The ”head” table contains font header information common to the
    whole font.

    @param[in,out] font The Font struct to which to add the table.
    @param[in] locaFormat The ”loca” offset index location table.
    @param[in] xMin The minimum x-coordinate for a glyph.
*/
void
fillHeadTable (Font *font, enum LocaFormat locaFormat, pixels_t xMin)
{
    Buffer *head = newBuffer (56);
    addTable (font, ”head”, head);
    cacheU16 (head, 1); // majorVersion
    cacheU16 (head, 0); // minorVersion
```

```
    cacheZeros (head, 4); // fontRevision (unused)
    // The 'checksumAdjustment' field is a checksum of the entire file.
    // It is later calculated and written directly in the 'writeFont' function.
    cacheU32 (head, 0); // checksumAdjustment (placeholder)
    cacheU32 (head, 0x5f0f3cf5); // magicNumber
    const uint_fast16_t flags =
        + B1 ( 0) // baseline at y=0
        + B1 ( 1) // LSB at x=0 (doubtful; probably should be LSB=xMin)
        + B0 ( 2) // instructions may depend on point size
        + B0 ( 3) // force internal ppem to integers
        + B0 ( 4) // instructions may alter advance width
        + B0 ( 5) // not used in OpenType
        + B0 ( 6) // not used in OpenType
        + B0 ( 7) // not used in OpenType
        + B0 ( 8) // not used in OpenType
        + B0 ( 9) // not used in OpenType
        + B0 (10) // not used in OpenType
        + B0 (11) // font transformed
        + B0 (12) // font converted
        + B0 (13) // font optimized for ClearType
        + B0 (14) // last resort font
        + B0 (15) // reserved
    ;
    cacheU16 (head, flags); // flags
    cacheU16 (head, FUPEM); // unitsPerEm
    cacheZeros (head, 8); // created (unused)
    cacheZeros (head, 8); // modified (unused)
    cacheU16 (head, FU (xMin)); // xMin
    cacheU16 (head, FU (-DESCENDER)); // yMin
    cacheU16 (head, FU (font->maxWidth)); // xMax
    cacheU16 (head, FU (ASCENDER)); // yMax
    // macStyle (must agree with 'fsSelection' in 'OS/2' table)
    const uint_fast16_t macStyle =
        + B0 (0) // bold
        + B0 (1) // italic
        + B0 (2) // underline
        + B0 (3) // outline
        + B0 (4) // shadow
        + B0 (5) // condensed
        + B0 (6) // extended
        //    7-15  reserved
    ;
    cacheU16 (head, macStyle);
    cacheU16 (head, GLYPH_HEIGHT); // lowestRecPPEM
    cacheU16 (head, 2); // fontDirectionHint
    cacheU16 (head, locaFormat); // indexToLocFormat
    cacheU16 (head, 0); // glyphDataFormat
}

/**
    @brief Fill a "hhea" font table.

    The "hhea" table contains horizontal header information,
    for example left and right side bearings.

    @param[in,out] font The Font struct to which to add the table.
    @param[in] xMin The minimum x-coordinate for a glyph.
*/
void
fillHheaTable (Font *font, pixels_t xMin)
{
    Buffer *hhea = newBuffer (36);
    addTable (font, "hhea", hhea);
    cacheU16 (hhea, 1); // majorVersion
    cacheU16 (hhea, 0); // minorVersion
    cacheU16 (hhea, FU (ASCENDER)); // ascender
    cacheU16 (hhea, FU (-DESCENDER)); // descender
    cacheU16 (hhea, FU (0)); // lineGap
    cacheU16 (hhea, FU (font->maxWidth)); // advanceWidthMax
    cacheU16 (hhea, FU (xMin)); // minLeftSideBearing
    cacheU16 (hhea, FU (0)); // minRightSideBearing (unused)
    cacheU16 (hhea, FU (font->maxWidth)); // xMaxExtent
    cacheU16 (hhea, 1); // caretSlopeRise
    cacheU16 (hhea, 0); // caretSlopeRun
    cacheU16 (hhea, 0); // caretOffset
    cacheU16 (hhea, 0); // reserved
    cacheU16 (hhea, 0); // reserved
    cacheU16 (hhea, 0); // reserved
    cacheU16 (hhea, 0); // reserved
    cacheU16 (hhea, 0); // metricDataFormat
    cacheU16 (hhea, font->glyphCount); // numberOfHMetrics
```

```
}

/**
    @brief Fill a "maxp" font table.

    The "maxp" table contains maximum profile information,
    such as the memory required to contain the font.

    @param[in,out] font The Font struct to which to add the table.
    @param[in] isCFF true if a CFF font is included, false otherwise.
    @param[in] maxPoints Maximum points in a non-composite glyph.
    @param[in] maxContours Maximum contours in a non-composite glyph.
*/
void
fillMaxpTable (Font *font, bool isCFF, uint_fast16_t maxPoints,
    uint_fast16_t maxContours)
{
    Buffer *maxp = newBuffer (32);
    addTable (font, "maxp", maxp);
    cacheU32 (maxp, isCFF ? 0x00005000 : 0x00010000); // version
    cacheU16 (maxp, font->glyphCount); // numGlyphs
    if (isCFF)
        return;
    cacheU16 (maxp, maxPoints); // maxPoints
    cacheU16 (maxp, maxContours); // maxContours
    cacheU16 (maxp, 0); // maxCompositePoints
    cacheU16 (maxp, 0); // maxCompositeContours
    cacheU16 (maxp, 0); // maxZones
    cacheU16 (maxp, 0); // maxTwilightPoints
    cacheU16 (maxp, 0); // maxStorage
    cacheU16 (maxp, 0); // maxFunctionDefs
    cacheU16 (maxp, 0); // maxInstructionDefs
    cacheU16 (maxp, 0); // maxStackElements
    cacheU16 (maxp, 0); // maxSizeOfInstructions
    cacheU16 (maxp, 0); // maxComponentElements
    cacheU16 (maxp, 0); // maxComponentDepth
}

/**
    @brief Fill an "OS/2" font table.

    The "OS/2" table contains OS/2 and Windows font metrics information.

    @param[in,out] font The Font struct to which to add the table.
*/
void
fillOS2Table (Font *font)
{
    Buffer *os2 = newBuffer (100);
    addTable (font, "OS/2", os2);
    cacheU16 (os2, 5); // version
    // HACK: Average glyph width is not actually calculated.
    cacheU16 (os2, FU (font->maxWidth)); // xAvgCharWidth
    cacheU16 (os2, 400); // usWeightClass = Normal
    cacheU16 (os2, 5); // usWidthClass = Medium
    const uint_fast16_t typeFlags =
        + B0 (0) // reserved
        // usage permissions, one of:
            // Default: Installable embedding
            + B0 (1) // Restricted License embedding
            + B0 (2) // Preview & Print embedding
            + B0 (3) // Editable embedding
        //    4-7    reserved
        + B0 (8) // no subsetting
        + B0 (9) // bitmap embedding only
        //    10-15 reserved
    ;
    cacheU16 (os2, typeFlags); // fsType
    cacheU16 (os2, FU (5)); // ySubscriptXSize
    cacheU16 (os2, FU (7)); // ySubscriptYSize
    cacheU16 (os2, FU (0)); // ySubscriptXOffset
    cacheU16 (os2, FU (1)); // ySubscriptYOffset
    cacheU16 (os2, FU (5)); // ySuperscriptXSize
    cacheU16 (os2, FU (7)); // ySuperscriptYSize
    cacheU16 (os2, FU (0)); // ySuperscriptXOffset
    cacheU16 (os2, FU (4)); // ySuperscriptYOffset
    cacheU16 (os2, FU (1)); // yStrikeoutSize
    cacheU16 (os2, FU (5)); // yStrikeoutPosition
    cacheU16 (os2, 0x080a); // sFamilyClass = Sans Serif, Matrix
    const byte panose[] =
    {
```

```
        2, // Family Kind = Latin Text
        11, // Serif Style = Normal Sans
        4, // Weight = Thin
        // Windows would render all glyphs to the same width,
        // if 'Proportion' is set to 'Monospaced' (as Unifont should be).
        // 'Condensed' is the best alternative according to metrics.
        6, // Proportion = Condensed
        2, // Contrast = None
        2, // Stroke = No Variation
        2, // Arm Style = Straight Arms
        8, // Letterform = Normal/Square
        2, // Midline = Standard/Trimmed
        4, // X-height = Constant/Large
    };
    cacheBytes (os2, panose, sizeof panose); // panose
    // HACK: All defined Unicode ranges are marked functional for convenience.
    cacheU32 (os2, 0xffffffff); // ulUnicodeRange1
    cacheU32 (os2, 0xffffffff); // ulUnicodeRange2
    cacheU32 (os2, 0xffffffff); // ulUnicodeRange3
    cacheU32 (os2, 0x0effffff); // ulUnicodeRange4
    cacheBytes (os2, "GNU ", 4); // achVendID
    // fsSelection (must agree with 'macStyle' in 'head' table)
    const uint_fast16_t selection =
        + B0 (0) // italic
        + B0 (1) // underscored
        + B0 (2) // negative
        + B0 (3) // outlined
        + B0 (4) // strikeout
        + B0 (5) // bold
        + B1 (6) // regular
        + B1 (7) // use sTypo* metrics in this table
        + B1 (8) // font name conforms to WWS model
        + B0 (9) // oblique
        //    10-15 reserved
    ;
    cacheU16 (os2, selection);
    const Glyph *glyphs = getBufferHead (font->glyphs);
    uint_fast32_t first = glyphs[1].codePoint;
    uint_fast32_t last = glyphs[font->glyphCount - 1].codePoint;
    cacheU16 (os2, first < U16MAX ? first : U16MAX); // usFirstCharIndex
    cacheU16 (os2, last  < U16MAX ? last  : U16MAX); // usLastCharIndex
    cacheU16 (os2, FU (ASCENDER)); // sTypoAscender
    cacheU16 (os2, FU (-DESCENDER)); // sTypoDescender
    cacheU16 (os2, FU (0)); // sTypoLineGap
    cacheU16 (os2, FU (ASCENDER)); // usWinAscent
    cacheU16 (os2, FU (DESCENDER)); // usWinDescent
    // HACK: All reasonable code pages are marked functional for convenience.
    cacheU32 (os2, 0x603f01ff); // ulCodePageRange1
    cacheU32 (os2, 0xffff0000); // ulCodePageRange2
    cacheU16 (os2, FU (8)); // sxHeight
    cacheU16 (os2, FU (10)); // sCapHeight
    cacheU16 (os2, 0); // usDefaultChar
    cacheU16 (os2, 0x20); // usBreakChar
    cacheU16 (os2, 0); // usMaxContext
    cacheU16 (os2, 0); // usLowerOpticalPointSize
    cacheU16 (os2, 0xffff); // usUpperOpticalPointSize
}

/**
    @brief Fill an "hmtx" font table.

    The "hmtx" table contains horizontal metrics information.

    @param[in,out] font The Font struct to which to add the table.
*/
void
fillHmtxTable (Font *font)
{
    Buffer *hmtx = newBuffer (4 * font->glyphCount);
    addTable (font, "hmtx", hmtx);
    const Glyph *const glyphs = getBufferHead (font->glyphs);
    const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
    for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
    {
        int_fast16_t aw = glyph->combining ? 0 : PW (glyph->byteCount);
        cacheU16 (hmtx, FU (aw)); // advanceWidth
        cacheU16 (hmtx, FU (glyph->lsb)); // lsb
    }
}

/**
```

```
    @brief Fill a ”cmap” font table.

    The ”cmap” table contains character to glyph index mapping information.

    @param[in,out] font The Font struct to which to add the table.
*/
void
fillCmapTable (Font *font)
{
    Glyph *const glyphs = getBufferHead (font->glyphs);
    Buffer *rangeHeads = newBuffer (16);
    uint_fast32_t rangeCount = 0;
    uint_fast32_t bmpRangeCount = 1; // 1 for the last 0xffff-0xffff range
    glyphs[0].codePoint = glyphs[1].codePoint; // to start a range at glyph 1
    for (uint_fast16_t i = 1; i < font->glyphCount; i++)
    {
        if (glyphs[i].codePoint != glyphs[i - 1].codePoint + 1)
        {
            storeU16 (rangeHeads, i);
            rangeCount++;
            bmpRangeCount += glyphs[i].codePoint < 0xffff;
        }
    }
    Buffer *cmap = newBuffer (256);
    addTable (font, ”cmap”, cmap);
    // Format 4 table is always generated for compatibility.
    bool hasFormat12 = glyphs[font->glyphCount - 1].codePoint > 0xffff;
    cacheU16 (cmap, 0); // version
    cacheU16 (cmap, 1 + hasFormat12); // numTables
    { // encodingRecords[0]
        cacheU16 (cmap, 3); // platformID
        cacheU16 (cmap, 1); // encodingID
        cacheU32 (cmap, 12 + 8 * hasFormat12); // subtableOffset
    }
    if (hasFormat12) // encodingRecords[1]
    {
        cacheU16 (cmap, 3); // platformID
        cacheU16 (cmap, 10); // encodingID
        cacheU32 (cmap, 36 + 8 * bmpRangeCount); // subtableOffset
    }
    const uint_least16_t *ranges = getBufferHead (rangeHeads);
    const uint_least16_t *const rangesEnd = getBufferTail (rangeHeads);
    storeU16 (rangeHeads, font->glyphCount);
    { // format 4 table
        cacheU16 (cmap, 4); // format
        cacheU16 (cmap, 16 + 8 * bmpRangeCount); // length
        cacheU16 (cmap, 0); // language
        if (bmpRangeCount * 2 > U16MAX)
            fail (”Too many ranges in ’cmap’ table.”);
        cacheU16 (cmap, bmpRangeCount * 2); // segCountX2
        uint_fast16_t searchRange = 1, entrySelector = -1;
        while (searchRange <= bmpRangeCount)
        {
            searchRange «= 1;
            entrySelector++;
        }
        cacheU16 (cmap, searchRange); // searchRange
        cacheU16 (cmap, entrySelector); // entrySelector
        cacheU16 (cmap, bmpRangeCount * 2 - searchRange); // rangeShift
        { // endCode[]
            const uint_least16_t *p = ranges;
            for (p++; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
                cacheU16 (cmap, glyphs[*p - 1].codePoint);
            uint_fast32_t cp = glyphs[*p - 1].codePoint;
            if (cp > 0xfffe)
                cp = 0xfffe;
            cacheU16 (cmap, cp);
            cacheU16 (cmap, 0xffff);
        }
        cacheU16 (cmap, 0); // reservedPad
        { // startCode[]
            for (uint_fast32_t i = 0; i < bmpRangeCount - 1; i++)
                cacheU16 (cmap, glyphs[ranges[i]].codePoint);
            cacheU16 (cmap, 0xffff);
        }
        { // idDelta[]
            const uint_least16_t *p = ranges;
            for (; p < rangesEnd && glyphs[*p].codePoint < 0xffff; p++)
                cacheU16 (cmap, *p - glyphs[*p].codePoint);
            uint_fast16_t delta = 1;
            if (p < rangesEnd && *p == 0xffff)
```

```
                delta = *p - glyphs[*p].codePoint;
            cacheU16 (cmap, delta);
        }
        { // idRangeOffsets[]
            for (uint_least16_t i = 0; i < bmpRangeCount; i++)
                cacheU16 (cmap, 0);
        }
    }
    if (hasFormat12) // format 12 table
    {
        cacheU16 (cmap, 12); // format
        cacheU16 (cmap, 0); // reserved
        cacheU32 (cmap, 16 + 12 * rangeCount); // length
        cacheU32 (cmap, 0); // language
        cacheU32 (cmap, rangeCount); // numGroups

        // groups[]
        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
        {
            cacheU32 (cmap, glyphs[*p].codePoint); // startCharCode
            cacheU32 (cmap, glyphs[p[1] - 1].codePoint); // endCharCode
            cacheU32 (cmap, *p); // startGlyphID
        }
    }
    freeBuffer (rangeHeads);
}

/**
    @brief Fill a "post" font table.

    The "post" table contains information for PostScript printers.

    @param[in,out] font The Font struct to which to add the table.
*/
void
fillPostTable (Font *font)
{
    Buffer *post = newBuffer (32);
    addTable (font, "post", post);
    cacheU32 (post, 0x00030000); // version = 3.0
    cacheU32 (post, 0); // italicAngle
    cacheU16 (post, 0); // underlinePosition
    cacheU16 (post, 1); // underlineThickness
    cacheU32 (post, 1); // isFixedPitch
    cacheU32 (post, 0); // minMemType42
    cacheU32 (post, 0); // maxMemType42
    cacheU32 (post, 0); // minMemType1
    cacheU32 (post, 0); // maxMemType1
}

/**
    @brief Fill a "GPOS" font table.

    The "GPOS" table contains information for glyph positioning.

    @param[in,out] font The Font struct to which to add the table.
*/
void
fillGposTable (Font *font)
{
    Buffer *gpos = newBuffer (16);
    addTable (font, "GPOS", gpos);
    cacheU16 (gpos, 1); // majorVersion
    cacheU16 (gpos, 0); // minorVersion
    cacheU16 (gpos, 10); // scriptListOffset
    cacheU16 (gpos, 12); // featureListOffset
    cacheU16 (gpos, 14); // lookupListOffset
    { // ScriptList table
        cacheU16 (gpos, 0); // scriptCount
    }
    { // Feature List table
        cacheU16 (gpos, 0); // featureCount
    }
    { // Lookup List Table
        cacheU16 (gpos, 0); // lookupCount
    }
}

/**
    @brief Fill a "GSUB" font table.

```

```
    The ”GSUB” table contains information for glyph substitution.

    @param[in,out] font The Font struct to which to add the table.
*/
void
fillGsubTable (Font *font)
{
    Buffer *gsub = newBuffer (38);
    addTable (font, ”GSUB”, gsub);
    cacheU16 (gsub, 1); // majorVersion
    cacheU16 (gsub, 0); // minorVersion
    cacheU16 (gsub, 10); // scriptListOffset
    cacheU16 (gsub, 34); // featureListOffset
    cacheU16 (gsub, 36); // lookupListOffset
    { // ScriptList table
        cacheU16 (gsub, 2); // scriptCount
        { // scriptRecords[0]
            cacheBytes (gsub, ”DFLT”, 4); // scriptTag
            cacheU16 (gsub, 14); // scriptOffset
        }
        { // scriptRecords[1]
            cacheBytes (gsub, ”thai”, 4); // scriptTag
            cacheU16 (gsub, 14); // scriptOffset
        }
        { // Script table
            cacheU16 (gsub, 4); // defaultLangSysOffset
            cacheU16 (gsub, 0); // langSysCount
            { // Default Language System table
                cacheU16 (gsub, 0); // lookupOrderOffset
                cacheU16 (gsub, 0); // requiredFeatureIndex
                cacheU16 (gsub, 0); // featureIndexCount
            }
        }
    }
    { // Feature List table
        cacheU16 (gsub, 0); // featureCount
    }
    { // Lookup List Table
        cacheU16 (gsub, 0); // lookupCount
    }
}

/**
    @brief Cache a string as a big-ending UTF-16 surrogate pair.

    This function encodes a UTF-8 string as a big-endian UTF-16
    surrogate pair.

    @param[in,out] buf Pointer to a Buffer struct to update.
    @param[in] str The character array to encode.
*/
void
cacheStringAsUTF16BE (Buffer *buf, const char *str)
{
    for (const char *p = str; *p; p++)
    {
        byte c = *p;
        if (c < 0x80)
        {
            cacheU16 (buf, c);
            continue;
        }
        int length = 1;
        byte mask = 0x40;
        for (; c & mask; mask »= 1)
            length++;
        if (length == 1 || length > 4)
            fail (”Ill-formed UTF-8 sequence.”);
        uint_fast32_t codePoint = c & (mask - 1);
        for (int i = 1; i < length; i++)
        {
            c = *++p;
            if ((c & 0xc0) != 0x80) // NUL checked here
                fail (”Ill-formed UTF-8 sequence.”);
            codePoint = (codePoint « 6) | (c & 0x3f);
        }
        const int lowerBits = length==2 ? 7 : length==3 ? 11 : 16;
        if (codePoint » lowerBits == 0)
            fail (”Ill-formed UTF-8 sequence.”); // sequence should be shorter
        if (codePoint >= 0xd800 && codePoint <= 0xdfff)
            fail (”Ill-formed UTF-8 sequence.”);
```

```c
        if (codePoint > 0x10ffff)
            fail ("Ill-formed UTF-8 sequence.");
        if (codePoint > 0xffff)
        {
            cacheU16 (buf, 0xd800 | (codePoint - 0x10000) >> 10);
            cacheU16 (buf, 0xdc00 | (codePoint & 0x3ff));
        }
        else
            cacheU16 (buf, codePoint);
    }
}

/**
    @brief Fill a "name" font table.

    The "name" table contains name information, for example for Name IDs.

    @param[in,out] font The Font struct to which to add the table.
    @param[in] names List of NameStrings.
*/
void
fillNameTable (Font *font, NameStrings nameStrings)
{
    Buffer *name = newBuffer (2048);
    addTable (font, "name", name);
    size_t nameStringCount = 0;
    for (size_t i = 0; i < MAX_NAME_IDS; i++)
        nameStringCount += !!nameStrings[i];
    cacheU16 (name, 0); // version
    cacheU16 (name, nameStringCount); // count
    cacheU16 (name, 2 * 3 + 12 * nameStringCount); // storageOffset
    Buffer *stringData = newBuffer (1024);
    // nameRecord[]
    for (size_t i = 0; i < MAX_NAME_IDS; i++)
    {
        if (!nameStrings[i])
            continue;
        size_t offset = countBufferedBytes (stringData);
        cacheStringAsUTF16BE (stringData, nameStrings[i]);
        size_t length = countBufferedBytes (stringData) - offset;
        if (offset > U16MAX || length > U16MAX)
            fail ("Name strings are too long.");
        // Platform ID 0 (Unicode) is not well supported.
        // ID 3 (Windows) seems to be the best for compatibility.
        cacheU16 (name, 3); // platformID = Windows
        cacheU16 (name, 1); // encodingID = Unicode BMP
        cacheU16 (name, 0x0409); // languageID = en-US
        cacheU16 (name, i); // nameID
        cacheU16 (name, length); // length
        cacheU16 (name, offset); // stringOffset
    }
    cacheBuffer (name, stringData);
    freeBuffer (stringData);
}

/**
    @brief Print program version string on stdout.

    Print program version if invoked with the "--version" option,
    and then exit successfully.
*/
void
printVersion () {
    printf ("hex2otf (GNU Unifont) %s\n", VERSION);
    printf ("Copyright \u00A9 2022 \u4F55\u5FD7\u7FD4 (He Zhixiang)\n");
    printf ("License GPLv2+: GNU GPL version 2 or later\n");
    printf ("<https://gnu.org/licenses/gpl.html>\n");
    printf ("This is free software: you are free to change and\n");
    printf ("redistribute it.  There is NO WARRANTY, to the extent\n");
    printf ("permitted by law.\n");

    exit (EXIT_SUCCESS);
}

/**
    @brief Print help message to stdout and then exit.

    Print help message if invoked with the "--help" option,
    and then exit successfully.
*/
void
```

```c
printHelp () {
    printf ("Synopsis: hex2otf <options>:\n\n");
    printf ("    hex=<filename>        Specify Unifont .hex input file.\n");
    printf ("    pos=<filename>        Specify combining file. (Optional)\n");
    printf ("    out=<filename>        Specify output font file.\n");
    printf ("    format=<f1>,<f2>,...  Specify font format(s); values:\n");
    printf ("                          cff\n");
    printf ("                          cff2\n");
    printf ("                          truetype\n");
    printf ("                          blank\n");
    printf ("                          bitmap\n");
    printf ("                          gpos\n");
    printf ("                          gsub\n");
    printf ("\nExample:\n\n");
    printf ("    hex2otf hex=Myfont.hex out=Myfont.otf format=cff\n\n");
    printf ("For more information, consult the hex2otf(1) man page.\n\n");

    exit (EXIT_SUCCESS);
}

/**
   @brief Data structure to hold options for OpenType font output.

   This data structure holds the status of options that can be
   specified as command line arguments for creating the output
   OpenType font file.
*/
typedef struct Options
{
    bool truetype, blankOutline, bitmap, gpos, gsub;
    int cff; // 0 = no CFF outline; 1 = use 'CFF' table; 2 = use 'CFF2' table
    const char *hex, *pos, *out; // file names
    NameStrings nameStrings; // indexed directly by Name IDs
} Options;

/**
    @brief Match a command line option with its key for enabling.

    @param[in] operand A pointer to the specified operand.
    @param[in] key Pointer to the option structure.
    @param[in] delimeter The delimiter to end searching.
    @return Pointer to the first character of the desired option.
*/
const char *
matchToken (const char *operand, const char *key, char delimiter)
{
    while (*key)
        if (*operand++ != *key++)
            return NULL;
    if (!*operand || *operand++ == delimiter)
        return operand;
    return NULL;
}

/**
    @brief Parse command line options.

        Option          Data Type       Description
        ------          ---------       -----------
        truetype        bool            Generate TrueType outlines
        blankOutline    bool            Generate blank outlines
        bitmap          bool            Generate embedded bitmap
        gpos            bool            Generate a dummy GPOS table
        gsub            bool            Generate a dummy GSUB table
        cff             int             Generate CFF 1 or CFF 2 outlines
        hex             const char *    Name of Unifont .hex file
        pos             const char *    Name of Unifont combining data file
        out             const char *    Name of output font file
        nameStrings     NameStrings     Array of TrueType font Name IDs

    @param[in] argv Pointer to array of command line options.
    @return Data structure to hold requested command line options.
*/
Options
parseOptions (char *const argv[const])
{
    Options opt = {0}; // all options default to 0, false and NULL
    const char *format = NULL;
    struct StringArg
    {
        const char *const key;
```

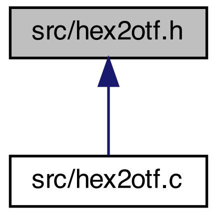
```
        const char **const value;
    } strArgs[] =
    {
        {"hex", &opt.hex},
        {"pos", &opt.pos},
        {"out", &opt.out},
        {"format", &format},
        {NULL, NULL} // sentinel
    };
    for (char *const *argp = argv + 1; *argp; argp++)
    {
        const char *const arg = *argp;
        struct StringArg *p;
        const char *value = NULL;
        if (strcmp (arg, "--help") == 0)
            printHelp ();
        if (strcmp (arg, "--version") == 0)
            printVersion ();
        for (p = strArgs; p->key; p++)
            if ((value = matchToken (arg, p->key, '=')))
                break;
        if (p->key)
        {
            if (!*value)
                fail ("Empty argument: '%s'.", p->key);
            if (*p->value)
                fail ("Duplicate argument: '%s'.", p->key);
            *p->value = value;
        }
        else // shall be a name string
        {
            char *endptr;
            unsigned long id = strtoul (arg, &endptr, 10);
            if (endptr == arg || id >= MAX_NAME_IDS || *endptr != '=')
                fail ("Invalid argument: '%s'.", arg);
            endptr++; // skip '='
            if (opt.nameStrings[id])
                fail ("Duplicate name ID: %lu.", id);
            opt.nameStrings[id] = endptr;
        }
    }
    if (!opt.hex)
        fail ("Hex file is not specified.");
    if (opt.pos && opt.pos[0] == '\0')
        opt.pos = NULL; // Position file is optional. Empty path means none.
    if (!opt.out)
        fail ("Output file is not specified.");
    if (!format)
        fail ("Format is not specified.");
    for (const NamePair *p = defaultNames; p->str; p++)
        if (!opt.nameStrings[p->id])
            opt.nameStrings[p->id] = p->str;
    bool cff = false, cff2 = false;
    struct Symbol
    {
        const char *const key;
        bool *const found;
    } symbols[] =
    {
        {"cff", &cff},
        {"cff2", &cff2},
        {"truetype", &opt.truetype},
        {"blank", &opt.blankOutline},
        {"bitmap", &opt.bitmap},
        {"gpos", &opt.gpos},
        {"gsub", &opt.gsub},
        {NULL, NULL} // sentinel
    };
    while (*format)
    {
        const struct Symbol *p;
        const char *next = NULL;
        for (p = symbols; p->key; p++)
            if ((next = matchToken (format, p->key, ',')))
                break;
        if (!p->key)
            fail ("Invalid format.");
        *p->found = true;
        format = next;
    }
    if (cff + cff2 + opt.truetype + opt.blankOutline > 1)
```

```
        fail ("At most one outline format can be accepted.");
    if (!(cff || cff2 || opt.truetype || opt.bitmap))
        fail ("Invalid format.");
    opt.cff = cff + cff2 * 2;
    return opt;
}

/**
  @brief The main function.

  @param[in] argc The number of command-line arguments.
  @param[in] argv The array of command-line arguments.
  @return EXIT_FAILURE upon fatal error, EXIT_SUCCESS otherwise.
*/
int
main (int argc, char *argv[])
{
    initBuffers (16);
    atexit (cleanBuffers);
    Options opt = parseOptions (argv);
    Font font;
    font.tables = newBuffer (sizeof (Table) * 16);
    font.glyphs = newBuffer (sizeof (Glyph) * MAX_GLYPHS);
    readGlyphs (&font, opt.hex);
    sortGlyphs (&font);
    enum LocaFormat loca = LOCA_OFFSET16;
    uint_fast16_t maxPoints = 0, maxContours = 0;
    pixels_t xMin = 0;
    if (opt.pos)
        positionGlyphs (&font, opt.pos, &xMin);
    if (opt.gpos)
        fillGposTable (&font);
    if (opt.gsub)
        fillGsubTable (&font);
    if (opt.cff)
        fillCFF (&font, opt.cff, opt.nameStrings);
    if (opt.truetype)
        fillTrueType (&font, &loca, &maxPoints, &maxContours);
    if (opt.blankOutline)
        fillBlankOutline (&font);
    if (opt.bitmap)
        fillBitmap (&font);
    fillHeadTable (&font, loca, xMin);
    fillHheaTable (&font, xMin);
    fillMaxpTable (&font, opt.cff, maxPoints, maxContours);
    fillOS2Table (&font);
    fillNameTable (&font, opt.nameStrings);
    fillHmtxTable (&font);
    fillCmapTable (&font);
    fillPostTable (&font);
    organizeTables (&font, opt.cff);
    writeFont (&font, opt.cff, opt.out);
    return EXIT_SUCCESS;
}
```

## 5.3 src/hex2otf.h File Reference

hex2otf.h - Header file for hex2otf.c

This graph shows which files directly or indirectly include this file:




## Data Structures

- struct NamePair

  Data structure for a font ID number and name character string.

## Macros

- #define UNIFONT_VERSION ”15.0.02”

  Current Unifont version.
- #define DEFAULT_ID0 ”Copyright © 1998-2022 Roman Czyborra, Paul Hardy, \Qianqian Fang, Andrew Miller, Johnnie Weaver, David Corbett, \Nils Moskopp, Rebecca Bettencourt, et al.”
- #define DEFAULT_ID1 ”Unifont”

  Default NameID 1 string (Font Family)
- #define DEFAULT_ID2 ”Regular”

  Default NameID 2 string (Font Subfamily)
- #define DEFAULT_ID5 ”Version ”UNIFONT_VERSION

  Default NameID 5 string (Version of the Name Table)
- #define DEFAULT_ID11 ”https://unifoundry.com/unifont/”

  Default NameID 11 string (Font Vendor URL)
- #define DEFAULT_ID13 ”Dual license: SIL Open Font License version 1.1, \and GNU GPL version 2 or later with the GNU Font Embedding Exception.”

  Default NameID 13 string (License Description)
- #define DEFAULT_ID14 ”http://unifoundry.com/LICENSE.txt, \https://scripts.sil.org/OFL”

  Default NameID 14 string (License Information URLs)
- #define NAMEPAIR(n) {(n), DEFAULT_ID##n}

  Macro to initialize name identifier codes to default values defined above.

## Typedefs

- typedef struct NamePair NamePair

  Data structure for a font ID number and name character string.

### Variables

- const NamePair defaultNames []
  Allocate array of NameID codes with default values.

## 5.3.1 Detailed Description

hex2otf.h - Header file for hex2otf.c

**Copyright**

Copyright © 2022 何志翔 (He Zhixiang)

**Author**

何志翔 (He Zhixiang)

Definition in file hex2otf.h.

## 5.3.2 Macro Definition Documentation

### 5.3.2.1 DEFAULT_ID0

#define DEFAULT_ID0 "Copyright © 1998-2022 Roman Czyborra, Paul Hardy, \Qianqian Fang, Andrew Miller, Johnnie Weaver, David Corbett, \Nils Moskopp, Rebecca Bettencourt, et al."

Define default strings for some TrueType font NameID strings.

```
NameID   Description
------   -----------
   0     Copyright Notice
   1     Font Family
   2     Font Subfamily
   5     Version of the Name Table
  11     URL of the Font Vendor
  13     License Description
  14     License Information URL
```

Default NameID 0 string (Copyright Notice)

Definition at line 53 of file hex2otf.h.

#### 5.3.2.2 DEFAULT_ID1

#define DEFAULT_ID1 "Unifont"

Default NameID 1 string (Font Family)

Definition at line 57 of file hex2otf.h.

#### 5.3.2.3 DEFAULT_ID11

#define DEFAULT_ID11 "https://unifoundry.com/unifont/"

Default NameID 11 string (Font Vendor URL)

Definition at line 64 of file hex2otf.h.

#### 5.3.2.4 DEFAULT_ID13

#define DEFAULT_ID13 "Dual license: SIL Open Font License version 1.1, \and GNU GPL version 2 or later with the GNU Font Embedding Exception."

Default NameID 13 string (License Description)

Definition at line 67 of file hex2otf.h.

#### 5.3.2.5 DEFAULT_ID14

#define DEFAULT_ID14 "http://unifoundry.com/LICENSE.txt, \https://scripts.sil.org/OFL"

Default NameID 14 string (License Information URLs)

Definition at line 71 of file hex2otf.h.

#### 5.3.2.6 DEFAULT_ID2

#define DEFAULT_ID2 "Regular"

Default NameID 2 string (Font Subfamily)

Definition at line 58 of file hex2otf.h.

#### 5.3.2.7 DEFAULT_ID5

#define DEFAULT_ID5 "Version "UNIFONT_VERSION

Default NameID 5 string (Version of the Name Table)

Definition at line 61 of file hex2otf.h.

#### 5.3.2.8 NAMEPAIR

```
#define NAMEPAIR(
                    n ) {(n), DEFAULT_ID##n}
```

Macro to initialize name identifier codes to default values defined above.

Definition at line 84 of file hex2otf.h.

#### 5.3.2.9 UNIFONT_VERSION

#define UNIFONT_VERSION "15.0.02"

Current Unifont version.

Definition at line 36 of file hex2otf.h.

### 5.3.3 Variable Documentation

#### 5.3.3.1 defaultNames

const NamePair defaultNames[]

Initial value:

```
=
{
    NAMEPAIR (0),
    NAMEPAIR (1),
    NAMEPAIR (2),
    NAMEPAIR (5),
    NAMEPAIR (11),
    NAMEPAIR (13),
    NAMEPAIR (14),
    {0, NULL}
}
```

Allocate array of NameID codes with default values.

This array contains the default values for several TrueType NameID strings, as defined above in this file. Strings are assigned using the NAMEPAIR macro defined above.

Definition at line 93 of file hex2otf.h.

## 5.4 hex2otf.h

Go to the documentation of this file.

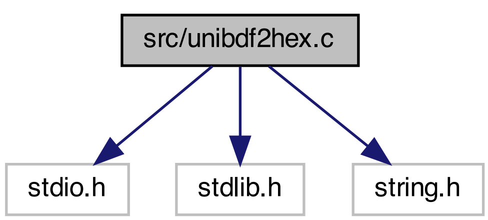
```
/**
   @file hex2otf.h

   @brief hex2otf.h - Header file for hex2otf.c

   @copyright Copyright © 2022 何志翔 (He Zhixiang)

   @author 何志翔 (He Zhixiang)
*/

/*
   LICENSE:

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License
   as published by the Free Software Foundation; either version 2
   of the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
   02110-1301, USA.

   NOTE: It is a violation of the license terms of this software
   to delete license and copyright information below if creating
   a font derived from Unifont glyphs.
*/
#ifndef _HEX2OTF_H_
#define _HEX2OTF_H_

#define UNIFONT_VERSION "15.0.02"   ///< Current Unifont version.

/**
   Define default strings for some TrueType font NameID strings.

      NameID   Description
      ------   -----------
         0     Copyright Notice
         1     Font Family
         2     Font Subfamily
         5     Version of the Name Table
        11     URL of the Font Vendor
        13     License Description
        14     License Information URL

   Default NameID 0 string (Copyright Notice)
*/
#define DEFAULT_ID0 "Copyright © 1998-2022 Roman Czyborra, Paul Hardy, \
Qianqian Fang, Andrew Miller, Johnnie Weaver, David Corbett, \
Nils Moskopp, Rebecca Bettencourt, et al."

#define DEFAULT_ID1 "Unifont"   ///< Default NameID 1 string (Font Family)
#define DEFAULT_ID2 "Regular"   ///< Default NameID 2 string (Font Subfamily)

/// Default NameID 5 string (Version of the Name Table)
#define DEFAULT_ID5 "Version "UNIFONT_VERSION

/// Default NameID 11 string (Font Vendor URL)
#define DEFAULT_ID11 "https://unifoundry.com/unifont/"

/// Default NameID 13 string (License Description)
#define DEFAULT_ID13 "Dual license: SIL Open Font License version 1.1, \
and GNU GPL version 2 or later with the GNU Font Embedding Exception."

/// Default NameID 14 string (License Information URLs)
#define DEFAULT_ID14 "http://unifoundry.com/LICENSE.txt, \
https://scripts.sil.org/OFL"

/**
   @brief Data structure for a font ID number and name character string.
*/
```

```
typedef struct NamePair
{
    int id;
    const char *str;
} NamePair;

/// Macro to initialize name identifier codes to default values defined above.
#define NAMEPAIR(n) {(n), DEFAULT_ID##n}

/**
    @brief Allocate array of NameID codes with default values.

    This array contains the default values for several TrueType NameID
    strings, as defined above in this file.  Strings are assigned using
    the NAMEPAIR macro defined above.
*/
const NamePair defaultNames[] =
{
    NAMEPAIR (0),  // Copyright notice; required (used in CFF)
    NAMEPAIR (1),  // Font family; required (used in CFF)
    NAMEPAIR (2),  // Font subfamily
    NAMEPAIR (5),  // Version of the name table
    NAMEPAIR (11), // URL of font vendor
    NAMEPAIR (13), // License description
    NAMEPAIR (14), // License information URL
    {0, NULL}      // Sentinel
};

#undef NAMEPAIR

#endif
```

## 5.5 src/unibdf2hex.c File Reference

unibdf2hex - Convert a BDF file into a unifont.hex file

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Include dependency graph for unibdf2hex.c:




### Macros

- #define UNISTART 0x3400

    First Unicode code point to examine.

- #define UNISTOP 0x4DBF

  Last Unicode code point to examine.
- #define MAXBUF 256

  Maximum allowable input file line length - 1.

## Functions

- int main ()

  The main function.

### 5.5.1 Detailed Description

unibdf2hex - Convert a BDF file into a unifont.hex file

Author

Paul Hardy, January 2008

Copyright

Copyright (C) 2008, 2013 Paul Hardy

Note: currently this has hard-coded code points for glyphs extracted from Wen Quan Yi to create the Unifont source file "wqy.hex".

Definition in file unibdf2hex.c.

### 5.5.2 Macro Definition Documentation

#### 5.5.2.1 MAXBUF

#define MAXBUF 256

Maximum allowable input file line length - 1.

Definition at line 37 of file unibdf2hex.c.

#### 5.5.2.2 UNISTART

#define UNISTART 0x3400

First Unicode code point to examine.

Definition at line 34 of file unibdf2hex.c.

#### 5.5.2.3 UNISTOP

#define UNISTOP 0x4DBF

Last Unicode code point to examine.

Definition at line 35 of file unibdf2hex.c.

### 5.5.3 Function Documentation

#### 5.5.3.1 main()

int main ( )

The main function.

**Returns**

Exit status is always 0 (successful termination).

Definition at line 46 of file unibdf2hex.c.

```
{
   int i;
   int digitsout;  /* how many hex digits we output in a bitmap */
   int thispoint;
   char inbuf[MAXBUF];
   int bbxx, bbxy, bbxxoff, bbxyoff;

   int descent=4; /* font descent wrt baseline */
   int startrow;  /* row to start glyph         */
   unsigned rowout;

   while (fgets (inbuf, MAXBUF - 1, stdin) != NULL) {
      if (strncmp (inbuf, "ENCODING ", 9) == 0) {
         sscanf (&inbuf[9], "%d", &thispoint); /* get code point */
         /*
            If we want this code point, get the BBX (bounding box) and
            BITMAP information.
         */
         if ((thispoint >= 0x2E80 && thispoint <= 0x2EFF) || // CJK Radicals Supplement
             (thispoint >= 0x2F00 && thispoint <= 0x2FDF) || // Kangxi Radicals
             (thispoint >= 0x2FF0 && thispoint <= 0x2FFF) || // Ideographic Description Characters
             (thispoint >= 0x3001 && thispoint <= 0x303F) || // CJK Symbols and Punctuation (U+3000 is a space)
             (thispoint >= 0x3100 && thispoint <= 0x312F) || // Bopomofo
```

```
               (thispoint >= 0x31A0 && thispoint <= 0x31BF) || // Bopomofo extend
               (thispoint >= 0x31C0 && thispoint <= 0x31EF) || // CJK Strokes
               (thispoint >= 0x3400 && thispoint <= 0x4DBF) || // CJK Unified Ideographs Extension A
               (thispoint >= 0x4E00 && thispoint <= 0x9FCF) || // CJK Unified Ideographs
               (thispoint >= 0xF900 && thispoint <= 0xFAFF))   // CJK Compatibility Ideographs
            {
            while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
                  strncmp (inbuf, "BBX ", 4) != 0); /* find bounding box */

            sscanf (&inbuf[4], "%d %d %d %d", &bbxx, &bbxy, &bbxxoff, &bbxyoff);
            while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
                  strncmp (inbuf, "BITMAP", 6) != 0); /* find bitmap start */
            fprintf (stdout, "%04X:", thispoint);
            digitsout = 0;
            /* Print initial blank rows */
            startrow = descent + bbxyoff + bbxy;

            /* Force everything to 16 pixels wide */
            for (i = 16; i > startrow; i--) {
               fprintf (stdout,"0000");
               digitsout += 4;
            }
            while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
                  strncmp (inbuf, "END", 3) != 0) { /* copy bitmap until END */
               sscanf (inbuf, "%X", &rowout);
               /* Now force glyph to a 16x16 grid even if they'd fit in 8x16 */
               if (bbxx <= 8) rowout «= 8;  /* shift left for 16x16 glyph */
               rowout »= bbxxoff;
               fprintf (stdout, "%04X", rowout);
               digitsout += 4;
            }

            /* Pad for 16x16 glyph */
            while (digitsout < 64) {
               fprintf (stdout,"0000");
               digitsout += 4;
            }
            fprintf (stdout,"\n");
         }
      }
   }
   exit (0);
}
```

## 5.6 unibdf2hex.c

Go to the documentation of this file.

```
/**
   @file unibdf2hex.c

   @brief unibdf2hex - Convert a BDF file into a unifont.hex file

   @author Paul Hardy, January 2008

   @copyright Copyright (C) 2008, 2013 Paul Hardy

   Note: currently this has hard-coded code points for glyphs extracted
   from Wen Quan Yi to create the Unifont source file "wqy.hex".
*/
/*
   LICENSE:

      This program is free software: you can redistribute it and/or modify
      it under the terms of the GNU General Public License as published by
      the Free Software Foundation, either version 2 of the License, or
      (at your option) any later version.

      This program is distributed in the hope that it will be useful,
      but WITHOUT ANY WARRANTY; without even the implied warranty of
      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
      GNU General Public License for more details.

      You should have received a copy of the GNU General Public License
      along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
```

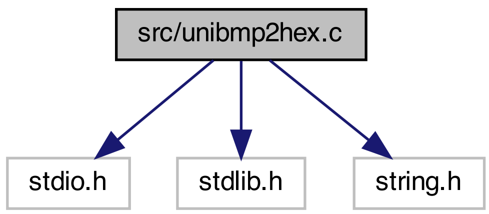
```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define UNISTART 0x3400 ///< First Unicode code point to examine
#define UNISTOP 0x4DBF  ///< Last Unicode code point to examine

#define MAXBUF 256  ///< Maximum allowable input file line length - 1


/**
   @brief The main function.

   @return Exit status is always 0 (successful termination).
*/
int
main()
{
   int i;
   int digitsout;  /* how many hex digits we output in a bitmap */
   int thispoint;
   char inbuf[MAXBUF];
   int bbxx, bbxy, bbxxoff, bbxyoff;

   int descent=4; /* font descent wrt baseline */
   int startrow;  /* row to start glyph         */
   unsigned rowout;

   while (fgets (inbuf, MAXBUF - 1, stdin) != NULL) {
      if (strncmp (inbuf, ”ENCODING ”, 9) == 0) {
         sscanf (&inbuf[9], ”%d”, &thispoint); /* get code point */
         /*
            If we want this code point, get the BBX (bounding box) and
            BITMAP information.
         */
         if ((thispoint >= 0x2E80 && thispoint <= 0x2EFF) || // CJK Radicals Supplement
             (thispoint >= 0x2F00 && thispoint <= 0x2FDF) || // Kangxi Radicals
             (thispoint >= 0x2FF0 && thispoint <= 0x2FFF) || // Ideographic Description Characters
             (thispoint >= 0x3001 && thispoint <= 0x303F) || // CJK Symbols and Punctuation (U+3000 is a space)
             (thispoint >= 0x3100 && thispoint <= 0x312F) || // Bopomofo
             (thispoint >= 0x31A0 && thispoint <= 0x31BF) || // Bopomofo extend
             (thispoint >= 0x31C0 && thispoint <= 0x31EF) || // CJK Strokes
             (thispoint >= 0x3400 && thispoint <= 0x4DBF) || // CJK Unified Ideographs Extension A
             (thispoint >= 0x4E00 && thispoint <= 0x9FCF) || // CJK Unified Ideographs
             (thispoint >= 0xF900 && thispoint <= 0xFAFF))   // CJK Compatibility Ideographs
            {
            while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
                   strncmp (inbuf, ”BBX ”, 4) != 0); /* find bounding box */

            sscanf (&inbuf[4], ”%d %d %d %d”, &bbxx, &bbxy, &bbxxoff, &bbxyoff);
            while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
                   strncmp (inbuf, ”BITMAP”, 6) != 0); /* find bitmap start */
            fprintf (stdout, ”%04X:”, thispoint);
            digitsout = 0;
            /* Print initial blank rows */
            startrow = descent + bbxyoff + bbxy;

            /* Force everything to 16 pixels wide */
            for (i = 16; i > startrow; i--) {
               fprintf (stdout,”0000”);
               digitsout += 4;
            }
            while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
                   strncmp (inbuf, ”END”, 3) != 0) { /* copy bitmap until END */
               sscanf (inbuf, ”%X”, &rowout);
               /* Now force glyph to a 16x16 grid even if they’d fit in 8x16 */
               if (bbxx <= 8) rowout «= 8;  /* shift left for 16x16 glyph */
               rowout »= bbxxoff;
               fprintf (stdout, ”%04X”, rowout);
               digitsout += 4;
            }

            /* Pad for 16x16 glyph */
            while (digitsout < 64) {
               fprintf (stdout,”0000”);
               digitsout += 4;
            }
            fprintf (stdout,”\n”);
         }
      }
```

```
    }
    exit (0);
}
```

## 5.7 src/unibmp2hex.c File Reference

unibmp2hex - Turn a .bmp or .wbmp glyph matrix into a GNU Unifont hex glyph set of 256 characters

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Include dependency graph for unibmp2hex.c:




### Macros

- #define MAXBUF 256

    Maximum input file line length - 1.

### Functions

- int main (int argc, char ∗argv[ ])

    The main function.

### Variables

- unsigned hexdigit [16][4]

    32 bit representation of 16x8 0..F bitmap
- unsigned uniplane =0

    Unicode plane number, 0..0xff ff ff.
- unsigned planeset =0

    =1: use plane specified with -p parameter
- unsigned flip =0

=1 if we're transposing glyph matrix

- unsigned forcewide =0

  =1 to set each glyph to 16 pixels wide
- unsigned unidigit [6][4]
- struct {
  char filetype [2]
  int file_size
  int image_offset
  int info_size
  int width
  int height
  int nplanes
  int bits_per_pixel
  int compression
  int image_size
  int x_ppm
  int y_ppm
  int ncolors
  int important_colors
  } bmp_header
- unsigned char color_table [256][4]

### 5.7.1 Detailed Description

unibmp2hex - Turn a .bmp or .wbmp glyph matrix into a GNU Unifont hex glyph set of 256 characters

Author

Paul Hardy, unifoundry <at> unifoundry.com, December 2007

Copyright

Copyright (C) 2007, 2008, 2013, 2017, 2019, 2022 Paul Hardy

Synopsis: unibmp2hex [-iin_file.bmp] [-oout_file.hex] [-phex_page_num] [-w]

Definition in file unibmp2hex.c.

### 5.7.2 Macro Definition Documentation

#### 5.7.2.1 MAXBUF

#define MAXBUF 256

Maximum input file line length - 1.

Definition at line 104 of file unibmp2hex.c.

## 5.7.3 Function Documentation

### 5.7.3.1 main()

```
int main (
        int argc,
        char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status 0.

Definition at line 149 of file unibmp2hex.c.

```
00150 {
00151
00152    int i, j, k;                /* loop variables                       */
00153    unsigned char inchar;       /* temporary input character */
00154    char header[MAXBUF];        /* input buffer for bitmap file header */
00155    int wbmp=0; /* =0 for Windows Bitmap (.bmp); 1 for Wireless Bitmap (.wbmp) */
00156    int fatal; /* =1 if a fatal error occurred */
00157    int match; /* =1 if we're still matching a pattern, 0 if no match */
00158    int empty1, empty2; /* =1 if bytes tested are all zeroes */
00159    unsigned char thischar1[16], thischar2[16]; /* bytes of hex char */
00160    unsigned char thischar0[16], thischar3[16]; /* bytes for quadruple-width */
00161    int thisrow; /* index to point into thischar1[] and thischar2[] */
00162    int tmpsum;  /* temporary sum to see if a character is blank */
00163    unsigned this_pixel;  /* color of one pixel, if > 1 bit per pixel */
00164    unsigned next_pixels; /* pending group of 8 pixels being read */
00165    unsigned color_mask = 0x00;  /* to invert monochrome bitmap, set to 0xFF */
00166
00167    unsigned char bitmap[17*32][18*32/8]; /* final bitmap */
00168    /* For wide array:
00169          0 = don't force glyph to double-width;
00170          1 = force glyph to double-width;
```

```
      4 = force glyph to quadruple-width.
*/
char wide[0x200000]={0x200000 * 0};

char *infile="", *outfile="";  /* names of input and output files */
FILE *infp, *outfp;       /* file pointers of input and output files */

if (argc > 1) {
   for (i = 1; i < argc; i++) {
      if (argv[i][0] == '-') {  /* this is an option argument */
         switch (argv[i][1]) {
            case 'i':  /* name of input file */
               infile = &argv[i][2];
               break;
            case 'o':  /* name of output file */
               outfile = &argv[i][2];
               break;
            case 'p':  /* specify a Unicode plane */
               sscanf (&argv[i][2], "%x", &uniplane); /* Get Unicode plane */
               planeset = 1; /* Use specified range, not what's in bitmap */
               break;
            case 'w': /* force wide (16 pixels) for each glyph */
               forcewide = 1;
               break;
            default:   /* if unrecognized option, print list and exit */
               fprintf (stderr, "\nSyntax:\n\n");
               fprintf (stderr, "   %s -p<Unicode_Page> ", argv[0]);
               fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
               fprintf (stderr, "   -w specifies .wbmp output instead of ");
               fprintf (stderr, "default Windows .bmp output.\n\n");
               fprintf (stderr, "   -p is followed by 1 to 6 ");
               fprintf (stderr, "Unicode plane hex digits ");
               fprintf (stderr, "(default is Page 0).\n\n");
               fprintf (stderr, "\nExample:\n\n");
               fprintf (stderr, "   %s -p83 -iunifont.hex -ou83.bmp\n\n\n",
                      argv[0]);
               exit (1);
         }
      }
   }
}
/*
   Make sure we can open any I/O files that were specified before
   doing anything else.
*/
if (strlen (infile) > 0) {
   if ((infp = fopen (infile, "r")) == NULL) {
      fprintf (stderr, "Error: can't open %s for input.\n", infile);
      exit (1);
   }
}
else {
   infp = stdin;
}
if (strlen (outfile) > 0) {
   if ((outfp = fopen (outfile, "w")) == NULL) {
      fprintf (stderr, "Error: can't open %s for output.\n", outfile);
      exit (1);
   }
}
else {
   outfp = stdout;
}
/*
   Initialize selected code points for double width (16x16).
   Double-width is forced in cases where a glyph (usually a combining
   glyph) only occupies the left-hand side of a 16x16 grid, but must
   be rendered as double-width to appear properly with other glyphs
   in a given script.  If additions were made to a script after
   Unicode 5.0, the Unicode version is given in parentheses after
   the script name.
*/
for (i = 0x0700; i <= 0x074F; i++) wide[i] = 1; /* Syriac                 */
for (i = 0x0800; i <= 0x083F; i++) wide[i] = 1; /* Samaritan (5.2)        */
for (i = 0x0900; i <= 0x0DFF; i++) wide[i] = 1; /* Indic                  */
for (i = 0x1000; i <= 0x109F; i++) wide[i] = 1; /* Myanmar                */
for (i = 0x1100; i <= 0x11FF; i++) wide[i] = 1; /* Hangul Jamo            */
for (i = 0x1400; i <= 0x167F; i++) wide[i] = 1; /* Canadian Aboriginal    */
for (i = 0x1700; i <= 0x171F; i++) wide[i] = 1; /* Tagalog                */
for (i = 0x1720; i <= 0x173F; i++) wide[i] = 1; /* Hanunoo                */
for (i = 0x1740; i <= 0x175F; i++) wide[i] = 1; /* Buhid                  */
```

```
    for (i = 0x1760; i <= 0x177F; i++) wide[i] = 1; /* Tagbanwa              */
    for (i = 0x1780; i <= 0x17FF; i++) wide[i] = 1; /* Khmer                 */
    for (i = 0x18B0; i <= 0x18FF; i++) wide[i] = 1; /* Ext. Can. Aboriginal  */
    for (i = 0x1800; i <= 0x18AF; i++) wide[i] = 1; /* Mongolian             */
    for (i = 0x1900; i <= 0x194F; i++) wide[i] = 1; /* Limbu                 */
// for (i = 0x1980; i <= 0x19DF; i++) wide[i] = 1; /* New Tai Lue           */
    for (i = 0x1A00; i <= 0x1A1F; i++) wide[i] = 1; /* Buginese              */
    for (i = 0x1A20; i <= 0x1AAF; i++) wide[i] = 1; /* Tai Tham (5.2)        */
    for (i = 0x1B00; i <= 0x1B7F; i++) wide[i] = 1; /* Balinese              */
    for (i = 0x1B80; i <= 0x1BBF; i++) wide[i] = 1; /* Sundanese (5.1)       */
    for (i = 0x1BC0; i <= 0x1BFF; i++) wide[i] = 1; /* Batak (6.0)           */
    for (i = 0x1C00; i <= 0x1C4F; i++) wide[i] = 1; /* Lepcha (5.1)          */
    for (i = 0x1CC0; i <= 0x1CCF; i++) wide[i] = 1; /* Sundanese Supplement  */
    for (i = 0x1CD0; i <= 0x1CFF; i++) wide[i] = 1; /* Vedic Extensions (5.2) */
    wide[0x2329] = wide[0x232A] = 1; /* Left- & Right-pointing Angle Brackets */
    for (i = 0x2E80; i <= 0xA4CF; i++) wide[i] = 1; /* CJK                   */
// for (i = 0x9FD8; i <= 0x9FE9; i++) wide[i] = 4; /* CJK quadruple-width   */
    for (i = 0xA900; i <= 0xA92F; i++) wide[i] = 1; /* Kayah Li (5.1)        */
    for (i = 0xA930; i <= 0xA95F; i++) wide[i] = 1; /* Rejang (5.1)          */
    for (i = 0xA960; i <= 0xA97F; i++) wide[i] = 1; /* Hangul Jamo Extended-A */
    for (i = 0xA980; i <= 0xA9DF; i++) wide[i] = 1; /* Javanese (5.2)        */
    for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham (5.1)            */
    for (i = 0xA9E0; i <= 0xA9FF; i++) wide[i] = 1; /* Myanmar Extended-B    */
    for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham                  */
    for (i = 0xAA60; i <= 0xAA7F; i++) wide[i] = 1; /* Myanmar Extended-A    */
    for (i = 0xAAE0; i <= 0xAAFF; i++) wide[i] = 1; /* Meetei Mayek Ext (6.0) */
    for (i = 0xABC0; i <= 0xABFF; i++) wide[i] = 1; /* Meetei Mayek (5.2)    */
    for (i = 0xAC00; i <= 0xD7AF; i++) wide[i] = 1; /* Hangul Syllables      */
    for (i = 0xD7B0; i <= 0xD7FF; i++) wide[i] = 1; /* Hangul Jamo Extended-B */
    for (i = 0xF900; i <= 0xFAFF; i++) wide[i] = 1; /* CJK Compatibility     */
    for (i = 0xFE10; i <= 0xFE1F; i++) wide[i] = 1; /* Vertical Forms        */
    for (i = 0xFE30; i <= 0xFE60; i++) wide[i] = 1; /* CJK Compatibility Forms*/
    for (i = 0xFFE0; i <= 0xFFE6; i++) wide[i] = 1; /* CJK Compatibility Forms*/

    wide[0x303F] = 0; /* CJK half-space fill */

    /* Supplemental Multilingual Plane (Plane 01) */
    for (i = 0x010A00; i <= 0x010A5F; i++) wide[i] = 1; /* Kharoshthi         */
    for (i = 0x011000; i <= 0x01107F; i++) wide[i] = 1; /* Brahmi             */
    for (i = 0x011080; i <= 0x0110CF; i++) wide[i] = 1; /* Kaithi             */
    for (i = 0x011100; i <= 0x01114F; i++) wide[i] = 1; /* Chakma             */
    for (i = 0x011180; i <= 0x0111DF; i++) wide[i] = 1; /* Sharada            */
    for (i = 0x011200; i <= 0x01124F; i++) wide[i] = 1; /* Khojki             */
    for (i = 0x0112B0; i <= 0x0112FF; i++) wide[i] = 1; /* Khudawadi          */
    for (i = 0x011300; i <= 0x01137F; i++) wide[i] = 1; /* Grantha            */
    for (i = 0x011400; i <= 0x01147F; i++) wide[i] = 1; /* Newa               */
    for (i = 0x011480; i <= 0x0114DF; i++) wide[i] = 1; /* Tirhuta            */
    for (i = 0x011580; i <= 0x0115FF; i++) wide[i] = 1; /* Siddham            */
    for (i = 0x011600; i <= 0x01165F; i++) wide[i] = 1; /* Modi               */
    for (i = 0x011660; i <= 0x01167F; i++) wide[i] = 1; /* Mongolian Suppl.   */
    for (i = 0x011680; i <= 0x0116CF; i++) wide[i] = 1; /* Takri              */
    for (i = 0x011700; i <= 0x01173F; i++) wide[i] = 1; /* Ahom               */
    for (i = 0x011800; i <= 0x01184F; i++) wide[i] = 1; /* Dogra              */
    for (i = 0x011900; i <= 0x01195F; i++) wide[i] = 1; /* Dives Akuru        */
    for (i = 0x0119A0; i <= 0x0119FF; i++) wide[i] = 1; /* Nandinagari        */
    for (i = 0x011A00; i <= 0x011A4F; i++) wide[i] = 1; /* Zanabazar Square   */
    for (i = 0x011A50; i <= 0x011AAF; i++) wide[i] = 1; /* Soyombo            */
    for (i = 0x011B00; i <= 0x011B5F; i++) wide[i] = 1; /* Devanagari Extended-A*/
    for (i = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi               */
    for (i = 0x011C00; i <= 0x011C6F; i++) wide[i] = 1; /* Bhaiksuki          */
    for (i = 0x011C70; i <= 0x011CBF; i++) wide[i] = 1; /* Marchen            */
    for (i = 0x011D00; i <= 0x011D5F; i++) wide[i] = 1; /* Masaram Gondi      */
    for (i = 0x011EE0; i <= 0x011EFF; i++) wide[i] = 1; /* Makasar            */
    for (i = 0x012F90; i <= 0x012FFF; i++) wide[i] = 1; /* Cypro-Minoan       */
    /* Make Bassa Vah all single width or all double width */
    for (i = 0x016AD0; i <= 0x016AFF; i++) wide[i] = 1; /* Bassa Vah          */
    for (i = 0x016B00; i <= 0x016B8F; i++) wide[i] = 1; /* Pahawh Hmong       */
    for (i = 0x016F00; i <= 0x016F9F; i++) wide[i] = 1; /* Miao               */
    for (i = 0x016FE0; i <= 0x016FFF; i++) wide[i] = 1; /* Ideograph Sym/Punct*/
    for (i = 0x017000; i <= 0x0187FF; i++) wide[i] = 1; /* Tangut             */
    for (i = 0x018800; i <= 0x018AFF; i++) wide[i] = 1; /* Tangut Components  */
    for (i = 0x01AFF0; i <= 0x01AFFF; i++) wide[i] = 1; /* Kana Extended-B    */
    for (i = 0x01B000; i <= 0x01B0FF; i++) wide[i] = 1; /* Kana Supplement    */
    for (i = 0x01B100; i <= 0x01B12F; i++) wide[i] = 1; /* Kana Extended-A    */
    for (i = 0x01B170; i <= 0x01B2FF; i++) wide[i] = 1; /* Nushu              */
    for (i = 0x01CF00; i <= 0x01CFCF; i++) wide[i] = 1; /* Znamenny Musical   */
    for (i = 0x01D100; i <= 0x01D1FF; i++) wide[i] = 1; /* Musical Symbols    */
    for (i = 0x01D800; i <= 0x01DAAF; i++) wide[i] = 1; /* Sutton SignWriting */
    for (i = 0x01E2C0; i <= 0x01E2FF; i++) wide[i] = 1; /* Wancho             */
    for (i = 0x01E800; i <= 0x01E8DF; i++) wide[i] = 1; /* Mende Kikakui      */
    for (i = 0x01F200; i <= 0x01F2FF; i++) wide[i] = 1; /* Encl Ideograp Suppl*/
```

```
wide[0x01F5E7] = 1;                    /* Three Rays Right   */

/*
   Determine whether or not the file is a Microsoft Windows Bitmap file.
   If it starts with ’B’, ’M’, assume it’s a Windows Bitmap file.
   Otherwise, assume it’s a Wireless Bitmap file.

   WARNING: There isn’t much in the way of error checking here --
   if you give it a file that wasn’t first created by hex2bmp.c,
   all bets are off.
*/
fatal = 0;  /* assume everything is okay with reading input file */
if ((header[0] = fgetc (infp)) != EOF) {
   if ((header[1] = fgetc (infp)) != EOF) {
      if (header[0] == ’B’ && header[1] == ’M’) {
         wbmp = 0; /* Not a Wireless Bitmap -- it’s a Windows Bitmap */
      }
      else {
         wbmp = 1; /* Assume it’s a Wireless Bitmap */
      }
   }
   else
      fatal = 1;
}
else
   fatal = 1;

if (fatal) {
   fprintf (stderr, ”Fatal error; end of input file.\n\n”);
   exit (1);
}
/*
   If this is a Wireless Bitmap (.wbmp) format file,
   skip the header and point to the start of the bitmap itself.
*/
if (wbmp) {
   for (i=2; i<6; i++)
      header[i] = fgetc (infp);
   /*
      Now read the bitmap.
   */
   for (i=0; i < 32*17; i++) {
      for (j=0; j < 32*18/8; j++) {
         inchar = fgetc (infp);
         bitmap[i][j] = ~inchar;  /* invert bits for proper color */
      }
   }
}
/*
   Otherwise, treat this as a Windows Bitmap file, because we checked
   that it began with ”BM”.  Save the header contents for future use.
   Expect a 14 byte standard BITMAPFILEHEADER format header followed
   by a 40 byte standard BITMAPINFOHEADER Device Independent Bitmap
   header, with data stored in little-endian format.
*/
else {
   for (i = 2; i < 54; i++)
      header[i] = fgetc (infp);

   bmp_header.filetype[0] = ’B’;
   bmp_header.filetype[1] = ’M’;

   bmp_header.file_size =
       (header[2] & 0xFF)        | ((header[3] & 0xFF) «  8) |
      ((header[4] & 0xFF) « 16) | ((header[5] & 0xFF) « 24);

   /* header bytes 6..9 are reserved */

   bmp_header.image_offset =
       (header[10] & 0xFF)        | ((header[11] & 0xFF) «  8) |
      ((header[12] & 0xFF) « 16) | ((header[13] & 0xFF) « 24);

   bmp_header.info_size =
       (header[14] & 0xFF)        | ((header[15] & 0xFF) «  8) |
      ((header[16] & 0xFF) « 16) | ((header[17] & 0xFF) « 24);

   bmp_header.width =
       (header[18] & 0xFF)        | ((header[19] & 0xFF) «  8) |
      ((header[20] & 0xFF) « 16) | ((header[21] & 0xFF) « 24);

   bmp_header.height =
```

```
        (header[22] & 0xFF)        | ((header[23] & 0xFF) «  8) |
       ((header[24] & 0xFF) « 16) | ((header[25] & 0xFF) « 24);

    bmp_header.nplanes =
        (header[26] & 0xFF)        | ((header[27] & 0xFF) «  8);

    bmp_header.bits_per_pixel =
        (header[28] & 0xFF)        | ((header[29] & 0xFF) «  8);

    bmp_header.compression =
        (header[30] & 0xFF)        | ((header[31] & 0xFF) «  8) |
       ((header[32] & 0xFF) « 16) | ((header[33] & 0xFF) « 24);

    bmp_header.image_size =
        (header[34] & 0xFF)        | ((header[35] & 0xFF) «  8) |
       ((header[36] & 0xFF) « 16) | ((header[37] & 0xFF) « 24);

    bmp_header.x_ppm =
        (header[38] & 0xFF)        | ((header[39] & 0xFF) «  8) |
       ((header[40] & 0xFF) « 16) | ((header[41] & 0xFF) « 24);

    bmp_header.y_ppm =
        (header[42] & 0xFF)        | ((header[43] & 0xFF) «  8) |
       ((header[44] & 0xFF) « 16) | ((header[45] & 0xFF) « 24);

    bmp_header.ncolors =
        (header[46] & 0xFF)        | ((header[47] & 0xFF) «  8) |
       ((header[48] & 0xFF) « 16) | ((header[49] & 0xFF) « 24);

    bmp_header.important_colors =
        (header[50] & 0xFF)        | ((header[51] & 0xFF) «  8) |
       ((header[52] & 0xFF) « 16) | ((header[53] & 0xFF) « 24);

    if (bmp_header.ncolors == 0)
       bmp_header.ncolors = 1 « bmp_header.bits_per_pixel;

    /* If a Color Table exists, read it */
    if (bmp_header.ncolors > 0 && bmp_header.bits_per_pixel <= 8) {
       for (i = 0; i < bmp_header.ncolors; i++) {
          color_table[i][0] = fgetc (infp);  /* Red   */
          color_table[i][1] = fgetc (infp);  /* Green */
          color_table[i][2] = fgetc (infp);  /* Blue  */
          color_table[i][3] = fgetc (infp);  /* Alpha */
       }
       /*
          Determine from the first color table entry whether we
          are inverting the resulting bitmap image.
       */
       if ( (color_table[0][0] + color_table[0][1] + color_table[0][2])
            < (3 * 128) ) {
          color_mask = 0xFF;
       }
    }

#ifdef DEBUG

    /*
       Print header info for possibly adding support for
       additional file formats in the future, to determine
       how the bitmap is encoded.
    */
    fprintf (stderr, "Filetype: '%c%c'\n",
                     bmp_header.filetype[0], bmp_header.filetype[1]);
    fprintf (stderr, "File Size: %d\n", bmp_header.file_size);
    fprintf (stderr, "Image Offset: %d\n", bmp_header.image_offset);
    fprintf (stderr, "Info Header Size: %d\n", bmp_header.info_size);
    fprintf (stderr, "Image Width: %d\n", bmp_header.width);
    fprintf (stderr, "Image Height: %d\n", bmp_header.height);
    fprintf (stderr, "Number of Planes: %d\n", bmp_header.nplanes);
    fprintf (stderr, "Bits per Pixel: %d\n", bmp_header.bits_per_pixel);
    fprintf (stderr, "Compression Method: %d\n", bmp_header.compression);
    fprintf (stderr, "Image Size: %d\n", bmp_header.image_size);
    fprintf (stderr, "X Pixels per Meter: %d\n", bmp_header.x_ppm);
    fprintf (stderr, "Y Pixels per Meter: %d\n", bmp_header.y_ppm);
    fprintf (stderr, "Number of Colors: %d\n", bmp_header.ncolors);
    fprintf (stderr, "Important Colors: %d\n", bmp_header.important_colors);

#endif

    /*
       Now read the bitmap.
```

```
      */
      for (i = 32*17-1; i >= 0; i--) {
         for (j=0; j < 32*18/8; j++) {
            next_pixels = 0x00;  /* initialize next group of 8 pixels */
            /* Read a monochrome image -- the original case */
            if (bmp_header.bits_per_pixel == 1) {
               next_pixels = fgetc (infp);
            }
            /* Read a 32 bit per pixel RGB image; convert to monochrome */
            else if ( bmp_header.bits_per_pixel == 24 ||
                      bmp_header.bits_per_pixel == 32) {
               next_pixels = 0;
               for (k = 0; k < 8; k++) {  /* get next 8 pixels */
                  this_pixel = (fgetc (infp) & 0xFF) +
                               (fgetc (infp) & 0xFF) +
                               (fgetc (infp) & 0xFF);

                  if (bmp_header.bits_per_pixel == 32) {
                     (void) fgetc (infp);  /* ignore alpha value */
                  }

                  /* convert RGB color space to monochrome */
                  if (this_pixel >= (128 * 3))
                     this_pixel = 0;
                  else
                     this_pixel = 1;

                  /* shift next pixel color into place for 8 pixels total */
                  next_pixels = (next_pixels « 1) | this_pixel;
               }
            }
            if (bmp_header.height < 0) {  /* Bitmap drawn top to bottom */
               bitmap [(32*17-1) - i] [j] = next_pixels;
            }
            else {  /* Bitmap drawn bottom to top */
               bitmap [i][j] = next_pixels;
            }
         }
      }

      /*
         If any bits are set in color_mask, apply it to
         entire bitmap to invert black <--> white.
      */
      if (color_mask != 0x00) {
         for (i = 32*17-1; i >= 0; i--) {
            for (j=0; j < 32*18/8; j++) {
               bitmap [i][j] ^= color_mask;
            }
         }
      }

   }

   /*
      We've read the entire file.  Now close the input file pointer.
   */
   fclose (infp);
  /*
      We now have the header portion in the header[] array,
      and have the bitmap portion from top-to-bottom in the bitmap[] array.
   */
   /*
      If no Unicode range (U+nnnnnn00 through U+nnnnnnFF) was specified
      with a -p parameter, determine the range from the digits in the
      bitmap itself.

      Store bitmaps for the hex digit patterns that this file uses.
   */
   if (!planeset) {  /* If Unicode range not specified with -p parameter */
      for (i = 0x0; i <= 0xF; i++) {  /* hex digit pattern we're storing */
         for (j = 0; j < 4; j++) {
            hexdigit[i][j]   =
               ((unsigned)bitmap[32 * (i+1) + 4 * j + 8    ][6] « 24 ) |
               ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 1][6] « 16 ) |
               ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 2][6] «  8 ) |
               ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 3][6]       );
         }
      }
      /*
         Read the Unicode plane digits into arrays for comparison, to
```

```
         determine the upper four hex digits of the glyph addresses.
      */
      for (i = 0; i < 4; i++) {
         for (j = 0; j < 4; j++) {
            unidigit[i][j] =
               ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 1][i + 3] « 24 ) |
               ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 2][i + 3] « 16 ) |
               ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 3][i + 3] «  8 ) |
               ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 4][i + 3]       );
         }
      }

      tmpsum = 0;
      for (i = 4; i < 6; i++) {
         for (j = 0; j < 4; j++) {
            unidigit[i][j] =
               ((unsigned)bitmap[32 * 1 + 4 * j + 8    ][i] « 24 ) |
               ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 1][i] « 16 ) |
               ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 2][i] «  8 ) |
               ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 3][i]       );
            tmpsum |= unidigit[i][j];
         }
      }
      if (tmpsum == 0) {  /* the glyph matrix is transposed */
         flip = 1;  /* note transposed order for processing glyphs in matrix */
         /*
            Get 5th and 6th hex digits by shifting first column header left by
            1.5 columns, thereby shifting the hex digit right after the leading
            "U+nnnn" page number.
         */
         for (i = 0x08; i < 0x18; i++) {
            bitmap[i][7] = (bitmap[i][8] « 4) | ((bitmap[i][ 9] » 4) & 0xf);
            bitmap[i][8] = (bitmap[i][9] « 4) | ((bitmap[i][10] » 4) & 0xf);
         }
         for (i = 4; i < 6; i++) {
            for (j = 0; j < 4; j++) {
               unidigit[i][j] =
                  ((unsigned)bitmap[4 * j + 8 + 1][i + 3] « 24 ) |
                  ((unsigned)bitmap[4 * j + 8 + 2][i + 3] « 16 ) |
                  ((unsigned)bitmap[4 * j + 8 + 3][i + 3] «  8 ) |
                  ((unsigned)bitmap[4 * j + 8 + 4][i + 3]       );
            }
         }
      }

      /*
         Now determine the Unicode plane by comparing unidigit[0..5] to
         the hexdigit[0x0..0xF] array.
      */
      uniplane = 0;
      for (i=0; i<6; i++) { /* go through one bitmap digit at a time */
         match = 0; /* haven't found pattern yet */
         for (j = 0x0; !match && j <= 0xF; j++) {
            if (unidigit[i][0] == hexdigit[j][0] &&
                unidigit[i][1] == hexdigit[j][1] &&
                unidigit[i][2] == hexdigit[j][2] &&
                unidigit[i][3] == hexdigit[j][3]) { /* we found the digit */
               uniplane |= j;
               match = 1;
            }
         }
         uniplane «= 4;
      }
      uniplane »= 4;
   }
   /*
      Now read each glyph and print it as hex.
   */
   for (i = 0x0; i <= 0xf; i++) {
      for (j = 0x0; j <= 0xf; j++) {
         for (k = 0; k < 16; k++) {
            if (flip) {  /* transpose glyph matrix */
               thischar0[k] = bitmap[32*(j+1) + k + 7][4 * (i+2)    ];
               thischar1[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 1];
               thischar2[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 2];
               thischar3[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 3];
            }
            else {
               thischar0[k] = bitmap[32*(i+1) + k + 7][4 * (j+2)    ];
               thischar1[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 1];
               thischar2[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 2];
```

```
                thischar3[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 3];
            }
        }
        /*
            If the second half of the 16*16 character is all zeroes, this
            character is only 8 bits wide, so print a half-width character.
        */
        empty1 = empty2 = 1;
        for (k=0; (empty1 || empty2) && k < 16; k++) {
            if (thischar1[k] != 0) empty1 = 0;
            if (thischar2[k] != 0) empty2 = 0;
            }
        /*
            Only print this glyph if it isn't blank.
        */
        if (!empty1 || !empty2) {
            /*
                If the second half is empty, this is a half-width character.
                Only print the first half.
            */
            /*
                Original GNU Unifont format is four hexadecimal digit character
                code followed by a colon followed by a hex string.  Add support
                for codes beyond the Basic Multilingual Plane.

                Unicode ranges from U+0000 to U+10FFFF, so print either a
                4-digit or a 6-digit code point.  Note that this software
                should support up to an 8-digit code point, extending beyond
                the normal Unicode range, but this has not been fully tested.
            */
            if (uniplane > 0xff)
                fprintf (outfp, "%04X%X%X:", uniplane, i, j); // 6 digit code pt.
            else
                fprintf (outfp, "%02X%X%X:", uniplane, i, j); // 4 digit code pt.
            for (thisrow=0; thisrow<16; thisrow++) {
                /*
                    If second half is empty and we're not forcing this
                    code point to double width, print as single width.
                */
                if (!forcewide &&
                     empty2 && !wide[(uniplane << 8) | (i << 4) | j]) {
                    fprintf (outfp,
                             "%02X",
                             thischar1[thisrow]);
                }
                else if (wide[(uniplane << 8) | (i << 4) | j] == 4) {
                    /* quadruple-width; force 32nd pixel to zero */
                    fprintf (outfp,
                             "%02X%02X%02X%02X",
                             thischar0[thisrow], thischar1[thisrow],
                             thischar2[thisrow], thischar3[thisrow] & 0xFE);
                }
                else { /* treat as double-width */
                    fprintf (outfp,
                             "%02X%02X",
                             thischar1[thisrow], thischar2[thisrow]);
                }
            }
            fprintf (outfp, "\n");
        }
      }
    }
    exit (0);
}
```

### 5.7.4 Variable Documentation

#### 5.7.4.1 bits_per_pixel

int bits_per_pixel

Definition at line 127 of file unibmp2hex.c.

#### 5.7.4.2

struct { ... } bmp_header

Bitmap Header parameters

#### 5.7.4.3 color_table

unsigned char color_table[256][4]

Bitmap Color Table – maximum of 256 colors in a BMP file

Definition at line 137 of file unibmp2hex.c.

#### 5.7.4.4 compression

int compression

Definition at line 128 of file unibmp2hex.c.

#### 5.7.4.5 file_size

int file_size

Definition at line 121 of file unibmp2hex.c.

#### 5.7.4.6 filetype

char filetype[2]

Definition at line 120 of file unibmp2hex.c.

#### 5.7.4.7 flip

unsigned flip =0

=1 if we're transposing glyph matrix

Definition at line 111 of file unibmp2hex.c.

#### 5.7.4.8 forcewide

unsigned forcewide =0

=1 to set each glyph to 16 pixels wide

Definition at line 112 of file unibmp2hex.c.

#### 5.7.4.9 height

int height

Definition at line 125 of file unibmp2hex.c.

#### 5.7.4.10 hexdigit

unsigned hexdigit[16][4]

32 bit representation of 16x8 0..F bitmap

Definition at line 107 of file unibmp2hex.c.

#### 5.7.4.11 image_offset

int image_offset

Definition at line 122 of file unibmp2hex.c.

#### 5.7.4.12 image_size

int image_size

Definition at line 129 of file unibmp2hex.c.

#### 5.7.4.13 important_colors

int important_colors

Definition at line 133 of file unibmp2hex.c.

#### 5.7.4.14 info_size

int info_size

Definition at line 123 of file unibmp2hex.c.

#### 5.7.4.15 ncolors

int ncolors

Definition at line 132 of file unibmp2hex.c.

#### 5.7.4.16 nplanes

int nplanes

Definition at line 126 of file unibmp2hex.c.

#### 5.7.4.17 planeset

unsigned planeset =0

=1: use plane specified with -p parameter

Definition at line 110 of file unibmp2hex.c.

#### 5.7.4.18 unidigit

unsigned unidigit[6][4]

The six Unicode plane digits, from left-most (0) to right-most (5)

Definition at line 115 of file unibmp2hex.c.

#### 5.7.4.19 uniplane

unsigned uniplane =0

Unicode plane number, 0..0xff ff ff.

Definition at line 109 of file unibmp2hex.c.

#### 5.7.4.20 width

int width

Definition at line 124 of file unibmp2hex.c.

#### 5.7.4.21 x_ppm

int x_ppm

Definition at line 130 of file unibmp2hex.c.

#### 5.7.4.22 y_ppm

int y_ppm

Definition at line 131 of file unibmp2hex.c.

## 5.8 unibmp2hex.c

Go to the documentation of this file.

```
/**
  @file unibmp2hex.c

  @brief unibmp2hex - Turn a .bmp or .wbmp glyph matrix into a
                      GNU Unifont hex glyph set of 256 characters

  @author Paul Hardy, unifoundry <at> unifoundry.com, December 2007

  @copyright Copyright (C) 2007, 2008, 2013, 2017, 2019, 2022 Paul Hardy

  Synopsis: unibmp2hex [-iin__file.bmp] [-oout__file.hex] [-phex__page__num] [-w]
*/
/*

  LICENSE:

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

/*
  6 September 2021 [Paul Hardy]:
    - Set U+12F90..U+12FFF (Cypro-Minoan) to be double width.
    - Set U+1CF00..U+1CFCF (Znamenny Musical Notation) to be double width.
    - Set U+1AFF0..U+1AFFF (Kana Extended-B) to be double width.

  20 June 2017 [Paul Hardy]:
    - Modify to allow hard-coding of quadruple-width hex glyphs.
      The 32nd column (rightmost column) is cleared to zero, because
      that column contains the vertical cell border.
    - Set U+9FD8..U+9FE9 (complex CJK) to be quadruple-width.
    - Set U+011A00..U+011A4F (Masaram Gondi, non-digits) to be wide.
    - Set U+011A50..U+011AAF (Soyombo) to be wide.

  8 July 2017 [Paul Hardy]:
    - All CJK glyphs in the range U+4E00..u+9FFF are double width
      again; commented out the line that sets U+9FD8..U+9FE9 to be
      quadruple width.

  6 August 2017 [Paul Hardy]:
    - Remove hard-coding of U+01D200..U+01D24F Ancient Greek Musical
      Notation to double-width; allow range to be dual-width.

  12 August 2017 [Paul Hardy]:
    - Remove Miao script from list of wide scripts, so it can contain
      single-width glyphs.

  26 December 2017 Paul Hardy:
    - Removed Tibetan from list of wide scripts, so it can contain
      single-width glyphs.
    - Added a number of scripts to be explicitly double-width in case
      they are redrawn.
    - Added Miao script back as wide, because combining glyphs are
      added back to font/plane01/plane01-combining.txt.

  05 June 2018 Paul Hardy:
    - Made U+2329] and U+232A wide.
    - Added to wide settings for CJK Compatibility Forms over entire range.
    - Made Kayah Li script double-width.
    - Made U+232A (Right-pointing Angle Bracket) double-width.
    - Made U+01F5E7 (Three Rays Right) double-width.

  July 2018 Paul Hardy:
    - Changed 2017 to 2018 in previous change entry.
    - Added Dogra (U+011800..U+01184F) as double width.
    - Added Makasar (U+011EE0..U+011EFF) as dobule width.
```

```

  23 February 2019 [Paul Hardy]:
     - Set U+119A0..U+119FF (Nandinagari) to be wide.
     - Set U+1E2C0..U+1E2FF (Wancho) to be wide.

  25 May 2019 [Paul Hardy]:
     - Added support for the case when the original .bmp monochrome
       file has been converted to a 32 bit per pixel RGB file.
     - Added support for bitmap images stored from either top to bottom
       or bottom to top.
     - Add DEBUG compile flag to print header information, to ease
       adding support for additional bitmap formats in the future.

  13 March 2022 [Paul Hardy]:
     - Added support for 24 bits per pixel RGB file.

  12 June 2022 [Paul Hardy]:
     - Set U+11B00..U+11B5F (Devanagari Extended-A) to be wide.
     - Set U+11F00..U+11F5F (Kawi) to be wide.


*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXBUF 256  ///< Maximum input file line length - 1


unsigned hexdigit[16][4]; ///< 32 bit representation of 16x8 0..F bitmap

unsigned uniplane=0;       ///< Unicode plane number, 0..0xffffff
unsigned planeset=0;       ///< =1: use plane specified with -p parameter
unsigned flip=0;           ///< =1 if we're transposing glyph matrix
unsigned forcewide=0;      ///< =1 to set each glyph to 16 pixels wide

/** The six Unicode plane digits, from left-most (0) to right-most (5) */
unsigned unidigit[6][4];


/** Bitmap Header parameters */
struct {
   char filetype[2];
   int file_size;
   int image_offset;
   int info_size;
   int width;
   int height;
   int nplanes;
   int bits_per_pixel;
   int compression;
   int image_size;
   int x_ppm;
   int y_ppm;
   int ncolors;
   int important_colors;
} bmp_header;

/** Bitmap Color Table -- maximum of 256 colors in a BMP file */
unsigned char color_table[256][4];  /* R, G, B, alpha for up to 256 colors */

// #define DEBUG

/**
   @brief The main function.

   @param[in] argc The count of command line arguments.
   @param[in] argv Pointer to array of command line arguments.
   @return This program exits with status 0.
*/
int
main (int argc, char *argv[])
{

   int i, j, k;                /* loop variables                      */
   unsigned char inchar;       /* temporary input character */
   char header[MAXBUF];        /* input buffer for bitmap file header */
   int wbmp=0; /* =0 for Windows Bitmap (.bmp); 1 for Wireless Bitmap (.wbmp) */
   int fatal; /* =1 if a fatal error occurred */
   int match; /* =1 if we're still matching a pattern, 0 if no match */
```

```
int empty1, empty2; /* =1 if bytes tested are all zeroes */
unsigned char thischar1[16], thischar2[16]; /* bytes of hex char */
unsigned char thischar0[16], thischar3[16]; /* bytes for quadruple-width */
int thisrow; /* index to point into thischar1[] and thischar2[] */
int tmpsum;  /* temporary sum to see if a character is blank */
unsigned this_pixel;  /* color of one pixel, if > 1 bit per pixel */
unsigned next_pixels; /* pending group of 8 pixels being read */
unsigned color_mask = 0x00;  /* to invert monochrome bitmap, set to 0xFF */

unsigned char bitmap[17*32][18*32/8]; /* final bitmap */
/* For wide array:
      0 = don't force glyph to double-width;
      1 = force glyph to double-width;
      4 = force glyph to quadruple-width.
*/
char wide[0x200000]={0x200000 * 0};

char *infile="", *outfile="";  /* names of input and output files */
FILE *infp, *outfp;       /* file pointers of input and output files */

if (argc > 1) {
   for (i = 1; i < argc; i++) {
      if (argv[i][0] == '-') {  /* this is an option argument */
         switch (argv[i][1]) {
            case 'i':  /* name of input file */
               infile = &argv[i][2];
               break;
            case 'o':  /* name of output file */
               outfile = &argv[i][2];
               break;
            case 'p':  /* specify a Unicode plane */
               sscanf (&argv[i][2], "%x", &uniplane); /* Get Unicode plane */
               planeset = 1; /* Use specified range, not what's in bitmap */
               break;
            case 'w': /* force wide (16 pixels) for each glyph */
               forcewide = 1;
               break;
            default:   /* if unrecognized option, print list and exit */
               fprintf (stderr, "\nSyntax:\n\n");
               fprintf (stderr, "   %s -p<Unicode_Page> ", argv[0]);
               fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
               fprintf (stderr, "   -w specifies .wbmp output instead of ");
               fprintf (stderr, "default Windows .bmp output.\n\n");
               fprintf (stderr, "   -p is followed by 1 to 6 ");
               fprintf (stderr, "Unicode plane hex digits ");
               fprintf (stderr, "(default is Page 0).\n\n");
               fprintf (stderr, "\nExample:\n\n");
               fprintf (stderr, "   %s -p83 -iunifont.hex -ou83.bmp\n\n\n",
                      argv[0]);
               exit (1);
         }
      }
   }
}
/*
   Make sure we can open any I/O files that were specified before
   doing anything else.
*/
if (strlen (infile) > 0) {
   if ((infp = fopen (infile, "r")) == NULL) {
      fprintf (stderr, "Error: can't open %s for input.\n", infile);
      exit (1);
   }
}
else {
   infp = stdin;
}
if (strlen (outfile) > 0) {
   if ((outfp = fopen (outfile, "w")) == NULL) {
      fprintf (stderr, "Error: can't open %s for output.\n", outfile);
      exit (1);
   }
}
else {
   outfp = stdout;
}
/*
   Initialize selected code points for double width (16x16).
   Double-width is forced in cases where a glyph (usually a combining
   glyph) only occupies the left-hand side of a 16x16 grid, but must
   be rendered as double-width to appear properly with other glyphs
```

```
     in a given script.  If additions were made to a script after
     Unicode 5.0, the Unicode version is given in parentheses after
     the script name.
  */
  for (i = 0x0700; i <= 0x074F; i++) wide[i] = 1; /* Syriac                */
  for (i = 0x0800; i <= 0x083F; i++) wide[i] = 1; /* Samaritan (5.2)       */
  for (i = 0x0900; i <= 0x0DFF; i++) wide[i] = 1; /* Indic                 */
  for (i = 0x1000; i <= 0x109F; i++) wide[i] = 1; /* Myanmar               */
  for (i = 0x1100; i <= 0x11FF; i++) wide[i] = 1; /* Hangul Jamo           */
  for (i = 0x1400; i <= 0x167F; i++) wide[i] = 1; /* Canadian Aboriginal   */
  for (i = 0x1700; i <= 0x171F; i++) wide[i] = 1; /* Tagalog               */
  for (i = 0x1720; i <= 0x173F; i++) wide[i] = 1; /* Hanunoo               */
  for (i = 0x1740; i <= 0x175F; i++) wide[i] = 1; /* Buhid                 */
  for (i = 0x1760; i <= 0x177F; i++) wide[i] = 1; /* Tagbanwa              */
  for (i = 0x1780; i <= 0x17FF; i++) wide[i] = 1; /* Khmer                 */
  for (i = 0x18B0; i <= 0x18FF; i++) wide[i] = 1; /* Ext. Can. Aboriginal  */
  for (i = 0x1800; i <= 0x18AF; i++) wide[i] = 1; /* Mongolian             */
  for (i = 0x1900; i <= 0x194F; i++) wide[i] = 1; /* Limbu                 */
// for (i = 0x1980; i <= 0x19DF; i++) wide[i] = 1; /* New Tai Lue           */
  for (i = 0x1A00; i <= 0x1A1F; i++) wide[i] = 1; /* Buginese              */
  for (i = 0x1A20; i <= 0x1AAF; i++) wide[i] = 1; /* Tai Tham (5.2)        */
  for (i = 0x1B00; i <= 0x1B7F; i++) wide[i] = 1; /* Balinese              */
  for (i = 0x1B80; i <= 0x1BBF; i++) wide[i] = 1; /* Sundanese (5.1)       */
  for (i = 0x1BC0; i <= 0x1BFF; i++) wide[i] = 1; /* Batak (6.0)           */
  for (i = 0x1C00; i <= 0x1C4F; i++) wide[i] = 1; /* Lepcha (5.1)          */
  for (i = 0x1CC0; i <= 0x1CCF; i++) wide[i] = 1; /* Sundanese Supplement  */
  for (i = 0x1CD0; i <= 0x1CFF; i++) wide[i] = 1; /* Vedic Extensions (5.2) */
  wide[0x2329] = wide[0x232A] = 1; /* Left- & Right-pointing Angle Brackets */
  for (i = 0x2E80; i <= 0xA4CF; i++) wide[i] = 1; /* CJK                   */
// for (i = 0x9FD8; i <= 0x9FE9; i++) wide[i] = 4; /* CJK quadruple-width   */
  for (i = 0xA900; i <= 0xA92F; i++) wide[i] = 1; /* Kayah Li (5.1)        */
  for (i = 0xA930; i <= 0xA95F; i++) wide[i] = 1; /* Rejang (5.1)          */
  for (i = 0xA960; i <= 0xA97F; i++) wide[i] = 1; /* Hangul Jamo Extended-A */
  for (i = 0xA980; i <= 0xA9DF; i++) wide[i] = 1; /* Javanese (5.2)        */
  for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham (5.1)            */
  for (i = 0xA9E0; i <= 0xA9FF; i++) wide[i] = 1; /* Myanmar Extended-B    */
  for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham                  */
  for (i = 0xAA60; i <= 0xAA7F; i++) wide[i] = 1; /* Myanmar Extended-A    */
  for (i = 0xAAE0; i <= 0xAAFF; i++) wide[i] = 1; /* Meetei Mayek Ext (6.0) */
  for (i = 0xABC0; i <= 0xABFF; i++) wide[i] = 1; /* Meetei Mayek (5.2)    */
  for (i = 0xAC00; i <= 0xD7AF; i++) wide[i] = 1; /* Hangul Syllables      */
  for (i = 0xD7B0; i <= 0xD7FF; i++) wide[i] = 1; /* Hangul Jamo Extended-B */
  for (i = 0xF900; i <= 0xFAFF; i++) wide[i] = 1; /* CJK Compatibility     */
  for (i = 0xFE10; i <= 0xFE1F; i++) wide[i] = 1; /* Vertical Forms        */
  for (i = 0xFE30; i <= 0xFE60; i++) wide[i] = 1; /* CJK Compatibility Forms*/
  for (i = 0xFFE0; i <= 0xFFE6; i++) wide[i] = 1; /* CJK Compatibility Forms*/

  wide[0x303F] = 0; /* CJK half-space fill */

  /* Supplemental Multilingual Plane (Plane 01) */
  for (i = 0x010A00; i <= 0x010A5F; i++) wide[i] = 1; /* Kharoshthi       */
  for (i = 0x011000; i <= 0x01107F; i++) wide[i] = 1; /* Brahmi           */
  for (i = 0x011080; i <= 0x0110CF; i++) wide[i] = 1; /* Kaithi           */
  for (i = 0x011100; i <= 0x01114F; i++) wide[i] = 1; /* Chakma           */
  for (i = 0x011180; i <= 0x0111DF; i++) wide[i] = 1; /* Sharada          */
  for (i = 0x011200; i <= 0x01124F; i++) wide[i] = 1; /* Khojki           */
  for (i = 0x0112B0; i <= 0x0112FF; i++) wide[i] = 1; /* Khudawadi        */
  for (i = 0x011300; i <= 0x01137F; i++) wide[i] = 1; /* Grantha          */
  for (i = 0x011400; i <= 0x01147F; i++) wide[i] = 1; /* Newa             */
  for (i = 0x011480; i <= 0x0114DF; i++) wide[i] = 1; /* Tirhuta          */
  for (i = 0x011580; i <= 0x0115FF; i++) wide[i] = 1; /* Siddham          */
  for (i = 0x011600; i <= 0x01165F; i++) wide[i] = 1; /* Modi             */
  for (i = 0x011660; i <= 0x01167F; i++) wide[i] = 1; /* Mongolian Suppl.  */
  for (i = 0x011680; i <= 0x0116CF; i++) wide[i] = 1; /* Takri            */
  for (i = 0x011700; i <= 0x01173F; i++) wide[i] = 1; /* Ahom             */
  for (i = 0x011800; i <= 0x01184F; i++) wide[i] = 1; /* Dogra            */
  for (i = 0x011900; i <= 0x01195F; i++) wide[i] = 1; /* Dives Akuru      */
  for (i = 0x0119A0; i <= 0x0119FF; i++) wide[i] = 1; /* Nandinagari      */
  for (i = 0x011A00; i <= 0x011A4F; i++) wide[i] = 1; /* Zanabazar Square  */
  for (i = 0x011A50; i <= 0x011AAF; i++) wide[i] = 1; /* Soyombo          */
  for (i = 0x011B00; i <= 0x011B5F; i++) wide[i] = 1; /* Devanagari Extended-A*/
  for (i = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi             */
  for (i = 0x011C00; i <= 0x011C6F; i++) wide[i] = 1; /* Bhaiksuki        */
  for (i = 0x011C70; i <= 0x011CBF; i++) wide[i] = 1; /* Marchen          */
  for (i = 0x011D00; i <= 0x011D5F; i++) wide[i] = 1; /* Masaram Gondi    */
  for (i = 0x011EE0; i <= 0x011EFF; i++) wide[i] = 1; /* Makasar          */
  for (i = 0x012F90; i <= 0x012FFF; i++) wide[i] = 1; /* Cypro-Minoan     */
  /* Make Bassa Vah all single width or all double width */
  for (i = 0x016AD0; i <= 0x016AFF; i++) wide[i] = 1; /* Bassa Vah        */
  for (i = 0x016B00; i <= 0x016B8F; i++) wide[i] = 1; /* Pahawh Hmong     */
  for (i = 0x016F00; i <= 0x016F9F; i++) wide[i] = 1; /* Miao             */
```

```
for (i = 0x016FE0; i <= 0x016FFF; i++) wide[i] = 1; /* Ideograph Sym/Punct*/
for (i = 0x017000; i <= 0x0187FF; i++) wide[i] = 1; /* Tangut             */
for (i = 0x018800; i <= 0x018AFF; i++) wide[i] = 1; /* Tangut Components  */
for (i = 0x01AFF0; i <= 0x01AFFF; i++) wide[i] = 1; /* Kana Extended-B    */
for (i = 0x01B000; i <= 0x01B0FF; i++) wide[i] = 1; /* Kana Supplement    */
for (i = 0x01B100; i <= 0x01B12F; i++) wide[i] = 1; /* Kana Extended-A    */
for (i = 0x01B170; i <= 0x01B2FF; i++) wide[i] = 1; /* Nushu              */
for (i = 0x01CF00; i <= 0x01CFCF; i++) wide[i] = 1; /* Znamenny Musical   */
for (i = 0x01D100; i <= 0x01D1FF; i++) wide[i] = 1; /* Musical Symbols    */
for (i = 0x01D800; i <= 0x01DAAF; i++) wide[i] = 1; /* Sutton SignWriting */
for (i = 0x01E2C0; i <= 0x01E2FF; i++) wide[i] = 1; /* Wancho             */
for (i = 0x01E800; i <= 0x01E8DF; i++) wide[i] = 1; /* Mende Kikakui      */
for (i = 0x01F200; i <= 0x01F2FF; i++) wide[i] = 1; /* Encl Ideograp Suppl*/
wide[0x01F5E7] = 1;                                  /* Three Rays Right   */

/*
   Determine whether or not the file is a Microsoft Windows Bitmap file.
   If it starts with ’B’, ’M’, assume it’s a Windows Bitmap file.
   Otherwise, assume it’s a Wireless Bitmap file.

   WARNING: There isn’t much in the way of error checking here --
   if you give it a file that wasn’t first created by hex2bmp.c,
   all bets are off.
*/
fatal = 0;  /* assume everything is okay with reading input file */
if ((header[0] = fgetc (infp)) != EOF) {
   if ((header[1] = fgetc (infp)) != EOF) {
      if (header[0] == ’B’ && header[1] == ’M’) {
         wbmp = 0; /* Not a Wireless Bitmap -- it’s a Windows Bitmap */
      }
      else {
         wbmp = 1; /* Assume it’s a Wireless Bitmap */
      }
   }
   else
      fatal = 1;
}
else
   fatal = 1;

if (fatal) {
   fprintf (stderr, ”Fatal error; end of input file.\n\n”);
   exit (1);
}
/*
   If this is a Wireless Bitmap (.wbmp) format file,
   skip the header and point to the start of the bitmap itself.
*/
if (wbmp) {
   for (i=2; i<6; i++)
      header[i] = fgetc (infp);
   /*
      Now read the bitmap.
   */
   for (i=0; i < 32*17; i++) {
      for (j=0; j < 32*18/8; j++) {
         inchar = fgetc (infp);
         bitmap[i][j] = ~inchar;  /* invert bits for proper color */
      }
   }
}
/*
   Otherwise, treat this as a Windows Bitmap file, because we checked
   that it began with ”BM”.  Save the header contents for future use.
   Expect a 14 byte standard BITMAPFILEHEADER format header followed
   by a 40 byte standard BITMAPINFOHEADER Device Independent Bitmap
   header, with data stored in little-endian format.
*/
else {
   for (i = 2; i < 54; i++)
      header[i] = fgetc (infp);

   bmp_header.filetype[0] = ’B’;
   bmp_header.filetype[1] = ’M’;

   bmp_header.file_size =
       (header[2] & 0xFF)        | ((header[3] & 0xFF) «  8) |
      ((header[4] & 0xFF) « 16) | ((header[5] & 0xFF) « 24);

   /* header bytes 6..9 are reserved */

```

```
    bmp_header.image_offset =
        (header[10] & 0xFF)        | ((header[11] & 0xFF) <<  8) |
        ((header[12] & 0xFF) << 16) | ((header[13] & 0xFF) << 24);

    bmp_header.info_size =
        (header[14] & 0xFF)        | ((header[15] & 0xFF) <<  8) |
        ((header[16] & 0xFF) << 16) | ((header[17] & 0xFF) << 24);

    bmp_header.width =
        (header[18] & 0xFF)        | ((header[19] & 0xFF) <<  8) |
        ((header[20] & 0xFF) << 16) | ((header[21] & 0xFF) << 24);

    bmp_header.height =
        (header[22] & 0xFF)        | ((header[23] & 0xFF) <<  8) |
        ((header[24] & 0xFF) << 16) | ((header[25] & 0xFF) << 24);

    bmp_header.nplanes =
        (header[26] & 0xFF)        | ((header[27] & 0xFF) <<  8);

    bmp_header.bits_per_pixel =
        (header[28] & 0xFF)        | ((header[29] & 0xFF) <<  8);

    bmp_header.compression =
        (header[30] & 0xFF)        | ((header[31] & 0xFF) <<  8) |
        ((header[32] & 0xFF) << 16) | ((header[33] & 0xFF) << 24);

    bmp_header.image_size =
        (header[34] & 0xFF)        | ((header[35] & 0xFF) <<  8) |
        ((header[36] & 0xFF) << 16) | ((header[37] & 0xFF) << 24);

    bmp_header.x_ppm =
        (header[38] & 0xFF)        | ((header[39] & 0xFF) <<  8) |
        ((header[40] & 0xFF) << 16) | ((header[41] & 0xFF) << 24);

    bmp_header.y_ppm =
        (header[42] & 0xFF)        | ((header[43] & 0xFF) <<  8) |
        ((header[44] & 0xFF) << 16) | ((header[45] & 0xFF) << 24);

    bmp_header.ncolors =
        (header[46] & 0xFF)        | ((header[47] & 0xFF) <<  8) |
        ((header[48] & 0xFF) << 16) | ((header[49] & 0xFF) << 24);

    bmp_header.important_colors =
        (header[50] & 0xFF)        | ((header[51] & 0xFF) <<  8) |
        ((header[52] & 0xFF) << 16) | ((header[53] & 0xFF) << 24);

    if (bmp_header.ncolors == 0)
        bmp_header.ncolors = 1 << bmp_header.bits_per_pixel;

    /* If a Color Table exists, read it */
    if (bmp_header.ncolors > 0 && bmp_header.bits_per_pixel <= 8) {
        for (i = 0; i < bmp_header.ncolors; i++) {
            color_table[i][0] = fgetc (infp);  /* Red   */
            color_table[i][1] = fgetc (infp);  /* Green */
            color_table[i][2] = fgetc (infp);  /* Blue  */
            color_table[i][3] = fgetc (infp);  /* Alpha */
        }
        /*
            Determine from the first color table entry whether we
            are inverting the resulting bitmap image.
        */
        if ( (color_table[0][0] + color_table[0][1] + color_table[0][2])
             < (3 * 128) ) {
            color_mask = 0xFF;
        }
    }

#ifdef DEBUG

    /*
        Print header info for possibly adding support for
        additional file formats in the future, to determine
        how the bitmap is encoded.
    */
    fprintf (stderr, "Filetype: '%c%c'\n",
                     bmp_header.filetype[0], bmp_header.filetype[1]);
    fprintf (stderr, "File Size: %d\n", bmp_header.file_size);
    fprintf (stderr, "Image Offset: %d\n", bmp_header.image_offset);
    fprintf (stderr, "Info Header Size: %d\n", bmp_header.info_size);
    fprintf (stderr, "Image Width: %d\n", bmp_header.width);
    fprintf (stderr, "Image Height: %d\n", bmp_header.height);
```

```
    fprintf (stderr, "Number of Planes: %d\n", bmp_header.nplanes);
    fprintf (stderr, "Bits per Pixel: %d\n", bmp_header.bits_per_pixel);
    fprintf (stderr, "Compression Method: %d\n", bmp_header.compression);
    fprintf (stderr, "Image Size: %d\n", bmp_header.image_size);
    fprintf (stderr, "X Pixels per Meter: %d\n", bmp_header.x_ppm);
    fprintf (stderr, "Y Pixels per Meter: %d\n", bmp_header.y_ppm);
    fprintf (stderr, "Number of Colors: %d\n", bmp_header.ncolors);
    fprintf (stderr, "Important Colors: %d\n", bmp_header.important_colors);

#endif

    /*
      Now read the bitmap.
    */
    for (i = 32*17-1; i >= 0; i--) {
      for (j=0; j < 32*18/8; j++) {
        next_pixels = 0x00;  /* initialize next group of 8 pixels */
        /* Read a monochrome image -- the original case */
        if (bmp_header.bits_per_pixel == 1) {
          next_pixels = fgetc (infp);
        }
        /* Read a 32 bit per pixel RGB image; convert to monochrome */
        else if ( bmp_header.bits_per_pixel == 24 ||
                 bmp_header.bits_per_pixel == 32) {
          next_pixels = 0;
          for (k = 0; k < 8; k++) {  /* get next 8 pixels */
            this_pixel = (fgetc (infp) & 0xFF) +
                         (fgetc (infp) & 0xFF) +
                         (fgetc (infp) & 0xFF);

            if (bmp_header.bits_per_pixel == 32) {
              (void) fgetc (infp);  /* ignore alpha value */
            }

            /* convert RGB color space to monochrome */
            if (this_pixel >= (128 * 3))
              this_pixel = 0;
            else
              this_pixel = 1;

            /* shift next pixel color into place for 8 pixels total */
            next_pixels = (next_pixels « 1) | this_pixel;
          }
        }
        if (bmp_header.height < 0) {  /* Bitmap drawn top to bottom */
          bitmap [(32*17-1) - i] [j] = next_pixels;
        }
        else {  /* Bitmap drawn bottom to top */
          bitmap [i][j] = next_pixels;
        }
      }
    }

    /*
      If any bits are set in color_mask, apply it to
      entire bitmap to invert black <--> white.
    */
    if (color_mask != 0x00) {
      for (i = 32*17-1; i >= 0; i--) {
        for (j=0; j < 32*18/8; j++) {
          bitmap [i][j] ^= color_mask;
        }
      }
    }

  }

  /*
    We've read the entire file.  Now close the input file pointer.
  */
  fclose (infp);
 /*
    We now have the header portion in the header[] array,
    and have the bitmap portion from top-to-bottom in the bitmap[] array.
  */
  /*
    If no Unicode range (U+nnnnnn00 through U+nnnnnnFF) was specified
    with a -p parameter, determine the range from the digits in the
    bitmap itself.

    Store bitmaps for the hex digit patterns that this file uses.
```

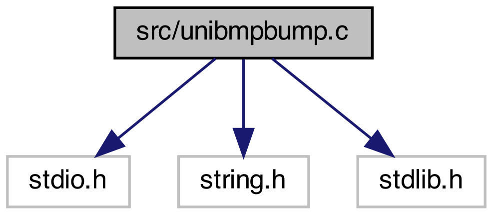
```
*/
if (!planeset) {  /* If Unicode range not specified with -p parameter */
   for (i = 0x0; i <= 0xF; i++) {  /* hex digit pattern we're storing */
      for (j = 0; j < 4; j++) {
         hexdigit[i][j]   =
            ((unsigned)bitmap[32 * (i+1) + 4 * j + 8    ][6] « 24 ) |
            ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 1][6] « 16 ) |
            ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 2][6] «  8 ) |
            ((unsigned)bitmap[32 * (i+1) + 4 * j + 8 + 3][6]       );
      }
   }
   /*
      Read the Unicode plane digits into arrays for comparison, to
      determine the upper four hex digits of the glyph addresses.
   */
   for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++) {
         unidigit[i][j] =
            ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 1][i + 3] « 24 ) |
            ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 2][i + 3] « 16 ) |
            ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 3][i + 3] «  8 ) |
            ((unsigned)bitmap[32 * 0 + 4 * j + 8 + 4][i + 3]       );
      }
   }

   tmpsum = 0;
   for (i = 4; i < 6; i++) {
      for (j = 0; j < 4; j++) {
         unidigit[i][j] =
            ((unsigned)bitmap[32 * 1 + 4 * j + 8    ][i] « 24 ) |
            ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 1][i] « 16 ) |
            ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 2][i] «  8 ) |
            ((unsigned)bitmap[32 * 1 + 4 * j + 8 + 3][i]       );
         tmpsum |= unidigit[i][j];
      }
   }
   if (tmpsum == 0) {  /* the glyph matrix is transposed */
      flip = 1;  /* note transposed order for processing glyphs in matrix */
      /*
         Get 5th and 6th hex digits by shifting first column header left by
         1.5 columns, thereby shifting the hex digit right after the leading
         "U+nnnn" page number.
      */
      for (i = 0x08; i < 0x18; i++) {
         bitmap[i][7] = (bitmap[i][8] « 4) | ((bitmap[i][ 9] » 4) & 0xf);
         bitmap[i][8] = (bitmap[i][9] « 4) | ((bitmap[i][10] » 4) & 0xf);
      }
      for (i = 4; i < 6; i++) {
         for (j = 0; j < 4; j++) {
            unidigit[i][j] =
               ((unsigned)bitmap[4 * j + 8 + 1][i + 3] « 24 ) |
               ((unsigned)bitmap[4 * j + 8 + 2][i + 3] « 16 ) |
               ((unsigned)bitmap[4 * j + 8 + 3][i + 3] «  8 ) |
               ((unsigned)bitmap[4 * j + 8 + 4][i + 3]       );
         }
      }
   }

   /*
      Now determine the Unicode plane by comparing unidigit[0..5] to
      the hexdigit[0x0..0xF] array.
   */
   uniplane = 0;
   for (i=0; i<6; i++) { /* go through one bitmap digit at a time */
      match = 0; /* haven't found pattern yet */
      for (j = 0x0; !match && j <= 0xF; j++) {
         if (unidigit[i][0] == hexdigit[j][0] &&
             unidigit[i][1] == hexdigit[j][1] &&
             unidigit[i][2] == hexdigit[j][2] &&
             unidigit[i][3] == hexdigit[j][3]) { /* we found the digit */
            uniplane |= j;
            match = 1;
         }
      }
      uniplane «= 4;
   }
   uniplane »= 4;
}
/*
   Now read each glyph and print it as hex.
*/
```

```
    for (i = 0x0; i <= 0xf; i++) {
       for (j = 0x0; j <= 0xf; j++) {
          for (k = 0; k < 16; k++) {
             if (flip) {  /* transpose glyph matrix */
                thischar0[k] = bitmap[32*(j+1) + k + 7][4 * (i+2)    ];
                thischar1[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 1];
                thischar2[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 2];
                thischar3[k] = bitmap[32*(j+1) + k + 7][4 * (i+2) + 3];
             }
             else {
                thischar0[k] = bitmap[32*(i+1) + k + 7][4 * (j+2)    ];
                thischar1[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 1];
                thischar2[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 2];
                thischar3[k] = bitmap[32*(i+1) + k + 7][4 * (j+2) + 3];
             }
          }
          /*
             If the second half of the 16*16 character is all zeroes, this
             character is only 8 bits wide, so print a half-width character.
          */
          empty1 = empty2 = 1;
          for (k=0; (empty1 || empty2) && k < 16; k++) {
             if (thischar1[k] != 0) empty1 = 0;
             if (thischar2[k] != 0) empty2 = 0;
             }
          /*
             Only print this glyph if it isn't blank.
          */
          if (!empty1 || !empty2) {
             /*
                If the second half is empty, this is a half-width character.
                Only print the first half.
             */
             /*
                Original GNU Unifont format is four hexadecimal digit character
                code followed by a colon followed by a hex string.  Add support
                for codes beyond the Basic Multilingual Plane.

                Unicode ranges from U+0000 to U+10FFFF, so print either a
                4-digit or a 6-digit code point.  Note that this software
                should support up to an 8-digit code point, extending beyond
                the normal Unicode range, but this has not been fully tested.
             */
             if (uniplane > 0xff)
                fprintf (outfp, "%04X%X%X:", uniplane, i, j); // 6 digit code pt.
             else
                fprintf (outfp, "%02X%X%X:", uniplane, i, j); // 4 digit code pt.
             for (thisrow=0; thisrow<16; thisrow++) {
                /*
                   If second half is empty and we're not forcing this
                   code point to double width, print as single width.
                */
                if (!forcewide &&
                     empty2 && !wide[(uniplane « 8) | (i « 4) | j]) {
                   fprintf (outfp,
                           "%02X",
                           thischar1[thisrow]);
                }
                else if (wide[(uniplane « 8) | (i « 4) | j] == 4) {
                   /* quadruple-width; force 32nd pixel to zero */
                   fprintf (outfp,
                           "%02X%02X%02X%02X",
                           thischar0[thisrow], thischar1[thisrow],
                           thischar2[thisrow], thischar3[thisrow] & 0xFE);
                }
                else { /* treat as double-width */
                   fprintf (outfp,
                           "%02X%02X",
                           thischar1[thisrow], thischar2[thisrow]);
                }
             }
             fprintf (outfp, "\n");
          }
       }
    }
    exit (0);
}
```

## 5.9 src/unibmpbump.c File Reference

unibmpbump - Adjust a Microsoft bitmap (.bmp) file that was created by unihex2png but converted to .bmp

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
Include dependency graph for unibmpbump.c:




### Macros

- #define VERSION "1.0"

  Version of this program.
- #define MAX_COMPRESSION_METHOD 13

  Maximum supported compression method.

### Functions

- int main (int argc, char *argv[])

  The main function.
- unsigned get_bytes (FILE *infp, int nbytes)

  Get from 1 to 4 bytes, inclusive, from input file.
- void regrid (unsigned *image_bytes)

  After reading in the image, shift it.

### 5.9.1 Detailed Description

unibmpbump - Adjust a Microsoft bitmap (.bmp) file that was created by unihex2png but converted to .bmp

Author

Paul Hardy, unifoundry <at> unifoundry.com

Copyright

Copyright (C) 2019 Paul Hardy

This program shifts the glyphs in a bitmap file to adjust an original PNG file that was saved in BMP format. This is so the result matches the format of a unihex2bmp image. This conversion then lets unibmp2hex decode the result.

Synopsis: unibmpbump [-iin_file.bmp] [-oout_file.bmp]

Definition in file unibmpbump.c.

### 5.9.2 Macro Definition Documentation

#### 5.9.2.1 MAX_COMPRESSION_METHOD

```
#define MAX_COMPRESSION_METHOD 13
```

Maximum supported compression method.

Definition at line 40 of file unibmpbump.c.

#### 5.9.2.2 VERSION

```
#define VERSION "1.0"
```

Version of this program.

Definition at line 38 of file unibmpbump.c.

### 5.9.3 Function Documentation

#### 5.9.3.1 get_bytes()

```
unsigned get_bytes (
        FILE * infp,
        int nbytes )
```

Get from 1 to 4 bytes, inclusive, from input file.

Parameters

| | | |
|---|---|---|
| in | infp | Pointer to input file. |
| in | nbytes | Number of bytes to read, from 1 to 4, inclusive. |

Returns

The unsigned 1 to 4 bytes in machine native endian format.

Definition at line 487 of file unibmpbump.c.

```
00487 {
00488    int i;
00489    unsigned char inchar[4];
00490    unsigned inword;
00491
00492    for (i = 0; i < nbytes; i++) {
00493       if (fread (&inchar[i], 1, 1, infp) != 1) {
00494          inchar[i] = 0;
00495       }
00496    }
00497    for (i = nbytes; i < 4; i++) inchar[i] = 0;
00498
00499    inword = ((inchar[3] & 0xFF) « 24) | ((inchar[2] & 0xFF) « 16) |
00500             ((inchar[1] & 0xFF) «  8) |  (inchar[0] & 0xFF);
00501
00502    return inword;
00503 }
```

### 5.9.3.2 main()

int main (
    int argc,
    char ∗ argv[] )

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 50 of file unibmpbump.c.

```
00050                                 {
00051
00052    /*
00053       Values preserved from file header (first 14 bytes).
00054    */
00055    char file_format[3];        /* "BM" for original Windows format            */
00056    unsigned filesize;          /* size of file in bytes                       */
00057    unsigned char rsvd_hdr[4];  /* 4 reserved bytes                            */
00058    unsigned image_start;       /* byte offset of image in file                */
00059
00060    /*
00061       Values preserved from Device Independent Bitmap (DIB) Header.
00062
00063       The DIB fields below are in the standard 40-byte header.  Version
00064       4 and version 5 headers have more information, mainly for color
00065       information.  That is skipped over, because a valid glyph image
00066       is just monochrome.
00067    */
00068    int dib_length;             /* in bytes, for parsing by header version     */
00069    int image_width = 0;        /* Signed image width                          */
00070    int image_height = 0;       /* Signed image height                         */
00071    int num_planes = 0;         /* number of planes; must be 1                 */
00072    int bits_per_pixel = 0;     /* for palletized color maps (< 2^16 colors)   */
00073    /*
00074       The following fields are not in the original spec, so initialize
00075       them to 0 so we can correctly parse an original file format.
00076    */
00077    int compression_method=0;   /* 0 --> uncompressed RGB/monochrome           */
00078    int image_size = 0;         /* 0 is a valid size if no compression         */
00079    int hres = 0;               /* image horizontal resolution                 */
00080    int vres = 0;               /* image vertical resolution                   */
00081    int num_colors = 0;         /* Number of colors for pallettized images     */
00082    int important_colors = 0;   /* Number of significant colors (0 or 2)       */
00083
00084    int true_colors = 0;        /* interpret num_colors, which can equal 0     */
00085
00086    /*
00087       Color map.  This should be a monochrome file, so only two
00088       colors are stored.
00089    */
00090    unsigned char color_map[2][4]; /* two of R, G, B, and possibly alpha  */
00091
00092    /*
```

```
   The monochrome image bitmap, stored as a vector 544 rows by
   72*8 columns.
*/
unsigned image_bytes[544*72];

/*
   Flags for conversion & I/O.
*/
int verbose      = 0;     /* Whether to print file info on stderr     */
unsigned image_xor = 0x00; /* Invert (= 0xFF) if color 0 is not black  */

/*
   Temporary variables.
*/
int i, j, k;          /* loop variables */

/* Compression type, for parsing file */
char *compression_type[MAX_COMPRESSION_METHOD + 1] = {
   "BI_RGB",           /*  0 */
   "BI_RLE8",          /*  1 */
   "BI_RLE4",          /*  2 */
   "BI_BITFIELDS",     /*  3 */
   "BI_JPEG",          /*  4 */
   "BI_PNG",           /*  5 */
   "BI_ALPHABITFIELDS", /*  6 */
   "", "", "", "",     /*  7 - 10 */
   "BI_CMYK",          /* 11 */
   "BI_CMYKRLE8",      /* 12 */
   "BI_CMYKRLE4",      /* 13 */
};

/* Standard unihex2bmp.c header for BMP image */
unsigned standard_header [62] = {
   /*  0 */ 0x42, 0x4d, 0x3e, 0x99, 0x00, 0x00, 0x00, 0x00,
   /*  8 */ 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x28, 0x00,
   /* 16 */ 0x00, 0x00, 0x40, 0x02, 0x00, 0x00, 0x20, 0x02,
   /* 24 */ 0x00, 0x00, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00,
   /* 32 */ 0x00, 0x00, 0x00, 0x99, 0x00, 0x00, 0xc4, 0x0e,
   /* 40 */ 0x00, 0x00, 0xc4, 0x0e, 0x00, 0x00, 0x00, 0x00,
   /* 48 */ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   /* 56 */ 0x00, 0x00, 0xff, 0xff, 0xff, 0x00
};

unsigned get_bytes (FILE *, int);
void     regrid    (unsigned *);

char *infile="", *outfile="";  /* names of input and output files         */
FILE *infp, *outfp;            /* file pointers of input and output files */

/*
   Process command line arguments.
*/
if (argc > 1) {
   for (i = 1; i < argc; i++) {
      if (argv[i][0] == '-') {  /* this is an option argument */
         switch (argv[i][1]) {
            case 'i':  /* name of input file */
               infile = &argv[i][2];
               break;
            case 'o':  /* name of output file */
               outfile = &argv[i][2];
               break;
            case 'v':  /* verbose output */
               verbose = 1;
               break;
            case 'V':  /* print version & quit */
               fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
               exit (EXIT_SUCCESS);
               break;
            case '-':  /* see if "--verbose" */
               if (strcmp (argv[i], "--verbose") == 0) {
                  verbose = 1;
               }
               else if (strcmp (argv[i], "--version") == 0) {
                  fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
                  exit (EXIT_SUCCESS);
               }
               break;
            default:   /* if unrecognized option, print list and exit */
               fprintf (stderr, "\nSyntax:\n\n");
               fprintf (stderr, "   unibmpbump ");
```

```
               fprintf (stderr, "-i<Input_File> -o<Output_File>\n\n");
               fprintf (stderr, "-v or --verbose gives verbose output");
               fprintf (stderr, " on stderr\n\n");
               fprintf (stderr, "-V or --version prints version");
               fprintf (stderr, " on stderr and exits\n\n");
               fprintf (stderr, "\nExample:\n\n");
               fprintf (stderr, "   unibmpbump -iuni0101.bmp");
               fprintf (stderr, " -onew-uni0101.bmp\n\n");
               exit (EXIT_SUCCESS);
            }
         }
      }
   }

   /*
      Make sure we can open any I/O files that were specified before
      doing anything else.
   */
   if (strlen (infile) > 0) {
      if ((infp = fopen (infile, "r")) == NULL) {
         fprintf (stderr, "Error: can't open %s for input.\n", infile);
         exit (EXIT_FAILURE);
      }
   }
   else {
      infp = stdin;
   }
   if (strlen (outfile) > 0) {
      if ((outfp = fopen (outfile, "w")) == NULL) {
         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
         exit (EXIT_FAILURE);
      }
   }
   else {
      outfp = stdout;
   }


   /* Read bitmap file header */
   file_format[0] = get_bytes (infp, 1);
   file_format[1] = get_bytes (infp, 1);
   file_format[2] = '\0';  /* Terminate string with null */

   /* Read file size */
   filesize = get_bytes (infp, 4);

   /* Read Reserved bytes */
   rsvd_hdr[0] = get_bytes (infp, 1);
   rsvd_hdr[1] = get_bytes (infp, 1);
   rsvd_hdr[2] = get_bytes (infp, 1);
   rsvd_hdr[3] = get_bytes (infp, 1);

   /* Read Image Offset Address within file */
   image_start = get_bytes (infp, 4);

   /*
      See if this looks like a valid image file based on
      the file header first two bytes.
   */
   if (strncmp (file_format, "BM", 2) != 0) {
      fprintf (stderr, "\nInvalid file format: not file type \"BM\".\n\n");
      exit (EXIT_FAILURE);
   }

   if (verbose) {
      fprintf (stderr, "\nFile Header:\n");
      fprintf (stderr, "   File Type:   \"%s\"\n", file_format);
      fprintf (stderr, "   File Size:   %d bytes\n", filesize);
      fprintf (stderr, "   Reserved:   ");
      for (i = 0; i < 4; i++) fprintf (stderr, " 0x%02X", rsvd_hdr[i]);
      fputc ('\n', stderr);
      fprintf (stderr, "   Image Start: %d. = 0x%02X = 0%05o\n\n",
               image_start, image_start, image_start);
   }  /* if (verbose) */

   /*
      Device Independent Bitmap (DIB) Header: bitmap information header
      ("BM" format file DIB Header is 12 bytes long).
   */
   dib_length = get_bytes (infp, 4);

```

```
/*
   Parse one of three versions of Device Independent Bitmap (DIB) format:

        Length  Format
        ------  ------
           12   BITMAPCOREHEADER
           40   BITMAPINFOHEADER
          108   BITMAPV4HEADER
          124   BITMAPV5HEADER
*/
if (dib_length == 12) { /* BITMAPCOREHEADER format -- UNTESTED */
   image_width    = get_bytes (infp, 2);
   image_height   = get_bytes (infp, 2);
   num_planes     = get_bytes (infp, 2);
   bits_per_pixel = get_bytes (infp, 2);
}
else if (dib_length >= 40) { /* BITMAPINFOHEADER format or later */
   image_width = get_bytes (infp, 4);
   image_height       = get_bytes (infp, 4);
   num_planes         = get_bytes (infp, 2);
   bits_per_pixel     = get_bytes (infp, 2);
   compression_method = get_bytes (infp, 4);  /* BI_BITFIELDS */
   image_size         = get_bytes (infp, 4);
   hres               = get_bytes (infp, 4);
   vres               = get_bytes (infp, 4);
   num_colors         = get_bytes (infp, 4);
   important_colors   = get_bytes (infp, 4);

   /* true_colors is true number of colors in image */
   if (num_colors == 0)
      true_colors = 1 « bits_per_pixel;
   else
      true_colors = num_colors;

   /*
      If dib_length > 40, the format is BITMAPV4HEADER or
      BITMAPV5HEADER.  As this program is only designed
      to handle a monochrome image, we can ignore the rest
      of the header but must read past the remaining bytes.
   */
   for (i = 40; i < dib_length; i++) (void)get_bytes (infp, 1);
}

if (verbose) {
   fprintf (stderr, "Device Independent Bitmap (DIB) Header:\n");
   fprintf (stderr, "   DIB Length:  %9d bytes (version = ", dib_length);

   if      (dib_length ==  12) fprintf (stderr, "\"BITMAPCOREHEADER\")\n");
   else if (dib_length ==  40) fprintf (stderr, "\"BITMAPINFOHEADER\")\n");
   else if (dib_length == 108) fprintf (stderr, "\"BITMAPV4HEADER\")\n");
   else if (dib_length == 124) fprintf (stderr, "\"BITMAPV5HEADER\")\n");
   else fprintf (stderr, "unknown)");
   fprintf (stderr, "   Bitmap Width:   %6d pixels\n", image_width);
   fprintf (stderr, "   Bitmap Height:  %6d pixels\n", image_height);
   fprintf (stderr, "   Color Planes:   %6d\n",        num_planes);
   fprintf (stderr, "   Bits per Pixel: %6d\n",        bits_per_pixel);
   fprintf (stderr, "   Compression Method: %2d --> ", compression_method);
   if (compression_method <= MAX_COMPRESSION_METHOD) {
      fprintf (stderr, "%s", compression_type [compression_method]);
   }
   /*
      Supported compression method values:
           0 --> uncompressed RGB
          11 --> uncompressed CMYK
   */
   if (compression_method == 0 || compression_method == 11) {
      fprintf (stderr, " (no compression)");
   }
   else {
      fprintf (stderr, "Image uses compression; this is unsupported.\n\n");
      exit (EXIT_FAILURE);
   }
   fprintf (stderr, "\n");
   fprintf (stderr, "   Image Size:            %5d bytes\n", image_size);
   fprintf (stderr, "   Horizontal Resolution: %5d pixels/meter\n", hres);
   fprintf (stderr, "   Vertical Resolution:   %5d pixels/meter\n", vres);
   fprintf (stderr, "   Number of Colors:      %5d", num_colors);
   if (num_colors != true_colors) {
      fprintf (stderr, " --> %d", true_colors);
   }
   fputc ('\n', stderr);
```

```
    fprintf (stderr, "   Important Colors:      %5d", important_colors);
    if (important_colors == 0)
        fprintf (stderr, " (all colors are important)");
    fprintf (stderr, "\n\n");
}  /* if (verbose) */

/*
    Print Color Table information for images with pallettized colors.
*/
if (bits_per_pixel <= 8) {
    for (i = 0; i < 2; i++) {
        color_map [i][0] = get_bytes (infp, 1);
        color_map [i][1] = get_bytes (infp, 1);
        color_map [i][2] = get_bytes (infp, 1);
        color_map [i][3] = get_bytes (infp, 1);
    }
    /* Skip remaining color table entries if more than 2 */
    while (i < true_colors) {
        (void) get_bytes (infp, 4);
        i++;
    }

    if (color_map [0][0] >= 128) image_xor = 0xFF;  /* Invert colors */
}

if (verbose) {
    fprintf (stderr, "Color Palette [R, G, B, %s] Values:\n",
            (dib_length <= 40) ? "reserved" : "Alpha");
    for (i = 0; i < 2; i++) {
        fprintf (stderr, "%7d: [", i);
        fprintf (stderr, "%3d,",   color_map [i][0] & 0xFF);
        fprintf (stderr, "%3d,",   color_map [i][1] & 0xFF);
        fprintf (stderr, "%3d,",   color_map [i][2] & 0xFF);
        fprintf (stderr, "%3d]\n", color_map [i][3] & 0xFF);
    }
    if (image_xor == 0xFF) fprintf (stderr, "Will Invert Colors.\n");
    fputc ('\n', stderr);

}  /* if (verbose) */


/*
    Check format before writing output file.
*/
if (image_width != 560 && image_width != 576) {
    fprintf (stderr, "\nUnsupported image width: %d\n", image_width);
    fprintf (stderr, "Width should be 560 or 576 pixels.\n\n");
    exit (EXIT_FAILURE);
}

if (image_height != 544) {
    fprintf (stderr, "\nUnsupported image height: %d\n", image_height);
    fprintf (stderr, "Height should be 544 pixels.\n\n");
    exit (EXIT_FAILURE);
}

if (num_planes != 1) {
    fprintf (stderr, "\nUnsupported number of planes: %d\n", num_planes);
    fprintf (stderr, "Number of planes should be 1.\n\n");
    exit (EXIT_FAILURE);
}

if (bits_per_pixel != 1) {
    fprintf (stderr, "\nUnsupported number of bits per pixel: %d\n",
            bits_per_pixel);
    fprintf (stderr, "Bits per pixel should be 1.\n\n");
    exit (EXIT_FAILURE);
}

if (compression_method != 0 && compression_method != 11) {
    fprintf (stderr, "\nUnsupported compression method: %d\n",
            compression_method);
    fprintf (stderr, "Compression method should be 1 or 11.\n\n");
    exit (EXIT_FAILURE);
}

if (true_colors != 2) {
    fprintf (stderr, "\nUnsupported number of colors: %d\n", true_colors);
    fprintf (stderr, "Number of colors should be 2.\n\n");
    exit (EXIT_FAILURE);
}
```

```


   /*
      If we made it this far, things look okay, so write out
      the standard header for image conversion.
   */
   for (i = 0; i < 62; i++) fputc (standard_header[i], outfp);


   /*
      Image Data.  Each row must be a multiple of 4 bytes, with
      padding at the end of each row if necessary.
   */
   k = 0;  /* byte number within the binary image */
   for (i = 0; i < 544; i++) {
      /*
         If original image is 560 pixels wide (not 576), add
         2 white bytes at beginning of row.
      */
      if (image_width == 560) {  /* Insert 2 white bytes */
         image_bytes[k++] = 0xFF;
         image_bytes[k++] = 0xFF;
      }
      for (j = 0; j < 70; j++) {  /* Copy next 70 bytes */
         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
      }
      /*
         If original image is 560 pixels wide (not 576), skip
         2 padding bytes at end of row in file because we inserted
         2 white bytes at the beginning of the row.
      */
      if (image_width == 560) {
         (void) get_bytes (infp, 2);
      }
      else {  /* otherwise, next 2 bytes are part of the image so copy them */
         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
      }
   }


   /*
      Change the image to match the unihex2bmp.c format if original wasn't
   */
   if (image_width == 560) {
      regrid (image_bytes);
   }

   for (i = 0; i < 544 * 576 / 8; i++) {
      fputc (image_bytes[i], outfp);
   }


   /*
      Wrap up.
   */
   fclose (infp);
   fclose (outfp);

   exit (EXIT_SUCCESS);
}
```

#### 5.9.3.3 regrid()

void regrid (
    unsigned ∗ image_bytes )

After reading in the image, shift it.

This function adjusts the input image from an original PNG file to match unihex2bmp.c format.

Parameters

| | | |
|---|---|---|
| in,out | image_bytes | The pixels in an image. |

Definition at line 514 of file unibmpbump.c.

```
00514 {
00515     int i, j, k;  /* loop variables */
00516     int offset;
00517     unsigned glyph_row; /* one grid row of 32 pixels */
00518     unsigned last_pixel; /* last pixel in a byte, to preserve */
00519
00520     /* To insert ”00” after ”U+” at top of image */
00521     char zero_pattern[16] = {
00522         0x00, 0x00, 0x00, 0x00, 0x18, 0x24, 0x42, 0x42,
00523         0x42, 0x42, 0x42, 0x42, 0x24, 0x18, 0x00, 0x00
00524     };
00525
00526     /* This is the horizontal grid pattern on glyph boundaries */
00527     unsigned hgrid[72] = {
00528        /*  0 */ 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xfe,
00529        /*  8 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00530        /* 16 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00531        /* 24 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00532        /* 32 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00533        /* 40 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00534        /* 48 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00535        /* 56 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00536        /* 64 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00
00537     };
00538
00539
00540     /*
00541        First move ”U+” left and insert ”00” after it.
00542     */
00543     j = 15; /* rows are written bottom to top, so we'll decrement j */
00544     for (i = 543 - 8; i > 544 - 24; i--) {
00545        offset = 72 * i;
00546        image_bytes [offset + 0] = image_bytes [offset + 2];
00547        image_bytes [offset + 1] = image_bytes [offset + 3];
00548        image_bytes [offset + 2] = image_bytes [offset + 4];
00549        image_bytes [offset + 3] = image_bytes [offset + 4] =
00550           ~zero_pattern[15 - j--] & 0xFF;
00551     }
00552
00553     /*
00554        Now move glyph bitmaps to the right by 8 pixels.
00555     */
00556     for (i = 0; i < 16; i++) { /* for each glyph row */
00557        for (j = 0; j < 16; j++) { /* for each glyph column */
00558           /* set offset to lower left-hand byte of next glyph */
00559           offset = (32 * 72 * i) + (9 * 72) + (4 * j) + 8;
00560           for (k = 0; k < 16; k++) { /* for each glyph row */
00561              glyph_row = (image_bytes [offset + 0] « 24) |
00562                          (image_bytes [offset + 1] « 16) |
00563                          (image_bytes [offset + 2] «  8) |
00564                          (image_bytes [offset + 3]);
00565              last_pixel = glyph_row & 1;  /* preserve border */
00566              glyph_row »= 4;
00567              glyph_row &=  0x0FFFFFFE;
00568              /* Set left 4 pixels to white and preserve last pixel */
00569              glyph_row |=  0xF0000000 | last_pixel;
00570              image_bytes [offset + 3] = glyph_row & 0xFF;
00571              glyph_row »= 8;
00572              image_bytes [offset + 2] = glyph_row & 0xFF;
00573              glyph_row »= 8;
00574              image_bytes [offset + 1] = glyph_row & 0xFF;
00575              glyph_row »= 8;
00576              image_bytes [offset + 0] = glyph_row & 0xFF;
00577              offset += 72;  /* move up to next row in current glyph */
00578           }
00579        }
00580     }
00581
```

```
    /* Replace horizontal grid with unihex2bmp.c grid */
    for (i = 0; i <= 16; i++) {
       offset = 32 * 72 * i;
       for (j = 0; j < 72; j++) {
          image_bytes [offset + j] = hgrid [j];
       }
    }

    return;
}
```

## 5.10 unibmpbump.c

Go to the documentation of this file.

```
/**
   @file unibmpbump.c

   @brief unibmpbump - Adjust a Microsoft bitmap (.bmp) file that
                       was created by unihex2png but converted to .bmp

   @author Paul Hardy, unifoundry <at> unifoundry.com

   @copyright Copyright (C) 2019 Paul Hardy

   This program shifts the glyphs in a bitmap file to adjust an
   original PNG file that was saved in BMP format.  This is so the
   result matches the format of a unihex2bmp image.  This conversion
   then lets unibmp2hex decode the result.

   Synopsis: unibmpbump [-iin_file.bmp] [-oout_file.bmp]
*/
/*
   LICENSE:

      This program is free software: you can redistribute it and/or modify
      it under the terms of the GNU General Public License as published by
      the Free Software Foundation, either version 2 of the License, or
      (at your option) any later version.

      This program is distributed in the hope that it will be useful,
      but WITHOUT ANY WARRANTY; without even the implied warranty of
      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
      GNU General Public License for more details.

      You should have received a copy of the GNU General Public License
      along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define VERSION "1.0"   ///< Version of this program

#define MAX_COMPRESSION_METHOD 13   ///< Maximum supported compression method


/**
   @brief The main function.

   @param[in] argc The count of command line arguments.
   @param[in] argv Pointer to array of command line arguments.
   @return This program exits with status EXIT_SUCCESS.
*/
int main (int argc, char *argv[]) {

   /*
      Values preserved from file header (first 14 bytes).
   */
   char file_format[3];        /* "BM" for original Windows format            */
   unsigned filesize;          /* size of file in bytes                       */
   unsigned char rsvd_hdr[4];  /* 4 reserved bytes                            */
   unsigned image_start;       /* byte offset of image in file                */

   /*
      Values preserved from Device Independent Bitmap (DIB) Header.
```

```

   The DIB fields below are in the standard 40-byte header.  Version
   4 and version 5 headers have more information, mainly for color
   information.  That is skipped over, because a valid glyph image
   is just monochrome.
*/
int dib_length;             /* in bytes, for parsing by header version    */
int image_width = 0;        /* Signed image width                         */
int image_height = 0;       /* Signed image height                        */
int num_planes = 0;         /* number of planes; must be 1                */
int bits_per_pixel = 0;     /* for palletized color maps (< 2^16 colors)  */
/*
   The following fields are not in the original spec, so initialize
   them to 0 so we can correctly parse an original file format.
*/
int compression_method=0;  /* 0 --> uncompressed RGB/monochrome           */
int image_size = 0;         /* 0 is a valid size if no compression        */
int hres = 0;               /* image horizontal resolution                */
int vres = 0;               /* image vertical resolution                  */
int num_colors = 0;         /* Number of colors for pallettized images    */
int important_colors = 0;  /* Number of significant colors (0 or 2)       */

int true_colors = 0;        /* interpret num_colors, which can equal 0    */

/*
   Color map.  This should be a monochrome file, so only two
   colors are stored.
*/
unsigned char color_map[2][4]; /* two of R, G, B, and possibly alpha  */

/*
   The monochrome image bitmap, stored as a vector 544 rows by
   72*8 columns.
*/
unsigned image_bytes[544*72];

/*
   Flags for conversion & I/O.
*/
int verbose      = 0;       /* Whether to print file info on stderr      */
unsigned image_xor = 0x00; /* Invert (= 0xFF) if color 0 is not black  */

/*
   Temporary variables.
*/
int i, j, k;            /* loop variables */

/* Compression type, for parsing file */
char *compression_type[MAX_COMPRESSION_METHOD + 1] = {
   "BI_RGB",            /*  0 */
   "BI_RLE8",           /*  1 */
   "BI_RLE4",           /*  2 */
   "BI_BITFIELDS",      /*  3 */
   "BI_JPEG",           /*  4 */
   "BI_PNG",            /*  5 */
   "BI_ALPHABITFIELDS", /*  6 */
   "", "", "", "",      /*  7 - 10 */
   "BI_CMYK",           /* 11 */
   "BI_CMYKRLE8",       /* 12 */
   "BI_CMYKRLE4",       /* 13 */
};

/* Standard unihex2bmp.c header for BMP image */
unsigned standard_header [62] = {
   /*  0 */ 0x42, 0x4d, 0x3e, 0x99, 0x00, 0x00, 0x00, 0x00,
   /*  8 */ 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x28, 0x00,
   /* 16 */ 0x00, 0x00, 0x40, 0x02, 0x00, 0x00, 0x20, 0x02,
   /* 24 */ 0x00, 0x00, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00,
   /* 32 */ 0x00, 0x00, 0x00, 0x99, 0x00, 0x00, 0xc4, 0x0e,
   /* 40 */ 0x00, 0x00, 0xc4, 0x0e, 0x00, 0x00, 0x00, 0x00,
   /* 48 */ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   /* 56 */ 0x00, 0x00, 0xff, 0xff, 0xff, 0x00
};

unsigned get_bytes (FILE *, int);
void     regrid    (unsigned *);

char *infile="", *outfile="";  /* names of input and output files         */
FILE *infp, *outfp;            /* file pointers of input and output files */

/*
```

```
   Process command line arguments.
*/
if (argc > 1) {
   for (i = 1; i < argc; i++) {
      if (argv[i][0] == '-') {  /* this is an option argument */
         switch (argv[i][1]) {
            case 'i':  /* name of input file */
               infile = &argv[i][2];
               break;
            case 'o':  /* name of output file */
               outfile = &argv[i][2];
               break;
            case 'v':  /* verbose output */
               verbose = 1;
               break;
            case 'V':  /* print version & quit */
               fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
               exit (EXIT_SUCCESS);
               break;
            case '-':  /* see if "--verbose" */
               if (strcmp (argv[i], "--verbose") == 0) {
                  verbose = 1;
               }
               else if (strcmp (argv[i], "--version") == 0) {
                  fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
                  exit (EXIT_SUCCESS);
               }
               break;
            default:   /* if unrecognized option, print list and exit */
               fprintf (stderr, "\nSyntax:\n\n");
               fprintf (stderr, "   unibmpbump ");
               fprintf (stderr, "-i<Input_File> -o<Output_File>\n\n");
               fprintf (stderr, "-v or --verbose gives verbose output");
               fprintf (stderr, " on stderr\n\n");
               fprintf (stderr, "-V or --version prints version");
               fprintf (stderr, " on stderr and exits\n\n");
               fprintf (stderr, "\nExample:\n\n");
               fprintf (stderr, "   unibmpbump -iuni0101.bmp");
               fprintf (stderr, " -onew-uni0101.bmp\n\n");
               exit (EXIT_SUCCESS);
         }
      }
   }
}

/*
   Make sure we can open any I/O files that were specified before
   doing anything else.
*/
if (strlen (infile) > 0) {
   if ((infp = fopen (infile, "r")) == NULL) {
      fprintf (stderr, "Error: can't open %s for input.\n", infile);
      exit (EXIT_FAILURE);
   }
}
else {
   infp = stdin;
}
if (strlen (outfile) > 0) {
   if ((outfp = fopen (outfile, "w")) == NULL) {
      fprintf (stderr, "Error: can't open %s for output.\n", outfile);
      exit (EXIT_FAILURE);
   }
}
else {
   outfp = stdout;
}


/* Read bitmap file header */
file_format[0] = get_bytes (infp, 1);
file_format[1] = get_bytes (infp, 1);
file_format[2] = '\0';  /* Terminate string with null */

/* Read file size */
filesize = get_bytes (infp, 4);

/* Read Reserved bytes */
rsvd_hdr[0] = get_bytes (infp, 1);
rsvd_hdr[1] = get_bytes (infp, 1);
rsvd_hdr[2] = get_bytes (infp, 1);
```

```
rsvd_hdr[3] = get_bytes (infp, 1);

/* Read Image Offset Address within file */
image_start = get_bytes (infp, 4);

/*
   See if this looks like a valid image file based on
   the file header first two bytes.
*/
if (strncmp (file_format, "BM", 2) != 0) {
   fprintf (stderr, "\nInvalid file format: not file type \"BM\".\n\n");
   exit (EXIT_FAILURE);
}

if (verbose) {
   fprintf (stderr, "\nFile Header:\n");
   fprintf (stderr, "   File Type:   \"%s\"\n", file_format);
   fprintf (stderr, "   File Size:   %d bytes\n", filesize);
   fprintf (stderr, "   Reserved:   ");
   for (i = 0; i < 4; i++) fprintf (stderr, " 0x%02X", rsvd_hdr[i]);
   fputc ('\n', stderr);
   fprintf (stderr, "   Image Start: %d. = 0x%02X = 0%05o\n\n",
            image_start, image_start, image_start);
}  /* if (verbose) */

/*
   Device Independent Bitmap (DIB) Header: bitmap information header
   ("BM" format file DIB Header is 12 bytes long).
*/
dib_length = get_bytes (infp, 4);

/*
   Parse one of three versions of Device Independent Bitmap (DIB) format:

        Length  Format
        ------  ------
           12   BITMAPCOREHEADER
           40   BITMAPINFOHEADER
          108   BITMAPV4HEADER
          124   BITMAPV5HEADER
*/
if (dib_length == 12) { /* BITMAPCOREHEADER format -- UNTESTED */
   image_width    = get_bytes (infp, 2);
   image_height   = get_bytes (infp, 2);
   num_planes     = get_bytes (infp, 2);
   bits_per_pixel = get_bytes (infp, 2);
}
else if (dib_length >= 40) { /* BITMAPINFOHEADER format or later */
   image_width = get_bytes (infp, 4);
   image_height       = get_bytes (infp, 4);
   num_planes         = get_bytes (infp, 2);
   bits_per_pixel     = get_bytes (infp, 2);
   compression_method = get_bytes (infp, 4);  /* BI_BITFIELDS */
   image_size         = get_bytes (infp, 4);
   hres               = get_bytes (infp, 4);
   vres               = get_bytes (infp, 4);
   num_colors         = get_bytes (infp, 4);
   important_colors   = get_bytes (infp, 4);

   /* true_colors is true number of colors in image */
   if (num_colors == 0)
      true_colors = 1 << bits_per_pixel;
   else
      true_colors = num_colors;

   /*
      If dib_length > 40, the format is BITMAPV4HEADER or
      BITMAPV5HEADER.  As this program is only designed
      to handle a monochrome image, we can ignore the rest
      of the header but must read past the remaining bytes.
   */
   for (i = 40; i < dib_length; i++) (void)get_bytes (infp, 1);
}

if (verbose) {
   fprintf (stderr, "Device Independent Bitmap (DIB) Header:\n");
   fprintf (stderr, "   DIB Length:  %9d bytes (version = ", dib_length);

   if      (dib_length ==  12) fprintf (stderr, "\"BITMAPCOREHEADER\")\n");
   else if (dib_length ==  40) fprintf (stderr, "\"BITMAPINFOHEADER\")\n");
   else if (dib_length == 108) fprintf (stderr, "\"BITMAPV4HEADER\")\n");
```

```
    else if (dib_length == 124) fprintf (stderr, "\"BITMAPV5HEADER\")\n");
    else fprintf (stderr, "unknown)");
    fprintf (stderr, "   Bitmap Width:   %6d pixels\n", image_width);
    fprintf (stderr, "   Bitmap Height:  %6d pixels\n", image_height);
    fprintf (stderr, "   Color Planes:   %6d\n",        num_planes);
    fprintf (stderr, "   Bits per Pixel: %6d\n",        bits_per_pixel);
    fprintf (stderr, "   Compression Method: %2d --> ", compression_method);
    if (compression_method <= MAX_COMPRESSION_METHOD) {
       fprintf (stderr, "%s", compression_type [compression_method]);
    }
    /*
       Supported compression method values:
            0 --> uncompressed RGB
           11 --> uncompressed CMYK
    */
    if (compression_method == 0 || compression_method == 11) {
       fprintf (stderr, " (no compression)");
    }
    else {
       fprintf (stderr, "Image uses compression; this is unsupported.\n\n");
       exit (EXIT_FAILURE);
    }
    fprintf (stderr, "\n");
    fprintf (stderr, "   Image Size:            %5d bytes\n", image_size);
    fprintf (stderr, "   Horizontal Resolution: %5d pixels/meter\n", hres);
    fprintf (stderr, "   Vertical Resolution:   %5d pixels/meter\n", vres);
    fprintf (stderr, "   Number of Colors:      %5d", num_colors);
    if (num_colors != true_colors) {
       fprintf (stderr, " --> %d", true_colors);
    }
    fputc ('\n', stderr);
    fprintf (stderr, "   Important Colors:      %5d", important_colors);
    if (important_colors == 0)
       fprintf (stderr, " (all colors are important)");
    fprintf (stderr, "\n\n");
 }  /* if (verbose) */

 /*
    Print Color Table information for images with pallettized colors.
 */
 if (bits_per_pixel <= 8) {
    for (i = 0; i < 2; i++) {
       color_map [i][0] = get_bytes (infp, 1);
       color_map [i][1] = get_bytes (infp, 1);
       color_map [i][2] = get_bytes (infp, 1);
       color_map [i][3] = get_bytes (infp, 1);
    }
    /* Skip remaining color table entries if more than 2 */
    while (i < true_colors) {
       (void) get_bytes (infp, 4);
       i++;
    }

    if (color_map [0][0] >= 128) image_xor = 0xFF;  /* Invert colors */
 }

 if (verbose) {
    fprintf (stderr, "Color Palette [R, G, B, %s] Values:\n",
             (dib_length <= 40) ? "reserved" : "Alpha");
    for (i = 0; i < 2; i++) {
       fprintf (stderr, "%7d: [", i);
       fprintf (stderr, "%3d,",   color_map [i][0] & 0xFF);
       fprintf (stderr, "%3d,",   color_map [i][1] & 0xFF);
       fprintf (stderr, "%3d,",   color_map [i][2] & 0xFF);
       fprintf (stderr, "%3d]\n", color_map [i][3] & 0xFF);
    }
    if (image_xor == 0xFF) fprintf (stderr, "Will Invert Colors.\n");
    fputc ('\n', stderr);

 }  /* if (verbose) */


 /*
    Check format before writing output file.
 */
 if (image_width != 560 && image_width != 576) {
    fprintf (stderr, "\nUnsupported image width: %d\n", image_width);
    fprintf (stderr, "Width should be 560 or 576 pixels.\n\n");
    exit (EXIT_FAILURE);
 }

```

```
if (image_height != 544) {
   fprintf (stderr, "\nUnsupported image height: %d\n", image_height);
   fprintf (stderr, "Height should be 544 pixels.\n\n");
   exit (EXIT_FAILURE);
}

if (num_planes != 1) {
   fprintf (stderr, "\nUnsupported number of planes: %d\n", num_planes);
   fprintf (stderr, "Number of planes should be 1.\n\n");
   exit (EXIT_FAILURE);
}

if (bits_per_pixel != 1) {
   fprintf (stderr, "\nUnsupported number of bits per pixel: %d\n",
            bits_per_pixel);
   fprintf (stderr, "Bits per pixel should be 1.\n\n");
   exit (EXIT_FAILURE);
}

if (compression_method != 0 && compression_method != 11) {
   fprintf (stderr, "\nUnsupported compression method: %d\n",
            compression_method);
   fprintf (stderr, "Compression method should be 1 or 11.\n\n");
   exit (EXIT_FAILURE);
}

if (true_colors != 2) {
   fprintf (stderr, "\nUnsupported number of colors: %d\n", true_colors);
   fprintf (stderr, "Number of colors should be 2.\n\n");
   exit (EXIT_FAILURE);
}


/*
   If we made it this far, things look okay, so write out
   the standard header for image conversion.
*/
for (i = 0; i < 62; i++) fputc (standard_header[i], outfp);


/*
   Image Data.  Each row must be a multiple of 4 bytes, with
   padding at the end of each row if necessary.
*/
k = 0;  /* byte number within the binary image */
for (i = 0; i < 544; i++) {
   /*
      If original image is 560 pixels wide (not 576), add
      2 white bytes at beginning of row.
   */
   if (image_width == 560) {  /* Insert 2 white bytes */
      image_bytes[k++] = 0xFF;
      image_bytes[k++] = 0xFF;
   }
   for (j = 0; j < 70; j++) {  /* Copy next 70 bytes */
      image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
   }
   /*
      If original image is 560 pixels wide (not 576), skip
      2 padding bytes at end of row in file because we inserted
      2 white bytes at the beginning of the row.
   */
   if (image_width == 560) {
      (void) get_bytes (infp, 2);
   }
   else {  /* otherwise, next 2 bytes are part of the image so copy them */
      image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
      image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
   }
}


/*
   Change the image to match the unihex2bmp.c format if original wasn't
*/
if (image_width == 560) {
   regrid (image_bytes);
}

for (i = 0; i < 544 * 576 / 8; i++) {
   fputc (image_bytes[i], outfp);
```

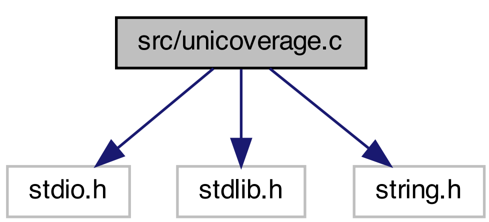
```
    }


    /*
       Wrap up.
    */
    fclose (infp);
    fclose (outfp);

    exit (EXIT_SUCCESS);
}


/**
   @brief Get from 1 to 4 bytes, inclusive, from input file.

   @param[in] infp Pointer to input file.
   @param[in] nbytes Number of bytes to read, from 1 to 4, inclusive.
   @return The unsigned 1 to 4 bytes in machine native endian format.
*/
unsigned get_bytes (FILE *infp, int nbytes) {
   int i;
   unsigned char inchar[4];
   unsigned inword;

   for (i = 0; i < nbytes; i++) {
      if (fread (&inchar[i], 1, 1, infp) != 1) {
         inchar[i] = 0;
      }
   }
   for (i = nbytes; i < 4; i++) inchar[i] = 0;

   inword = ((inchar[3] & 0xFF) « 24) | ((inchar[2] & 0xFF) « 16) |
            ((inchar[1] & 0xFF) «  8) |  (inchar[0] & 0xFF);

   return inword;
}


/**
   @brief After reading in the image, shift it.

   This function adjusts the input image from an original PNG file
   to match unihex2bmp.c format.

   @param[in,out] image_bytes The pixels in an image.
*/
void regrid (unsigned *image_bytes) {
   int i, j, k;  /* loop variables */
   int offset;
   unsigned glyph_row; /* one grid row of 32 pixels */
   unsigned last_pixel; /* last pixel in a byte, to preserve */

   /* To insert ”00” after ”U+” at top of image */
   char zero_pattern[16] = {
       0x00, 0x00, 0x00, 0x00, 0x18, 0x24, 0x42, 0x42,
       0x42, 0x42, 0x42, 0x42, 0x24, 0x18, 0x00, 0x00
   };

   /* This is the horizontal grid pattern on glyph boundaries */
   unsigned hgrid[72] = {
      /*  0 */ 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xfe,
      /*  8 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 16 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 24 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 32 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 40 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 48 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 56 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 64 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00
   };


   /*
      First move ”U+” left and insert ”00” after it.
   */
   j = 15; /* rows are written bottom to top, so we'll decrement j */
   for (i = 543 - 8; i > 544 - 24; i--) {
      offset = 72 * i;
      image_bytes [offset + 0] = image_bytes [offset + 2];
      image_bytes [offset + 1] = image_bytes [offset + 3];
```

```
        image_bytes [offset + 2] = image_bytes [offset + 4];
        image_bytes [offset + 3] = image_bytes [offset + 4] =
           ~zero_pattern[15 - j--] & 0xFF;
    }

    /*
        Now move glyph bitmaps to the right by 8 pixels.
    */
    for (i = 0; i < 16; i++) { /* for each glyph row */
        for (j = 0; j < 16; j++) { /* for each glyph column */
           /* set offset to lower left-hand byte of next glyph */
           offset = (32 * 72 * i) + (9 * 72) + (4 * j) + 8;
           for (k = 0; k < 16; k++) { /* for each glyph row */
              glyph_row = (image_bytes [offset + 0] « 24) |
                          (image_bytes [offset + 1] « 16) |
                          (image_bytes [offset + 2] «  8) |
                          (image_bytes [offset + 3]);
              last_pixel = glyph_row & 1;  /* preserve border */
              glyph_row »= 4;
              glyph_row &=  0x0FFFFFFE;
              /* Set left 4 pixels to white and preserve last pixel */
              glyph_row |=  0xF0000000 | last_pixel;
              image_bytes [offset + 3] = glyph_row & 0xFF;
              glyph_row »= 8;
              image_bytes [offset + 2] = glyph_row & 0xFF;
              glyph_row »= 8;
              image_bytes [offset + 1] = glyph_row & 0xFF;
              glyph_row »= 8;
              image_bytes [offset + 0] = glyph_row & 0xFF;
              offset += 72;  /* move up to next row in current glyph */
           }
        }
    }

    /* Replace horizontal grid with unihex2bmp.c grid */
    for (i = 0; i <= 16; i++) {
        offset = 32 * 72 * i;
        for (j = 0; j < 72; j++) {
           image_bytes [offset + j] = hgrid [j];
        }
    }

    return;
}
```

## 5.11 src/unicoverage.c File Reference

unicoverage - Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Include dependency graph for unicoverage.c:



## Macros

- #define MAXBUF 256
  Maximum input line length - 1.

## Functions

- int main (int argc, char *argv[ ])
  The main function.
- int nextrange (FILE *coveragefp, int *cstart, int *cend, char *coverstring)
  Get next Unicode range.
- void print_subtotal (FILE *outfp, int print_n, int nglyphs, int cstart, int cend, char *coverstring)
  Print the subtotal for one Unicode script range.

### 5.11.1 Detailed Description

unicoverage - Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file

Author

Paul Hardy, unifoundry <at> unifoundry.com, 6 January 2008

Copyright

Copyright (C) 2008, 2013 Paul Hardy

Synopsis: unicoverage [-ifont_file.hex] [-ocoverage_file.txt]

This program requires the file "coverage.dat" to be present in the directory from which it is run.

Definition in file unicoverage.c.

### 5.11.2 Macro Definition Documentation

#### 5.11.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum input line length - 1.

Definition at line 57 of file unicoverage.c.

### 5.11.3 Function Documentation

#### 5.11.3.1 main()

```
int main (
        int argc,
        char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

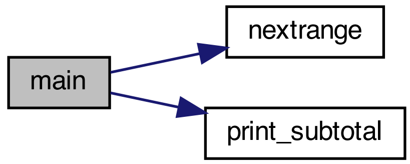
This program exits with status 0.

Definition at line 68 of file unicoverage.c.

```
{

   int       print_n=0;          /* print # of glyphs, not percentage   */
   unsigned i;                   /* loop variable                        */
   unsigned slen;                /* string length of coverage file line */
   char     inbuf[256];          /* input buffer                         */
   unsigned thischar;            /* the current character                */

   char *infile="", *outfile="";  /* names of input and output files        */
   FILE *infp, *outfp;           /* file pointers of input and output files    */
   FILE *coveragefp;             /* file pointer to coverage.dat file          */
   int cstart, cend;             /* current coverage start and end code points */
   char coverstring[MAXBUF];     /* description of current coverage range      */
   int nglyphs;                  /* number of glyphs in this section           */
   int nextrange();              /* to get next range & name of Unicode glyphs */

   void print_subtotal (FILE *outfp, int print_n, int nglyphs,
                        int cstart, int cend, char *coverstring);

   if ((coveragefp = fopen ("coverage.dat", "r")) == NULL) {
      fprintf (stderr, "\nError: data file \"coverage.dat\" not found.\n\n");
      exit (0);
   }

   if (argc > 1) {
      for (i = 1; i < argc; i++) {
         if (argv[i][0] == '-') {  /* this is an option argument */
            switch (argv[i][1]) {
               case 'i':  /* name of input file */
                  infile = &argv[i][2];
                  break;
               case 'n':  /* print number of glyphs instead of percentage */
                  print_n = 1;
               case 'o':  /* name of output file */
                  outfile = &argv[i][2];
                  break;
               default:   /* if unrecognized option, print list and exit */
                  fprintf (stderr, "\nSyntax:\n\n");
                  fprintf (stderr, "   %s -p<Unicode_Page> ", argv[0]);
                  fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
                  exit (1);
            }
         }
```

```
        }
    }
    /*
        Make sure we can open any I/O files that were specified before
        doing anything else.
    */
    if (strlen (infile) > 0) {
        if ((infp = fopen (infile, "r")) == NULL) {
            fprintf (stderr, "Error: can't open %s for input.\n", infile);
            exit (1);
        }
    }
    else {
        infp = stdin;
    }
    if (strlen (outfile) > 0) {
        if ((outfp = fopen (outfile, "w")) == NULL) {
            fprintf (stderr, "Error: can't open %s for output.\n", outfile);
            exit (1);
        }
    }
    else {
        outfp = stdout;
    }

    /*
        Print header row.
    */
    if (print_n) {
        fprintf (outfp, "# Glyphs      Range          Script\n");
        fprintf (outfp, "--------      -----          ------\n");
    }
    else {
        fprintf (outfp, "Covered      Range          Script\n");
        fprintf (outfp, "-------      -----          ------\n\n");
    }

    slen = nextrange (coveragefp, &cstart, &cend, coverstring);
    nglyphs = 0;

    /*
        Read in the glyphs in the file
    */
    while (slen != 0 && fgets (inbuf, MAXBUF-1, infp) != NULL) {
        sscanf (inbuf, "%x", &thischar);

        /* Read a character beyond end of current script. */
        while (cend < thischar && slen != 0) {
            print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);

            /* start new range total */
            slen = nextrange (coveragefp, &cstart, &cend, coverstring);
            nglyphs = 0;
        }
        nglyphs++;
    }

    print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);

    exit (0);
}
```

Here is the call graph for this function:




#### 5.11.3.2 nextrange()

```
int nextrange (
        FILE * coveragefp,
        int * cstart,
        int * cend,
        char * coverstring )
```

Get next Unicode range.

This function reads the next Unicode script range to count its glyph coverage.

Parameters

| | | |
|---|---|---|
| in | coveragefp | File pointer to Unicode script range data file. |
| in | cstart | Starting code point in current Unicode script range. |

Parameters

| | | |
|---|---|---|
| in | cend | Ending code point in current Unicode script range. |
| out | coverstring | String containing <cstart>-<cend> substring. |

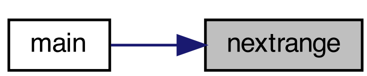
Returns

Length of the last string read, or 0 for end of file.

Definition at line 187 of file unicoverage.c.

```
{
   int i;
   static char inbuf[MAXBUF];
   int retval;         /* the return value */

   retval = 0;

   do {
      if (fgets (inbuf, MAXBUF-1, coveragefp) != NULL) {
         retval = strlen (inbuf);
         if ((inbuf[0] >= '0' && inbuf[0] <= '9') ||
             (inbuf[0] >= 'A' && inbuf[0] <= 'F') ||
             (inbuf[0] >= 'a' && inbuf[0] <= 'f')) {
            sscanf (inbuf, "%x-%x", cstart, cend);
            i = 0;
            while (inbuf[i] != ' ') i++;  /* find first blank */
            while (inbuf[i] == ' ') i++;  /* find next non-blank */
            strncpy (coverstring, &inbuf[i], MAXBUF);
         }
         else retval = 0;
      }
      else retval = 0;
   } while (retval == 0 && !feof (coveragefp));

   return (retval);
}
```

Here is the caller graph for this function:



#### 5.11.3.3 print_subtotal()

```
void print_subtotal (
            FILE * outfp,
            int print_n,
            int nglyphs,
            int cstart,
            int cend,
            char * coverstring )
```

Print the subtotal for one Unicode script range.

Parameters

| | | |
|---|---|---|
| in | outfp | Pointer to output file. |
| in | print_n | 1 = print number of glyphs, 0 = print percentage. |
| in | nglyphs | Number of glyphs in current range. |
| in | cstart | Starting code point for current range. |
| in | cend | Ending code point for current range. |

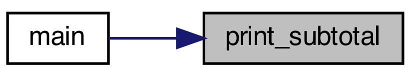
Parameters

| | | |
|---|---|---|
| in | coverstring | Character string of "<cstart>-<cend>". |

Definition at line 228 of file unicoverage.c.

```
                                   {

   /* print old range total */
   if (print_n) {  /* Print number of glyphs, not percentage */
      fprintf (outfp, " %6d ", nglyphs);
   }
   else {
      fprintf (outfp, " %5.1f%%", 100.0*nglyphs/(1+cend-cstart));
   }

   if (cend < 0x10000)
      fprintf (outfp, "  U+%04X..U+%04X   %s",
               cstart, cend, coverstring);
   else
      fprintf (outfp, " U+%05X..U+%05X  %s",
               cstart, cend, coverstring);

   return;
}
```

Here is the caller graph for this function:




## 5.12 unicoverage.c

Go to the documentation of this file.

```
/**
   @file unicoverage.c

   @brief unicoverage - Show the coverage of Unicode plane scripts
                        for a GNU Unifont hex glyph file

   @author Paul Hardy, unifoundry <at> unifoundry.com, 6 January 2008

   @copyright Copyright (C) 2008, 2013 Paul Hardy

   Synopsis: unicoverage [-ifont_file.hex] [-ocoverage_file.txt]

   This program requires the file "coverage.dat" to be present
   in the directory from which it is run.
*/
/*
   LICENSE:

      This program is free software: you can redistribute it and/or modify
```

```
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

/*
  2016 (Paul Hardy): Modified in Unifont 9.0.01 release to remove non-existent
  "-p" option and empty example from help printout.

  2018 (Paul Hardy): Modified to cover entire Unicode range, not just Plane 0.

  11 May 2019: [Paul Hardy] changed strcpy function call to strlcpy
  for better error handling.

  31 May 2019: [Paul Hardy] replaced strlcpy call with strncpy
  for compilation on more systems.

  4 June 2022: [Paul Hardy] Adjusted column spacing for better alignment
  of Unicode Plane 1-15 scripts.  Added "-n" option to print number of
  glyphs in each range instead of percent coverage.

  18 September 2022: [Paul Hardy] in nextrange function, initialize retval.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#define MAXBUF 256   ///< Maximum input line length - 1


/**
  @brief The main function.

  @param[in] argc The count of command line arguments.
  @param[in] argv Pointer to array of command line arguments.
  @return This program exits with status 0.
*/
int
main (int argc, char *argv[])
{

  int      print_n=0;        /* print # of glyphs, not percentage   */
  unsigned i;                /* loop variable                       */
  unsigned slen;             /* string length of coverage file line */
  char     inbuf[256];       /* input buffer                        */
  unsigned thischar;         /* the current character               */

  char *infile="", *outfile="";  /* names of input and output files        */
  FILE *infp, *outfp;           /* file pointers of input and output files */
  FILE *coveragefp;             /* file pointer to coverage.dat file       */
  int cstart, cend;             /* current coverage start and end code points */
  char coverstring[MAXBUF];     /* description of current coverage range     */
  int nglyphs;                  /* number of glyphs in this section        */
  int nextrange();              /* to get next range & name of Unicode glyphs */

  void print_subtotal (FILE *outfp, int print_n, int nglyphs,
                       int cstart, int cend, char *coverstring);

  if ((coveragefp = fopen ("coverage.dat", "r")) == NULL) {
     fprintf (stderr, "\nError: data file \"coverage.dat\" not found.\n\n");
     exit (0);
  }

  if (argc > 1) {
     for (i = 1; i < argc; i++) {
        if (argv[i][0] == '-') {  /* this is an option argument */
           switch (argv[i][1]) {
              case 'i':  /* name of input file */
                 infile = &argv[i][2];
                 break;
              case 'n':  /* print number of glyphs instead of percentage */
```

```
                  print_n = 1;
               case 'o':  /* name of output file */
                  outfile = &argv[i][2];
                  break;
               default:   /* if unrecognized option, print list and exit */
                  fprintf (stderr, "\nSyntax:\n\n");
                  fprintf (stderr, "   %s -p<Unicode_Page> ", argv[0]);
                  fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
                  exit (1);
            }
         }
      }
   }
   /*
      Make sure we can open any I/O files that were specified before
      doing anything else.
   */
   if (strlen (infile) > 0) {
      if ((infp = fopen (infile, "r")) == NULL) {
         fprintf (stderr, "Error: can't open %s for input.\n", infile);
         exit (1);
      }
   }
   else {
      infp = stdin;
   }
   if (strlen (outfile) > 0) {
      if ((outfp = fopen (outfile, "w")) == NULL) {
         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
         exit (1);
      }
   }
   else {
      outfp = stdout;
   }

   /*
      Print header row.
   */
   if (print_n) {
      fprintf (outfp, "# Glyphs      Range         Script\n");
      fprintf (outfp, "--------      -----         ------\n");
   }
   else {
      fprintf (outfp, "Covered      Range         Script\n");
      fprintf (outfp, "-------      -----         ------\n\n");
   }

   slen = nextrange (coveragefp, &cstart, &cend, coverstring);
   nglyphs = 0;

   /*
      Read in the glyphs in the file
   */
   while (slen != 0 && fgets (inbuf, MAXBUF-1, infp) != NULL) {
      sscanf (inbuf, "%x", &thischar);

      /* Read a character beyond end of current script. */
      while (cend < thischar && slen != 0) {
         print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);

         /* start new range total */
         slen = nextrange (coveragefp, &cstart, &cend, coverstring);
         nglyphs = 0;
      }
      nglyphs++;
   }

   print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);

   exit (0);
}

/**
   @brief Get next Unicode range.

   This function reads the next Unicode script range to count its
   glyph coverage.

   @param[in] coveragefp File pointer to Unicode script range data file.
   @param[in] cstart Starting code point in current Unicode script range.
```

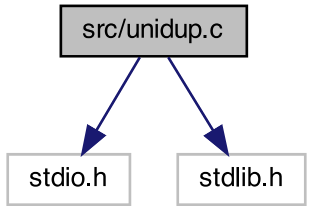
```c
   @param[in] cend Ending code point in current Unicode script range.
   @param[out] coverstring String containing <cstart>-<cend> substring.
   @return Length of the last string read, or 0 for end of file.
*/
int
nextrange (FILE *coveragefp,
           int *cstart, int *cend,
           char *coverstring)
{
   int i;
   static char inbuf[MAXBUF];
   int retval;         /* the return value */

   retval = 0;

   do {
      if (fgets (inbuf, MAXBUF-1, coveragefp) != NULL) {
         retval = strlen (inbuf);
         if ((inbuf[0] >= '0' && inbuf[0] <= '9') ||
             (inbuf[0] >= 'A' && inbuf[0] <= 'F') ||
             (inbuf[0] >= 'a' && inbuf[0] <= 'f')) {
            sscanf (inbuf, "%x-%x", cstart, cend);
            i = 0;
            while (inbuf[i] != ' ') i++;  /* find first blank */
            while (inbuf[i] == ' ') i++;  /* find next non-blank */
            strncpy (coverstring, &inbuf[i], MAXBUF);
         }
         else retval = 0;
      }
      else retval = 0;
   } while (retval == 0 && !feof (coveragefp));

   return (retval);
}


/**
   @brief Print the subtotal for one Unicode script range.

   @param[in] outfp Pointer to output file.
   @param[in] print_n 1 = print number of glyphs, 0 = print percentage.
   @param[in] nglyphs Number of glyphs in current range.
   @param[in] cstart Starting code point for current range.
   @param[in] cend Ending code point for current range.
   @param[in] coverstring Character string of "<cstart>-<cend>".
*/
void print_subtotal (FILE *outfp, int print_n, int nglyphs,
                     int cstart, int cend, char *coverstring) {

   /* print old range total */
   if (print_n) {  /* Print number of glyphs, not percentage */
      fprintf (outfp, " %6d ", nglyphs);
   }
   else {
      fprintf (outfp, " %5.1f%%", 100.0*nglyphs/(1+cend-cstart));
   }

   if (cend < 0x10000)
      fprintf (outfp, "  U+%04X..U+%04X   %s",
               cstart, cend, coverstring);
   else
      fprintf (outfp, " U+%05X..U+%05X  %s",
               cstart, cend, coverstring);

   return;
}
```

## 5.13 src/unidup.c File Reference

unidup - Check for duplicate code points in sorted unifont.hex file

#include <stdio.h>
#include <stdlib.h>

Include dependency graph for unidup.c:




## Macros

- #define MAXBUF 256

  Maximum input line length - 1.

## Functions

- int main (int argc, char **argv)

  The main function.

### 5.13.1 Detailed Description

unidup - Check for duplicate code points in sorted unifont.hex file

**Author**

Paul Hardy, unifoundry <at> unifoundry.com, December 2007

**Copyright**

Copyright (C) 2007, 2008, 2013 Paul Hardy

This program reads a sorted list of glyphs in Unifont .hex format and prints duplicate code points on stderr if any were detected.

Synopsis: unidup < unifont_file.hex

[Hopefully there won't be any output!]

Definition in file unidup.c.

### 5.13.2 Macro Definition Documentation

#### 5.13.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum input line length - 1.

Definition at line 37 of file unidup.c.

### 5.13.3 Function Documentation

#### 5.13.3.1 main()

```
int main (
        int argc,
        char ** argv )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status 0.

Definition at line 48 of file unidup.c.

```
{

   int ix, iy;
   char inbuf[MAXBUF];
   char *infile;   /* the input file name */
   FILE *infilefp; /* file pointer to input file */

   if (argc > 1) {
      infile = argv[1];
      if ((infilefp = fopen (infile, "r")) == NULL) {
         fprintf (stderr, "\nERROR: Can't open file %s\n\n", infile);
         exit (EXIT_FAILURE);
      }
   }
   else {
      infilefp = stdin;
   }

   ix = -1;

   while (fgets (inbuf, MAXBUF-1, infilefp) != NULL) {
      sscanf (inbuf, "%X", &iy);
      if (ix == iy) fprintf (stderr, "Duplicate code point: %04X\n", ix);
      else ix = iy;
   }
   exit (0);
}
```

## 5.14 unidup.c

Go to the documentation of this file.

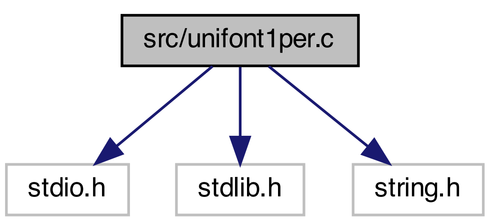
```
/**
   @file unidup.c

   @brief unidup - Check for duplicate code points in sorted unifont.hex file

   @author Paul Hardy, unifoundry <at> unifoundry.com, December 2007

   @copyright Copyright (C) 2007, 2008, 2013 Paul Hardy

   This program reads a sorted list of glyphs in Unifont .hex format
   and prints duplicate code points on stderr if any were detected.

   Synopsis: unidup < unifont_file.hex

             [Hopefully there won't be any output!]
*/
/*
   LICENSE:

      This program is free software: you can redistribute it and/or modify
      it under the terms of the GNU General Public License as published by
      the Free Software Foundation, either version 2 of the License, or
      (at your option) any later version.

      This program is distributed in the hope that it will be useful,
      but WITHOUT ANY WARRANTY; without even the implied warranty of
      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
      GNU General Public License for more details.

      You should have received a copy of the GNU General Public License
      along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#include <stdio.h>
#include <stdlib.h>

#define MAXBUF 256   ///< Maximum input line length - 1


/**
   @brief The main function.

   @param[in] argc The count of command line arguments.
```

```
   @param[in] argv Pointer to array of command line arguments.
   @return This program exits with status 0.
*/
int
main (int argc, char **argv)
{

   int ix, iy;
   char inbuf[MAXBUF];
   char *infile;   /* the input file name */
   FILE *infilefp; /* file pointer to input file */

   if (argc > 1) {
      infile = argv[1];
      if ((infilefp = fopen (infile, "r")) == NULL) {
         fprintf (stderr, "\nERROR: Can't open file %s\n\n", infile);
         exit (EXIT_FAILURE);
      }
   }
   else {
      infilefp = stdin;
   }

   ix = -1;

   while (fgets (inbuf, MAXBUF-1, infilefp) != NULL) {
      sscanf (inbuf, "%X", &iy);
      if (ix == iy) fprintf (stderr, "Duplicate code point: %04X\n", ix);
      else ix = iy;
   }
   exit (0);
}
```

## 5.15 src/unifont1per.c File Reference

unifont1per - Read a Unifont .hex file from standard input and produce one glyph per ".bmp" bitmap file as output

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Include dependency graph for unifont1per.c:




### Macros

- #define MAXSTRING 266
- #define MAXFILENAME 20

## Functions

- int main ()
  The main function.

### 5.15.1 Detailed Description

unifont1per - Read a Unifont .hex file from standard input and produce one glyph per ".bmp" bitmap file as output

Author

Paul Hardy, unifoundry <at> unifoundry.com, December 2016

Copyright



Each glyph is 16 pixels tall, and can be 8, 16, 24, or 32 pixels wide. The width of each output graphic file is determined automatically by the width of each Unifont hex representation.

This program creates files of the form "U+<codepoint>.bmp", 1 per glyph.

Synopsis: unifont1per < unifont.hex

Definition in file unifont1per.c.

### 5.15.2 Macro Definition Documentation

#### 5.15.2.1 MAXFILENAME

#define MAXFILENAME 20

Maximum size of a filename of the form "U+%06X.bmp".

Definition at line 60 of file unifont1per.c.

#### 5.15.2.2 MAXSTRING

#define MAXSTRING 266

Maximum size of an input line in a Unifont .hex file - 1.

Definition at line 57 of file unifont1per.c.

## 5.15.3 Function Documentation

### 5.15.3.1 main()

int main ( )

The main function.

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 69 of file unifont1per.c.

```
00069 {
00070
00071    int i; /* loop variable */
00072
00073    /*
00074       Define bitmap header bytes
00075    */
00076    unsigned char header [62] = {
00077       /*
00078          Bitmap File Header -- 14 bytes
00079       */
00080       'B', 'M',        /* Signature        */
00081       0x7E, 0, 0, 0,   /* File Size        */
00082          0, 0, 0, 0,   /* Reserved         */
00083       0x3E, 0, 0, 0,   /* Pixel Array Offset */
00084
00085       /*
00086          Device Independent Bitmap Header -- 40 bytes
00087
00088          Image Width and Image Height are assigned final values
00089          based on the dimensions of each glyph.
00090       */
00091       0x28,    0,    0,    0,  /* DIB Header Size              */
00092       0x10,    0,    0,    0,  /* Image Width = 16 pixels      */
00093       0xF0, 0xFF, 0xFF, 0xFF,  /* Image Height = -16 pixels    */
00094       0x01,    0,              /* Planes                       */
00095       0x01,    0,              /* Bits Per Pixel               */
00096          0,    0,    0,    0,  /* Compression                  */
00097       0x40,    0,    0,    0,  /* Image Size                   */
00098       0x14, 0x0B,    0,    0,  /* X Pixels Per Meter = 72 dpi  */
00099       0x14, 0x0B,    0,    0,  /* Y Pixels Per Meter = 72 dpi  */
00100       0x02,    0,    0,    0,  /* Colors In Color Table        */
00101          0,    0,    0,    0,  /* Important Colors             */
00102
00103       /*
00104          Color Palette -- 8 bytes
00105       */
00106       0xFF, 0xFF, 0xFF, 0,  /* White */
00107          0,    0,    0, 0   /* Black */
00108    };
00109
00110    char instring[MAXSTRING]; /* input string                     */
00111    int  code_point;          /* current Unicode code point       */
00112    char glyph[MAXSTRING];    /* bitmap string for this glyph     */
00113    int  glyph_height=16;     /* for now, fixed at 16 pixels high */
00114    int  glyph_width;         /* 8, 16, 24, or 32 pixels wide     */
00115    char filename[MAXFILENAME];/* name of current output file     */
00116    FILE *outfp;              /* file pointer to current output file */
00117
00118    int string_index;  /* pointer into hexadecimal glyph string */
00119    int nextbyte;      /* next set of 8 bits to print out       */
00120
00121    /* Repeat for each line in the input stream */
00122    while (fgets (instring, MAXSTRING - 1, stdin) != NULL) {
00123       /* Read next Unicode ASCII hexadecimal format glyph description */
```

```
      sscanf (instring, "%X:%s", &code_point, glyph);
      /* Calculate width of a glyph in pixels; 4 bits per ASCII hex digit */
      glyph_width = strlen (glyph) / (glyph_height / 4);
      snprintf (filename, MAXFILENAME, "U+%06X.bmp", code_point);
      header [18] =  glyph_width;  /* bitmap width */
      header [22] = -glyph_height; /* negative height --> draw top to bottom */
      if ((outfp = fopen (filename, "w")) != NULL) {
         for (i = 0; i < 62; i++) fputc (header[i], outfp);
         /*
            Bitmap, with each row padded with zeroes if necessary
            so each row is four bytes wide.  (Each row must end
            on a four-byte boundary, and four bytes is the maximum
            possible row length for up to 32 pixels in a row.)
         */
         string_index = 0;
         for (i = 0; i < glyph_height; i++) {
            /* Read 2 ASCII hexadecimal digits (1 byte of output pixels) */
            sscanf (&glyph[string_index], "%2X", &nextbyte);
            string_index += 2;
            fputc (nextbyte, outfp);  /* write out the 8 pixels    */
            if (glyph_width <= 8) {   /* pad row with 3 zero bytes */
               fputc (0x00, outfp); fputc (0x00, outfp); fputc (0x00, outfp);
            }
            else { /* get 8 more pixels */
               sscanf (&glyph[string_index], "%2X", &nextbyte);
               string_index += 2;
               fputc (nextbyte, outfp);  /* write out the 8 pixels    */
               if (glyph_width <= 16) {  /* pad row with 2 zero bytes */
                  fputc (0x00, outfp); fputc (0x00, outfp);
               }
               else { /* get 8 more pixels */
                  sscanf (&glyph[string_index], "%2X", &nextbyte);
                  string_index += 2;
                  fputc (nextbyte, outfp);  /* write out the 8 pixels   */
                  if (glyph_width <= 24) {  /* pad row with 1 zero byte */
                     fputc (0x00, outfp);
                  }
                  else { /* get 8 more pixels */
                     sscanf (&glyph[string_index], "%2X", &nextbyte);
                     string_index += 2;
                     fputc (nextbyte, outfp); /* write out the 8 pixels */
                  }  /* glyph is 32 pixels wide */
               }  /* glyph is 24 pixels wide */
            }  /* glyph is 16 pixels wide */
         }  /* glyph is 8 pixels wide */

         fclose (outfp);
      }
   }

   exit (EXIT_SUCCESS);
}
```

## 5.16 unifont1per.c

Go to the documentation of this file.

```
/**
   @file unifont1per.c

   @brief unifont1per - Read a Unifont .hex file from standard input and
          produce one glyph per ".bmp" bitmap file as output

   @author Paul Hardy, unifoundry <at> unifoundry.com, December 2016

   @copyright Copyright (C) 2016, 2017 Paul Hardy

   Each glyph is 16 pixels tall, and can be 8, 16, 24,
   or 32 pixels wide.  The width of each output graphic
   file is determined automatically by the width of each
   Unifont hex representation.

   This program creates files of the form "U+<codepoint>.bmp", 1 per glyph.

   Synopsis: unifont1per < unifont.hex
*/
```

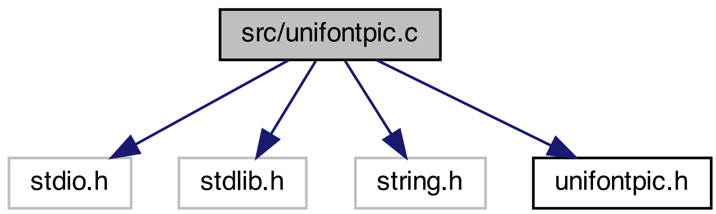
```
/*
   LICENSE:

      This program is free software: you can redistribute it and/or modify
      it under the terms of the GNU General Public License as published by
      the Free Software Foundation, either version 2 of the License, or
      (at your option) any later version.

      This program is distributed in the hope that it will be useful,
      but WITHOUT ANY WARRANTY; without even the implied warranty of
      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
      GNU General Public License for more details.

      You should have received a copy of the GNU General Public License
      along with this program.  If not, see <http://www.gnu.org/licenses/>.

   Example:

      mkdir my-bmp
      cd my-bmp
      unifont1per < ../glyphs.hex

*/

/*
   11 May 2019 [Paul Hardy]:
      - Changed sprintf function call to snprintf for writing
    "filename" character string.
      - Defined MAXFILENAME to hold size of "filename" array
    for snprintf function call.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/** Maximum size of an input line in a Unifont .hex file - 1. */
#define MAXSTRING 266

/** Maximum size of a filename of the form "U+%06X.bmp". */
#define MAXFILENAME 20


/**
   @brief The main function.

   @return This program exits with status EXIT_SUCCESS.
*/
int
main () {

   int i; /* loop variable */

   /*
      Define bitmap header bytes
   */
   unsigned char header [62] = {
      /*
         Bitmap File Header -- 14 bytes
      */
      'B', 'M',        /* Signature          */
      0x7E, 0, 0, 0,  /* File Size          */
         0, 0, 0, 0,  /* Reserved           */
      0x3E, 0, 0, 0,  /* Pixel Array Offset */

      /*
         Device Independent Bitmap Header -- 40 bytes

         Image Width and Image Height are assigned final values
         based on the dimensions of each glyph.
      */
      0x28,    0,    0,    0,  /* DIB Header Size                */
      0x10,    0,    0,    0,  /* Image Width = 16 pixels        */
      0xF0, 0xFF, 0xFF, 0xFF,  /* Image Height = -16 pixels      */
      0x01,    0,              /* Planes                         */
      0x01,    0,              /* Bits Per Pixel                 */
         0,    0,    0,    0,  /* Compression                    */
      0x40,    0,    0,    0,  /* Image Size                     */
      0x14, 0x0B,    0,    0,  /* X Pixels Per Meter = 72 dpi    */
      0x14, 0x0B,    0,    0,  /* Y Pixels Per Meter = 72 dpi    */
      0x02,    0,    0,    0,  /* Colors In Color Table          */
```

```
      0,    0,    0,    0,  /* Important Colors                */

   /*
      Color Palette -- 8 bytes
   */
   0xFF, 0xFF, 0xFF, 0,  /* White */
      0,    0,    0, 0   /* Black */
};

char instring[MAXSTRING]; /* input string                        */
int  code_point;           /* current Unicode code point           */
char glyph[MAXSTRING];     /* bitmap string for this glyph         */
int  glyph_height=16;      /* for now, fixed at 16 pixels high     */
int  glyph_width;          /* 8, 16, 24, or 32 pixels wide         */
char filename[MAXFILENAME];/* name of current output file          */
FILE *outfp;               /* file pointer to current output file */

int string_index;  /* pointer into hexadecimal glyph string */
int nextbyte;      /* next set of 8 bits to print out        */

/* Repeat for each line in the input stream */
while (fgets (instring, MAXSTRING - 1, stdin) != NULL) {
   /* Read next Unifont ASCII hexadecimal format glyph description */
   sscanf (instring, "%X:%s", &code_point, glyph);
   /* Calculate width of a glyph in pixels; 4 bits per ASCII hex digit */
   glyph_width = strlen (glyph) / (glyph_height / 4);
   snprintf (filename, MAXFILENAME, "U+%06X.bmp", code_point);
   header [18] =  glyph_width;  /* bitmap width */
   header [22] = -glyph_height; /* negative height --> draw top to bottom */
   if ((outfp = fopen (filename, "w")) != NULL) {
      for (i = 0; i < 62; i++) fputc (header[i], outfp);
      /*
         Bitmap, with each row padded with zeroes if necessary
         so each row is four bytes wide.  (Each row must end
         on a four-byte boundary, and four bytes is the maximum
         possible row length for up to 32 pixels in a row.)
      */
      string_index = 0;
      for (i = 0; i < glyph_height; i++) {
         /* Read 2 ASCII hexadecimal digits (1 byte of output pixels) */
         sscanf (&glyph[string_index], "%2X", &nextbyte);
         string_index += 2;
         fputc (nextbyte, outfp);  /* write out the 8 pixels    */
         if (glyph_width <= 8) {   /* pad row with 3 zero bytes */
            fputc (0x00, outfp); fputc (0x00, outfp); fputc (0x00, outfp);
         }
         else { /* get 8 more pixels */
            sscanf (&glyph[string_index], "%2X", &nextbyte);
            string_index += 2;
            fputc (nextbyte, outfp);  /* write out the 8 pixels    */
            if (glyph_width <= 16) {  /* pad row with 2 zero bytes */
               fputc (0x00, outfp); fputc (0x00, outfp);
            }
            else { /* get 8 more pixels */
               sscanf (&glyph[string_index], "%2X", &nextbyte);
               string_index += 2;
               fputc (nextbyte, outfp);  /* write out the 8 pixels   */
               if (glyph_width <= 24) {  /* pad row with 1 zero byte */
                  fputc (0x00, outfp);
               }
               else { /* get 8 more pixels */
                  sscanf (&glyph[string_index], "%2X", &nextbyte);
                  string_index += 2;
                  fputc (nextbyte, outfp); /* write out the 8 pixels */
               }  /* glyph is 32 pixels wide */
            }  /* glyph is 24 pixels wide */
         }  /* glyph is 16 pixels wide */
      }  /* glyph is 8 pixels wide */

      fclose (outfp);
   }
}

exit (EXIT_SUCCESS);
}
```

## 5.17 src/unifontpic.c File Reference

unifontpic - See the "Big Picture": the entire Unifont in one BMP bitmap

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "unifontpic.h"
Include dependency graph for unifontpic.c:




### Macros

- #define HDR_LEN 33

### Functions

- int main (int argc, char **argv)

    The main function.
- void output4 (int thisword)

    Output a 4-byte integer in little-endian order.
- void output2 (int thisword)

    Output a 2-byte integer in little-endian order.
- void gethex (char *instring, int plane_array[0x10000][16], int plane)

    Read a Unifont .hex-format input file from stdin.
- void genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)

    Generate the BMP output file in long format.
- void genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)

    Generate the BMP output file in wide format.

### 5.17.1 Detailed Description

unifontpic - See the "Big Picture": the entire Unifont in one BMP bitmap

Author

Paul Hardy, 2013

Copyright

Copyright (C) 2013, 2017 Paul Hardy

Definition in file unifontpic.c.

### 5.17.2 Macro Definition Documentation

#### 5.17.2.1 HDR_LEN

```
#define HDR_LEN 33
```

Define length of header string for top of chart.

Definition at line 67 of file unifontpic.c.

### 5.17.3 Function Documentation

#### 5.17.3.1 genlongbmp()

```
void genlongbmp (
        int plane_array[0x10000][16],
        int dpi,
        int tinynum,
        int plane )
```

Generate the BMP output file in long format.

This function generates the BMP output file from a bitmap parameter. This is a long bitmap, 16 glyphs wide by 4,096 glyphs tall.

Parameters

| | | |
|---|---|---|
| in | plane_array | The array of glyph bitmaps for a plane. |
| in | dpi | Dots per inch, for encoding in the BMP output file header. |
| in | tinynum | Whether to generate tiny numbers in wide grid (unused). |
| in | plane | The Unicode plane, 0..17. |

Definition at line 294 of file unifontpic.c.

```
{

    char header_string[HDR_LEN]; /* centered header              */
    char raw_header[HDR_LEN];    /* left-aligned header          */
    int header[16][16];      /* header row, for chart title */
    int hdrlen;              /* length of HEADER_STRING      */
    int startcol;            /* column to start printing header, for centering */

    unsigned leftcol[0x1000][16]; /* code point legend on left side of chart */
    int d1, d2, d3, d4;           /* digits for filling leftcol[][] legend   */
    int codept;                   /* current starting code point for legend  */
    int thisrow;                  /* glyph row currently being rendered       */
    unsigned toprow[16][16];      /* code point legend on top of chart        */
    int digitrow;        /* row we're in (0..4) for the above hexdigit digits */

    /*
       DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
    */
    int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
```

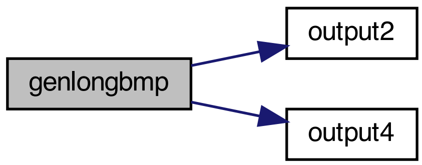
```
int ImageSize;
int FileSize;
int Width, Height; /* bitmap image width and height in pixels */
int ppm;      /* integer pixels per meter */

int i, j, k;

unsigned bytesout;

void output4(int), output2(int);

/*
   Image width and height, in pixels.

      N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
*/
Width  =   18 * 16;  /* (2 legend +   16 glyphs) * 16 pixels/glyph */
Height = 4099 * 16;  /* (1 header + 4096 glyphs) * 16 rows/glyph   */

ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */

FileSize = DataOffset + ImageSize;

/* convert dots/inch to pixels/meter */
if (dpi == 0) dpi = 96;
ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);

/*
   Generate the BMP Header
*/
putchar (’B’);
putchar (’M’);

/*
   Calculate file size:

      BMP Header + InfoHeader + Color Table + Raster Data
*/
output4 (FileSize);  /* FileSize */
output4 (0x0000); /* reserved */

/* Calculate DataOffset */
output4 (DataOffset);

/*
   InfoHeader
*/
output4 (40);         /* Size of InfoHeader                        */
output4 (Width);      /* Width of bitmap in pixels                 */
output4 (Height);     /* Height of bitmap in pixels                */
output2 (1);          /* Planes (1 plane)                          */
output2 (1);          /* BitCount (1 = monochrome)                 */
output4 (0);          /* Compression (0 = none)                    */
output4 (ImageSize);  /* ImageSize, in bytes                       */
output4 (ppm);        /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (ppm);        /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (2);          /* ColorsUsed (= 2)                          */
output4 (2);          /* ColorsImportant (= 2)                     */
output4 (0x00000000); /* black (reserved, B, G, R)                 */
output4 (0x00FFFFFF); /* white (reserved, B, G, R)                 */

/*
   Create header row bits.
*/
snprintf (raw_header, HDR_LEN, ”%s Plane %d”, HEADER_STRING, plane);
memset ((void *)header, 0, 16 * 16 * sizeof (int)); /* fill with white */
memset ((void *)header_string, ’ ’, 32 * sizeof (char)); /* 32 spaces */
header_string[32] = ’\0’;  /* null-terminated */

hdrlen = strlen (raw_header);
if (hdrlen > 32) hdrlen = 32;        /* only 32 columns to print header */
startcol = 16 - ((hdrlen + 1) » 1); /* to center header                */
/* center up to 32 chars */
memcpy (&header_string[startcol], raw_header, hdrlen);

/* Copy each letter’s bitmap from the plane_array[][] we constructed. */
/* Each glyph must be single-width, to fit two glyphs in 16 pixels */
for (j = 0; j < 16; j++) {
   for (i = 0; i < 16; i++) {
      header[i][j] =
         (ascii_bits[header_string[j+j  ] & 0x7F][i] & 0xFF00) |
```

```
            (ascii_bits[header_string[j+j+1] & 0x7F][i] » 8);
      }
   }

   /*
      Create the left column legend.
   */
   memset ((void *)leftcol, 0, 4096 * 16 * sizeof (unsigned));

   for (codept = 0x0000; codept < 0x10000; codept += 0x10) {
      d1 = (codept » 12) & 0xF; /* most significant hex digit */
      d2 = (codept »  8) & 0xF;
      d3 = (codept »  4) & 0xF;

      thisrow = codept » 4; /* rows of 16 glyphs */

      /* fill in first and second digits */
      for (digitrow = 0; digitrow < 5; digitrow++) {
         leftcol[thisrow][2 + digitrow] =
            (hexdigit[d1][digitrow] « 10) |
            (hexdigit[d2][digitrow] «  4);
      }

      /* fill in third digit */
      for (digitrow = 0; digitrow < 5; digitrow++) {
         leftcol[thisrow][9 + digitrow] = hexdigit[d3][digitrow] « 10;
      }
      leftcol[thisrow][9 + 4] |= 0xF « 4; /* underscore as 4th digit */

      for (i = 0; i < 15; i ++) {
         leftcol[thisrow][i] |= 0x00000002;      /* right border */
      }

      leftcol[thisrow][15] = 0x0000FFFE;        /* bottom border */

      if (d3 == 0xF) {                     /* 256-point boundary */
         leftcol[thisrow][15] |= 0x00FF0000;  /* longer tic mark */
      }

      if ((thisrow % 0x40) == 0x3F) {    /* 1024-point boundary */
         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
      }
   }

   /*
      Create the top row legend.
   */
   memset ((void *)toprow, 0, 16 * 16 * sizeof (unsigned));

   for (codept = 0x0; codept <= 0xF; codept++) {
      d1 = (codept » 12) & 0xF; /* most significant hex digit */
      d2 = (codept »  8) & 0xF;
      d3 = (codept »  4) & 0xF;
      d4 =  codept       & 0xF; /* least significant hex digit */

      /* fill in last digit */
      for (digitrow = 0; digitrow < 5; digitrow++) {
         toprow[6 + digitrow][codept] = hexdigit[d4][digitrow] « 6;
      }
   }

   for (j = 0; j < 16; j++) {
      /* force bottom pixel row to be white, for separation from glyphs */
      toprow[15][j] = 0x0000;
   }

   /* 1 pixel row with left-hand legend line */
   for (j = 0; j < 16; j++) {
      toprow[14][j] |= 0xFFFF;
   }

   /* 14 rows with line on left to fill out this character row */
   for (i = 13; i >= 0; i--) {
      for (j = 0; j < 16; j++) {
         toprow[i][j] |= 0x0001;
      }
   }

   /*
      Now write the raster image.

```

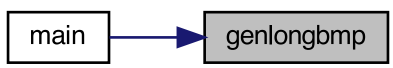
```
   XOR each byte with 0xFF because black = 0, white = 1 in BMP.
*/

/* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
for (i = 0xFFF0; i >= 0; i -= 0x10) {
   thisrow = i » 4; /* 16 glyphs per row */
   for (j = 15; j >= 0; j--) {
      /* left-hand legend */
      putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
      putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
      putchar ((~leftcol[thisrow][j] »  8) & 0xFF);
      putchar ( ~leftcol[thisrow][j]        & 0xFF);
      /* Unifont glyph */
      for (k = 0; k < 16; k++) {
         bytesout = ~plane_array[i+k][j] & 0xFFFF;
         putchar ((bytesout » 8) & 0xFF);
         putchar ( bytesout       & 0xFF);
      }
   }
}

/*
   Write the top legend.
*/
/* i == 15: bottom pixel row of header is output here */
/* left-hand legend: solid black line except for right-most pixel */
putchar (0x00);
putchar (0x00);
putchar (0x00);
putchar (0x01);
for (j = 0; j < 16; j++) {
   putchar ((~toprow[15][j] » 8) & 0xFF);
   putchar ( ~toprow[15][j]       & 0xFF);
}

putchar (0xFF);
putchar (0xFF);
putchar (0xFF);
putchar (0xFC);
for (j = 0; j < 16; j++) {
   putchar ((~toprow[14][j] » 8) & 0xFF);
   putchar ( ~toprow[14][j]       & 0xFF);
}

for (i = 13; i >= 0; i--) {
   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFD);
   for (j = 0; j < 16; j++) {
      putchar ((~toprow[i][j] » 8) & 0xFF);
      putchar ( ~toprow[i][j]       & 0xFF);
   }
}

/*
   Write the header.
*/

/* 7 completely white rows */
for (i = 7; i >= 0; i--) {
   for (j = 0; j < 18; j++) {
      putchar (0xFF);
      putchar (0xFF);
   }
}

for (i = 15; i >= 0; i--) {
   /* left-hand legend */
   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFF);
   /* header glyph */
   for (j = 0; j < 16; j++) {
      bytesout = ~header[i][j] & 0xFFFF;
      putchar ((bytesout » 8) & 0xFF);
      putchar ( bytesout       & 0xFF);
   }
}

```

```
    /* 8 completely white rows at very top */
    for (i = 7; i >= 0; i--) {
       for (j = 0; j < 18; j++) {
       putchar (0xFF);
       putchar (0xFF);
       }
    }

    return;
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.17.3.2 genwidebmp()

void genwidebmp (
    int plane_array[0x10000][16],
    int dpi,
    int tinynum,
    int plane )

Generate the BMP output file in wide format.

This function generates the BMP output file from a bitmap parameter. This is a wide bitmap, 256 glyphs wide by 256 glyphs tall.

Parameters

| | | |
|---|---|---|
| in | plane_array | The array of glyph bitmaps for a plane. |
| in | dpi | Dots per inch, for encoding in the BMP output file header. |
| in | tinynum | Whether to generate tiny numbers in 256x256 grid. |
| in | plane | The Unicode plane, 0..17. |

Definition at line 581 of file unifontpic.c.

```
{

   char header_string[257];
   char raw_header[HDR_LEN];
   int header[16][256]; /* header row, for chart title */
   int hdrlen;          /* length of HEADER_STRING */
   int startcol;        /* column to start printing header, for centering */

   unsigned leftcol[0x100][16]; /* code point legend on left side of chart */
   int d1, d2, d3, d4;          /* digits for filling leftcol[][] legend   */
   int codept;                  /* current starting code point for legend  */
   int thisrow;                 /* glyph row currently being rendered      */
   unsigned toprow[32][256];    /* code point legend on top of chart       */
   int digitrow;       /* row we're in (0..4) for the above hexdigit digits */
   int hexalpha1, hexalpha2;    /* to convert hex digits to ASCII          */

   /*
      DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
   */
   int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
   int ImageSize;
   int FileSize;
```

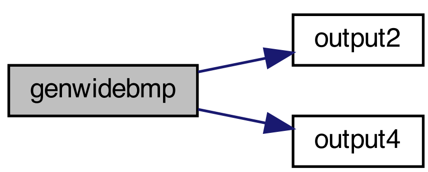
```
int Width, Height; /* bitmap image width and height in pixels */
int ppm;      /* integer pixels per meter */

int i, j, k;

unsigned bytesout;

void output4(int), output2(int);

/*
   Image width and height, in pixels.

      N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
*/
Width  = 258 * 16;  /* (           2 legend + 256 glyphs) * 16 pixels/glyph */
Height = 260 * 16;  /* (2 header + 2 legend + 256 glyphs) * 16 rows/glyph   */

ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */

FileSize = DataOffset + ImageSize;

/* convert dots/inch to pixels/meter */
if (dpi == 0) dpi = 96;
ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);

/*
   Generate the BMP Header
*/
putchar ('B');
putchar ('M');
/*
   Calculate file size:

      BMP Header + InfoHeader + Color Table + Raster Data
*/
output4 (FileSize);  /* FileSize */
output4 (0x0000); /* reserved */
/* Calculate DataOffset */
output4 (DataOffset);

/*
   InfoHeader
*/
output4 (40);         /* Size of InfoHeader                       */
output4 (Width);      /* Width of bitmap in pixels                */
output4 (Height);     /* Height of bitmap in pixels               */
output2 (1);          /* Planes (1 plane)                         */
output2 (1);          /* BitCount (1 = monochrome)                */
output4 (0);          /* Compression (0 = none)                   */
output4 (ImageSize);  /* ImageSize, in bytes                      */
output4 (ppm);        /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (ppm);        /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (2);          /* ColorsUsed (= 2)                         */
output4 (2);          /* ColorsImportant (= 2)                    */
output4 (0x00000000); /* black (reserved, B, G, R)                */
output4 (0x00FFFFFF); /* white (reserved, B, G, R)                */

/*
   Create header row bits.
*/
snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
memset ((void *)header, 0, 256 * 16 * sizeof (int)); /* fill with white */
memset ((void *)header_string, ' ', 256 * sizeof (char)); /* 256 spaces */
header_string[256] = '\0';  /* null-terminated */

hdrlen = strlen (raw_header);
/* Wide bitmap can print 256 columns, but limit to 32 columns for long bitmap. */
if (hdrlen > 32) hdrlen = 32;
startcol = 127 - ((hdrlen - 1) » 1);  /* to center header */
/* center up to 32 chars */
memcpy (&header_string[startcol], raw_header, hdrlen);

/* Copy each letter's bitmap from the plane_array[][] we constructed. */
for (j = 0; j < 256; j++) {
   for (i = 0; i < 16; i++) {
      header[i][j] = ascii_bits[header_string[j] & 0x7F][i];
   }
}

/*
   Create the left column legend.
```

```
*/
memset ((void *)leftcol, 0, 256 * 16 * sizeof (unsigned));

for (codept = 0x0000; codept < 0x10000; codept += 0x100) {
   d1 = (codept » 12) & 0xF; /* most significant hex digit */
   d2 = (codept »  8) & 0xF;

   thisrow = codept » 8; /* rows of 256 glyphs */

   /* fill in first and second digits */

   if (tinynum) { /* use 4x5 pixel glyphs */
      for (digitrow = 0; digitrow < 5; digitrow++) {
         leftcol[thisrow][6 + digitrow] =
            (hexdigit[d1][digitrow] « 10) |
            (hexdigit[d2][digitrow] «  4);
      }
   }
   else { /* bigger numbers -- use glyphs from Unifont itself */
      /* convert hexadecimal digits to ASCII equivalent */
      hexalpha1 = d1 < 0xA ? '0' + d1 : 'A' + d1 - 0xA;
      hexalpha2 = d2 < 0xA ? '0' + d2 : 'A' + d2 - 0xA;

      for (i = 0 ; i < 16; i++) {
         leftcol[thisrow][i] =
            (ascii_bits[hexalpha1][i] « 2) |
            (ascii_bits[hexalpha2][i] » 6);
      }
   }

   for (i = 0; i < 15; i ++) {
      leftcol[thisrow][i] |= 0x00000002;      /* right border */
   }

   leftcol[thisrow][15] = 0x0000FFFE;        /* bottom border */

   if (d2 == 0xF) {                     /* 4096-point boundary */
      leftcol[thisrow][15] |= 0x00FF0000;  /* longer tic mark */
   }

   if ((thisrow % 0x40) == 0x3F) {    /* 16,384-point boundary */
      leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
   }
}

/*
   Create the top row legend.
*/
memset ((void *)toprow, 0, 32 * 256 * sizeof (unsigned));

for (codept = 0x00; codept <= 0xFF; codept++) {
   d3 = (codept »  4) & 0xF;
   d4 =  codept       & 0xF; /* least significant hex digit */

   if (tinynum) {
      for (digitrow = 0; digitrow < 5; digitrow++) {
         toprow[16 + 6 + digitrow][codept] =
            (hexdigit[d3][digitrow] « 10) |
            (hexdigit[d4][digitrow] «  4);
      }
   }
   else {
      /* convert hexadecimal digits to ASCII equivalent */
      hexalpha1 = d3 < 0xA ? '0' + d3 : 'A' + d3 - 0xA;
      hexalpha2 = d4 < 0xA ? '0' + d4 : 'A' + d4 - 0xA;
      for (i = 0 ; i < 16; i++) {
         toprow[14 + i][codept] =
            (ascii_bits[hexalpha1][i]     ) |
            (ascii_bits[hexalpha2][i] » 7);
      }
   }
}

for (j = 0; j < 256; j++) {
   /* force bottom pixel row to be white, for separation from glyphs */
   toprow[16 + 15][j] = 0x0000;
}

/* 1 pixel row with left-hand legend line */
for (j = 0; j < 256; j++) {
   toprow[16 + 14][j] |= 0xFFFF;
```

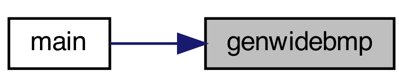
```
}

/* 14 rows with line on left to fill out this character row */
for (i = 13; i >= 0; i--) {
   for (j = 0; j < 256; j++) {
      toprow[16 + i][j] |= 0x0001;
   }
}

/* Form the longer tic marks in top legend */
for (i = 8; i < 16; i++) {
   for (j = 0x0F; j < 0x100; j += 0x10) {
      toprow[i][j] |= 0x0001;
   }
}

/*
   Now write the raster image.

   XOR each byte with 0xFF because black = 0, white = 1 in BMP.
*/

/* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
for (i = 0xFF00; i >= 0; i -= 0x100) {
   thisrow = i » 8; /* 256 glyphs per row */
   for (j = 15; j >= 0; j--) {
      /* left-hand legend */
      putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
      putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
      putchar ((~leftcol[thisrow][j] »  8) & 0xFF);
      putchar ( ~leftcol[thisrow][j]       & 0xFF);
      /* Unifont glyph */
      for (k = 0x00; k < 0x100; k++) {
         bytesout = ~plane_array[i+k][j] & 0xFFFF;
         putchar ((bytesout » 8) & 0xFF);
         putchar ( bytesout      & 0xFF);
      }
   }
}

/*
   Write the top legend.
*/
/* i == 15: bottom pixel row of header is output here */
/* left-hand legend: solid black line except for right-most pixel */
putchar (0x00);
putchar (0x00);
putchar (0x00);
putchar (0x01);
for (j = 0; j < 256; j++) {
   putchar ((~toprow[16 + 15][j] » 8) & 0xFF);
   putchar ( ~toprow[16 + 15][j]      & 0xFF);
}

putchar (0xFF);
putchar (0xFF);
putchar (0xFF);
putchar (0xFC);
for (j = 0; j < 256; j++) {
   putchar ((~toprow[16 + 14][j] » 8) & 0xFF);
   putchar ( ~toprow[16 + 14][j]      & 0xFF);
}

for (i = 16 + 13; i >= 0; i--) {
   if (i >= 8) { /* make vertical stroke on right */
      putchar (0xFF);
      putchar (0xFF);
      putchar (0xFF);
      putchar (0xFD);
   }
   else { /* all white */
      putchar (0xFF);
      putchar (0xFF);
      putchar (0xFF);
      putchar (0xFF);
   }
   for (j = 0; j < 256; j++) {
      putchar ((~toprow[i][j] » 8) & 0xFF);
      putchar ( ~toprow[i][j]      & 0xFF);
   }
}
```

```
    /*
       Write the header.
    */

    /* 8 completely white rows */
    for (i = 7; i >= 0; i--) {
       for (j = 0; j < 258; j++) {
          putchar (0xFF);
          putchar (0xFF);
       }
    }

    for (i = 15; i >= 0; i--) {
       /* left-hand legend */
       putchar (0xFF);
       putchar (0xFF);
       putchar (0xFF);
       putchar (0xFF);
       /* header glyph */
       for (j = 0; j < 256; j++) {
          bytesout = ~header[i][j] & 0xFFFF;
          putchar ((bytesout » 8) & 0xFF);
          putchar ( bytesout       & 0xFF);
       }
    }

    /* 8 completely white rows at very top */
    for (i = 7; i >= 0; i--) {
       for (j = 0; j < 258; j++) {
       putchar (0xFF);
       putchar (0xFF);
       }
    }

    return;
}
```

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.17.3.3 gethex()

void gethex (
char ∗ instring,
int plane_array[0x10000][16],
int plane )

Read a Unifont .hex-format input file from stdin.

Each glyph can be 2, 4, 6, or 8 ASCII hexadecimal digits wide. Glyph height is fixed at 16 pixels.

Parameters

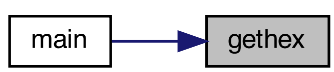
| | | |
|---|---|---|
| in | instring | One line from a Unifont .hex-format file. |
| in,out | plane_array | Bitmap for this plane, one bitmap row per element. |
| in | plane | The Unicode plane, 0..17. |

Definition at line 215 of file unifontpic.c.

```
{
   char *bitstring;  /* pointer into instring for glyph bitmap */
   int i;        /* loop variable                           */
   int codept;  /* the Unicode code point of the current glyph */
   int glyph_plane; /* Unicode plane of current glyph          */
   int ndigits; /* number of ASCII hexadecimal digits in glyph */
   int bytespl; /* bytes per line of pixels in a glyph        */
   int temprow; /* 1 row of a quadruple-width glyph           */
   int newrow;  /* 1 row of double-width output pixels        */
   unsigned bitmask; /* to mask off 2 bits of long width glyph */

   /*
      Read each input line and place its glyph into the bit array.
   */
   sscanf (instring, "%X", &codept);
   glyph_plane = codept » 16;
   if (glyph_plane == plane) {
      codept &= 0xFFFF;  /* array index will only have 16 bit address */
      /* find the colon separator */
      for (i = 0; (i < 9) && (instring[i] != ':'); i++);
      i++; /* position past it */
      bitstring = &instring[i];
```

```
      ndigits = strlen (bitstring);
      /* don't count '\n' at end of line if present */
      if (bitstring[ndigits - 1] == '\n') ndigits--;
      bytespl = ndigits » 5;  /* 16 rows per line, 2 digits per byte */

      if (bytespl >= 1 && bytespl <= 4) {
         for (i = 0; i < 16; i++) { /* 16 rows per glyph */
            /* Read correct number of hexadecimal digits given glyph width */
            switch (bytespl) {
               case 1: sscanf (bitstring, "%2X", &temprow);
                       bitstring += 2;
                       temprow «= 8; /* left-justify single-width glyph */
                       break;
               case 2: sscanf (bitstring, "%4X", &temprow);
                       bitstring += 4;
                       break;
               /* cases 3 and 4 widths will be compressed by 50% (see below) */
               case 3: sscanf (bitstring, "%6X", &temprow);
                       bitstring += 6;
                       temprow «= 8; /* left-justify */
                       break;
               case 4: sscanf (bitstring, "%8X", &temprow);
                       bitstring += 8;
                       break;
            }  /* switch on number of bytes per row */
            /* compress glyph width by 50% if greater than double-width */
            if (bytespl > 2) {
               newrow = 0x0000;
               /* mask off 2 bits at a time to convert each pair to 1 bit out */
               for (bitmask = 0xC0000000; bitmask != 0; bitmask »= 2) {
                  newrow «= 1;
                  if ((temprow & bitmask) != 0) newrow |= 1;
               }
               temprow = newrow;
            }  /* done conditioning glyphs beyond double-width */
            plane_array[codept][i] = temprow;  /* store glyph bitmap for output */
         }  /* for each row */
      }  /* if 1 to 4 bytes per row/line */
   }  /* if this is the plane we are seeking */

   return;
}
```

Here is the caller graph for this function:




#### 5.17.3.4 main()

int main (
    int argc,
    char ** argv )

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

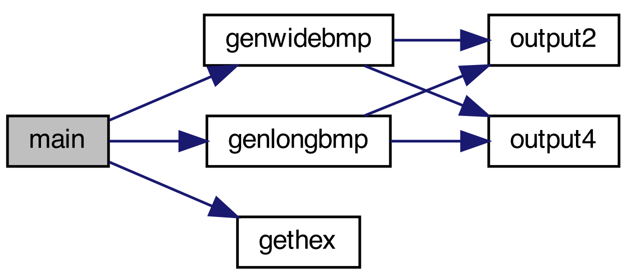
This program exits with status EXIT_SUCCESS.

Definition at line 87 of file unifontpic.c.

```
{
   /* Input line buffer */
   char instring[MAXSTRING];

   /* long and dpi are set from command-line options */
   int wide=1; /* =1 for a 256x256 grid, =0 for a 16x4096 grid */
   int dpi=96; /* change for 256x256 grid to fit paper if desired */
   int tinynum=0; /* whether to use tiny labels for 256x256 grid */

   int i, j; /* loop variables */

   int plane=0;      /* Unicode plane, 0..17; Plane 0 is default */
   /* 16 pixel rows for each of 65,536 glyphs in a Unicode plane */
   int plane_array[0x10000][16];

   void gethex();
   void genlongbmp();
   void genwidebmp();

   if (argc > 1) {
      for (i = 1; i < argc; i++) {
         if (strncmp (argv[i],"-l",2) == 0) { /* long display */
           wide = 0;
         }
         else if (strncmp (argv[i],"-d",2) == 0) {
            dpi = atoi (&argv[i][2]); /* dots/inch specified on command line */
         }
         else if (strncmp (argv[i],"-t",2) == 0) {
            tinynum = 1;
         }
         else if (strncmp (argv[i],"-P",2) == 0) {
            /* Get Unicode plane */
            for (j = 2; argv[i][j] != '\0'; j++) {
               if (argv[i][j] < '0' || argv[i][j] > '9') {
                  fprintf (stderr,
                           "ERROR: Specify Unicode plane as decimal number.\n\n");
                  exit (EXIT_FAILURE);
               }
            }
            plane = atoi (&argv[i][2]); /* Unicode plane, 0..17 */
            if (plane < 0 || plane > 17) {
               fprintf (stderr,
                        "ERROR: Plane out of Unicode range [0,17].\n\n");
```

```
                exit (EXIT_FAILURE);
            }
        }
    }
}


/*
   Initialize the ASCII bitmap array for chart titles
*/
for (i = 0; i < 128; i++) {
    gethex (ascii_hex[i], plane_array, 0); /* convert Unifont hexadecimal string to bitmap */
    for (j = 0; j < 16; j++) ascii_bits[i][j] = plane_array[i][j];
}


/*
   Read in the Unifont hex file to render from standard input
*/
memset ((void *)plane_array, 0, 0x10000 * 16 * sizeof (int));
while (fgets (instring, MAXSTRING, stdin) != NULL) {
    gethex (instring, plane_array, plane); /* read .hex input file and fill plane_array with glyph data */
}  /* while not EOF */


/*
   Write plane_array glyph data to BMP file as wide or long bitmap.
*/
if (wide) {
    genwidebmp (plane_array, dpi, tinynum, plane);
}
else {
    genlongbmp (plane_array, dpi, tinynum, plane);
}

exit (EXIT_SUCCESS);
}
```

Here is the call graph for this function:




### 5.17.3.5 output2()

void output2 (
    int thisword )

Output a 2-byte integer in little-endian order.

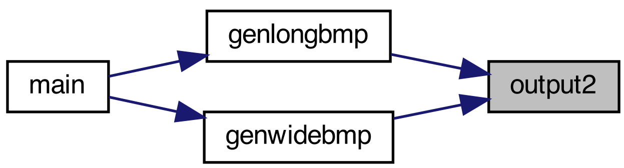
Parameters

| | | |
|---|---|---|
| in | thisword | The 2-byte integer to output as binary data. |

Definition at line 194 of file unifontpic.c.

```
00195 {
00196
00197     putchar ( thisword        & 0xFF);
00198     putchar ((thisword » 8) & 0xFF);
00199
00200     return;
00201 }
```

Here is the caller graph for this function:




#### 5.17.3.6 output4()

void output4 (
    int thisword )

Output a 4-byte integer in little-endian order.

Parameters

| | | |
|---|---|---|
| in | thisword | The 4-byte integer to output as binary data. |

Definition at line 176 of file unifontpic.c.

```
{

   putchar ( thisword        & 0xFF);
   putchar ((thisword »  8) & 0xFF);
   putchar ((thisword » 16) & 0xFF);
   putchar ((thisword » 24) & 0xFF);

   return;
}
```

Here is the caller graph for this function:


## 5.18 unifontpic.c

Go to the documentation of this file.

```
/**
   @file unifontpic.c

   @brief unifontpic - See the "Big Picture": the entire Unifont
                     in one BMP bitmap

   @author Paul Hardy, 2013

   @copyright Copyright (C) 2013, 2017 Paul Hardy
*/
/*
   LICENSE:

      This program is free software: you can redistribute it and/or modify
      it under the terms of the GNU General Public License as published by
      the Free Software Foundation, either version 2 of the License, or
      (at your option) any later version.

      This program is distributed in the hope that it will be useful,
      but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

/*
  11 June 2017 [Paul Hardy]:
     -  Modified to take glyphs that are 24 or 32 pixels wide and
        compress them horizontally by 50%.

  8 July 2017 [Paul Hardy]:
     - Modified to print Unifont charts above Unicode Plane 0.
     - Adds "-P" option to specify Unicode plane in decimal,
       as "-P0" through "-P17".  Omitting this argument uses
       plane 0 as the default.
     - Appends Unicode plane number to chart title.
     - Reads in "unifontpic.h", which was added mainly to
       store ASCII chart title glyphs in an embedded array
       rather than requiring these ASCII glyphs to be in
       the ".hex" file that is read in for the chart body
       (which was the case previously, when all that was
       able to print was Unicode place 0).
     - Fixes truncated header in long bitmap format, making
       the long chart title glyphs single-spaced.  This leaves
       room for the Unicode plane to appear even in the narrow
       chart title of the "long" format chart.  The wide chart
       title still has double-spaced ASCII glyphs.
     - Adjusts centering of title on long and wide charts.

  11 May 2019 [Paul Hardy]:
     - Changed strncpy calls to memcpy.
     - Added "HDR_LEN" to define length of header string
   for use in snprintf function call.
   - Changed sprintf function calls to snprintf function
   calls for writing chart header string.
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "unifontpic.h"

/** Define length of header string for top of chart. */
#define HDR_LEN 33


/*
   Stylistic Note:

   Many variables in this program use multiple words scrunched
   together, with each word starting with an upper-case letter.
   This is only done to match the canonical field names in the
   Windows Bitmap Graphics spec.
*/

/**
   @brief The main function.

   @param[in] argc The count of command line arguments.
   @param[in] argv Pointer to array of command line arguments.
   @return This program exits with status EXIT_SUCCESS.
*/
int
main (int argc, char **argv)
{
   /* Input line buffer */
   char instring[MAXSTRING];

   /* long and dpi are set from command-line options */
   int wide=1; /* =1 for a 256x256 grid, =0 for a 16x4096 grid */
   int dpi=96; /* change for 256x256 grid to fit paper if desired */
   int tinynum=0; /* whether to use tiny labels for 256x256 grid */

   int i, j; /* loop variables */

   int plane=0;      /* Unicode plane, 0..17; Plane 0 is default */
   /* 16 pixel rows for each of 65,536 glyphs in a Unicode plane */
   int plane_array[0x10000][16];
```

```

   void gethex();
   void genlongbmp();
   void genwidebmp();

   if (argc > 1) {
      for (i = 1; i < argc; i++) {
         if (strncmp (argv[i],"-l",2) == 0) { /* long display */
           wide = 0;
         }
         else if (strncmp (argv[i],"-d",2) == 0) {
            dpi = atoi (&argv[i][2]); /* dots/inch specified on command line */
         }
         else if (strncmp (argv[i],"-t",2) == 0) {
            tinynum = 1;
         }
         else if (strncmp (argv[i],"-P",2) == 0) {
            /* Get Unicode plane */
            for (j = 2; argv[i][j] != '\0'; j++) {
               if (argv[i][j] < '0' || argv[i][j] > '9') {
                  fprintf (stderr,
                           "ERROR: Specify Unicode plane as decimal number.\n\n");
                  exit (EXIT_FAILURE);
               }
            }
            plane = atoi (&argv[i][2]); /* Unicode plane, 0..17 */
            if (plane < 0 || plane > 17) {
               fprintf (stderr,
                        "ERROR: Plane out of Unicode range [0,17].\n\n");
               exit (EXIT_FAILURE);
            }
         }
      }
   }


   /*
      Initialize the ASCII bitmap array for chart titles
   */
   for (i = 0; i < 128; i++) {
      gethex (ascii_hex[i], plane_array, 0); /* convert Unifont hexadecimal string to bitmap */
      for (j = 0; j < 16; j++) ascii_bits[i][j] = plane_array[i][j];
   }


   /*
      Read in the Unifont hex file to render from standard input
   */
   memset ((void *)plane_array, 0, 0x10000 * 16 * sizeof (int));
   while (fgets (instring, MAXSTRING, stdin) != NULL) {
      gethex (instring, plane_array, plane); /* read .hex input file and fill plane_array with glyph data */
   }  /* while not EOF */


   /*
      Write plane_array glyph data to BMP file as wide or long bitmap.
   */
   if (wide) {
      genwidebmp (plane_array, dpi, tinynum, plane);
   }
   else {
      genlongbmp (plane_array, dpi, tinynum, plane);
   }

   exit (EXIT_SUCCESS);
}


/**
   @brief Output a 4-byte integer in little-endian order.

   @param[in] thisword The 4-byte integer to output as binary data.
*/
void
output4 (int thisword)
{

   putchar ( thisword        & 0xFF);
   putchar ((thisword >>  8) & 0xFF);
   putchar ((thisword >> 16) & 0xFF);
   putchar ((thisword >> 24) & 0xFF);
```

```

  return;
}


/**
  @brief Output a 2-byte integer in little-endian order.

  @param[in] thisword The 2-byte integer to output as binary data.
*/
void
output2 (int thisword)
{

  putchar ( thisword       & 0xFF);
  putchar ((thisword » 8) & 0xFF);

  return;
}


/**
  @brief Read a Unifont .hex-format input file from stdin.

  Each glyph can be 2, 4, 6, or 8 ASCII hexadecimal digits wide.
  Glyph height is fixed at 16 pixels.

  @param[in] instring One line from a Unifont .hex-format file.
  @param[in,out] plane_array Bitmap for this plane, one bitmap row per element.
  @param[in] plane The Unicode plane, 0..17.
*/
void
gethex (char *instring, int plane_array[0x10000][16], int plane)
{
  char *bitstring;  /* pointer into instring for glyph bitmap */
  int i;        /* loop variable                          */
  int codept;  /* the Unicode code point of the current glyph */
  int glyph_plane; /* Unicode plane of current glyph          */
  int ndigits; /* number of ASCII hexadecimal digits in glyph */
  int bytespl; /* bytes per line of pixels in a glyph         */
  int temprow; /* 1 row of a quadruple-width glyph            */
  int newrow;  /* 1 row of double-width output pixels         */
  unsigned bitmask; /* to mask off 2 bits of long width glyph */

  /*
    Read each input line and place its glyph into the bit array.
  */
  sscanf (instring, "%X", &codept);
  glyph_plane = codept » 16;
  if (glyph_plane == plane) {
    codept &= 0xFFFF;  /* array index will only have 16 bit address */
    /* find the colon separator */
    for (i = 0; (i < 9) && (instring[i] != ':'); i++);
    i++; /* position past it */
    bitstring = &instring[i];
    ndigits = strlen (bitstring);
    /* don't count '\n' at end of line if present */
    if (bitstring[ndigits - 1] == '\n') ndigits--;
    bytespl = ndigits » 5;  /* 16 rows per line, 2 digits per byte */

    if (bytespl >= 1 && bytespl <= 4) {
      for (i = 0; i < 16; i++) { /* 16 rows per glyph */
        /* Read correct number of hexadecimal digits given glyph width */
        switch (bytespl) {
          case 1: sscanf (bitstring, "%2X", &temprow);
                  bitstring += 2;
                  temprow «= 8; /* left-justify single-width glyph */
                  break;
          case 2: sscanf (bitstring, "%4X", &temprow);
                  bitstring += 4;
                  break;
          /* cases 3 and 4 widths will be compressed by 50% (see below) */
          case 3: sscanf (bitstring, "%6X", &temprow);
                  bitstring += 6;
                  temprow «= 8; /* left-justify */
                  break;
          case 4: sscanf (bitstring, "%8X", &temprow);
                  bitstring += 8;
                  break;
        }  /* switch on number of bytes per row */
        /* compress glyph width by 50% if greater than double-width */
```

```
            if (bytespl > 2) {
               newrow = 0x0000;
               /* mask off 2 bits at a time to convert each pair to 1 bit out */
               for (bitmask = 0xC0000000; bitmask != 0; bitmask »= 2) {
                  newrow «= 1;
                  if ((temprow & bitmask) != 0) newrow |= 1;
               }
               temprow = newrow;
            }  /* done conditioning glyphs beyond double-width */
            plane_array[codept][i] = temprow;  /* store glyph bitmap for output */
         }  /* for each row */
      }  /* if 1 to 4 bytes per row/line */
   }  /* if this is the plane we are seeking */

   return;
}


/**
   @brief Generate the BMP output file in long format.

   This function generates the BMP output file from a bitmap parameter.
   This is a long bitmap, 16 glyphs wide by 4,096 glyphs tall.

   @param[in] plane_array The array of glyph bitmaps for a plane.
   @param[in] dpi Dots per inch, for encoding in the BMP output file header.
   @param[in] tinynum Whether to generate tiny numbers in wide grid (unused).
   @param[in] plane The Unicode plane, 0..17.
*/
void
genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
{

   char header_string[HDR_LEN]; /* centered header              */
   char raw_header[HDR_LEN];    /* left-aligned header          */
   int header[16][16];      /* header row, for chart title */
   int hdrlen;              /* length of HEADER_STRING      */
   int startcol;            /* column to start printing header, for centering */

   unsigned leftcol[0x1000][16]; /* code point legend on left side of chart */
   int d1, d2, d3, d4;             /* digits for filling leftcol[][] legend   */
   int codept;                     /* current starting code point for legend  */
   int thisrow;                    /* glyph row currently being rendered       */
   unsigned toprow[16][16];        /* code point legend on top of chart       */
   int digitrow;        /* row we're in (0..4) for the above hexdigit digits */

   /*
      DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
   */
   int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
   int ImageSize;
   int FileSize;
   int Width, Height; /* bitmap image width and height in pixels */
   int ppm;      /* integer pixels per meter */

   int i, j, k;

   unsigned bytesout;

   void output4(int), output2(int);

   /*
      Image width and height, in pixels.

         N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
   */
   Width  =   18 * 16;  /* (2 legend +   16 glyphs) * 16 pixels/glyph */
   Height = 4099 * 16;  /* (1 header + 4096 glyphs) * 16 rows/glyph   */

   ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */

   FileSize = DataOffset + ImageSize;

   /* convert dots/inch to pixels/meter */
   if (dpi == 0) dpi = 96;
   ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);

   /*
      Generate the BMP Header
   */
   putchar ('B');
```

```
putchar ('M');

/*
   Calculate file size:

      BMP Header + InfoHeader + Color Table + Raster Data
*/
output4 (FileSize);  /* FileSize */
output4 (0x0000); /* reserved */

/* Calculate DataOffset */
output4 (DataOffset);

/*
   InfoHeader
*/
output4 (40);         /* Size of InfoHeader                       */
output4 (Width);      /* Width of bitmap in pixels                */
output4 (Height);     /* Height of bitmap in pixels               */
output2 (1);          /* Planes (1 plane)                         */
output2 (1);          /* BitCount (1 = monochrome)                */
output4 (0);          /* Compression (0 = none)                   */
output4 (ImageSize);  /* ImageSize, in bytes                      */
output4 (ppm);        /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (ppm);        /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (2);          /* ColorsUsed (= 2)                         */
output4 (2);          /* ColorsImportant (= 2)                    */
output4 (0x00000000); /* black (reserved, B, G, R)                */
output4 (0x00FFFFFF); /* white (reserved, B, G, R)                */

/*
   Create header row bits.
*/
snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
memset ((void *)header, 0, 16 * 16 * sizeof (int)); /* fill with white */
memset ((void *)header_string, ' ', 32 * sizeof (char)); /* 32 spaces */
header_string[32] = '\0';  /* null-terminated */

hdrlen = strlen (raw_header);
if (hdrlen > 32) hdrlen = 32;        /* only 32 columns to print header */
startcol = 16 - ((hdrlen + 1) » 1); /* to center header                */
/* center up to 32 chars */
memcpy (&header_string[startcol], raw_header, hdrlen);

/* Copy each letter's bitmap from the plane_array[][] we constructed. */
/* Each glyph must be single-width, to fit two glyphs in 16 pixels */
for (j = 0; j < 16; j++) {
   for (i = 0; i < 16; i++) {
      header[i][j] =
         (ascii_bits[header_string[j+j  ] & 0x7F][i] & 0xFF00) |
         (ascii_bits[header_string[j+j+1] & 0x7F][i] » 8);
   }
}

/*
   Create the left column legend.
*/
memset ((void *)leftcol, 0, 4096 * 16 * sizeof (unsigned));

for (codept = 0x0000; codept < 0x10000; codept += 0x10) {
   d1 = (codept » 12) & 0xF; /* most significant hex digit */
   d2 = (codept »  8) & 0xF;
   d3 = (codept »  4) & 0xF;

   thisrow = codept » 4; /* rows of 16 glyphs */

   /* fill in first and second digits */
   for (digitrow = 0; digitrow < 5; digitrow++) {
      leftcol[thisrow][2 + digitrow] =
         (hexdigit[d1][digitrow] « 10) |
         (hexdigit[d2][digitrow] «  4);
   }

   /* fill in third digit */
   for (digitrow = 0; digitrow < 5; digitrow++) {
      leftcol[thisrow][9 + digitrow] = hexdigit[d3][digitrow] « 10;
   }
   leftcol[thisrow][9 + 4] |= 0xF « 4; /* underscore as 4th digit */

   for (i = 0; i < 15; i ++) {
      leftcol[thisrow][i] |= 0x00000002;      /* right border */
```

```
   }

   leftcol[thisrow][15] = 0x0000FFFE;        /* bottom border */

   if (d3 == 0xF) {                          /* 256-point boundary */
      leftcol[thisrow][15] |= 0x00FF0000;  /* longer tic mark */
   }

   if ((thisrow % 0x40) == 0x3F) {    /* 1024-point boundary */
      leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
   }
}

/*
   Create the top row legend.
*/
memset ((void *)toprow, 0, 16 * 16 * sizeof (unsigned));

for (codept = 0x0; codept <= 0xF; codept++) {
   d1 = (codept » 12) & 0xF; /* most significant hex digit */
   d2 = (codept »  8) & 0xF;
   d3 = (codept »  4) & 0xF;
   d4 =  codept       & 0xF; /* least significant hex digit */

   /* fill in last digit */
   for (digitrow = 0; digitrow < 5; digitrow++) {
      toprow[6 + digitrow][codept] = hexdigit[d4][digitrow] « 6;
   }
}

for (j = 0; j < 16; j++) {
   /* force bottom pixel row to be white, for separation from glyphs */
   toprow[15][j] = 0x0000;
}

/* 1 pixel row with left-hand legend line */
for (j = 0; j < 16; j++) {
   toprow[14][j] |= 0xFFFF;
}

/* 14 rows with line on left to fill out this character row */
for (i = 13; i >= 0; i--) {
   for (j = 0; j < 16; j++) {
      toprow[i][j] |= 0x0001;
   }
}

/*
   Now write the raster image.

   XOR each byte with 0xFF because black = 0, white = 1 in BMP.
*/

/* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
for (i = 0xFFF0; i >= 0; i -= 0x10) {
   thisrow = i » 4; /* 16 glyphs per row */
   for (j = 15; j >= 0; j--) {
      /* left-hand legend */
      putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
      putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
      putchar ((~leftcol[thisrow][j] »  8) & 0xFF);
      putchar ( ~leftcol[thisrow][j]       & 0xFF);
      /* Unifont glyph */
      for (k = 0; k < 16; k++) {
         bytesout = ~plane_array[i+k][j] & 0xFFFF;
         putchar ((bytesout » 8) & 0xFF);
         putchar ( bytesout       & 0xFF);
      }
   }
}

/*
   Write the top legend.
*/
/* i == 15: bottom pixel row of header is output here */
/* left-hand legend: solid black line except for right-most pixel */
putchar (0x00);
putchar (0x00);
putchar (0x00);
putchar (0x01);
for (j = 0; j < 16; j++) {
```

```
      putchar ((~toprow[15][j] » 8) & 0xFF);
      putchar ( ~toprow[15][j]        & 0xFF);
   }

   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFC);
   for (j = 0; j < 16; j++) {
      putchar ((~toprow[14][j] » 8) & 0xFF);
      putchar ( ~toprow[14][j]        & 0xFF);
   }

   for (i = 13; i >= 0; i--) {
      putchar (0xFF);
      putchar (0xFF);
      putchar (0xFF);
      putchar (0xFD);
      for (j = 0; j < 16; j++) {
         putchar ((~toprow[i][j] » 8) & 0xFF);
         putchar ( ~toprow[i][j]        & 0xFF);
      }
   }

   /*
      Write the header.
   */

   /* 7 completely white rows */
   for (i = 7; i >= 0; i--) {
      for (j = 0; j < 18; j++) {
         putchar (0xFF);
         putchar (0xFF);
      }
   }

   for (i = 15; i >= 0; i--) {
      /* left-hand legend */
      putchar (0xFF);
      putchar (0xFF);
      putchar (0xFF);
      putchar (0xFF);
      /* header glyph */
      for (j = 0; j < 16; j++) {
         bytesout = ~header[i][j] & 0xFFFF;
         putchar ((bytesout » 8) & 0xFF);
         putchar ( bytesout        & 0xFF);
      }
   }

   /* 8 completely white rows at very top */
   for (i = 7; i >= 0; i--) {
      for (j = 0; j < 18; j++) {
      putchar (0xFF);
      putchar (0xFF);
      }
   }

   return;
}


/**
   @brief Generate the BMP output file in wide format.

   This function generates the BMP output file from a bitmap parameter.
   This is a wide bitmap, 256 glyphs wide by 256 glyphs tall.

   @param[in] plane_array The array of glyph bitmaps for a plane.
   @param[in] dpi Dots per inch, for encoding in the BMP output file header.
   @param[in] tinynum Whether to generate tiny numbers in 256x256 grid.
   @param[in] plane The Unicode plane, 0..17.
*/
void
genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
{

   char header_string[257];
   char raw_header[HDR_LEN];
   int header[16][256]; /* header row, for chart title */
   int hdrlen;          /* length of HEADER_STRING */
```

```
int startcol;      /* column to start printing header, for centering */

unsigned leftcol[0x100][16]; /* code point legend on left side of chart */
int d1, d2, d3, d4;          /* digits for filling leftcol[][] legend   */
int codept;                  /* current starting code point for legend  */
int thisrow;                 /* glyph row currently being rendered      */
unsigned toprow[32][256];    /* code point legend on top of chart       */
int digitrow;      /* row we're in (0..4) for the above hexdigit digits */
int hexalpha1, hexalpha2;    /* to convert hex digits to ASCII          */

/*
   DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
*/
int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
int ImageSize;
int FileSize;
int Width, Height; /* bitmap image width and height in pixels */
int ppm;     /* integer pixels per meter */

int i, j, k;

unsigned bytesout;

void output4(int), output2(int);

/*
   Image width and height, in pixels.

      N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
*/
Width  = 258 * 16;  /* (           2 legend + 256 glyphs) * 16 pixels/glyph */
Height = 260 * 16;  /* (2 header + 2 legend + 256 glyphs) * 16 rows/glyph   */

ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */

FileSize = DataOffset + ImageSize;

/* convert dots/inch to pixels/meter */
if (dpi == 0) dpi = 96;
ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);

/*
   Generate the BMP Header
*/
putchar ('B');
putchar ('M');
/*
   Calculate file size:

      BMP Header + InfoHeader + Color Table + Raster Data
*/
output4 (FileSize);  /* FileSize */
output4 (0x0000); /* reserved */
/* Calculate DataOffset */
output4 (DataOffset);

/*
   InfoHeader
*/
output4 (40);         /* Size of InfoHeader                       */
output4 (Width);      /* Width of bitmap in pixels                */
output4 (Height);     /* Height of bitmap in pixels               */
output2 (1);          /* Planes (1 plane)                         */
output2 (1);          /* BitCount (1 = monochrome)                */
output4 (0);          /* Compression (0 = none)                   */
output4 (ImageSize);  /* ImageSize, in bytes                      */
output4 (ppm);        /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (ppm);        /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (2);          /* ColorsUsed (= 2)                         */
output4 (2);          /* ColorsImportant (= 2)                    */
output4 (0x00000000); /* black (reserved, B, G, R)                */
output4 (0x00FFFFFF); /* white (reserved, B, G, R)                */

/*
   Create header row bits.
*/
snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
memset ((void *)header, 0, 256 * 16 * sizeof (int)); /* fill with white */
memset ((void *)header_string, ' ', 256 * sizeof (char)); /* 256 spaces */
header_string[256] = '\0';  /* null-terminated */

```

```
hdrlen = strlen (raw_header);
/* Wide bitmap can print 256 columns, but limit to 32 columns for long bitmap. */
if (hdrlen > 32) hdrlen = 32;
startcol = 127 - ((hdrlen - 1) » 1);  /* to center header */
/* center up to 32 chars */
memcpy (&header_string[startcol], raw_header, hdrlen);

/* Copy each letter’s bitmap from the plane_array[][] we constructed. */
for (j = 0; j < 256; j++) {
   for (i = 0; i < 16; i++) {
      header[i][j] = ascii_bits[header_string[j] & 0x7F][i];
   }
}

/*
   Create the left column legend.
*/
memset ((void *)leftcol, 0, 256 * 16 * sizeof (unsigned));

for (codept = 0x0000; codept < 0x10000; codept += 0x100) {
   d1 = (codept » 12) & 0xF; /* most significant hex digit */
   d2 = (codept »  8) & 0xF;

   thisrow = codept » 8; /* rows of 256 glyphs */

   /* fill in first and second digits */

   if (tinynum) { /* use 4x5 pixel glyphs */
      for (digitrow = 0; digitrow < 5; digitrow++) {
         leftcol[thisrow][6 + digitrow] =
            (hexdigit[d1][digitrow] « 10) |
            (hexdigit[d2][digitrow] «  4);
      }
   }
   else { /* bigger numbers -- use glyphs from Unifont itself */
      /* convert hexadecimal digits to ASCII equivalent */
      hexalpha1 = d1 < 0xA ? ’0’ + d1 : ’A’ + d1 - 0xA;
      hexalpha2 = d2 < 0xA ? ’0’ + d2 : ’A’ + d2 - 0xA;

      for (i = 0 ; i < 16; i++) {
         leftcol[thisrow][i] =
            (ascii_bits[hexalpha1][i] « 2) |
            (ascii_bits[hexalpha2][i] » 6);
      }
   }

   for (i = 0; i < 15; i ++) {
      leftcol[thisrow][i] |= 0x00000002;      /* right border */
   }

   leftcol[thisrow][15] = 0x0000FFFE;        /* bottom border */

   if (d2 == 0xF) {                     /* 4096-point boundary */
      leftcol[thisrow][15] |= 0x00FF0000;  /* longer tic mark */
   }

   if ((thisrow % 0x40) == 0x3F) {     /* 16,384-point boundary */
      leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
   }
}

/*
   Create the top row legend.
*/
memset ((void *)toprow, 0, 32 * 256 * sizeof (unsigned));

for (codept = 0x00; codept <= 0xFF; codept++) {
   d3 = (codept »  4) & 0xF;
   d4 =  codept        & 0xF; /* least significant hex digit */

   if (tinynum) {
      for (digitrow = 0; digitrow < 5; digitrow++) {
         toprow[16 + 6 + digitrow][codept] =
            (hexdigit[d3][digitrow] « 10) |
            (hexdigit[d4][digitrow] «  4);
      }
   }
   else {
      /* convert hexadecimal digits to ASCII equivalent */
      hexalpha1 = d3 < 0xA ? ’0’ + d3 : ’A’ + d3 - 0xA;
      hexalpha2 = d4 < 0xA ? ’0’ + d4 : ’A’ + d4 - 0xA;
```

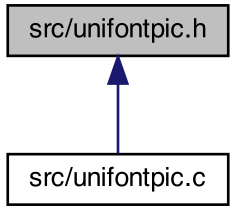
```
      for (i = 0 ; i < 16; i++) {
         toprow[14 + i][codept] =
            (ascii_bits[hexalpha1][i]      ) |
            (ascii_bits[hexalpha2][i] » 7);
      }
   }
}

for (j = 0; j < 256; j++) {
   /* force bottom pixel row to be white, for separation from glyphs */
   toprow[16 + 15][j] = 0x0000;
}

/* 1 pixel row with left-hand legend line */
for (j = 0; j < 256; j++) {
   toprow[16 + 14][j] |= 0xFFFF;
}

/* 14 rows with line on left to fill out this character row */
for (i = 13; i >= 0; i--) {
   for (j = 0; j < 256; j++) {
      toprow[16 + i][j] |= 0x0001;
   }
}

/* Form the longer tic marks in top legend */
for (i = 8; i < 16; i++) {
   for (j = 0x0F; j < 0x100; j += 0x10) {
      toprow[i][j] |= 0x0001;
   }
}

/*
   Now write the raster image.

   XOR each byte with 0xFF because black = 0, white = 1 in BMP.
*/

/* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
for (i = 0xFF00; i >= 0; i -= 0x100) {
   thisrow = i » 8; /* 256 glyphs per row */
   for (j = 15; j >= 0; j--) {
      /* left-hand legend */
      putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
      putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
      putchar ((~leftcol[thisrow][j] »  8) & 0xFF);
      putchar ( ~leftcol[thisrow][j]       & 0xFF);
      /* Unifont glyph */
      for (k = 0x00; k < 0x100; k++) {
         bytesout = ~plane_array[i+k][j] & 0xFFFF;
         putchar ((bytesout » 8) & 0xFF);
         putchar ( bytesout       & 0xFF);
      }
   }
}

/*
   Write the top legend.
*/
/* i == 15: bottom pixel row of header is output here */
/* left-hand legend: solid black line except for right-most pixel */
putchar (0x00);
putchar (0x00);
putchar (0x00);
putchar (0x01);
for (j = 0; j < 256; j++) {
   putchar ((~toprow[16 + 15][j] » 8) & 0xFF);
   putchar ( ~toprow[16 + 15][j]       & 0xFF);
}

putchar (0xFF);
putchar (0xFF);
putchar (0xFF);
putchar (0xFC);
for (j = 0; j < 256; j++) {
   putchar ((~toprow[16 + 14][j] » 8) & 0xFF);
   putchar ( ~toprow[16 + 14][j]       & 0xFF);
}

for (i = 16 + 13; i >= 0; i--) {
   if (i >= 8) { /* make vertical stroke on right */
```

```
            putchar (0xFF);
            putchar (0xFF);
            putchar (0xFF);
            putchar (0xFD);
        }
        else { /* all white */
            putchar (0xFF);
            putchar (0xFF);
            putchar (0xFF);
            putchar (0xFF);
        }
        for (j = 0; j < 256; j++) {
            putchar ((~toprow[i][j] » 8) & 0xFF);
            putchar ( ~toprow[i][j]      & 0xFF);
        }
    }

    /*
        Write the header.
    */

    /* 8 completely white rows */
    for (i = 7; i >= 0; i--) {
        for (j = 0; j < 258; j++) {
            putchar (0xFF);
            putchar (0xFF);
        }
    }

    for (i = 15; i >= 0; i--) {
        /* left-hand legend */
        putchar (0xFF);
        putchar (0xFF);
        putchar (0xFF);
        putchar (0xFF);
        /* header glyph */
        for (j = 0; j < 256; j++) {
            bytesout = ~header[i][j] & 0xFFFF;
            putchar ((bytesout » 8) & 0xFF);
            putchar ( bytesout      & 0xFF);
        }
    }

    /* 8 completely white rows at very top */
    for (i = 7; i >= 0; i--) {
        for (j = 0; j < 258; j++) {
        putchar (0xFF);
        putchar (0xFF);
        }
    }

    return;
}
```

## 5.19 src/unifontpic.h File Reference

unifontpic.h - Header file for unifontpic.c

This graph shows which files directly or indirectly include this file:




## Macros

- #define MAXSTRING 256

  Maximum input string allowed.
- #define HEADER_STRING "GNU Unifont 15.0.02"

  To be printed as chart title.

## Variables

- const char * ascii_hex [128]

  Array of Unifont ASCII glyphs for chart row & column headings.
- int ascii_bits [128][16]

  Array to hold ASCII bitmaps for chart title.
- char hexdigit [16][5]

  Array of 4x5 hexadecimal digits for legend.

### 5.19.1 Detailed Description

unifontpic.h - Header file for unifontpic.c

Author

Paul Hardy, July 2017

Copyright

Copyright (C) 2017 Paul Hardy

Definition in file unifontpic.h.

### 5.19.2 Macro Definition Documentation

#### 5.19.2.1 HEADER_STRING

#define HEADER_STRING "GNU Unifont 15.0.02"

To be printed as chart title.

Definition at line 30 of file unifontpic.h.

#### 5.19.2.2 MAXSTRING

#define MAXSTRING 256

Maximum input string allowed.

Definition at line 28 of file unifontpic.h.

### 5.19.3 Variable Documentation

#### 5.19.3.1 ascii_bits

int ascii_bits[128][16]

Array to hold ASCII bitmaps for chart title.

This array will be created from the strings in ascii_hex[] above.

Definition at line 177 of file unifontpic.h.

#### 5.19.3.2 ascii_hex

const char∗ ascii_hex[128]

Array of Unifont ASCII glyphs for chart row & column headings.

Define the array of Unifont ASCII glyphs, code points 0 through 127. This allows using unifontpic to print charts of glyphs above Unicode Plane 0. These were copied from font/plane00/unifont-base.hex, plus U+0020 (ASCII space character).

Definition at line 40 of file unifontpic.h.

### 5.19.3.3 hexdigit

char hexdigit[16][5]

Initial value:
```
= {
   {0x6,0x9,0x9,0x9,0x6},
   {0x2,0x6,0x2,0x2,0x7},
   {0xF,0x1,0xF,0x8,0xF},
   {0xE,0x1,0x7,0x1,0xE},
   {0x9,0x9,0xF,0x1,0x1},
   {0xF,0x8,0xF,0x1,0xF},
   {0x6,0x8,0xE,0x9,0x6},
   {0xF,0x1,0x2,0x4,0x4},
   {0x6,0x9,0x6,0x9,0x6},
   {0x6,0x9,0x7,0x1,0x6},
   {0xF,0x9,0xF,0x9,0x9},
   {0xE,0x9,0xE,0x9,0xE},
   {0x7,0x8,0x8,0x8,0x7},
   {0xE,0x9,0x9,0x9,0xE},
   {0xF,0x8,0xE,0x8,0xF},
   {0xF,0x8,0xE,0x8,0x8}
}
```

Array of 4x5 hexadecimal digits for legend.

hexdigit contains 4x5 pixel arrays of tiny digits for the legend. See unihexgen.c for a more detailed description in the comments.

Definition at line 186 of file unifontpic.h.

## 5.20 unifontpic.h

Go to the documentation of this file.
```
/**
  @file unifontpic.h

  @brief unifontpic.h - Header file for unifontpic.c

  @author Paul Hardy, July 2017

  @copyright Copyright (C) 2017 Paul Hardy
*/
/*
  LICENSE:

     This program is free software: you can redistribute it and/or modify
     it under the terms of the GNU General Public License as published by
     the Free Software Foundation, either version 2 of the License, or
     (at your option) any later version.

     This program is distributed in the hope that it will be useful,
     but WITHOUT ANY WARRANTY; without even the implied warranty of
     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
     GNU General Public License for more details.

     You should have received a copy of the GNU General Public License
     along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/


#define MAXSTRING 256  ///< Maximum input string allowed.

#define HEADER_STRING "GNU Unifont 15.0.02" ///< To be printed as chart title.

/**
  @brief Array of Unifont ASCII glyphs for chart row & column headings.

  Define the array of Unifont ASCII glyphs, code points 0 through 127.
```

```
    This allows using unifontpic to print charts of glyphs above Unicode
    Plane 0.  These were copied from font/plane00/unifont-base.hex, plus
    U+0020 (ASCII space character).
*/
const char *ascii_hex [128] = {
    ”0000:AAAA00018000000180004A51EA505A51C99E0001800000018000000180005555”,
    ”0001:AAAA00018000000180003993C252325F8A527193800000018000000180005555”,
    ”0002:AAAA00018000000180003BA5C124311989247125800000018000000180005555”,
    ”0003:AAAA00018000000180007BA5C1247919C124792580000001800000180005555”,
    ”0004:AAAA000180000001800079BFC2487A49C2487989800000018000000180005555”,
    ”0005:AAAA00018000000180007A4DC2527B53C2D67A4F800000018000000180005555”,
    ”0006:AAAA000180000001800031A5CA287A31CA2849A5800000018000000180005555”,
    ”0007:AAAA000180000001800073D1CA1073D1CA1073DF800000018000000180005555”,
    ”0008:AAAA00018000000180001E3991401E3191081E71800000018000000180005555”,
    ”0009:AAAA000180000001800022F9A2203E21A2202221800000018000000180005555”,
    ”000A:AAAA000180000001800020F9A08020F9A0803E81800000018000000180005555”,
    ”000B:AAAA000180000001800022F9A2202221942008218000000180000000180005555”,
    ”000C:AAAA00018000000180003EF9A0803EF9A08020818000000180000000180005555”,
    ”000D:AAAA00018000000180001EF1A08820F1A0901E89800000018000000180005555”,
    ”000E:AAAA00018000000180001E71A0881C8982883C71800000018000000180005555”,
    ”000F:AAAA00018000000180001EF9A0201C2182203CF9800000018000000180005555”,
    ”0010:AAAA0001800000018000391DA510251DA51039DD800000018000000180005555”,
    ”0011:AAAA00018000000180007189CA184A09CA08719D800000018000000180005555”,
    ”0012:AAAA00018000000180007199CA044A09CA10719D800000018000000180005555”,
    ”0013:AAAA00018000000180007199CA044A19CA047199800000018000000180005555”,
    ”0014:AAAA00018000000180007185CA0C4A15CA1C7185800000018000000180005555”,
    ”0015:AAAA00018000000180004993EA546A59DBD44A53800000018000000180005555”,
    ”0016:AAAA00018000000180003453C29A3117891271138000000180000000180005555”,
    ”0017:AAAA00018000000180007BB9C1247939C124793980000001800000180005555”,
    ”0018:AAAA00018000000180003325C4B447ADC4A434A5800000018000000180005555”,
    ”0019:AAAA00018000000180003E89A0D83EA9A0883E89800000018000000180005555”,
    ”001A:AAAA00018000000180003A5DC252325D8A52719D800000018000000180005555”,
    ”001B:AAAA000180000001800079CFC2107991C0507B8F800000018000000180005555”,
    ”001C:AAAA00018000000180001E7190801E61901010E1800000018000000180005555”,
    ”001D:AAAA00018000000180000E719080166192100EE1800000018000000180005555”,
    ”001E:AAAA00018000000180001C7192801C61941012E1800000018000000180005555”,
    ”001F:AAAA00018000000180001271928012619210 0CE1800000018000000180005555”,
    ”0020:00000000000000000000000000000000”,
    ”0021:00000000080808080808080008080000”,
    ”0022:00002222222200000000000000000000”,
    ”0023:000000001212127E24247E4848480000”,
    ”0024:00000000083E4948380E09493E080000”,
    ”0025:00000000314A4A340808162929460000”,
    ”0026:000000001C2222141829454246390000”,
    ”0027:00000808080800000000000000000000”,
    ”0028:00000004080810101010101008080400”,
    ”0029:00000020101008080808080810102000”,
    ”002A:00000000000008492A1C2A4908000000”,
    ”002B:0000000000000808087F080808000000”,
    ”002C:00000000000000000000000018080810”,
    ”002D:0000000000000000003C000000000000”,
    ”002E:00000000000000000000000018180000”,
    ”002F:00000000020204080810102040400000”,
    ”0030:00000000182442464A52624224180000”,
    ”0031:000000000818280808080808083E0000”,
    ”0032:000000003C4242020C102040407E0000”,
    ”0033:000000003C4242021C020242423C0000”,
    ”0034:00000000040C142444447E0404040000”,
    ”0035:000000007E4040407C020202423C0000”,
    ”0036:000000001C2040407C424242423C0000”,
    ”0037:000000007E0202040404080808080000”,
    ”0038:000000003C4242423C424242423C0000”,
    ”0039:000000003C4242423E02020204380000”,
    ”003A:00000000000018180000001818000000”,
    ”003B:00000000000018180000001808081000”,
    ”003C:00000000000204081020100804020000”,
    ”003D:000000000000007E0000007E00000000”,
    ”003E:00000000004020100804081020400000”,
    ”003F:000000003C4242020408080008080000”,
    ”0040:000000001C224A565252524E201E0000”,
    ”0041:0000000018242442427E424242420000”,
    ”0042:000000007C4242427C424242427C0000”,
    ”0043:000000003C42424040404042423C0000”,
    ”0044:00000000784442424242424244780000”,
    ”0045:000000007E4040407C404040407E0000”,
    ”0046:000000007E4040407C40404040400000”,
    ”0047:000000003C424240404E4242463A0000”,
    ”0048:00000000424242427E42424242420000”,
    ”0049:000000003E08080808080808083E0000”,
    ”004A:000000001F0404040404044444380000”,
    ”004B:00000000424448506060504844420000”,
```

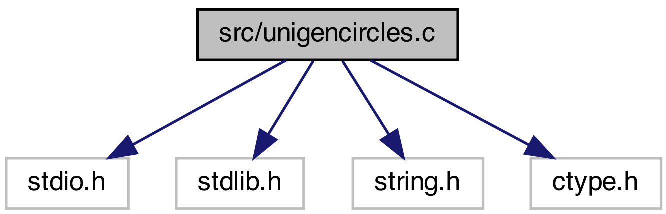
```c
    "004C:000000004040404040404040407E0000",
    "004D:00000000424266665A5A424242420000",
    "004E:0000000042626252524A4A4646420000",
    "004F:000000003C42424242424242423C0000",
    "0050:000000007C4242427C40404040400000",
    "0051:000000003C4242424242425A663C0300",
    "0052:000000007C4242427C48444442420000",
    "0053:000000003C424240300C0242423C0000",
    "0054:000000007F0808080808080808080000",
    "0055:000000004242424242424242423C0000",
    "0056:00000000414141222222141408080000",
    "0057:00000000424242425A5A666642420000",
    "0058:00000000424224241818242442420000",
    "0059:00000000414122221408080808080000",
    "005A:000000007E02020408102040407E0000",
    "005B:0000000E080808080808080808080E00",
    "005C:00000000404020101008080402020000",
    "005D:00000070101010101010101010107000",
    "005E:00001824420000000000000000000000",
    "005F:00000000000000000000000000007F00",
    "0060:00201008000000000000000000000000",
    "0061:0000000000003C42023E4242463A0000",
    "0062:0000004040405C6242424242625C0000",
    "0063:0000000000003C4240404040423C0000",
    "0064:0000000202023A4642424242463A0000",
    "0065:0000000000003C42427E4040423C0000",
    "0066:0000000C1010107C1010101010100000",
    "0067:0000000000023A44444438203C42423C",
    "0068:0000004040405C624242424242420000",
    "0069:000000080800180808080808083E0000",
    "006A:0000000404000C040404040404044830",
    "006B:00000040404044485060504844420000",
    "006C:000000180808080808080808083E0000",
    "006D:00000000000076494949494949490000",
    "006E:0000000000005C624242424242420000",
    "006F:0000000000003C4242424242423C0000",
    "0070:0000000000005C6242424242625C4040",
    "0071:0000000000003A4642424242463A0202",
    "0072:0000000000005C624240404040400000",
    "0073:0000000000003C4240300C02423C0000",
    "0074:000000001010107C10101010100C0000",
    "0075:000000000000424242424242463A0000",
    "0076:00000000000042424224242418180000",
    "0077:00000000000041494949494949360000",
    "0078:00000000000042422418182442420000",
    "0079:0000000000004242424242261A02023C",
    "007A:0000000000007E0204081020407E0000",
    "007B:0000000C10100808102010080810100C",
    "007C:00000808080808080808080808080808",
    "007D:00000030080810100804081010080830",
    "007E:00000031494600000000000000000000",
    "007F:AAAA000180000001800073D1CA104BD1CA1073DF800000018000000180005555"
};


/**
    @brief Array to hold ASCII bitmaps for chart title.

    This array will be created from the strings in ascii_hex[] above.
*/
int ascii_bits[128][16];


/**
    @brief Array of 4x5 hexadecimal digits for legend.

    hexdigit contains 4x5 pixel arrays of tiny digits for the legend.
    See unihexgen.c for a more detailed description in the comments.
*/
char hexdigit[16][5] = {
    {0x6,0x9,0x9,0x9,0x6},  /* 0x0 */
    {0x2,0x6,0x2,0x2,0x7},  /* 0x1 */
    {0xF,0x1,0xF,0x8,0xF},  /* 0x2 */
    {0xE,0x1,0x7,0x1,0xE},  /* 0x3 */
    {0x9,0x9,0xF,0x1,0x1},  /* 0x4 */
    {0xF,0x8,0xF,0x1,0xF},  /* 0x5 */
    {0x6,0x8,0xE,0x9,0x6},  /* 0x6 */
    {0xF,0x1,0x2,0x4,0x4},  /* 0x7 */
    {0x6,0x9,0x6,0x9,0x6},  /* 0x8 */
    {0x6,0x9,0x7,0x1,0x6},  /* 0x9 */
    {0xF,0x9,0xF,0x9,0x9},  /* 0xA */
```

```
   {0xE,0x9,0xE,0x9,0xE}, /* 0xB */
   {0x7,0x8,0x8,0x8,0x7}, /* 0xC */
   {0xE,0x9,0x9,0x9,0xE}, /* 0xD */
   {0xF,0x8,0xE,0x8,0xF}, /* 0xE */
   {0xF,0x8,0xE,0x8,0x8}  /* 0xF */
};

```

## 5.21 src/unigencircles.c File Reference

unigencircles - Superimpose dashed combining circles on combining glyphs

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
Include dependency graph for unigencircles.c:




### Macros

- #define MAXSTRING 256

  Maximum input line length - 1.

### Functions

- int main (int argc, char **argv)

  The main function.
- void add_single_circle (char *glyphstring)

  Superimpose a single-width dashed combining circle on a glyph bitmap.
- void add_double_circle (char *glyphstring, int offset)

  Superimpose a double-width dashed combining circle on a glyph bitmap.

### 5.21.1 Detailed Description

unigencircles - Superimpose dashed combining circles on combining glyphs

Author

Paul Hardy

Copyright

Copyright (C) 2013, Paul Hardy.

Definition in file unigencircles.c.

### 5.21.2 Macro Definition Documentation

#### 5.21.2.1 MAXSTRING

```
#define MAXSTRING 256
```

Maximum input line length - 1.

Definition at line 62 of file unigencircles.c.

### 5.21.3 Function Documentation

#### 5.21.3.1 add_double_circle()

```
void add_double_circle (
        char * glyphstring,
        int offset )
```

Superimpose a double-width dashed combining circle on a glyph bitmap.

Parameters

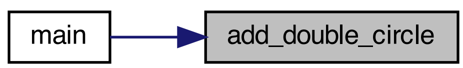
| | | |
|---|---|---|
| in,out | glyphstring | A double-width glyph, 16x16 pixels. |

Definition at line 221 of file unigencircles.c.

```
{

   char newstring[256];
   /* Circle hex string pattern is "00000008000024004200240000000000" */

   /* For double diacritical glyphs (offset = -8) */
   /* Combining circle is left-justified.         */
   char circle08[64]={0x0,0x0,0x0,0x0,  /* row  1 */
                      0x0,0x0,0x0,0x0,  /* row  2 */
                      0x0,0x0,0x0,0x0,  /* row  3 */
                      0x0,0x0,0x0,0x0,  /* row  4 */
                      0x0,0x0,0x0,0x0,  /* row  5 */
                      0x0,0x0,0x0,0x0,  /* row  6 */
                      0x2,0x4,0x0,0x0,  /* row  7 */
                      0x0,0x0,0x0,0x0,  /* row  8 */
                      0x4,0x2,0x0,0x0,  /* row  9 */
                      0x0,0x0,0x0,0x0,  /* row 10 */
                      0x2,0x4,0x0,0x0,  /* row 11 */
                      0x0,0x0,0x0,0x0,  /* row 12 */
                      0x0,0x0,0x0,0x0,  /* row 13 */
                      0x0,0x0,0x0,0x0,  /* row 14 */
                      0x0,0x0,0x0,0x0,  /* row 15 */
                      0x0,0x0,0x0,0x0}; /* row 16 */

   /* For all other combining glyphs (offset = -16) */
   /* Combining circle is centered in 16 columns.   */
   char circle16[64]={0x0,0x0,0x0,0x0,  /* row  1 */
                      0x0,0x0,0x0,0x0,  /* row  2 */
                      0x0,0x0,0x0,0x0,  /* row  3 */
                      0x0,0x0,0x0,0x0,  /* row  4 */
                      0x0,0x0,0x0,0x0,  /* row  5 */
                      0x0,0x0,0x0,0x0,  /* row  6 */
                      0x0,0x2,0x4,0x0,  /* row  7 */
                      0x0,0x0,0x0,0x0,  /* row  8 */
                      0x0,0x4,0x2,0x0,  /* row  9 */
                      0x0,0x0,0x0,0x0,  /* row 10 */
                      0x0,0x2,0x4,0x0,  /* row 11 */
                      0x0,0x0,0x0,0x0,  /* row 12 */
                      0x0,0x0,0x0,0x0,  /* row 13 */
                      0x0,0x0,0x0,0x0,  /* row 14 */
                      0x0,0x0,0x0,0x0,  /* row 15 */
                      0x0,0x0,0x0,0x0}; /* row 16 */

   char *circle; /* points into circle16 or circle08 */

   int digit1, digit2; /* corresponding digits in each string */

   int i; /* index variables */


   /*
      Determine if combining circle is left-justified (offset = -8)
      or centered (offset = -16).
   */
   circle = (offset >= -8) ? circle08 : circle16;

   /* for each character position, OR the corresponding circle glyph value */
   for (i = 0; i < 64; i++) {
      glyphstring[i] = toupper (glyphstring[i]);

      /* Convert ASCII character to a hexadecimal integer */
      digit1 = (glyphstring[i] <= '9') ?
               (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);

      /* Superimpose dashed circle */
```

```
00287         digit2 = digit1 | circle[i];
00288 
00289         /* Convert hexadecimal integer to an ASCII character */
00290         newstring[i] = (digit2 <= 9) ?
00291                        ('0' + digit2) : ('A' + digit2 - 0xA);
00292     }
00293 
00294     /* Terminate string for output */
00295     newstring[i++] = '\n';
00296     newstring[i++] = '\0';
00297 
00298     memcpy (glyphstring, newstring, i);
00299 
00300     return;
00301 }
```

Here is the caller graph for this function:




#### 5.21.3.2 add_single_circle()

void add_single_circle (
    char ∗ glyphstring )

Superimpose a single-width dashed combining circle on a glyph bitmap.

Parameters

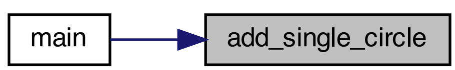
| | | |
|---|---|---|
| in,out | glyphstring | A single-width glyph, 8x16 pixels. |

Definition at line 163 of file unigencircles.c.

```
00164 {
00165 
00166     char newstring[256];
00167     /* Circle hex string pattern is "00000000800002400420024000000000" */
00168     char circle[32]={0x0,0x0,  /* row  1 */
00169                      0x0,0x0,  /* row  2 */
00170                      0x0,0x0,  /* row  3 */
00171                      0x0,0x0,  /* row  4 */
00172                      0x0,0x0,  /* row  5 */
00173                      0x0,0x0,  /* row  6 */
00174                      0x2,0x4,  /* row  7 */
```

```
                  0x0,0x0,  /* row  8 */
                  0x4,0x2,  /* row  9 */
                  0x0,0x0,  /* row 10 */
                  0x2,0x4,  /* row 11 */
                  0x0,0x0,  /* row 12 */
                  0x0,0x0,  /* row 13 */
                  0x0,0x0,  /* row 14 */
                  0x0,0x0,  /* row 15 */
                  0x0,0x0}; /* row 16 */

   int digit1, digit2; /* corresponding digits in each string */

   int i; /* index variables */

   /* for each character position, OR the corresponding circle glyph value */
   for (i = 0; i < 32; i++) {
      glyphstring[i] = toupper (glyphstring[i]);

      /* Convert ASCII character to a hexadecimal integer */
      digit1 = (glyphstring[i] <= '9') ?
            (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);

      /* Superimpose dashed circle */
      digit2 = digit1 | circle[i];

      /* Convert hexadecimal integer to an ASCII character */
      newstring[i] = (digit2 <= 9) ?
                ('0' + digit2) : ('A' + digit2 - 0xA);
   }

   /* Terminate string for output */
   newstring[i++] = '\n';
   newstring[i++] = '\0';

   memcpy (glyphstring, newstring, i);

   return;
}
```

Here is the caller graph for this function:




### 5.21.3.3 main()

int main (
    int argc,
    char ** argv )

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

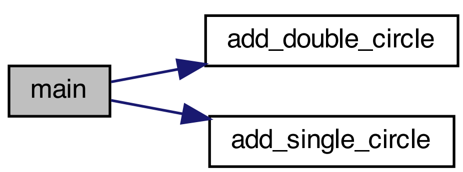
This program exits with status EXIT_SUCCESS.

Definition at line 73 of file unigencircles.c.

```
{

   char teststring[MAXSTRING];  /* current input line                       */
   int  loc;                    /* Unicode code point of current input line */
   int  offset;                 /* offset value of a combining character    */
   char *gstart;                /* glyph start, pointing into teststring    */

   char combining[0x110000];    /* 1 --> combining glyph; 0 --> non-combining */
   char x_offset [0x110000];    /* second value in *combining.txt files       */

   void add_single_circle(char *);      /* add a single-width dashed circle */
   void add_double_circle(char *, int); /* add a double-width dashed circle */

   FILE *infilefp;

   /*
      if (argc != 3) {
         fprintf (stderr,
                 "\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n");
         exit (EXIT_FAILURE);
      }
   */

   /*
      Read the combining characters list.
   */
   /* Start with no combining code points flagged */
   memset (combining, 0, 0x110000 * sizeof (char));
   memset (x_offset , 0, 0x110000 * sizeof (char));

   if ((infilefp = fopen (argv[1],"r")) == NULL) {
      fprintf (stderr,"ERROR - combining characters file %s not found.\n\n",
              argv[1]);
      exit (EXIT_FAILURE);
   }

   /* Flag list of combining characters to add a dashed circle. */
   while (fscanf (infilefp, "%X:%d", &loc, &offset) != EOF) {
      /*
         U+01107F and U+01D1A0 are not defined as combining characters
         in Unicode; they were added in a combining.txt file as the
         only way to make them look acceptable in proximity to other
         glyphs in their script.
```

```
      */
      if (loc != 0x01107F && loc != 0x01D1A0) {
         combining[loc] = 1;
         x_offset [loc] = offset;
      }
   }
   fclose (infilefp); /* all done reading combining.txt */

   /* Now read the non-printing glyphs; they never have dashed circles */
   if ((infilefp = fopen (argv[2],"r")) == NULL) {
      fprintf (stderr,"ERROR - nonprinting characters file %s not found.\n\n",
              argv[1]);
      exit (EXIT_FAILURE);
   }

   /* Reset list of nonprinting characters to avoid adding a dashed circle. */
   while (fscanf (infilefp, "%X:%*s", &loc) != EOF) combining[loc] = 0;

   fclose (infilefp); /* all done reading nonprinting.hex */

   /*
      Read the hex glyphs.
   */
   teststring[MAXSTRING - 1] = '\0';    /* so there's no chance we leave array  */
   while (fgets (teststring, MAXSTRING-1, stdin) != NULL) {
      sscanf (teststring, "%X", &loc);      /* loc == the Uniocde code point    */
      gstart = strchr (teststring,':') + 1; /* start of glyph bitmap            */
      if (combining[loc]) {                 /* if a combining character         */
         if (strlen (gstart) < 35)
            add_single_circle (gstart);                 /* single-width */
         else
            add_double_circle (gstart, x_offset[loc]);  /* double-width */
      }
      printf ("%s", teststring); /* output the new character .hex string */
   }

   exit (EXIT_SUCCESS);
}
```

Here is the call graph for this function:




## 5.22 unigencircles.c

Go to the documentation of this file.

```
/**
   @file unigencircles.c

   @brief unigencircles - Superimpose dashed combining circles
                          on combining glyphs

   @author Paul Hardy

   @copyright Copyright (C) 2013, Paul Hardy.
*/
```

```
/*
  LICENSE:

     This program is free software: you can redistribute it and/or modify
     it under the terms of the GNU General Public License as published by
     the Free Software Foundation, either version 2 of the License, or
     (at your option) any later version.

     This program is distributed in the hope that it will be useful,
     but WITHOUT ANY WARRANTY; without even the implied warranty of
     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
     GNU General Public License for more details.

     You should have received a copy of the GNU General Public License
     along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

/*
  8 July 2017 [Paul Hardy]:
     - Reads new second field that contains an x-axis offset for
       each combining character in "*combining.txt" files.
     - Uses the above x-axis offset value for a combining character
       to print combining circle in the left half of a double
       diacritic combining character grid, or in the center for
       other combining characters.
     - Adds exceptions for U+01107F (Brahmi number joiner) and
       U+01D1A0 (vertical stroke musical ornament); they are in
       a combining.txt file for positioning, but are not actually
       Unicode combining characters.
     - Typo fix: "single-width"-->"double-width" in comment for
       add_double_circle function.

  12 August 2017 [Paul Hardy]:
     - Hard-code Miao vowels to show combining circles after
       removing them from font/plane01/plane01-combining.txt.

  26 December 2017 [Paul Hardy]:
     - Remove Miao hard-coding; they are back in unibmp2hex.c and
       in font/plane01/plane01-combining.txt.

  11 May 2019 [Paul Hardy]:
     - Changed strncpy calls to memcpy calls to avoid a compiler
       warning.
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXSTRING   256   ///< Maximum input line length - 1.


/**
  @brief The main function.

  @param[in] argc The count of command line arguments.
  @param[in] argv Pointer to array of command line arguments.
  @return This program exits with status EXIT_SUCCESS.
*/
int
main (int argc, char **argv)
{

  char teststring[MAXSTRING];  /* current input line                      */
  int  loc;                    /* Unicode code point of current input line */
  int  offset;                 /* offset value of a combining character    */
  char *gstart;                /* glyph start, pointing into teststring    */

  char combining[0x110000];     /* 1 --> combining glyph; 0 --> non-combining */
  char x_offset [0x110000];     /* second value in *combining.txt files       */

  void add_single_circle(char *);       /* add a single-width dashed circle */
  void add_double_circle(char *, int); /* add a double-width dashed circle */

  FILE *infilefp;

  /*
     if (argc != 3) {
        fprintf (stderr,
```

```
               ”\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n”);
         exit (EXIT_FAILURE);
      }
   */

   /*
      Read the combining characters list.
   */
   /* Start with no combining code points flagged */
   memset (combining, 0, 0x110000 * sizeof (char));
   memset (x_offset , 0, 0x110000 * sizeof (char));

   if ((infilefp = fopen (argv[1],”r”)) == NULL) {
      fprintf (stderr,”ERROR - combining characters file %s not found.\n\n”,
              argv[1]);
      exit (EXIT_FAILURE);
   }

   /* Flag list of combining characters to add a dashed circle. */
   while (fscanf (infilefp, ”%X:%d”, &loc, &offset) != EOF) {
      /*
         U+01107F and U+01D1A0 are not defined as combining characters
         in Unicode; they were added in a combining.txt file as the
         only way to make them look acceptable in proximity to other
         glyphs in their script.
      */
      if (loc != 0x01107F && loc != 0x01D1A0) {
         combining[loc] = 1;
         x_offset [loc] = offset;
      }
   }
   fclose (infilefp); /* all done reading combining.txt */

   /* Now read the non-printing glyphs; they never have dashed circles */
   if ((infilefp = fopen (argv[2],”r”)) == NULL) {
      fprintf (stderr,”ERROR - nonprinting characters file %s not found.\n\n”,
              argv[1]);
      exit (EXIT_FAILURE);
   }

   /* Reset list of nonprinting characters to avoid adding a dashed circle. */
   while (fscanf (infilefp, ”%X:%*s”, &loc) != EOF) combining[loc] = 0;

   fclose (infilefp); /* all done reading nonprinting.hex */

   /*
      Read the hex glyphs.
   */
   teststring[MAXSTRING - 1] = ’\0’;   /* so there’s no chance we leave array  */
   while (fgets (teststring, MAXSTRING-1, stdin) != NULL) {
      sscanf (teststring, ”%X”, &loc);      /* loc == the Uniocde code point    */
      gstart = strchr (teststring,’:’) + 1; /* start of glyph bitmap            */
      if (combining[loc]) {                  /* if a combining character        */
         if (strlen (gstart) < 35)
            add_single_circle (gstart);                  /* single-width */
         else
            add_double_circle (gstart, x_offset[loc]);  /* double-width */
      }
      printf (”%s”, teststring); /* output the new character .hex string */
   }

   exit (EXIT_SUCCESS);
}


/**
   @brief Superimpose a single-width dashed combining circle on a glyph bitmap.

   @param[in,out] glyphstring A single-width glyph, 8x16 pixels.
*/
void
add_single_circle (char *glyphstring)
{

   char newstring[256];
   /* Circle hex string pattern is ”00000008000024004200240000000000” */
   char circle[32]={0x0,0x0,  /* row  1 */
                  0x0,0x0,  /* row  2 */
                  0x0,0x0,  /* row  3 */
                  0x0,0x0,  /* row  4 */
                  0x0,0x0,  /* row  5 */
```

```
                  0x0,0x0,  /* row  6 */
                  0x2,0x4,  /* row  7 */
                  0x0,0x0,  /* row  8 */
                  0x4,0x2,  /* row  9 */
                  0x0,0x0,  /* row 10 */
                  0x2,0x4,  /* row 11 */
                  0x0,0x0,  /* row 12 */
                  0x0,0x0,  /* row 13 */
                  0x0,0x0,  /* row 14 */
                  0x0,0x0,  /* row 15 */
                  0x0,0x0}; /* row 16 */

   int digit1, digit2; /* corresponding digits in each string */

   int i; /* index variables */

   /* for each character position, OR the corresponding circle glyph value */
   for (i = 0; i < 32; i++) {
      glyphstring[i] = toupper (glyphstring[i]);

      /* Convert ASCII character to a hexadecimal integer */
      digit1 = (glyphstring[i] <= '9') ?
            (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);

      /* Superimpose dashed circle */
      digit2 = digit1 | circle[i];

      /* Convert hexadecimal integer to an ASCII character */
      newstring[i] = (digit2 <= 9) ?
                ('0' + digit2) : ('A' + digit2 - 0xA);
   }

   /* Terminate string for output */
   newstring[i++] = '\n';
   newstring[i++] = '\0';

   memcpy (glyphstring, newstring, i);

   return;
}


/**
   @brief Superimpose a double-width dashed combining circle on a glyph bitmap.

   @param[in,out] glyphstring A double-width glyph, 16x16 pixels.
*/
void
add_double_circle (char *glyphstring, int offset)
{

   char newstring[256];
   /* Circle hex string pattern is "00000008000024004200240000000000" */

   /* For double diacritical glyphs (offset = -8) */
   /* Combining circle is left-justified.          */
   char circle08[64]={0x0,0x0,0x0,0x0,  /* row  1 */
                      0x0,0x0,0x0,0x0,  /* row  2 */
                      0x0,0x0,0x0,0x0,  /* row  3 */
                      0x0,0x0,0x0,0x0,  /* row  4 */
                      0x0,0x0,0x0,0x0,  /* row  5 */
                      0x0,0x0,0x0,0x0,  /* row  6 */
                      0x2,0x4,0x0,0x0,  /* row  7 */
                      0x0,0x0,0x0,0x0,  /* row  8 */
                      0x4,0x2,0x0,0x0,  /* row  9 */
                      0x0,0x0,0x0,0x0,  /* row 10 */
                      0x2,0x4,0x0,0x0,  /* row 11 */
                      0x0,0x0,0x0,0x0,  /* row 12 */
                      0x0,0x0,0x0,0x0,  /* row 13 */
                      0x0,0x0,0x0,0x0,  /* row 14 */
                      0x0,0x0,0x0,0x0,  /* row 15 */
                      0x0,0x0,0x0,0x0}; /* row 16 */

   /* For all other combining glyphs (offset = -16) */
   /* Combining circle is centered in 16 columns.   */
   char circle16[64]={0x0,0x0,0x0,0x0,  /* row  1 */
                      0x0,0x0,0x0,0x0,  /* row  2 */
                      0x0,0x0,0x0,0x0,  /* row  3 */
                      0x0,0x0,0x0,0x0,  /* row  4 */
                      0x0,0x0,0x0,0x0,  /* row  5 */
                      0x0,0x0,0x0,0x0,  /* row  6 */
```

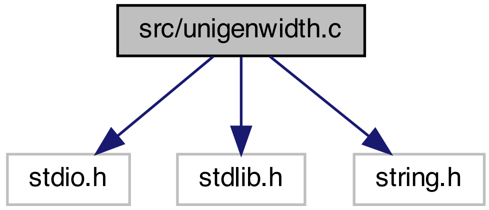
```
                  0x0,0x2,0x4,0x0,  /* row  7 */
                  0x0,0x0,0x0,0x0,  /* row  8 */
                  0x0,0x4,0x2,0x0,  /* row  9 */
                  0x0,0x0,0x0,0x0,  /* row 10 */
                  0x0,0x2,0x4,0x0,  /* row 11 */
                  0x0,0x0,0x0,0x0,  /* row 12 */
                  0x0,0x0,0x0,0x0,  /* row 13 */
                  0x0,0x0,0x0,0x0,  /* row 14 */
                  0x0,0x0,0x0,0x0,  /* row 15 */
                  0x0,0x0,0x0,0x0}; /* row 16 */

   char *circle; /* points into circle16 or circle08 */

   int digit1, digit2; /* corresponding digits in each string */

   int i; /* index variables */


   /*
      Determine if combining circle is left-justified (offset = -8)
      or centered (offset = -16).
   */
   circle = (offset >= -8) ? circle08 : circle16;

   /* for each character position, OR the corresponding circle glyph value */
   for (i = 0; i < 64; i++) {
      glyphstring[i] = toupper (glyphstring[i]);

      /* Convert ASCII character to a hexadecimal integer */
      digit1 = (glyphstring[i] <= '9') ?
               (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);

      /* Superimpose dashed circle */
      digit2 = digit1 | circle[i];

      /* Convert hexadecimal integer to an ASCII character */
      newstring[i] = (digit2 <= 9) ?
                     ('0' + digit2) : ('A' + digit2 - 0xA);
   }

   /* Terminate string for output */
   newstring[i++] = '\n';
   newstring[i++] = '\0';

   memcpy (glyphstring, newstring, i);

   return;
}
```

# 5.23 src/unigenwidth.c File Reference

unigenwidth - IEEE 1003.1-2008 setup to calculate wchar_t string widths

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Include dependency graph for unigenwidth.c:




## Macros

- #define MAXSTRING 256
  Maximum input line length - 1.
- #define PIKTO_START 0x0F0E70
  Start of Pikto code point range.
- #define PIKTO_END 0x0F11EF
  End of Pikto code point range.
- #define PIKTO_SIZE (PIKTO_END - PIKTO_START + 1)

## Functions

- int main (int argc, char **argv)
  The main function.

### 5.23.1 Detailed Description

unigenwidth - IEEE 1003.1-2008 setup to calculate wchar_t string widths

Author

Paul Hardy.

Copyright

Copyright (C) 2013, 2017 Paul Hardy.

All glyphs are treated as 16 pixels high, and can be 8, 16, 24, or 32 pixels wide (resulting in widths of 1, 2, 3, or 4, respectively).

Definition in file unigenwidth.c.

## 5.23.2 Macro Definition Documentation

#### 5.23.2.1 MAXSTRING

```
#define MAXSTRING 256
```

Maximum input line length - 1.

Definition at line 46 of file unigenwidth.c.

#### 5.23.2.2 PIKTO_END

```
#define PIKTO_END 0x0F11EF
```

End of Pikto code point range.

Definition at line 50 of file unigenwidth.c.

#### 5.23.2.3 PIKTO_SIZE

```
#define PIKTO_SIZE (PIKTO_END - PIKTO_START + 1)
```

Number of code points in Pikto range.

Definition at line 52 of file unigenwidth.c.

#### 5.23.2.4 PIKTO_START

```
#define PIKTO_START 0x0F0E70
```

Start of Pikto code point range.

Definition at line 49 of file unigenwidth.c.

## 5.23.3 Function Documentation

#### 5.23.3.1 main()

```
int main (
        int argc,
        char ** argv )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 63 of file unigenwidth.c.

```
00064 {
00065
00066    int i; /* loop variable */
00067
00068    char teststring[MAXSTRING];
00069    int  loc;
00070    char *gstart;
00071
00072    char glyph_width[0x20000];
00073    char pikto_width[PIKTO_SIZE];
00074
00075    FILE *infilefp;
00076
00077    if (argc != 3) {
00078       fprintf (stderr, "\n\nUsage: %s <unifont.hex> <combining.txt>\n\n", argv[0]);
00079       exit (EXIT_FAILURE);
00080    }
00081
00082    /*
00083       Read the collection of hex glyphs.
00084    */
00085    if ((infilefp = fopen (argv[1],"r")) == NULL) {
00086       fprintf (stderr,"ERROR - hex input file %s not found.\n\n", argv[1]);
00087       exit (EXIT_FAILURE);
00088    }
00089
00090    /* Flag glyph as non-existent until found. */
00091    memset (glyph_width, -1, 0x20000 * sizeof (char));
00092    memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));
00093
00094    teststring[MAXSTRING-1] = '\0';
00095    while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
00096       sscanf (teststring, "%X:%*s", &loc);
00097       if (loc < 0x20000) {
00098          gstart = strchr (teststring,':') + 1;
00099          /*
00100             16 rows per glyph, 2 ASCII hexadecimal digits per byte,
00101             so divide number of digits by 32 (shift right 5 bits).
00102          */
00103          glyph_width[loc] = (strlen (gstart) - 1) >> 5;
00104       }
00105       else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
00106          gstart = strchr (teststring,':') + 1;
```

```
            pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
         }
      }

      fclose (infilefp);

      /*
         Now read the combining character code points.  These have width of 0.
      */
      if ((infilefp = fopen (argv[2],"r")) == NULL) {
         fprintf (stderr,"ERROR - combining characters file %s not found.\n\n", argv[2]);
         exit (EXIT_FAILURE);
      }

      while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
         sscanf (teststring, "%X:%*s", &loc);
         if (loc < 0x20000) glyph_width[loc] = 0;
      }

      fclose (infilefp);

      /*
         Code Points with Unusual Properties (Unicode Standard, Chapter 4).

         As of Unifont 10.0.04, use the widths in the "*-nonprinting.hex"
         files.  If an application is smart enough to know how to handle
         these special cases, it will not render the "nonprinting" glyph
         and will treat the code point as being zero-width.
      */
//    glyph_width[0]=0; /* NULL character */
//    for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
//    for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */

//    glyph_width[0x034F]=0; /* combining grapheme joiner              */
//    glyph_width[0x180B]=0; /* Mongolian free variation selector one   */
//    glyph_width[0x180C]=0; /* Mongolian free variation selector two   */
//    glyph_width[0x180D]=0; /* Mongolian free variation selector three */
//    glyph_width[0x180E]=0; /* Mongolian vowel separator               */
//    glyph_width[0x200B]=0; /* zero width space                        */
//    glyph_width[0x200C]=0; /* zero width non-joiner                   */
//    glyph_width[0x200D]=0; /* zero width joiner                       */
//    glyph_width[0x200E]=0; /* left-to-right mark                      */
//    glyph_width[0x200F]=0; /* right-to-left mark                      */
//    glyph_width[0x202A]=0; /* left-to-right embedding                 */
//    glyph_width[0x202B]=0; /* right-to-left embedding                 */
//    glyph_width[0x202C]=0; /* pop directional formatting              */
//    glyph_width[0x202D]=0; /* left-to-right override                  */
//    glyph_width[0x202E]=0; /* right-to-left override                  */
//    glyph_width[0x2060]=0; /* word joiner                             */
//    glyph_width[0x2061]=0; /* function application                    */
//    glyph_width[0x2062]=0; /* invisible times                         */
//    glyph_width[0x2063]=0; /* invisible separator                     */
//    glyph_width[0x2064]=0; /* invisible plus                          */
//    glyph_width[0x206A]=0; /* inhibit symmetric swapping              */
//    glyph_width[0x206B]=0; /* activate symmetric swapping             */
//    glyph_width[0x206C]=0; /* inhibit arabic form shaping             */
//    glyph_width[0x206D]=0; /* activate arabic form shaping            */
//    glyph_width[0x206E]=0; /* national digit shapes                   */
//    glyph_width[0x206F]=0; /* nominal digit shapes                    */

//    /* Variation Selector-1 to Variation Selector-16 */
//    for (i = 0xFE00; i <= 0xFE0F; i++) glyph_width[i] = 0;

//    glyph_width[0xFEFF]=0; /* zero width no-break space         */
//    glyph_width[0xFFF9]=0; /* interlinear annotation anchor     */
//    glyph_width[0xFFFA]=0; /* interlinear annotation separator  */
//    glyph_width[0xFFFB]=0; /* interlinear annotation terminator */
      /*
         Let glyph widths represent 0xFFFC (object replacement character)
         and 0xFFFD (replacement character).
      */

      /*
         Hangul Jamo:

            Leading Consonant (Choseong): leave spacing as is.

            Hangul Choseong Filler (U+115F): set width to 2.

            Hangul Jungseong Filler, Hangul Vowel (Jungseong), and
            Final Consonant (Jongseong): set width to 0, because these
```

```
      combine with the leading consonant as one composite syllabic
      glyph.  As of Unicode 5.2, the Hangul Jamo block (U+1100..U+11FF)
      is completely filled.
*/
// for (i = 0x1160; i <= 0x11FF; i++) glyph_width[i]=0; /* Vowels & Final Consonants */

/*
   Private Use Area -- the width is undefined, but likely
   to be 2 charcells wide either from a graphic glyph or
   from a four-digit hexadecimal glyph representing the
   code point.  Therefore if any PUA glyph does not have
   a non-zero width yet, assign it a default width of 2.
   The Unicode Standard allows giving PUA characters
   default property values; see for example The Unicode
   Standard Version 5.0, p. 91.  This same default is
   used for higher plane PUA code points below.
*/
// for (i = 0xE000; i <= 0xF8FF; i++) {
//    if (glyph_width[i] == 0) glyph_width[i]=2;
// }

/*
   <not a character>
*/
for (i = 0xFDD0; i <= 0xFDEF; i++) glyph_width[i] = -1;
glyph_width[0xFFFE] = -1; /* Byte Order Mark */
glyph_width[0xFFFF] = -1; /* Byte Order Mark */

/* Surrogate Code Points */
for (i = 0xD800; i <= 0xDFFF; i++) glyph_width[i]=-1;

/* CJK Code Points */
for (i = 0x4E00; i <= 0x9FFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
for (i = 0x3400; i <= 0x4DBF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
for (i = 0xF900; i <= 0xFAFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;

/*
   Now generate the output file.
*/
printf ("/*\n");
printf ("   wcwidth and wcswidth functions, as per IEEE 1003.1-2008\n");
printf ("   System Interfaces, pp. 2241 and 2251.\n\n");
printf ("   Author: Paul Hardy, 2013\n\n");
printf ("   Copyright (c) 2013 Paul Hardy\n\n");
printf ("   LICENSE:\n");
printf ("\n");
printf ("      This program is free software: you can redistribute it and/or modify\n");
printf ("      it under the terms of the GNU General Public License as published by\n");
printf ("      the Free Software Foundation, either version 2 of the License, or\n");
printf ("      (at your option) any later version.\n");
printf ("\n");
printf ("      This program is distributed in the hope that it will be useful,\n");
printf ("      but WITHOUT ANY WARRANTY; without even the implied warranty of\n");
printf ("      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the\n");
printf ("      GNU General Public License for more details.\n");
printf ("\n");
printf ("      You should have received a copy of the GNU General Public License\n");
printf ("      along with this program.  If not, see <http://www.gnu.org/licenses/>.\n");
printf ("*/\n\n");

printf ("#include <wchar.h>\n\n");
printf ("/* Definitions for Pikto CSUR Private Use Area glyphs */\n");
printf ("#define PIKTO_START\t0x%06X\n", PIKTO_START);
printf ("#define PIKTO_END\t0x%06X\n", PIKTO_END);
printf ("#define PIKTO_SIZE\t(PIKTO_END - PIKTO_START + 1)\n");
printf ("\n\n");
printf ("/* wcwidth -- return charcell positions of one code point */\n");
printf ("inline int\nwcwidth (wchar_t wc)\n{\n");
printf ("   return (wcswidth (&wc, 1));\n");
printf ("}\n");
printf ("\n\n");
printf ("int\nwcswidth (const wchar_t *pwcs, size_t n)\n{\n\n");
printf ("   int i;                    /* loop variable                                      */\n");
printf ("   unsigned codept;          /* Unicode code point of current character            */\n");
printf ("   unsigned plane;           /* Unicode plane, 0x00..0x10                          */\n");
printf ("   unsigned lower17;         /* lower 17 bits of Unicode code point                */\n");
printf ("   unsigned lower16;         /* lower 16 bits of Unicode code point                */\n");
printf ("   int lowpt, midpt, highpt; /* for binary searching in plane1zeroes[]             */\n");
printf ("   int found;                /* for binary searching in plane1zeroes[]             */\n");
printf ("   int totalwidth;           /* total width of string, in charcells (1 or 2/glyph) */\n");
printf ("   int illegalchar;          /* Whether or not this code point is illegal          */\n");
```

```
putchar ('\n');

/*
   Print the glyph_width[] array for glyphs widths in the
   Basic Multilingual Plane (Plane 0).
*/
printf ("   char glyph_width[0x20000] = {");
for (i = 0; i < 0x10000; i++) {
   if ((i & 0x1F) == 0)
      printf ("\n      /* U+%04X */ ", i);
   printf ("%d,", glyph_width[i]);
}
for (i = 0x10000; i < 0x20000; i++) {
   if ((i & 0x1F) == 0)
      printf ("\n      /* U+%06X */ ", i);
   printf ("%d", glyph_width[i]);
   if (i < 0x1FFFF) putchar (',');
}
printf ("\n   };\n\n");

/*
   Print the pikto_width[] array for Pikto glyph widths.
*/
printf ("   char pikto_width[PIKTO_SIZE] = {");
for (i = 0; i < PIKTO_SIZE; i++) {
   if ((i & 0x1F) == 0)
      printf ("\n      /* U+%06X */ ", PIKTO_START + i);
   printf ("%d", pikto_width[i]);
   if ((PIKTO_START + i) < PIKTO_END) putchar (',');
}
printf ("\n   };\n\n");

/*
   Execution part of wcswidth.
*/
printf ("\n");
printf ("   illegalchar = totalwidth = 0;\n");
printf ("   for (i = 0; !illegalchar && i < n; i++) {\n");
printf ("      codept  = pwcs[i];\n");
printf ("      plane   = codept >> 16;\n");
printf ("      lower17 = codept & 0x1FFFF;\n");
printf ("      lower16 = codept & 0xFFFF;\n");
printf ("      if (plane < 2) { /* the most common case */\n");
printf ("         if (glyph_width[lower17] < 0) illegalchar = 1;\n");
printf ("         else totalwidth += glyph_width[lower17];\n");
printf ("      }\n");
printf ("      else { /* a higher plane or beyond Unicode range */\n");
printf ("         if  ((lower16 == 0xFFFE) || (lower16 == 0xFFFF)) {\n");
printf ("            illegalchar = 1;\n");
printf ("         }\n");
printf ("         else if (plane < 4) {  /* Ideographic Plane */\n");
printf ("            totalwidth += 2; /* Default ideographic width */\n");
printf ("         }\n");
printf ("         else if (plane == 0x0F) {  /* CSUR Private Use Area */\n");
printf ("            if (lower16 <= 0x0E6F) { /* Kinya */\n");
printf ("               totalwidth++; /* all Kinya syllables have width 1 */\n");
printf ("            }\n");
printf ("            else if (lower16 <= (PIKTO_END & 0xFFFF)) { /* Pikto */\n");
printf ("               if (pikto_width[lower16 - (PIKTO_START & 0xFFFF)] < 0) illegalchar = 1;\n");
printf ("               else totalwidth += pikto_width[lower16 - (PIKTO_START & 0xFFFF)];\n");
printf ("            }\n");
printf ("         }\n");
printf ("         else if (plane > 0x10) {\n");
printf ("            illegalchar = 1;\n");
printf ("         }\n");
printf ("         /* Other non-printing in higher planes; return -1 as per IEEE 1003.1-2008. */\n");
printf ("         else if (/* language tags */\n");
printf ("                  codept == 0x0E0001 || (codept >= 0x0E0020 && codept <= 0x0E007F) ||\n");
printf ("                  /* variation selectors, 0x0E0100..0x0E01EF */\n");
printf ("                  (codept >= 0x0E0100 && codept <= 0x0E01EF)) {\n");
printf ("            illegalchar = 1;\n");
printf ("         }\n");
printf ("         /*\n");
printf ("            Unicode plane 0x02..0x10 printing character\n");
printf ("         */\n");
printf ("         else {\n");
printf ("            illegalchar = 1; /* code is not in font */\n");
printf ("         }\n");
printf ("\n");
printf ("      }\n");
printf ("   }\n");
```

```
    printf ("    if (illegalchar) totalwidth = -1;\n");
    printf ("\n");
    printf ("    return (totalwidth);\n");
    printf ("\n");
    printf ("}\n");

    exit (EXIT_SUCCESS);
}
```

## 5.24 unigenwidth.c

Go to the documentation of this file.

```
/**
   @file unigenwidth.c

   @brief unigenwidth - IEEE 1003.1-2008 setup to calculate
                        wchar_t string widths

   @author Paul Hardy.

   @copyright Copyright (C) 2013, 2017 Paul Hardy.

   All glyphs are treated as 16 pixels high, and can be
   8, 16, 24, or 32 pixels wide (resulting in widths of
   1, 2, 3, or 4, respectively).
*/
/*
   LICENSE:

      This program is free software: you can redistribute it and/or modify
      it under the terms of the GNU General Public License as published by
      the Free Software Foundation, either version 2 of the License, or
      (at your option) any later version.

      This program is distributed in the hope that it will be useful,
      but WITHOUT ANY WARRANTY; without even the implied warranty of
      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
      GNU General Public License for more details.

      You should have received a copy of the GNU General Public License
      along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

/*
   20 June 2017 [Paul Hardy]:
      - Now handles glyphs that are 24 or 32 pixels wide.

   8 July 2017 [Paul Hardy]:
      - Modifies sscanf format strings to ignore second field after
        the ":" field separator, newly added to "*combining.txt" files
        and already present in "*.hex" files.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSTRING    256    ///< Maximum input line length - 1.

/* Definitions for Pikto in Plane 15 */
#define PIKTO_START 0x0F0E70    ///< Start of Pikto code point range.
#define PIKTO_END   0x0F11EF    ///< End of Pikto code point range.
/** Number of code points in Pikto range. */
#define PIKTO_SIZE  (PIKTO_END - PIKTO_START + 1)


/**
   @brief The main function.

   @param[in] argc The count of command line arguments.
   @param[in] argv Pointer to array of command line arguments.
   @return This program exits with status EXIT_SUCCESS.
*/
int
main (int argc, char **argv)
```

```
{

   int i; /* loop variable */

   char teststring[MAXSTRING];
   int  loc;
   char *gstart;

   char glyph_width[0x20000];
   char pikto_width[PIKTO_SIZE];

   FILE *infilefp;

   if (argc != 3) {
      fprintf (stderr, "\n\nUsage: %s <unifont.hex> <combining.txt>\n\n", argv[0]);
      exit (EXIT_FAILURE);
   }

   /*
      Read the collection of hex glyphs.
   */
   if ((infilefp = fopen (argv[1],"r")) == NULL) {
      fprintf (stderr,"ERROR - hex input file %s not found.\n\n", argv[1]);
      exit (EXIT_FAILURE);
   }

   /* Flag glyph as non-existent until found. */
   memset (glyph_width, -1, 0x20000 * sizeof (char));
   memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));

   teststring[MAXSTRING-1] = '\0';
   while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
      sscanf (teststring, "%X:%*s", &loc);
      if (loc < 0x20000) {
         gstart = strchr (teststring,':') + 1;
         /*
            16 rows per glyph, 2 ASCII hexadecimal digits per byte,
            so divide number of digits by 32 (shift right 5 bits).
         */
         glyph_width[loc] = (strlen (gstart) - 1) » 5;
      }
      else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
         gstart = strchr (teststring,':') + 1;
         pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
      }
   }

   fclose (infilefp);

   /*
      Now read the combining character code points.  These have width of 0.
   */
   if ((infilefp = fopen (argv[2],"r")) == NULL) {
      fprintf (stderr,"ERROR - combining characters file %s not found.\n\n", argv[2]);
      exit (EXIT_FAILURE);
   }

   while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
      sscanf (teststring, "%X:%*s", &loc);
      if (loc < 0x20000) glyph_width[loc] = 0;
   }

   fclose (infilefp);

   /*
      Code Points with Unusual Properties (Unicode Standard, Chapter 4).

      As of Unifont 10.0.04, use the widths in the "*-nonprinting.hex"
      files.  If an application is smart enough to know how to handle
      these special cases, it will not render the "nonprinting" glyph
      and will treat the code point as being zero-width.
   */
// glyph_width[0]=0; /* NULL character */
// for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
// for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */

// glyph_width[0x034F]=0; /* combining grapheme joiner            */
// glyph_width[0x180B]=0; /* Mongolian free variation selector one   */
// glyph_width[0x180C]=0; /* Mongolian free variation selector two   */
// glyph_width[0x180D]=0; /* Mongolian free variation selector three */
// glyph_width[0x180E]=0; /* Mongolian vowel separator            */
```

```
// glyph_width[0x200B]=0; /* zero width space                   */
// glyph_width[0x200C]=0; /* zero width non-joiner              */
// glyph_width[0x200D]=0; /* zero width joiner                  */
// glyph_width[0x200E]=0; /* left-to-right mark                 */
// glyph_width[0x200F]=0; /* right-to-left mark                 */
// glyph_width[0x202A]=0; /* left-to-right embedding            */
// glyph_width[0x202B]=0; /* right-to-left embedding            */
// glyph_width[0x202C]=0; /* pop directional formatting         */
// glyph_width[0x202D]=0; /* left-to-right override             */
// glyph_width[0x202E]=0; /* right-to-left override             */
// glyph_width[0x2060]=0; /* word joiner                        */
// glyph_width[0x2061]=0; /* function application               */
// glyph_width[0x2062]=0; /* invisible times                    */
// glyph_width[0x2063]=0; /* invisible separator                */
// glyph_width[0x2064]=0; /* invisible plus                     */
// glyph_width[0x206A]=0; /* inhibit symmetric swapping         */
// glyph_width[0x206B]=0; /* activate symmetric swapping        */
// glyph_width[0x206C]=0; /* inhibit arabic form shaping        */
// glyph_width[0x206D]=0; /* activate arabic form shaping       */
// glyph_width[0x206E]=0; /* national digit shapes              */
// glyph_width[0x206F]=0; /* nominal digit shapes               */

// /* Variation Selector-1 to Variation Selector-16 */
// for (i = 0xFE00; i <= 0xFE0F; i++) glyph_width[i] = 0;

// glyph_width[0xFEFF]=0; /* zero width no-break space         */
// glyph_width[0xFFF9]=0; /* interlinear annotation anchor     */
// glyph_width[0xFFFA]=0; /* interlinear annotation separator  */
// glyph_width[0xFFFB]=0; /* interlinear annotation terminator */
   /*
      Let glyph widths represent 0xFFFC (object replacement character)
      and 0xFFFD (replacement character).
   */

   /*
      Hangul Jamo:

         Leading Consonant (Choseong): leave spacing as is.

         Hangul Choseong Filler (U+115F): set width to 2.

         Hangul Jungseong Filler, Hangul Vowel (Jungseong), and
         Final Consonant (Jongseong): set width to 0, because these
         combine with the leading consonant as one composite syllabic
         glyph.  As of Unicode 5.2, the Hangul Jamo block (U+1100..U+11FF)
         is completely filled.
   */
   // for (i = 0x1160; i <= 0x11FF; i++) glyph_width[i]=0; /* Vowels & Final Consonants */

   /*
      Private Use Area -- the width is undefined, but likely
      to be 2 charcells wide either from a graphic glyph or
      from a four-digit hexadecimal glyph representing the
      code point.  Therefore if any PUA glyph does not have
      a non-zero width yet, assign it a default width of 2.
      The Unicode Standard allows giving PUA characters
      default property values; see for example The Unicode
      Standard Version 5.0, p. 91.  This same default is
      used for higher plane PUA code points below.
   */
   // for (i = 0xE000; i <= 0xF8FF; i++) {
   //    if (glyph_width[i] == 0) glyph_width[i]=2;
   // }

   /*
      <not a character>
   */
   for (i = 0xFDD0; i <= 0xFDEF; i++) glyph_width[i] = -1;
   glyph_width[0xFFFE] = -1; /* Byte Order Mark */
   glyph_width[0xFFFF] = -1; /* Byte Order Mark */

   /* Surrogate Code Points */
   for (i = 0xD800; i <= 0xDFFF; i++) glyph_width[i]=-1;

   /* CJK Code Points */
   for (i = 0x4E00; i <= 0x9FFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
   for (i = 0x3400; i <= 0x4DBF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
   for (i = 0xF900; i <= 0xFAFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;

   /*
      Now generate the output file.
```

```c
*/
printf ("/*\n");
printf ("   wcwidth and wcswidth functions, as per IEEE 1003.1-2008\n");
printf ("   System Interfaces, pp. 2241 and 2251.\n\n");
printf ("   Author: Paul Hardy, 2013\n\n");
printf ("   Copyright (c) 2013 Paul Hardy\n\n");
printf ("   LICENSE:\n");
printf ("\n");
printf ("      This program is free software: you can redistribute it and/or modify\n");
printf ("      it under the terms of the GNU General Public License as published by\n");
printf ("      the Free Software Foundation, either version 2 of the License, or\n");
printf ("      (at your option) any later version.\n");
printf ("\n");
printf ("      This program is distributed in the hope that it will be useful,\n");
printf ("      but WITHOUT ANY WARRANTY; without even the implied warranty of\n");
printf ("      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the\n");
printf ("      GNU General Public License for more details.\n");
printf ("\n");
printf ("      You should have received a copy of the GNU General Public License\n");
printf ("      along with this program.  If not, see <http://www.gnu.org/licenses/>.\n");
printf ("*/\n\n");

printf ("#include <wchar.h>\n\n");
printf ("/* Definitions for Pikto CSUR Private Use Area glyphs */\n");
printf ("#define PIKTO_START\t0x%06X\n", PIKTO_START);
printf ("#define PIKTO_END\t0x%06X\n", PIKTO_END);
printf ("#define PIKTO_SIZE\t(PIKTO_END - PIKTO_START + 1)\n");
printf ("\n\n");
printf ("/* wcwidth -- return charcell positions of one code point */\n");
printf ("inline int\nwcwidth (wchar_t wc)\n{\n");
printf ("   return (wcswidth (&wc, 1));\n");
printf ("}\n");
printf ("\n\n");
printf ("int\nwcswidth (const wchar_t *pwcs, size_t n)\n{\n\n");
printf ("   int i;                    /* loop variable                                      */\n");
printf ("   unsigned codept;          /* Unicode code point of current character            */\n");
printf ("   unsigned plane;           /* Unicode plane, 0x00..0x10                          */\n");
printf ("   unsigned lower17;         /* lower 17 bits of Unicode code point                */\n");
printf ("   unsigned lower16;         /* lower 16 bits of Unicode code point                */\n");
printf ("   int lowpt, midpt, highpt; /* for binary searching in plane1zeroes[]             */\n");
printf ("   int found;                /* for binary searching in plane1zeroes[]             */\n");
printf ("   int totalwidth;           /* total width of string, in charcells (1 or 2/glyph) */\n");
printf ("   int illegalchar;          /* Whether or not this code point is illegal          */\n");
putchar ('\n');

/*
   Print the glyph_width[] array for glyphs widths in the
   Basic Multilingual Plane (Plane 0).
*/
printf ("   char glyph_width[0x20000] = {");
for (i = 0; i < 0x10000; i++) {
   if ((i & 0x1F) == 0)
      printf ("\n      /* U+%04X */ ", i);
   printf ("%d,", glyph_width[i]);
}
for (i = 0x10000; i < 0x20000; i++) {
   if ((i & 0x1F) == 0)
      printf ("\n      /* U+%06X */ ", i);
   printf ("%d", glyph_width[i]);
   if (i < 0x1FFFF) putchar (',');
}
printf ("\n   };\n\n");

/*
   Print the pikto_width[] array for Pikto glyph widths.
*/
printf ("   char pikto_width[PIKTO_SIZE] = {");
for (i = 0; i < PIKTO_SIZE; i++) {
   if ((i & 0x1F) == 0)
      printf ("\n      /* U+%06X */ ", PIKTO_START + i);
   printf ("%d", pikto_width[i]);
   if ((PIKTO_START + i) < PIKTO_END) putchar (',');
}
printf ("\n   };\n\n");

/*
   Execution part of wcswidth.
*/
printf ("\n");
printf ("   illegalchar = totalwidth = 0;\n");
printf ("   for (i = 0; !illegalchar && i < n; i++) {\n");
```

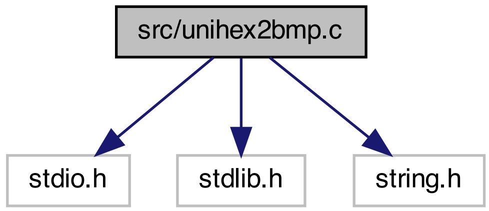
```
printf ("        codept  = pwcs[i];\n");
printf ("        plane   = codept » 16;\n");
printf ("        lower17 = codept & 0x1FFFF;\n");
printf ("        lower16 = codept & 0xFFFF;\n");
printf ("        if (plane < 2) { /* the most common case */\n");
printf ("           if (glyph_width[lower17] < 0) illegalchar = 1;\n");
printf ("           else totalwidth += glyph_width[lower17];\n");
printf ("        }\n");
printf ("        else { /* a higher plane or beyond Unicode range */\n");
printf ("           if  ((lower16 == 0xFFFE) || (lower16 == 0xFFFF)) {\n");
printf ("              illegalchar = 1;\n");
printf ("           }\n");
printf ("           else if (plane < 4) {  /* Ideographic Plane */\n");
printf ("              totalwidth += 2; /* Default ideographic width */\n");
printf ("           }\n");
printf ("           else if (plane == 0x0F) {  /* CSUR Private Use Area */\n");
printf ("              if (lower16 <= 0x0E6F) { /* Kinya */\n");
printf ("                 totalwidth++; /* all Kinya syllables have width 1 */\n");
printf ("              }\n");
printf ("              else if (lower16 <= (PIKTO_END & 0xFFFF)) { /* Pikto */\n");
printf ("                 if (pikto_width[lower16 - (PIKTO_START & 0xFFFF)] < 0) illegalchar = 1;\n");
printf ("                 else totalwidth += pikto_width[lower16 - (PIKTO_START & 0xFFFF)];\n");
printf ("              }\n");
printf ("           }\n");
printf ("           else if (plane > 0x10) {\n");
printf ("              illegalchar = 1;\n");
printf ("           }\n");
printf ("           /* Other non-printing in higher planes; return -1 as per IEEE 1003.1-2008. */\n");
printf ("           else if (/* language tags */\n");
printf ("                    codept == 0x0E0001 || (codept >= 0x0E0020 && codept <= 0x0E007F) ||\n");
printf ("                    /* variation selectors, 0x0E0100..0x0E01EF */\n");
printf ("                    (codept >= 0x0E0100 && codept <= 0x0E01EF)) {\n");
printf ("              illegalchar = 1;\n");
printf ("           }\n");
printf ("           /*\n");
printf ("              Unicode plane 0x02..0x10 printing character\n");
printf ("           */\n");
printf ("           else {\n");
printf ("              illegalchar = 1; /* code is not in font */\n");
printf ("           }\n");
printf ("\n");
printf ("        }\n");
printf ("     }\n");
printf ("     if (illegalchar) totalwidth = -1;\n");
printf ("\n");
printf ("     return (totalwidth);\n");
printf ("\n");
printf ("}\n");

exit (EXIT_SUCCESS);
}
```

## 5.25 src/unihex2bmp.c File Reference

unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Include dependency graph for unihex2bmp.c:




## Macros

- #define MAXBUF 256

## Functions

- int main (int argc, char ∗argv[ ])

  The main function.
- int hex2bit (char ∗instring, unsigned char character[32][4])

  Generate a bitmap for one glyph.
- int init (unsigned char bitmap[17 ∗32][18 ∗4])

  Initialize the bitmap grid.

## Variables

- char ∗ hex [18]

  GNU Unifont bitmaps for hexadecimal digits.
- unsigned char hexbits [18][32]

  The digits converted into bitmaps.
- unsigned unipage =0

  Unicode page number, 0x00..0xff.
- int flip =1

  Transpose entire matrix as in Unicode book.

### 5.25.1 Detailed Description

unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing

Author

Paul Hardy, unifoundry <at> unifoundry.com, December 2007

Copyright

Copyright (C) 2007, 2008, 2013, 2017 Paul Hardy

This program reads in a GNU Unifont .hex file, extracts a range of 256 code points, and converts it a Microsoft Bitmap (.bmp) or Wireless Bitmap file.

Synopsis: unihex2bmp [-iin_file.hex] [-oout_file.bmp] [-f] [-phex_page_num] [-w]

Definition in file unihex2bmp.c.

### 5.25.2 Macro Definition Documentation

#### 5.25.2.1 MAXBUF

```
#define MAXBUF 256
```

Definition at line 47 of file unihex2bmp.c.

### 5.25.3 Function Documentation

#### 5.25.3.1 hex2bit()

```
int hex2bit (
        char * instring,
        unsigned char character[32][4] )
```

Generate a bitmap for one glyph.

Convert the portion of a hex string after the ':' into a character bitmap.

If string is >= 128 characters, it will fill all 4 bytes per row. If string is >= 64 characters and < 128, it will fill 2 bytes per row. Otherwise, it will fill 1 byte per row.

Parameters

| | | |
|---|---|---|
| in | instring | The character array containing the glyph bitmap. |
| out | character | Glyph bitmap, 8, 16, or 32 columns by 16 rows tall. |

Returns

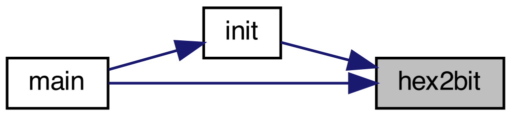
Always returns 0.

Definition at line 361 of file unihex2bmp.c.

```
{

   int i;  /* current row in bitmap character */
   int j;  /* current character in input string */
   int k;  /* current byte in bitmap character */
   int width;  /* number of output bytes to fill - 1: 0, 1, 2, or 3 */

   for (i=0; i<32; i++)  /* erase previous character */
      character[i][0] = character[i][1] = character[i][2] = character[i][3] = 0;
   j=0;  /* current location is at beginning of instring */

   if (strlen (instring) <= 34)  /* 32 + possible '\r', '\n' */
      width = 0;
   else if (strlen (instring) <= 66)  /* 64 + possible '\r', '\n' */
      width = 1;
   else if (strlen (instring) <= 98)  /* 96 + possible '\r', '\n' */
      width = 3;
   else  /* the maximum allowed is quadruple-width */
      width = 4;

   k = (width > 1) ? 0 : 1; /* if width > double, start at index 1 else at 0 */

   for (i=8; i<24; i++) {  /* 16 rows per input character, rows 8..23 */
      sscanf (&instring[j], "%2hhx", &character[i][k]);
      j += 2;
      if (width > 0) { /* add next pair of hex digits to this row */
         sscanf (&instring[j], "%2hhx", &character[i][k+1]);
         j += 2;
         if (width > 1) { /* add next pair of hex digits to this row */
            sscanf (&instring[j], "%2hhx", &character[i][k+2]);
            j += 2;
            if (width > 2) {  /* quadruple-width is maximum width */
               sscanf (&instring[j], "%2hhx", &character[i][k+3]);
               j += 2;
            }
         }
      }
   }

   return (0);
```

```
00402 }
```

Here is the caller graph for this function:




#### 5.25.3.2 init()

int init (
    unsigned char bitmap[17 ∗32][18 ∗4] )

Initialize the bitmap grid.

Parameters

| | | |
|---|---|---|
| out | bitmap | The bitmap to generate, with 32x32 pixel glyph areas. |

Returns

Always returns 0.

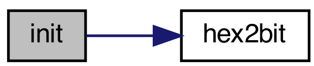
Definition at line 412 of file unihex2bmp.c.

```
00413 {
00414    int i, j;
00415    unsigned char charbits[32][4];  /* bitmap for one character, 4 bytes/row */
00416    unsigned toppixelrow;
00417    unsigned thiscol;
00418    unsigned char pnybble0, pnybble1, pnybble2, pnybble3;
00419
00420    for (i=0; i<18; i++) { /* bitmaps for '0'..'9', 'A'-'F', 'u', '+' */
00421
00422       hex2bit (&hex[i][5], charbits);  /* convert hex string to 32*4 bitmap */
```

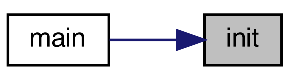
```

    for (j=0; j<32; j++) hexbits[i][j] = ~charbits[j][1];
}

/*
   Initialize bitmap to all white.
*/
for (toppixelrow=0; toppixelrow < 17*32; toppixelrow++) {
   for (thiscol=0; thiscol<18; thiscol++) {
      bitmap[toppixelrow][(thiscol « 2)    ] = 0xff;
      bitmap[toppixelrow][(thiscol « 2) | 1] = 0xff;
      bitmap[toppixelrow][(thiscol « 2) | 2] = 0xff;
      bitmap[toppixelrow][(thiscol « 2) | 3] = 0xff;
   }
}
/*
   Write the "u+nnnn" table header in the upper left-hand corner,
   where nnnn is the upper 16 bits of a 32-bit Unicode assignment.
*/
pnybble3 = (unipage » 20);
pnybble2 = (unipage » 16) & 0xf;
pnybble1 = (unipage » 12) & 0xf;
pnybble0 = (unipage »  8) & 0xf;
for (i=0; i<32; i++) {
   bitmap[i][1] = hexbits[16][i];  /* copy 'u' */
   bitmap[i][2] = hexbits[17][i];  /* copy '+' */
   bitmap[i][3] = hexbits[pnybble3][i];
   bitmap[i][4] = hexbits[pnybble2][i];
   bitmap[i][5] = hexbits[pnybble1][i];
   bitmap[i][6] = hexbits[pnybble0][i];
}
/*
   Write low-order 2 bytes of Unicode number assignments, as hex labels
*/
pnybble3 = (unipage » 4) & 0xf;  /* Highest-order hex digit */
pnybble2 = (unipage     ) & 0xf;  /* Next highest-order hex digit */
/*
   Write the column headers in bitmap[][] (row headers if flipped)
*/
toppixelrow = 32 * 17 - 1; /* maximum pixel row number */
/*
   Label the column headers. The hexbits[][] bytes are split across two
   bitmap[][] entries to center a the hex digits in a column of 4 bytes.
   OR highest byte with 0xf0 and lowest byte with 0x0f to make outer
   nybbles white (0=black, 1-white).
*/
for (i=0; i<16; i++) {
   for (j=0; j<32; j++) {
      if (flip) {  /* transpose matrix */
         bitmap[j][((i+2) « 2) | 0]  = (hexbits[pnybble3][j] » 4) | 0xf0;
         bitmap[j][((i+2) « 2) | 1]  = (hexbits[pnybble3][j] « 4) |
                                       (hexbits[pnybble2][j] » 4);
         bitmap[j][((i+2) « 2) | 2]  = (hexbits[pnybble2][j] « 4) |
                                       (hexbits[i][j] » 4);
         bitmap[j][((i+2) « 2) | 3]  = (hexbits[i][j] « 4) | 0x0f;
      }
      else {
         bitmap[j][((i+2) « 2) | 1] = (hexbits[i][j] » 4) | 0xf0;
         bitmap[j][((i+2) « 2) | 2] = (hexbits[i][j] « 4) | 0x0f;
      }
   }
}
/*
   Now use the single hex digit column graphics to label the row headers.
*/
for (i=0; i<16; i++) {
   toppixelrow = 32 * (i + 1) - 1; /* from bottom to top    */

   for (j=0; j<32; j++) {
      if (!flip) {  /* if not transposing matrix */
         bitmap[toppixelrow + j][4] = hexbits[pnybble3][j];
         bitmap[toppixelrow + j][5] = hexbits[pnybble2][j];
      }
      bitmap[toppixelrow + j][6] = hexbits[i][j];
   }
}
/*
   Now draw grid lines in bitmap, around characters we just copied.
*/
/* draw vertical lines 2 pixels wide */
for (i=1*32; i<17*32; i++) {
```

```
        if ((i & 0x1f) == 7)
            i++;
        else if ((i & 0x1f) == 14)
            i += 2;
        else if ((i & 0x1f) == 22)
            i++;
        for (j=1; j<18; j++) {
            bitmap[i][(j « 2) | 3] &= 0xfe;
        }
    }
    /* draw horizontal lines 1 pixel tall */
    for (i=1*32-1; i<18*32-1; i+=32) {
        for (j=2; j<18; j++) {
            bitmap[i][(j « 2)    ] = 0x00;
            bitmap[i][(j « 2) | 1] = 0x81;
            bitmap[i][(j « 2) | 2] = 0x81;
            bitmap[i][(j « 2) | 3] = 0x00;
        }
    }
    /* fill in top left corner pixel of grid */
    bitmap[31][7] = 0xfe;

    return (0);
}
```

Here is the call graph for this function:




Here is the caller graph for this function:




### 5.25.3.3 main()

int main (
    int argc,
    char ∗ argv[] )

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status 0.

Definition at line 96 of file unihex2bmp.c.

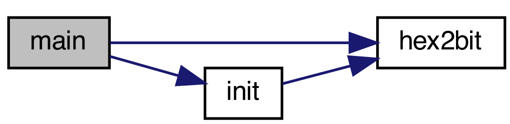
```
00097 {
00098
00099    int i, j;                    /* loop variables                       */
00100    unsigned k0;                 /* temp Unicode char variable           */
00101    unsigned swap;               /* temp variable for swapping values */
00102    char inbuf[256];             /* input buffer                         */
00103    unsigned filesize;           /* size of file in bytes                */
00104    unsigned bitmapsize;         /* size of bitmap image in bytes        */
00105    unsigned thischar;           /* the current character                */
00106    unsigned char thischarbyte; /* unsigned char lowest byte of Unicode char */
00107    int thischarrow;             /* row 0..15 where this character belongs  */
00108    int thiscol;                 /* column 0..15 where this character belongs */
00109    int toppixelrow;             /* pixel row, 0..16*32-1                */
00110    unsigned lastpage=0;         /* the last Unicode page read in font file */
00111    int wbmp=0;                  /* set to 1 if writing .wbmp format file */
00112
00113    unsigned char bitmap[17*32][18*4]; /* final bitmap */
00114    unsigned char charbits[32][4];  /* bitmap for one character, 4 bytes/row */
00115
00116    char *infile="", *outfile="";  /* names of input and output files */
00117    FILE *infp, *outfp;       /* file pointers of input and output files */
00118
00119    int init();                  /* initializes bitmap row/col labeling, &c. */
00120    int hex2bit();               /* convert hex string --> bitmap */
00121
00122    bitmapsize = 17*32*18*4;  /* 17 rows by 18 cols, each 4 bytes */
00123
00124    if (argc > 1) {
00125       for (i = 1; i < argc; i++) {
00126          if (argv[i][0] == '-') {  /* this is an option argument */
00127             switch (argv[i][1]) {
00128                case 'f':  /* flip (transpose) glyphs in bitmap as in standard */
00129                   flip = !flip;
00130                   break;
00131                case 'i':  /* name of input file */
00132                   infile = &argv[i][2];
00133                   break;
00134                case 'o':  /* name of output file */
00135                   outfile = &argv[i][2];
00136                   break;
00137                case 'p':  /* specify a Unicode page other than default of 0 */
00138                   sscanf (&argv[i][2], "%x", &unipage); /* Get Unicode page */
00139                   break;
```

```
            case 'w':  /* write a .wbmp file instead of a .bmp file */
               wbmp = 1;
               break;
            default:   /* if unrecognized option, print list and exit */
               fprintf (stderr, "\nSyntax:\n\n");
               fprintf (stderr, "   %s -p<Unicode_Page> ", argv[0]);
               fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
               fprintf (stderr, "   -w specifies .wbmp output instead of ");
               fprintf (stderr, "default Windows .bmp output.\n\n");
               fprintf (stderr, "   -p is followed by 1 to 6 ");
               fprintf (stderr, "Unicode page hex digits ");
               fprintf (stderr, "(default is Page 0).\n\n");
               fprintf (stderr, "\nExample:\n\n");
               fprintf (stderr, "   %s -p83 -iunifont.hex -ou83.bmp\n\n\n",
                      argv[0]);
               exit (1);
         }
      }
   }
}
/*
   Make sure we can open any I/O files that were specified before
   doing anything else.
*/
if (strlen (infile) > 0) {
   if ((infp = fopen (infile, "r")) == NULL) {
      fprintf (stderr, "Error: can't open %s for input.\n", infile);
      exit (1);
   }
}
else {
   infp = stdin;
}
if (strlen (outfile) > 0) {
   if ((outfp = fopen (outfile, "w")) == NULL) {
      fprintf (stderr, "Error: can't open %s for output.\n", outfile);
      exit (1);
   }
}
else {
   outfp = stdout;
}

(void)init(bitmap); /* initialize bitmap with row/column headers, etc. */

/*
   Read in the characters in the page
*/
while (lastpage <= unipage && fgets (inbuf, MAXBUF-1, infp) != NULL) {
   sscanf (inbuf, "%x", &thischar);
   lastpage = thischar » 8; /* keep Unicode page to see if we can stop */
   if (lastpage == unipage) {
      thischarbyte = (unsigned char)(thischar & 0xff);
      for (k0=0; inbuf[k0] != ':'; k0++);
      k0++;
      hex2bit (&inbuf[k0], charbits);  /* convert hex string to 32*4 bitmap */

      /*
         Now write character bitmap upside-down in page array, to match
         .bmp file order.  In the .wbmp' and .bmp files, white is a '1'
         bit and black is a '0' bit, so complement charbits[][].
      */

      thiscol = (thischarbyte & 0xf) + 2;  /* column number will be 1..16  */
      thischarrow = thischarbyte » 4;      /* charcter row number, 0..15   */
      if (flip) {  /* swap row and column placement */
         swap = thiscol;
         thiscol = thischarrow;
         thischarrow = swap;
         thiscol += 2;        /* column index starts at 1 */
         thischarrow -= 2;    /* row index starts at 0     */
      }
      toppixelrow = 32 * (thischarrow + 1) - 1; /* from bottom to top     */

      /*
         Copy the center of charbits[][] because hex characters only
         occupy rows 8 to 23 and column byte 2 (and for 16 bit wide
         characters, byte 3).  The charbits[][] array was given 32 rows
         and 4 column bytes for completeness in the beginning.
      */
      for (i=8; i<24; i++) {
```

```
               bitmap[toppixelrow + i][(thiscol « 2) | 0] =
                  ~charbits[i][0] & 0xff;
               bitmap[toppixelrow + i][(thiscol « 2) | 1] =
                  ~charbits[i][1] & 0xff;
               bitmap[toppixelrow + i][(thiscol « 2) | 2] =
                  ~charbits[i][2] & 0xff;
               /* Only use first 31 bits; leave vertical rule in 32nd column */
               bitmap[toppixelrow + i][(thiscol « 2) | 3] =
                  ~charbits[i][3] & 0xfe;
            }
            /*
               Leave white space in 32nd column of rows 8, 14, 15, and 23
               to leave 16 pixel height upper, middle, and lower guides.
            */
            bitmap[toppixelrow +  8][(thiscol « 2) | 3] |= 1;
            bitmap[toppixelrow + 14][(thiscol « 2) | 3] |= 1;
            bitmap[toppixelrow + 15][(thiscol « 2) | 3] |= 1;
            bitmap[toppixelrow + 23][(thiscol « 2) | 3] |= 1;
         }
      }
      /*
         Now write the appropriate bitmap file format, either
         Wireless Bitmap or Microsoft Windows bitmap.
      */
      if (wbmp) {  /* Write a Wireless Bitmap .wbmp format file */
         /*
            Write WBMP header
         */
         fprintf (outfp, "%c", 0x00); /* Type of image; always 0 (monochrome) */
         fprintf (outfp, "%c", 0x00); /* Reserved; always 0                   */
         fprintf (outfp, "%c%c", 0x84, 0x40); /* Width  = 576 pixels           */
         fprintf (outfp, "%c%c", 0x84, 0x20); /* Height = 544 pixels           */
         /*
            Write bitmap image
         */
         for (toppixelrow=0; toppixelrow <= 17*32-1; toppixelrow++) {
            for (j=0; j<18; j++) {
               fprintf (outfp, "%c", bitmap[toppixelrow][(j«2)    ]);
               fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 1]);
               fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 2]);
               fprintf (outfp, "%c", bitmap[toppixelrow][(j«2) | 3]);
            }
         }
      }
      else {  /* otherwise, write a Microsoft Windows .bmp format file */
         /*
            Write the .bmp file -- start with the header, then write the bitmap
         */

         /* 'B', 'M' appears at start of every .bmp file */
         fprintf (outfp, "%c%c", 0x42, 0x4d);

         /* Write file size in bytes */
         filesize   = 0x3E + bitmapsize;
         fprintf (outfp, "%c", (unsigned char)((filesize        ) & 0xff));
         fprintf (outfp, "%c", (unsigned char)((filesize » 0x08) & 0xff));
         fprintf (outfp, "%c", (unsigned char)((filesize » 0x10) & 0xff));
         fprintf (outfp, "%c", (unsigned char)((filesize » 0x18) & 0xff));

         /* Reserved - 0's */
         fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);

         /* Offset from start of file to bitmap data */
         fprintf (outfp, "%c%c%c%c", 0x3E, 0x00, 0x00, 0x00);

         /* Length of bitmap info header */
         fprintf (outfp, "%c%c%c%c", 0x28, 0x00, 0x00, 0x00);

         /* Width of bitmap in pixels */
         fprintf (outfp, "%c%c%c%c", 0x40, 0x02, 0x00, 0x00);

         /* Height of bitmap in pixels */
         fprintf (outfp, "%c%c%c%c", 0x20, 0x02, 0x00, 0x00);

         /* Planes in bitmap (fixed at 1) */
         fprintf (outfp, "%c%c", 0x01, 0x00);

         /* bits per pixel (1 = monochrome) */
         fprintf (outfp, "%c%c", 0x01, 0x00);

         /* Compression (0 = none) */
```

```
    fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);

    /* Size of bitmap data in bytes */
    fprintf (outfp, "%c", (unsigned char)((bitmapsize      ) & 0xff));
    fprintf (outfp, "%c", (unsigned char)((bitmapsize >> 0x08) & 0xff));
    fprintf (outfp, "%c", (unsigned char)((bitmapsize >> 0x10) & 0xff));
    fprintf (outfp, "%c", (unsigned char)((bitmapsize >> 0x18) & 0xff));

    /* Horizontal resolution in pixels per meter */
    fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);

    /* Vertical resolution in pixels per meter */
    fprintf (outfp, "%c%c%c%c", 0xC4, 0x0E, 0x00, 0x00);

    /* Number of colors used */
    fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);

    /* Number of important colors */
    fprintf (outfp, "%c%c%c%c", 0x02, 0x00, 0x00, 0x00);

    /* The color black: B=0x00, G=0x00, R=0x00, Filler=0xFF */
    fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);

    /* The color white: B=0xFF, G=0xFF, R=0xFF, Filler=0xFF */
    fprintf (outfp, "%c%c%c%c", 0xFF, 0xFF, 0xFF, 0x00);

    /*
       Now write the raw data bits.  Data is written from the lower
       left-hand corner of the image to the upper right-hand corner
       of the image.
    */
    for (toppixelrow=17*32-1; toppixelrow >= 0; toppixelrow--) {
       for (j=0; j<18; j++) {
          fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2)    ]);
          fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 1]);
          fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 2]);

          fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 3]);
       }
    }
  }
  exit (0);
}
```

Here is the call graph for this function:




## 5.25.4 Variable Documentation

### 5.25.4.1 flip

int flip =1

Transpose entire matrix as in Unicode book.

Definition at line 85 of file unihex2bmp.c.

#### 5.25.4.2 hex

char∗ hex[18]

Initial value:

```
= {
    "0030:00000000182442424242424224180000",
    "0031:000000000818280808080808083E0000",
    "0032:000000003C4242020C102040407E0000",
    "0033:000000003C4242021C020242423C0000",
    "0034:00000000040C142444447E0404040000",
    "0035:000000007E4040407C020202423C0000",
    "0036:000000001C2040407C424242423C0000",
    "0037:000000007E0202040404080808080000",
    "0038:000000003C4242423C424242423C0000",
    "0039:000000003C4242423E02020204380000",
    "0041:0000000018242442427E424242420000",
    "0042:000000007C4242427C424242427C0000",
    "0043:000000003C42424040404042423C0000",
    "0044:00000000784442424242424244780000",
    "0045:000000007E4040407C404040407E0000",
    "0046:000000007E4040407C40404040400000",
    "0055:000000004242424242424242423C0000",
    "002B:0000000000000808087F080808000000"
  }
```

GNU Unifont bitmaps for hexadecimal digits.

These are the GNU Unifont hex strings for '0'-'9' and 'A'-'F', for encoding as bit strings in row and column headers.

Looking at the final bitmap as a grid of 32∗32 bit tiles, the first row contains a hexadecimal character string of the first 3 hex digits in a 4 digit Unicode character name; the top column contains a hex character string of the 4th (low-order) hex digit of the Unicode character.

Definition at line 62 of file unihex2bmp.c.

#### 5.25.4.3 hexbits

unsigned char hexbits[18][32]

The digits converted into bitmaps.

Definition at line 82 of file unihex2bmp.c.

#### 5.25.4.4 unipage

unsigned unipage =0

Unicode page number, 0x00..0xff.

Definition at line 84 of file unihex2bmp.c.

## 5.26 unihex2bmp.c

Go to the documentation of this file.

```
/**
  @file unihex2bmp.c

  @brief unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points
                      into a bitmap for editing

  @author Paul Hardy, unifoundry <at> unifoundry.com, December 2007

  @copyright Copyright (C) 2007, 2008, 2013, 2017 Paul Hardy

  This program reads in a GNU Unifont .hex file, extracts a range of
  256 code points, and converts it a Microsoft Bitmap (.bmp) or Wireless
  Bitmap file.

  Synopsis: unihex2bmp [-iin_file.hex] [-oout_file.bmp]
             [-f] [-phex_page_num] [-w]
*/
/*
  LICENSE:

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

/*
  20 June 2017 [Paul Hardy]:
  - Adds capability to output triple-width and quadruple-width (31 pixels
    wide, not 32) glyphs.  The 32nd column in a glyph cell is occupied by
    the vertical cell border, so a quadruple-width glyph can only occupy
    the first 31 columns; the 32nd column is ignored.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXBUF 256


/**
  @brief GNU Unifont bitmaps for hexadecimal digits.

  These are the GNU Unifont hex strings for '0'-'9' and 'A'-'F',
  for encoding as bit strings in row and column headers.

  Looking at the final bitmap as a grid of 32*32 bit tiles, the
  first row contains a hexadecimal character string of the first
  3 hex digits in a 4 digit Unicode character name; the top column
  contains a hex character string of the 4th (low-order) hex digit
  of the Unicode character.
*/
```

```
char *hex[18]= {
        "0030:00000000182442424242424224180000",  /* Hex digit 0 */
        "0031:000000000818280808080808083E0000",  /* Hex digit 1 */
        "0032:000000003C4242020C102040407E0000",  /* Hex digit 2 */
        "0033:000000003C4242021C020242423C0000",  /* Hex digit 3 */
        "0034:00000000040C142444447E0404040000",  /* Hex digit 4 */
        "0035:000000007E4040407C020202423C0000",  /* Hex digit 5 */
        "0036:000000001C2040407C424242423C0000",  /* Hex digit 6 */
        "0037:000000007E0202040404080808080000",  /* Hex digit 7 */
        "0038:000000003C4242423C424242423C0000",  /* Hex digit 8 */
        "0039:000000003C4242423E02020204380000",  /* Hex digit 9 */
        "0041:0000000018242442427E424242420000",  /* Hex digit A */
        "0042:000000007C4242427C424242427C0000",  /* Hex digit B */
        "0043:000000003C42424040404042423C0000",  /* Hex digit C */
        "0044:00000000784442424242424244780000",  /* Hex digit D */
        "0045:000000007E4040407C404040407E0000",  /* Hex digit E */
        "0046:000000007E4040407C40404040400000",  /* Hex digit F */
        "0055:000000004242424242424242423C0000",  /* Unicode 'U' */
        "002B:0000000000000808087F080808000000"   /* Unicode '+' */
    };
unsigned char hexbits[18][32]; ///< The digits converted into bitmaps.

unsigned unipage=0;   ///< Unicode page number, 0x00..0xff.
int flip=1;           ///< Transpose entire matrix as in Unicode book.


/**
   @brief The main function.

   @param[in] argc The count of command line arguments.
   @param[in] argv Pointer to array of command line arguments.
   @return This program exits with status 0.
*/
int
main (int argc, char *argv[])
{

   int i, j;                  /* loop variables                    */
   unsigned k0;               /* temp Unicode char variable        */
   unsigned swap;             /* temp variable for swapping values */
   char inbuf[256];           /* input buffer                      */
   unsigned filesize;         /* size of file in bytes             */
   unsigned bitmapsize;       /* size of bitmap image in bytes     */
   unsigned thischar;         /* the current character             */
   unsigned char thischarbyte; /* unsigned char lowest byte of Unicode char */
   int thischarrow;           /* row 0..15 where this character belongs  */
   int thiscol;               /* column 0..15 where this character belongs */
   int toppixelrow;           /* pixel row, 0..16*32-1             */
   unsigned lastpage=0;       /* the last Unicode page read in font file */
   int wbmp=0;                /* set to 1 if writing .wbmp format file */

   unsigned char bitmap[17*32][18*4]; /* final bitmap */
   unsigned char charbits[32][4];  /* bitmap for one character, 4 bytes/row */

   char *infile="", *outfile="";  /* names of input and output files */
   FILE *infp, *outfp;       /* file pointers of input and output files */

   int init();                /* initializes bitmap row/col labeling, &c. */
   int hex2bit();             /* convert hex string --> bitmap */

   bitmapsize = 17*32*18*4;  /* 17 rows by 18 cols, each 4 bytes */

   if (argc > 1) {
      for (i = 1; i < argc; i++) {
         if (argv[i][0] == '-') {  /* this is an option argument */
            switch (argv[i][1]) {
               case 'f':  /* flip (transpose) glyphs in bitmap as in standard */
                  flip = !flip;
                  break;
               case 'i':  /* name of input file */
                  infile = &argv[i][2];
                  break;
               case 'o':  /* name of output file */
                  outfile = &argv[i][2];
                  break;
               case 'p':  /* specify a Unicode page other than default of 0 */
                  sscanf (&argv[i][2], "%x", &unipage); /* Get Unicode page */
                  break;
               case 'w':  /* write a .wbmp file instead of a .bmp file */
                  wbmp = 1;
                  break;
```

```
            default:   /* if unrecognized option, print list and exit */
              fprintf (stderr, ”\nSyntax:\n\n”);
              fprintf (stderr, ”   %s -p<Unicode_Page> ”, argv[0]);
              fprintf (stderr, ”-i<Input_File> -o<Output_File> -w\n\n”);
              fprintf (stderr, ”   -w specifies .wbmp output instead of ”);
              fprintf (stderr, ”default Windows .bmp output.\n\n”);
              fprintf (stderr, ”   -p is followed by 1 to 6 ”);
              fprintf (stderr, ”Unicode page hex digits ”);
              fprintf (stderr, ”(default is Page 0).\n\n”);
              fprintf (stderr, ”\nExample:\n\n”);
              fprintf (stderr, ”   %s -p83 -iunifont.hex -ou83.bmp\n\n\n”,
                   argv[0]);
              exit (1);
          }
        }
      }
    }
    /*
      Make sure we can open any I/O files that were specified before
      doing anything else.
    */
    if (strlen (infile) > 0) {
      if ((infp = fopen (infile, ”r”)) == NULL) {
        fprintf (stderr, ”Error: can’t open %s for input.\n”, infile);
        exit (1);
      }
    }
    else {
      infp = stdin;
    }
    if (strlen (outfile) > 0) {
      if ((outfp = fopen (outfile, ”w”)) == NULL) {
        fprintf (stderr, ”Error: can’t open %s for output.\n”, outfile);
        exit (1);
      }
    }
    else {
      outfp = stdout;
    }

    (void)init(bitmap); /* initialize bitmap with row/column headers, etc. */

    /*
      Read in the characters in the page
    */
    while (lastpage <= unipage && fgets (inbuf, MAXBUF-1, infp) != NULL) {
      sscanf (inbuf, ”%x”, &thischar);
      lastpage = thischar » 8; /* keep Unicode page to see if we can stop */
      if (lastpage == unipage) {
        thischarbyte = (unsigned char)(thischar & 0xff);
        for (k0=0; inbuf[k0] != ’:’; k0++);
        k0++;
        hex2bit (&inbuf[k0], charbits);  /* convert hex string to 32*4 bitmap */

        /*
          Now write character bitmap upside-down in page array, to match
          .bmp file order.  In the .wbmp‘ and .bmp files, white is a ’1’
          bit and black is a ’0’ bit, so complement charbits[][].
        */

        thiscol = (thischarbyte & 0xf) + 2;  /* column number will be 1..16  */
        thischarrow = thischarbyte » 4;      /* charcter row number, 0..15   */
        if (flip) {  /* swap row and column placement */
          swap = thiscol;
          thiscol = thischarrow;
          thischarrow = swap;
          thiscol += 2;        /* column index starts at 1 */
          thischarrow -= 2;   /* row index starts at 0    */
        }
        toppixelrow = 32 * (thischarrow + 1) - 1; /* from bottom to top    */

        /*
          Copy the center of charbits[][] because hex characters only
          occupy rows 8 to 23 and column byte 2 (and for 16 bit wide
          characters, byte 3).  The charbits[][] array was given 32 rows
          and 4 column bytes for completeness in the beginning.
        */
        for (i=8; i<24; i++) {
          bitmap[toppixelrow + i][(thiscol « 2) | 0] =
            ~charbits[i][0] & 0xff;
          bitmap[toppixelrow + i][(thiscol « 2) | 1] =
```

```
                ~charbits[i][1] & 0xff;
            bitmap[toppixelrow + i][(thiscol << 2) | 2] =
                ~charbits[i][2] & 0xff;
            /* Only use first 31 bits; leave vertical rule in 32nd column */
            bitmap[toppixelrow + i][(thiscol << 2) | 3] =
                ~charbits[i][3] & 0xfe;
        }
        /*
            Leave white space in 32nd column of rows 8, 14, 15, and 23
            to leave 16 pixel height upper, middle, and lower guides.
        */
        bitmap[toppixelrow +  8][(thiscol << 2) | 3] |= 1;
        bitmap[toppixelrow + 14][(thiscol << 2) | 3] |= 1;
        bitmap[toppixelrow + 15][(thiscol << 2) | 3] |= 1;
        bitmap[toppixelrow + 23][(thiscol << 2) | 3] |= 1;
    }
}
/*
    Now write the appropriate bitmap file format, either
    Wireless Bitmap or Microsoft Windows bitmap.
*/
if (wbmp) {  /* Write a Wireless Bitmap .wbmp format file */
    /*
        Write WBMP header
    */
    fprintf (outfp, "%c", 0x00); /* Type of image; always 0 (monochrome) */
    fprintf (outfp, "%c", 0x00); /* Reserved; always 0                    */
    fprintf (outfp, "%c%c", 0x84, 0x40); /* Width  = 576 pixels           */
    fprintf (outfp, "%c%c", 0x84, 0x20); /* Height = 544 pixels           */
    /*
        Write bitmap image
    */
    for (toppixelrow=0; toppixelrow <= 17*32-1; toppixelrow++) {
        for (j=0; j<18; j++) {
            fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2)    ]);
            fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 1]);
            fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 2]);
            fprintf (outfp, "%c", bitmap[toppixelrow][(j<<2) | 3]);
        }
    }
}
else {  /* otherwise, write a Microsoft Windows .bmp format file */
    /*
        Write the .bmp file -- start with the header, then write the bitmap
    */

    /* 'B', 'M' appears at start of every .bmp file */
    fprintf (outfp, "%c%c", 0x42, 0x4d);

    /* Write file size in bytes */
    filesize   = 0x3E + bitmapsize;
    fprintf (outfp, "%c", (unsigned char)((filesize        ) & 0xff));
    fprintf (outfp, "%c", (unsigned char)((filesize >> 0x08) & 0xff));
    fprintf (outfp, "%c", (unsigned char)((filesize >> 0x10) & 0xff));
    fprintf (outfp, "%c", (unsigned char)((filesize >> 0x18) & 0xff));

    /* Reserved - 0's */
    fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);

    /* Offset from start of file to bitmap data */
    fprintf (outfp, "%c%c%c%c", 0x3E, 0x00, 0x00, 0x00);

    /* Length of bitmap info header */
    fprintf (outfp, "%c%c%c%c", 0x28, 0x00, 0x00, 0x00);

    /* Width of bitmap in pixels */
    fprintf (outfp, "%c%c%c%c", 0x40, 0x02, 0x00, 0x00);

    /* Height of bitmap in pixels */
    fprintf (outfp, "%c%c%c%c", 0x20, 0x02, 0x00, 0x00);

    /* Planes in bitmap (fixed at 1) */
    fprintf (outfp, "%c%c", 0x01, 0x00);

    /* bits per pixel (1 = monochrome) */
    fprintf (outfp, "%c%c", 0x01, 0x00);

    /* Compression (0 = none) */
    fprintf (outfp, "%c%c%c%c", 0x00, 0x00, 0x00, 0x00);

    /* Size of bitmap data in bytes */
```

```
      fprintf (outfp, ”%c”, (unsigned char)((bitmapsize        ) & 0xff));
      fprintf (outfp, ”%c”, (unsigned char)((bitmapsize » 0x08) & 0xff));
      fprintf (outfp, ”%c”, (unsigned char)((bitmapsize » 0x10) & 0xff));
      fprintf (outfp, ”%c”, (unsigned char)((bitmapsize » 0x18) & 0xff));

      /* Horizontal resolution in pixels per meter */
      fprintf (outfp, ”%c%c%c%c”, 0xC4, 0x0E, 0x00, 0x00);

      /* Vertical resolution in pixels per meter */
      fprintf (outfp, ”%c%c%c%c”, 0xC4, 0x0E, 0x00, 0x00);

      /* Number of colors used */
      fprintf (outfp, ”%c%c%c%c”, 0x02, 0x00, 0x00, 0x00);

      /* Number of important colors */
      fprintf (outfp, ”%c%c%c%c”, 0x02, 0x00, 0x00, 0x00);

      /* The color black: B=0x00, G=0x00, R=0x00, Filler=0xFF */
      fprintf (outfp, ”%c%c%c%c”, 0x00, 0x00, 0x00, 0x00);

      /* The color white: B=0xFF, G=0xFF, R=0xFF, Filler=0xFF */
      fprintf (outfp, ”%c%c%c%c”, 0xFF, 0xFF, 0xFF, 0x00);

      /*
         Now write the raw data bits.  Data is written from the lower
         left-hand corner of the image to the upper right-hand corner
         of the image.
      */
      for (toppixelrow=17*32-1; toppixelrow >= 0; toppixelrow--) {
         for (j=0; j<18; j++) {
            fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2)    ]);
            fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2) | 1]);
            fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2) | 2]);

            fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2) | 3]);
         }
      }
   }
   exit (0);
}


/**
   @brief Generate a bitmap for one glyph.

   Convert the portion of a hex string after the ’:’ into a character bitmap.

   If string is >= 128 characters, it will fill all 4 bytes per row.
   If string is >= 64 characters and < 128, it will fill 2 bytes per row.
   Otherwise, it will fill 1 byte per row.

   @param[in] instring The character array containing the glyph bitmap.
   @param[out] character Glyph bitmap, 8, 16, or 32 columns by 16 rows tall.
   @return Always returns 0.
*/
int
hex2bit (char *instring, unsigned char character[32][4])
{

   int i;  /* current row in bitmap character */
   int j;  /* current character in input string */
   int k;  /* current byte in bitmap character */
   int width;  /* number of output bytes to fill - 1: 0, 1, 2, or 3 */

   for (i=0; i<32; i++)  /* erase previous character */
      character[i][0] = character[i][1] = character[i][2] = character[i][3] = 0;
   j=0;  /* current location is at beginning of instring */

   if (strlen (instring) <= 34)  /* 32 + possible ’\r’, ’\n’ */
      width = 0;
   else if (strlen (instring) <= 66)  /* 64 + possible ’\r’, ’\n’ */
      width = 1;
   else if (strlen (instring) <= 98)  /* 96 + possible ’\r’, ’\n’ */
      width = 3;
   else  /* the maximum allowed is quadruple-width */
      width = 4;

   k = (width > 1) ? 0 : 1; /* if width > double, start at index 1 else at 0 */

   for (i=8; i<24; i++) {  /* 16 rows per input character, rows 8..23 */
      sscanf (&instring[j], ”%2hhx”, &character[i][k]);
```

```
      j += 2;
      if (width > 0) { /* add next pair of hex digits to this row */
         sscanf (&instring[j], ”%2hhx”, &character[i][k+1]);
         j += 2;
         if (width > 1) { /* add next pair of hex digits to this row */
            sscanf (&instring[j], ”%2hhx”, &character[i][k+2]);
            j += 2;
            if (width > 2) {  /* quadruple-width is maximum width */
               sscanf (&instring[j], ”%2hhx”, &character[i][k+3]);
               j += 2;
            }
         }
      }
   }

   return (0);
}


/**
   @brief Initialize the bitmap grid.

   @param[out] bitmap The bitmap to generate, with 32x32 pixel glyph areas.
   @return Always returns 0.
*/
int
init (unsigned char bitmap[17*32][18*4])
{
   int i, j;
   unsigned char charbits[32][4];  /* bitmap for one character, 4 bytes/row */
   unsigned toppixelrow;
   unsigned thiscol;
   unsigned char pnybble0, pnybble1, pnybble2, pnybble3;

   for (i=0; i<18; i++) { /* bitmaps for '0'..'9', 'A'-'F', 'u', '+' */

      hex2bit (&hex[i][5], charbits);  /* convert hex string to 32*4 bitmap */

      for (j=0; j<32; j++) hexbits[i][j] = ~charbits[j][1];
   }

   /*
      Initialize bitmap to all white.
   */
   for (toppixelrow=0; toppixelrow < 17*32; toppixelrow++) {
      for (thiscol=0; thiscol<18; thiscol++) {
         bitmap[toppixelrow][(thiscol « 2)    ] = 0xff;
         bitmap[toppixelrow][(thiscol « 2) | 1] = 0xff;
         bitmap[toppixelrow][(thiscol « 2) | 2] = 0xff;
         bitmap[toppixelrow][(thiscol « 2) | 3] = 0xff;
      }
   }
   /*
      Write the ”u+nnnn” table header in the upper left-hand corner,
      where nnnn is the upper 16 bits of a 32-bit Unicode assignment.
   */
   pnybble3 = (unipage » 20);
   pnybble2 = (unipage » 16) & 0xf;
   pnybble1 = (unipage » 12) & 0xf;
   pnybble0 = (unipage »  8) & 0xf;
   for (i=0; i<32; i++) {
      bitmap[i][1] = hexbits[16][i];  /* copy 'u' */
      bitmap[i][2] = hexbits[17][i];  /* copy '+' */
      bitmap[i][3] = hexbits[pnybble3][i];
      bitmap[i][4] = hexbits[pnybble2][i];
      bitmap[i][5] = hexbits[pnybble1][i];
      bitmap[i][6] = hexbits[pnybble0][i];
   }
   /*
      Write low-order 2 bytes of Unicode number assignments, as hex labels
   */
   pnybble3 = (unipage » 4) & 0xf;  /* Highest-order hex digit */
   pnybble2 = (unipage     ) & 0xf;  /* Next highest-order hex digit */
   /*
      Write the column headers in bitmap[][] (row headers if flipped)
   */
   toppixelrow = 32 * 17 - 1; /* maximum pixel row number */
   /*
      Label the column headers. The hexbits[][] bytes are split across two
      bitmap[][] entries to center a the hex digits in a column of 4 bytes.
      OR highest byte with 0xf0 and lowest byte with 0x0f to make outer
```

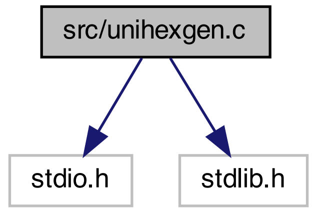
```
      nybbles white (0=black, 1-white).
   */
   for (i=0; i<16; i++) {
      for (j=0; j<32; j++) {
         if (flip) {  /* transpose matrix */
            bitmap[j][((i+2) « 2) | 0]  = (hexbits[pnybble3][j] » 4) | 0xf0;
            bitmap[j][((i+2) « 2) | 1]  = (hexbits[pnybble3][j] « 4) |
                                          (hexbits[pnybble2][j] » 4);
            bitmap[j][((i+2) « 2) | 2]  = (hexbits[pnybble2][j] « 4) |
                                          (hexbits[i][j] » 4);
            bitmap[j][((i+2) « 2) | 3]  = (hexbits[i][j] « 4) | 0x0f;
         }
         else {
            bitmap[j][((i+2) « 2) | 1] = (hexbits[i][j] » 4) | 0xf0;
            bitmap[j][((i+2) « 2) | 2] = (hexbits[i][j] « 4) | 0x0f;
         }
      }
   }
   /*
      Now use the single hex digit column graphics to label the row headers.
   */
   for (i=0; i<16; i++) {
      toppixelrow = 32 * (i + 1) - 1; /* from bottom to top    */

      for (j=0; j<32; j++) {
         if (!flip) {  /* if not transposing matrix */
            bitmap[toppixelrow + j][4] = hexbits[pnybble3][j];
            bitmap[toppixelrow + j][5] = hexbits[pnybble2][j];
         }
         bitmap[toppixelrow + j][6] = hexbits[i][j];
      }
   }
   /*
      Now draw grid lines in bitmap, around characters we just copied.
   */
   /* draw vertical lines 2 pixels wide */
   for (i=1*32; i<17*32; i++) {
      if ((i & 0x1f) == 7)
         i++;
      else if ((i & 0x1f) == 14)
         i += 2;
      else if ((i & 0x1f) == 22)
         i++;
      for (j=1; j<18; j++) {
         bitmap[i][(j « 2) | 3] &= 0xfe;
      }
   }
   /* draw horizontal lines 1 pixel tall */
   for (i=1*32-1; i<18*32-1; i+=32) {
      for (j=2; j<18; j++) {
         bitmap[i][(j « 2)    ] = 0x00;
         bitmap[i][(j « 2) | 1] = 0x81;
         bitmap[i][(j « 2) | 2] = 0x81;
         bitmap[i][(j « 2) | 3] = 0x00;
      }
   }
   /* fill in top left corner pixel of grid */
   bitmap[31][7] = 0xfe;

   return (0);
}
```

## 5.27 src/unihexgen.c File Reference

unihexgen - Generate a series of glyphs containing hexadecimal code points

#include <stdio.h>
#include <stdlib.h>

Include dependency graph for unihexgen.c:




## Functions

- int main (int argc, char ∗argv[])

  The main function.
- void hexprint4 (int thiscp)

  Generate a bitmap containing a 4-digit Unicode code point.
- void hexprint6 (int thiscp)

  Generate a bitmap containing a 6-digit Unicode code point.

## Variables

- char hexdigit [16][5]

  Bitmap pattern for each hexadecimal digit.

### 5.27.1 Detailed Description

unihexgen - Generate a series of glyphs containing hexadecimal code points

Author

Paul Hardy

Copyright

Copyright (C) 2013 Paul Hardy

This program generates glyphs in Unifont .hex format that contain four- or six-digit hexadecimal numbers in a 16x16 pixel area. These are rendered as white digits on a black background.

argv[1] is the starting code point (as a hexadecimal string, with no leading "0x".

argv[2] is the ending code point (as a hexadecimal string, with no leading "0x".

For example:

```
unihexgen e000 f8ff > pua.hex
```

This generates the Private Use Area glyph file.

This utility program works in Roman Czyborra's unifont.hex file format, the basis of the GNU Unifont package.

Definition in file unihexgen.c.

## 5.27.2 Function Documentation

### 5.27.2.1 hexprint4()

```
void hexprint4 (
        int thiscp )
```

Generate a bitmap containing a 4-digit Unicode code point.

Takes a 4-digit Unicode code point as an argument and prints a unifont.hex string for it to stdout.

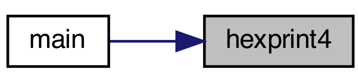
Parameters

| | | |
|---|---|---|
| in | thiscp | The current code point for which to generate a glyph. |

Definition at line 160 of file unihexgen.c.

```
{

   int grid[16]; /* the glyph grid we'll build */

   int row;       /* row number in current glyph */
   int digitrow; /* row number in current hex digit being rendered */
   int rowbits;  /* 1 & 0 bits to draw current glyph row */

   int d1, d2, d3, d4; /* four hexadecimal digits of each code point */

   d1 = (thiscp >> 12) & 0xF;
   d2 = (thiscp >>  8) & 0xF;
   d3 = (thiscp >>  4) & 0xF;
   d4 = (thiscp      ) & 0xF;

   /* top and bottom rows are white */
   grid[0] = grid[15] = 0x0000;

   /* 14 inner rows are 14-pixel wide black lines, centered */
   for (row = 1; row < 15; row++) grid[row] = 0x7FFE;

   printf ("%04X:", thiscp);

   /*
      Render the first row of 2 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 2; row < 7; row++) {
      rowbits = (hexdigit[d1][digitrow] << 9) |
                (hexdigit[d2][digitrow] << 3);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   /*
      Render the second row of 2 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 9; row < 14; row++) {
      rowbits = (hexdigit[d3][digitrow] << 9) |
                (hexdigit[d4][digitrow] << 3);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);

   putchar ('\n');

   return;
}
```

Here is the caller graph for this function:




#### 5.27.2.2 hexprint6()

void hexprint6 (
    int thiscp )

Generate a bitmap containing a 6-digit Unicode code point.

Takes a 6-digit Unicode code point as an argument and prints a unifont.hex string for it to stdout.

Parameters

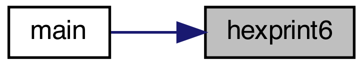
| | | |
|---|---|---|
| in | thiscp | The current code point for which to generate a glyph. |

Definition at line 223 of file unihexgen.c.

```
{

   int grid[16]; /* the glyph grid we'll build */

   int row;       /* row number in current glyph */
   int digitrow; /* row number in current hex digit being rendered */
   int rowbits;  /* 1 & 0 bits to draw current glyph row */

   int d1, d2, d3, d4, d5, d6; /* six hexadecimal digits of each code point */

   d1 = (thiscp » 20) & 0xF;
   d2 = (thiscp » 16) & 0xF;
   d3 = (thiscp » 12) & 0xF;
   d4 = (thiscp »  8) & 0xF;
   d5 = (thiscp »  4) & 0xF;
   d6 = (thiscp      ) & 0xF;

   /* top and bottom rows are white */
   grid[0] = grid[15] = 0x0000;

   /* 14 inner rows are 16-pixel wide black lines, centered */
   for (row = 1; row < 15; row++) grid[row] = 0xFFFF;


   printf ("%06X:", thiscp);

   /*
      Render the first row of 3 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 2; row < 7; row++) {
      rowbits = (hexdigit[d1][digitrow] « 11) |
                (hexdigit[d2][digitrow] «  6) |
                (hexdigit[d3][digitrow] «  1);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   /*
      Render the second row of 3 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 9; row < 14; row++) {
      rowbits = (hexdigit[d4][digitrow] « 11) |
                (hexdigit[d5][digitrow] «  6) |
                (hexdigit[d6][digitrow] «  1);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);

```

```
    putchar ('\n');

    return;
}
```

Here is the caller graph for this function:




#### 5.27.2.3 main()

int main (
 int argc,
 char * argv[] )

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments (code point range). |

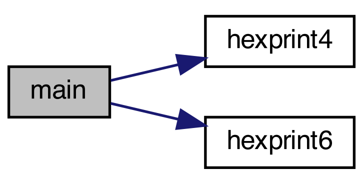
Returns

This program exits with status EXIT_SUCCESS.

Definition at line 112 of file unihexgen.c.

```
{

   int startcp, endcp, thiscp;
   void hexprint4(int); /* function to print one 4-digit unifont.hex code point */
   void hexprint6(int); /* function to print one 6-digit unifont.hex code point */

   if (argc != 3) {
      fprintf (stderr,"\n%s - generate unifont.hex code points as\n", argv[0]);
      fprintf (stderr,"four-digit hexadecimal numbers in a 2 by 2 grid,\n");
      fprintf (stderr,"or six-digit hexadecimal numbers in a 3 by 2 grid.\n");
      fprintf (stderr,"Syntax:\n\n");
      fprintf (stderr,"     %s first_code_point last_code_point > glyphs.hex\n\n", argv[0]);
      fprintf (stderr,"Example (to generate glyphs for the Private Use Area):\n\n");
      fprintf (stderr,"     %s e000 f8ff > pua.hex\n\n", argv[0]);
      exit (EXIT_FAILURE);
   }

   sscanf (argv[1], "%x", &startcp);
   sscanf (argv[2], "%x", &endcp);

   startcp &= 0xFFFFFF; /* limit to 6 hex digits */
   endcp   &= 0xFFFFFF; /* limit to 6 hex digits */

   /*
      For each code point in the desired range, generate a glyph.
   */
   for (thiscp = startcp; thiscp <= endcp; thiscp++) {
      if (thiscp <= 0xFFFF) {
         hexprint4 (thiscp); /* print digits 2/line, 2 lines */
      }
      else {
         hexprint6 (thiscp); /* print digits 3/line, 2 lines */
      }
   }
   exit (EXIT_SUCCESS);
}
```

Here is the call graph for this function:




## 5.27.3 Variable Documentation

#### 5.27.3.1 hexdigit

char hexdigit[16][5]

Initial value:
```
= {
   {0x6,0x9,0x9,0x9,0x6},
   {0x2,0x6,0x2,0x2,0x7},
   {0xF,0x1,0xF,0x8,0xF},
   {0xE,0x1,0x7,0x1,0xE},
   {0x9,0x9,0xF,0x1,0x1},
   {0xF,0x8,0xF,0x1,0xF},
   {0x6,0x8,0xE,0x9,0x6},
   {0xF,0x1,0x2,0x4,0x4},
   {0x6,0x9,0x6,0x9,0x6},
   {0x6,0x9,0x7,0x1,0x6},
   {0xF,0x9,0xF,0x9,0x9},
   {0xE,0x9,0xE,0x9,0xE},
   {0x7,0x8,0x8,0x8,0x7},
   {0xE,0x9,0x9,0x9,0xE},
   {0xF,0x8,0xE,0x8,0xF},
   {0xF,0x8,0xE,0x8,0x8}
}
```

Bitmap pattern for each hexadecimal digit.

hexdigit[][] definition: the bitmap pattern for each hexadecimal digit.

Each digit is drawn as a 4 wide by 5 high bitmap, so each digit row is one hexadecimal digit, and each entry has 5 rows.

For example, the entry for digit 1 is:

{0x2,0x6,0x2,0x2,0x7},

which corresponds graphically to:

–#- ==> 0010 ==> 0x2 -##- ==> 0110 ==> 0x6 –#- ==> 0010 ==> 0x2 –#- ==> 0010 ==> 0x2 -### ==> 0111 ==> 0x7

These row values will then be exclusive-ORed with four one bits (binary 1111, or 0xF) to form white digits on a black background.

Functions hexprint4 and hexprint6 share the hexdigit array; they print four-digit and six-digit hexadecimal code points in a single glyph, respectively.

Definition at line 84 of file unihexgen.c.

## 5.28 unihexgen.c

Go to the documentation of this file.

```
/**
  @file unihexgen.c

  @brief unihexgen - Generate a series of glyphs containing
                     hexadecimal code points

  @author Paul Hardy

  @copyright Copyright (C) 2013 Paul Hardy

  This program generates glyphs in Unifont .hex format that contain
  four- or six-digit hexadecimal numbers in a 16x16 pixel area.  These
  are rendered as white digits on a black background.

  argv[1] is the starting code point (as a hexadecimal
  string, with no leading ”0x”.

  argv[2] is the ending code point (as a hexadecimal
  string, with no leading ”0x”.

     For example:

        unihexgen e000 f8ff > pua.hex

     This generates the Private Use Area glyph file.

  This utility program works in Roman Czyborra’s unifont.hex file
  format, the basis of the GNU Unifont package.
*/
/*
  This program is released under the terms of the GNU General Public
  License version 2, or (at your option) a later version.

  LICENSE:

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#include <stdio.h>
#include <stdlib.h>


/**
  @brief Bitmap pattern for each hexadecimal digit.

  hexdigit[][] definition: the bitmap pattern for
  each hexadecimal digit.

  Each digit is drawn as a 4 wide by 5 high bitmap,
  so each digit row is one hexadecimal digit, and
  each entry has 5 rows.

  For example, the entry for digit 1 is:

    {0x2,0x6,0x2,0x2,0x7},

  which corresponds graphically to:

    --#-  ==>  0010  ==>  0x2
    -##-  ==>  0110  ==>  0x6
    --#-  ==>  0010  ==>  0x2
    --#-  ==>  0010  ==>  0x2
    -###  ==>  0111  ==>  0x7

  These row values will then be exclusive-ORed with four one bits
```

```
  (binary 1111, or 0xF) to form white digits on a black background.


  Functions hexprint4 and hexprint6 share the hexdigit array;
  they print four-digit and six-digit hexadecimal code points
  in a single glyph, respectively.
*/
char hexdigit[16][5] = {
  {0x6,0x9,0x9,0x9,0x6},  /* 0x0 */
  {0x2,0x6,0x2,0x2,0x7},  /* 0x1 */
  {0xF,0x1,0xF,0x8,0xF},  /* 0x2 */
  {0xE,0x1,0x7,0x1,0xE},  /* 0x3 */
  {0x9,0x9,0xF,0x1,0x1},  /* 0x4 */
  {0xF,0x8,0xF,0x1,0xF},  /* 0x5 */
  {0x6,0x8,0xE,0x9,0x6},  /* 0x6 */ // {0x8,0x8,0xF,0x9,0xF} [alternate square form of 6]
  {0xF,0x1,0x2,0x4,0x4},  /* 0x7 */
  {0x6,0x9,0x6,0x9,0x6},  /* 0x8 */
  {0x6,0x9,0x7,0x1,0x6},  /* 0x9 */ // {0xF,0x9,0xF,0x1,0x1} [alternate square form of 9]
  {0xF,0x9,0xF,0x9,0x9},  /* 0xA */
  {0xE,0x9,0xE,0x9,0xE},  /* 0xB */
  {0x7,0x8,0x8,0x8,0x7},  /* 0xC */
  {0xE,0x9,0x9,0x9,0xE},  /* 0xD */
  {0xF,0x8,0xE,0x8,0xF},  /* 0xE */
  {0xF,0x8,0xE,0x8,0x8}   /* 0xF */
};


/**
  @brief The main function.

  @param[in] argc The count of command line arguments.
  @param[in] argv Pointer to array of command line arguments (code point range).
  @return This program exits with status EXIT_SUCCESS.
*/
int
main (int argc, char *argv[])
{

  int startcp, endcp, thiscp;
  void hexprint4(int); /* function to print one 4-digit unifont.hex code point */
  void hexprint6(int); /* function to print one 6-digit unifont.hex code point */

  if (argc != 3) {
    fprintf (stderr,"\n%s - generate unifont.hex code points as\n", argv[0]);
    fprintf (stderr,"four-digit hexadecimal numbers in a 2 by 2 grid,\n");
    fprintf (stderr,"or six-digit hexadecimal numbers in a 3 by 2 grid.\n");
    fprintf (stderr,"Syntax:\n\n");
    fprintf (stderr,"     %s first_code_point last_code_point > glyphs.hex\n\n", argv[0]);
    fprintf (stderr,"Example (to generate glyphs for the Private Use Area):\n\n");
    fprintf (stderr,"     %s e000 f8ff > pua.hex\n\n", argv[0]);
    exit (EXIT_FAILURE);
  }

  sscanf (argv[1], "%x", &startcp);
  sscanf (argv[2], "%x", &endcp);

  startcp &= 0xFFFFFF; /* limit to 6 hex digits */
  endcp   &= 0xFFFFFF; /* limit to 6 hex digits */

  /*
    For each code point in the desired range, generate a glyph.
  */
  for (thiscp = startcp; thiscp <= endcp; thiscp++) {
    if (thiscp <= 0xFFFF) {
      hexprint4 (thiscp); /* print digits 2/line, 2 lines */
    }
    else {
      hexprint6 (thiscp); /* print digits 3/line, 2 lines */
    }
  }
  exit (EXIT_SUCCESS);
}


/**
  @brief Generate a bitmap containing a 4-digit Unicode code point.

  Takes a 4-digit Unicode code point as an argument
  and prints a unifont.hex string for it to stdout.

  @param[in] thiscp The current code point for which to generate a glyph.
```

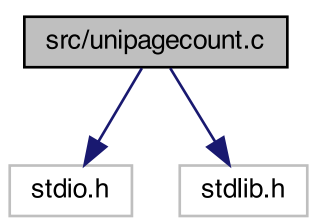
```
*/
void
hexprint4 (int thiscp)
{

   int grid[16]; /* the glyph grid we'll build */

   int row;      /* row number in current glyph */
   int digitrow; /* row number in current hex digit being rendered */
   int rowbits;  /* 1 & 0 bits to draw current glyph row */

   int d1, d2, d3, d4; /* four hexadecimal digits of each code point */

   d1 = (thiscp >> 12) & 0xF;
   d2 = (thiscp >>  8) & 0xF;
   d3 = (thiscp >>  4) & 0xF;
   d4 = (thiscp      ) & 0xF;

   /* top and bottom rows are white */
   grid[0] = grid[15] = 0x0000;

   /* 14 inner rows are 14-pixel wide black lines, centered */
   for (row = 1; row < 15; row++) grid[row] = 0x7FFE;

   printf ("%04X:", thiscp);

   /*
      Render the first row of 2 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 2; row < 7; row++) {
      rowbits = (hexdigit[d1][digitrow] << 9) |
                (hexdigit[d2][digitrow] << 3);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   /*
      Render the second row of 2 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 9; row < 14; row++) {
      rowbits = (hexdigit[d3][digitrow] << 9) |
                (hexdigit[d4][digitrow] << 3);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);

   putchar ('\n');

   return;
}


/**
   @brief Generate a bitmap containing a 6-digit Unicode code point.

   Takes a 6-digit Unicode code point as an argument
   and prints a unifont.hex string for it to stdout.

   @param[in] thiscp The current code point for which to generate a glyph.
*/
void
hexprint6 (int thiscp)
{

   int grid[16]; /* the glyph grid we'll build */

   int row;      /* row number in current glyph */
   int digitrow; /* row number in current hex digit being rendered */
   int rowbits;  /* 1 & 0 bits to draw current glyph row */

   int d1, d2, d3, d4, d5, d6; /* six hexadecimal digits of each code point */

   d1 = (thiscp >> 20) & 0xF;
   d2 = (thiscp >> 16) & 0xF;
   d3 = (thiscp >> 12) & 0xF;
   d4 = (thiscp >>  8) & 0xF;
   d5 = (thiscp >>  4) & 0xF;
```

```
   d6 = (thiscp      ) & 0xF;

   /* top and bottom rows are white */
   grid[0] = grid[15] = 0x0000;

   /* 14 inner rows are 16-pixel wide black lines, centered */
   for (row = 1; row < 15; row++) grid[row] = 0xFFFF;


   printf ("%06X:", thiscp);

   /*
      Render the first row of 3 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 2; row < 7; row++) {
      rowbits = (hexdigit[d1][digitrow] << 11) |
                (hexdigit[d2][digitrow] <<  6) |
                (hexdigit[d3][digitrow] <<  1);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   /*
      Render the second row of 3 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 9; row < 14; row++) {
      rowbits = (hexdigit[d4][digitrow] << 11) |
                (hexdigit[d5][digitrow] <<  6) |
                (hexdigit[d6][digitrow] <<  1);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);

   putchar ('\n');

   return;
}
```

## 5.29 src/unipagecount.c File Reference

unipagecount - Count the number of glyphs defined in each page of 256 code points

#include <stdio.h>
#include <stdlib.h>
Include dependency graph for unipagecount.c:



## Macros

- #define MAXBUF 256
  Maximum input line size - 1.

## Functions

- int main (int argc, char *argv[])
  The main function.
- void mkftable (unsigned plane, int pagecount[256], int links)
  Create an HTML table linked to PNG images.

### 5.29.1 Detailed Description

unipagecount - Count the number of glyphs defined in each page of 256 code points

Author

Paul Hardy, unifoundry <at> unifoundry.com, December 2007

Copyright

Copyright (C) 2007, 2008, 2013, 2014 Paul Hardy

This program counts the number of glyphs that are defined in each "page" of 256 code points, and prints the counts in an 8 x 8 grid. Input is from stdin. Output is to stdout.

The background color of each cell in a 16-by-16 grid of 256 code points is shaded to indicate percentage coverage. Red indicates 0% coverage, green represents 100% coverage, and colors in between pure red and pure green indicate partial coverage on a scale.

Each code point range number can be a hyperlink to a PNG file for that 256-code point range's corresponding bitmap glyph image.

Synopsis:

```
unipagecount < font_file.hex > count.txt
unipagecount -phex_page_num < font_file.hex  -- just 256 points
unipagecount -h < font_file.hex               -- HTML table
unipagecount -P1 -h < font.hex > count.html  -- Plane 1, HTML out
unipagecount -l < font_file.hex               -- linked HTML table
```

Definition in file unipagecount.c.

### 5.29.2 Macro Definition Documentation

#### 5.29.2.1 MAXBUF

#define MAXBUF 256

Maximum input line size - 1.

Definition at line 56 of file unipagecount.c.

### 5.29.3 Function Documentation

#### 5.29.3.1 main()

```
int main (
        int argc,
        char * argv[] )
```

The main function.

Parameters

| | | |
|---|---|---|
| in | argc | The count of command line arguments. |
| in | argv | Pointer to array of command line arguments. |

Returns

This program exits with status 0.

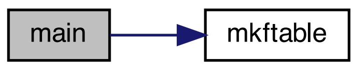
Definition at line 67 of file unipagecount.c.

```
{

    char inbuf[MAXBUF]; /* Max 256 characters in an input line */
    int i, j;  /* loop variables */
    unsigned plane=0; /* Unicode plane number, 0 to 0x16 */
    unsigned page;  /* unicode page (256 bytes wide) */
```

```
unsigned unichar; /* unicode character */
int pagecount[256] = {256 * 0};
int onepage=0; /* set to one if printing character grid for one page */
int pageno=0; /* page number selected if only examining one page */
int html=0;   /* =0: print plain text; =1: print HTML */
int links=0;  /* =1: print HTML links; =0: don’t print links */
void mkftable();  /* make (print) flipped HTML table */

size_t strlen();

if (argc > 1 && argv[1][0] == ’-’) {  /* Parse option */
   plane = 0;
   for (i = 1; i < argc; i++) {
      switch (argv[i][1]) {
         case ’p’:  /* specified -p<hexpage> -- use given page number */
            sscanf (&argv[1][2], ”%x”, &pageno);
            if (pageno >= 0 && pageno <= 255) onepage = 1;
            break;
         case ’h’:  /* print HTML table instead of text table */
            html = 1;
            break;
         case ’l’:  /* print hyperlinks in HTML table */
            links = 1;
            html = 1;
            break;
         case ’P’:  /* Plane number specified */
            plane = atoi(&argv[1][2]);
            break;
      }
   }
}
/*
   Initialize pagecount to account for noncharacters.
*/
if (!onepage && plane==0) {
   pagecount[0xfd] = 32;  /* for U+FDD0..U+FDEF */
}
pagecount[0xff] = 2;   /* for U+nnFFFE, U+nnFFFF */
/*
   Read one line at a time from input.  The format is:

      <hexpos>:<hexbitmap>

   where <hexpos> is the hexadecimal Unicode character position
   in the range 00..FF and <hexbitmap> is the sequence of hexadecimal
   digits of the character, laid out in a grid from left to right,
   top to bottom.  The character is assumed to be 16 rows of variable
   width.
*/
while (fgets (inbuf, MAXBUF-1, stdin) != NULL) {
   sscanf (inbuf, ”%X”, &unichar);
   page = unichar » 8;
   if (onepage) { /* only increment counter if this is page we want */
      if (page == pageno) { /* character is in the page we want */
         pagecount[unichar & 0xff]++; /* mark character as covered */
      }
   }
   else { /* counting all characters in all pages */
      if (plane == 0) {
         /* Don’t add in noncharacters (U+FDD0..U+FDEF, U+FFFE, U+FFFF) */
         if (unichar < 0xfdd0 || (unichar > 0xfdef && unichar < 0xfffe))
            pagecount[page]++;
      }
      else {
         if ((page » 8) == plane) { /* code point is in desired plane */
            pagecount[page & 0xFF]++;
         }
      }
   }
}
if (html) {
   mkftable (plane, pagecount, links);
}
else {  /* Otherwise, print plain text table */
   if (plane > 0) fprintf (stdout, ”  ”);
   fprintf (stdout,
      ”    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F\n”);
   for (i=0; i<0x10; i++) {
      fprintf (stdout,”%02X%X ”, plane, i); /* row header */
      for (j=0; j<0x10; j++) {
         if (onepage) {
```

```
                if (pagecount[i*16+j])
                    fprintf (stdout," *  ");
                else
                    fprintf (stdout," .  ");
            }
            else {
                fprintf (stdout, "%3X ", pagecount[i*16+j]);
            }
        }
        fprintf (stdout,"\n");
    }

  }
  exit (0);
}
```

Here is the call graph for this function:




#### 5.29.3.2 mkftable()

```
void mkftable (
        unsigned plane,
        int pagecount[256],
        int links )
```

Create an HTML table linked to PNG images.

This function creates an HTML table to show PNG files in a 16 by 16 grid. The background color of each "page" of 256 code points is shaded from red (for 0% coverage) to green (for 100% coverage).

Parameters

| | | |
|---|---|---|
| in | plane | The Unicode plane, 0..17. |
| in | pagecount | Array with count of glyphs in each 256 code point range. |

Parameters

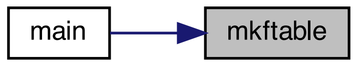
| | | |
|---|---|---|
| in | links | 1 = generate hyperlinks, 0 = do not generate hyperlinks. |

Definition at line 185 of file unipagecount.c.

```
{
   int i, j;
   int count;
   unsigned bgcolor;

   printf ("<html>\n");
   printf ("<body>\n");
   printf ("<table border=\"3\" align=\"center\">\n");
   printf ("  <tr><th colspan=\"16\" bgcolor=\"#ffcc80\">");
   printf ("GNU Unifont Glyphs<br>with Page Coverage for Plane %d<br>(Green=100%%, Red=0%%)</th></tr>\n",
  plane);
   for (i = 0x0; i <= 0xF; i++) {
      printf ("  <tr>\n");
      for (j = 0x0; j <= 0xF; j++) {
         count = pagecount[ (i « 4) | j ];

         /* print link in cell if links == 1 */
         if (plane != 0 || (i < 0xd || (i == 0xd && j < 0x8) || (i == 0xf && j > 0x8))) {
            /* background color is light green if completely done */
            if (count == 0x100) bgcolor = 0xccffcc;
            /* otherwise background is a shade of yellow to orange to red */
            else bgcolor = 0xff0000 | (count « 8) | (count » 1);
            printf ("    <td bgcolor=\"#%06X\">", bgcolor);
            if (plane == 0)
               printf ("<a href=\"png/plane%02X/uni%02X%X%X.png\">%X%X</a>", plane, plane, i, j, i, j);
            else
               printf ("<a href=\"png/plane%02X/uni%02X%X%X.png\">%02X%X%X</a>", plane, plane, i, j, plane, i, j);
            printf ("</td>\n");
         }
         else if (i == 0xd) {
            if (j == 0x8) {
               printf ("    <td align=\"center\" colspan=\"8\" bgcolor=\"#cccccc\">");
               printf ("<b>Surrogate Pairs</b>");
               printf ("</td>\n");
            }  /* otherwise don't print anything more columns in this row */
         }
         else if (i == 0xe) {
            if (j == 0x0) {
               printf ("    <td align=\"center\" colspan=\"16\" bgcolor=\"#cccccc\">");
               printf ("<b>Private Use Area</b>");
               printf ("</td>\n");
            }  /* otherwise don't print any more columns in this row */
         }
         else if (i == 0xf) {
            if (j == 0x0) {
               printf ("    <td align=\"center\" colspan=\"9\" bgcolor=\"#cccccc\">");
               printf ("<b>Private Use Area</b>");
               printf ("</td>\n");
            }
         }
      }
      printf ("  </tr>\n");
   }
   printf ("</table>\n");
   printf ("</body>\n");
   printf ("</html>\n");

   return;
```

```
}
```

Here is the caller graph for this function:




## 5.30 unipagecount.c

Go to the documentation of this file.

```
/**
  @file unipagecount.c

  @brief unipagecount - Count the number of glyphs defined in each page
                        of 256 code points

  @author Paul Hardy, unifoundry <at> unifoundry.com, December 2007

  @copyright Copyright (C) 2007, 2008, 2013, 2014 Paul Hardy

  This program counts the number of glyphs that are defined in each
  "page" of 256 code points, and prints the counts in an 8 x 8 grid.
  Input is from stdin.  Output is to stdout.

  The background color of each cell in a 16-by-16 grid of 256 code points
  is shaded to indicate percentage coverage.  Red indicates 0% coverage,
  green represents 100% coverage, and colors in between pure red and pure
  green indicate partial coverage on a scale.

  Each code point range number can be a hyperlink to a PNG file for
  that 256-code point range's corresponding bitmap glyph image.

  Synopsis:

        unipagecount < font_file.hex > count.txt
        unipagecount -phex_page_num < font_file.hex  -- just 256 points
        unipagecount -h < font_file.hex              -- HTML table
        unipagecount -P1 -h < font.hex > count.html  -- Plane 1, HTML out
        unipagecount -l < font_file.hex              -- linked HTML table
*/
/*
  LICENSE:

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

/*
  2018, Paul Hardy: Changed "Private Use" to "Private Use Area" in
  output HTML file.
*/

```

```
#include <stdio.h>
#include <stdlib.h>

#define MAXBUF 256 ///< Maximum input line size - 1.


/**
   @brief The main function.

   @param[in] argc The count of command line arguments.
   @param[in] argv Pointer to array of command line arguments.
   @return This program exits with status 0.
*/
int
main (int argc, char *argv[])
{

   char inbuf[MAXBUF]; /* Max 256 characters in an input line */
   int i, j;  /* loop variables */
   unsigned plane=0; /* Unicode plane number, 0 to 0x16 */
   unsigned page;  /* unicode page (256 bytes wide) */
   unsigned unichar; /* unicode character */
   int pagecount[256] = {256 * 0};
   int onepage=0; /* set to one if printing character grid for one page */
   int pageno=0; /* page number selected if only examining one page */
   int html=0;   /* =0: print plain text; =1: print HTML */
   int links=0;  /* =1: print HTML links; =0: don't print links */
   void mkftable();  /* make (print) flipped HTML table */

   size_t strlen();

   if (argc > 1 && argv[1][0] == '-') {  /* Parse option */
      plane = 0;
      for (i = 1; i < argc; i++) {
         switch (argv[i][1]) {
            case 'p':  /* specified -p<hexpage> -- use given page number */
               sscanf (&argv[1][2], "%x", &pageno);
               if (pageno >= 0 && pageno <= 255) onepage = 1;
               break;
            case 'h':  /* print HTML table instead of text table */
               html = 1;
               break;
            case 'l':  /* print hyperlinks in HTML table */
               links = 1;
               html = 1;
               break;
            case 'P':  /* Plane number specified */
               plane = atoi(&argv[1][2]);
               break;
         }
      }
   }
   /*
      Initialize pagecount to account for noncharacters.
   */
   if (!onepage && plane==0) {
      pagecount[0xfd] = 32;  /* for U+FDD0..U+FDEF */
   }
   pagecount[0xff] = 2;   /* for U+nnFFFE, U+nnFFFF */
   /*
      Read one line at a time from input.  The format is:

         <hexpos>:<hexbitmap>

      where <hexpos> is the hexadecimal Unicode character position
      in the range 00..FF and <hexbitmap> is the sequence of hexadecimal
      digits of the character, laid out in a grid from left to right,
      top to bottom.  The character is assumed to be 16 rows of variable
      width.
   */
   while (fgets (inbuf, MAXBUF-1, stdin) != NULL) {
      sscanf (inbuf, "%X", &unichar);
      page = unichar » 8;
      if (onepage) { /* only increment counter if this is page we want */
         if (page == pageno) { /* character is in the page we want */
            pagecount[unichar & 0xff]++; /* mark character as covered */
         }
      }
      else { /* counting all characters in all pages */
         if (plane == 0) {
            /* Don't add in noncharacters (U+FDD0..U+FDEF, U+FFFE, U+FFFF) */
```

```
            if (unichar < 0xfdd0 || (unichar > 0xfdef && unichar < 0xfffe))
               pagecount[page]++;
         }
         else {
            if ((page » 8) == plane) { /* code point is in desired plane */
               pagecount[page & 0xFF]++;
            }
         }
      }
   }
   if (html) {
      mkftable (plane, pagecount, links);
   }
   else { /* Otherwise, print plain text table */
      if (plane > 0) fprintf (stdout, "  ");
      fprintf (stdout,
         "    0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F\n");
      for (i=0; i<0x10; i++) {
         fprintf (stdout,"%02X%X ", plane, i); /* row header */
         for (j=0; j<0x10; j++) {
            if (onepage) {
               if (pagecount[i*16+j])
                  fprintf (stdout," *  ");
               else
                  fprintf (stdout," .  ");
            }
            else {
               fprintf (stdout, "%3X ", pagecount[i*16+j]);
            }
         }
         fprintf (stdout,"\n");
      }

   }
   exit (0);
}


/**
   @brief Create an HTML table linked to PNG images.

   This function creates an HTML table to show PNG files
   in a 16 by 16 grid.  The background color of each "page"
   of 256 code points is shaded from red (for 0% coverage)
   to green (for 100% coverage).

   @param[in] plane The Unicode plane, 0..17.
   @param[in] pagecount Array with count of glyphs in each 256 code point range.
   @param[in] links 1 = generate hyperlinks, 0 = do not generate hyperlinks.
*/
void
mkftable (unsigned plane, int pagecount[256], int links)
{
   int i, j;
   int count;
   unsigned bgcolor;

   printf ("<html>\n");
   printf ("<body>\n");
   printf ("<table border=\"3\" align=\"center\">\n");
   printf ("  <tr><th colspan=\"16\" bgcolor=\"#ffcc80\">");
   printf ("GNU Unifont Glyphs<br>with Page Coverage for Plane %d<br>(Green=100%%, Red=0%%)</th></tr>\n",
  plane);
   for (i = 0x0; i <= 0xF; i++) {
      printf ("  <tr>\n");
      for (j = 0x0; j <= 0xF; j++) {
         count = pagecount[ (i « 4) | j ];

         /* print link in cell if links == 1 */
         if (plane != 0 || (i < 0xd || (i == 0xd && j < 0x8) || (i == 0xf && j > 0x8))) {
            /* background color is light green if completely done */
            if (count == 0x100) bgcolor = 0xccffcc;
            /* otherwise background is a shade of yellow to orange to red */
            else bgcolor = 0xff0000 | (count « 8) | (count » 1);
            printf ("    <td bgcolor=\"#%06X\">", bgcolor);
            if (plane == 0)
               printf ("<a href=\"png/plane%02X/uni%02X%X%X.png\">%X%X</a>", plane, plane, i, j, i, j);
            else
               printf ("<a href=\"png/plane%02X/uni%02X%X%X.png\">%02X%X%X</a>", plane, plane, i, j, plane, i, j);
            printf ("</td>\n");
         }
```

```
         else if (i == 0xd) {
            if (j == 0x8) {
               printf ("    <td align=\"center\" colspan=\"8\" bgcolor=\"#cccccc\">");
               printf ("<b>Surrogate Pairs</b>");
               printf ("</td>\n");
            }  /* otherwise don't print anything more columns in this row */
         }
         else if (i == 0xe) {
            if (j == 0x0) {
               printf ("    <td align=\"center\" colspan=\"16\" bgcolor=\"#cccccc\">");
               printf ("<b>Private Use Area</b>");
               printf ("</td>\n");
            }  /* otherwise don't print any more columns in this row */
         }
         else if (i == 0xf) {
            if (j == 0x0) {
               printf ("    <td align=\"center\" colspan=\"9\" bgcolor=\"#cccccc\">");
               printf ("<b>Private Use Area</b>");
               printf ("</td>\n");
            }
         }
      }
      printf ("  </tr>\n");
   }
   printf ("</table>\n");
   printf ("</body>\n");
   printf ("</html>\n");

   return;
}
```

# Index