GNU Unifont

15.1.01

Generated by Doxygen 1.9.1

Chapter 1

Main Page

1.1 GNU Unifont C Utilities

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

1.2 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/.

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 Font↩ Forge.

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.

Minseo Lee created new Hangul glyphs for the original Unifont Johab 10/3 or 4/4 encoding. This was followed immediately after by Ho-Seok Ee, who created Hangul glyphs for a new, simpler Johab 6/3/1 encoding that are now in Unifont.

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
johab2syllables.c	Generate Hangul Syllables range with simple positioning
unibdf2hex.c	Convert a BDF file into a unifont.hex file
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
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
unigen-hangul.c	Generate modern and ancient Hangul syllables with shifting of final consonants combined with diphthongs having two long vertical strokes on the right
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

Program	Description
unihexgen.c	Generate a series of glyphs containing hexadecimal code points
unihexpose.c	Transpose Unifont .hex glyph bitmaps to simplify sending to graphics display controller chips that read bitmaps as a series of columns 8 rows (one byte) high
unijohab2html.c	Read a hangul-base.hex file and produce an HTML page as output showing juxtaposition and overlapping of all letter combinations in modern and ancient Hangul syllables
unipagecount.c	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
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
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
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
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.

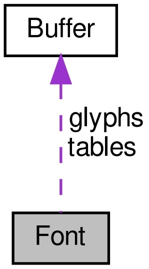
The documentation for this struct was generated from the following file:

- 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.

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 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.

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.

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

- src/hex2otf.c

4.6 PARAMS Struct Reference

Data Fields

- unsigned starting_codept
- unsigned cho_start
- unsigned cho_end
- unsigned jung_start
- unsigned jung_end
- unsigned jong_start
- unsigned jong_end
- FILE ∗ infp
- FILE ∗ outfp

4.6.1 Detailed Description

Definition at line 55 of file unigen-hangul.c.

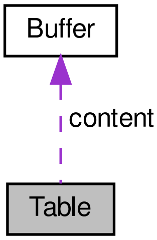
The documentation for this struct was generated from the following file:

- src/unigen-hangul.c

4.7 Table Struct Reference

Data structure for an OpenType table.

Collaboration diagram for Table:



Data Fields

- uint_fast32_t tag
- Buffer ∗ content

4.7.1 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.

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

- src/hex2otf.c

4.8 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

4.8.1 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.

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

- src/hex2otf.c

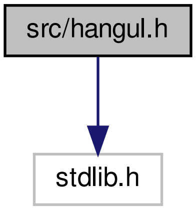
Chapter 5

File Documentation

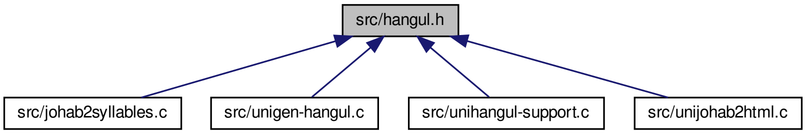
5.1 src/hangul.h File Reference

Define constants and function prototypes for using Hangul glyphs.

#include <stdlib.h>
Include dependency graph for hangul.h:



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



Macros

- #define MAXLINE 256

  Length of maximum file input line.
- #define EXTENDED_HANGUL /* Use rare Hangul code points beyond U+1100 */
- #define PUA_START 0xE000
- #define PUA_END 0xE8FF
- #define MAX_GLYPHS (PUA_END - PUA_START + 1) /* Maximum .hex file glyphs */
- #define CHO_UNICODE_START 0x1100

  Modern Hangul choseong start.
- #define CHO_UNICODE_END 0x115E

  Hangul Jamo choseong end.
- #define CHO_EXTA_UNICODE_START 0xA960

  Hangul Extended-A choseong start.
- #define CHO_EXTA_UNICODE_END 0xA97C

  Hangul Extended-A choseong end.
- #define JUNG_UNICODE_START 0x1161

  Modern Hangul jungseong start.
- #define JUNG_UNICODE_END 0x11A7

  Modern Hangul jungseong end.
- #define JUNG_EXTB_UNICODE_START 0xD7B0

  Hangul Extended-B jungseong start.
- #define JUNG_EXTB_UNICODE_END 0xD7C6

  Hangul Extended-B jungseong end.
- #define JONG_UNICODE_START 0x11A8

  Modern Hangul jongseong start.
- #define JONG_UNICODE_END 0x11FF

  Modern Hangul jongseong end.
- #define JONG_EXTB_UNICODE_START 0xD7CB

  Hangul Extended-B jongseong start.
- #define JONG_EXTB_UNICODE_END 0xD7FB

  Hangul Extended-B jongseong end.
- #define NCHO_MODERN 19

  19 modern Hangul Jamo choseong
- #define NCHO_ANCIENT 76

  ancient Hangul Jamo choseong
- #define NCHO_EXTA 29

  Hangul Extended-A choseong.
- #define NCHO_EXTA_RSRVD 3

  Reserved at end of Extended-A choseong.
- #define NJUNG_MODERN 21

  21 modern Hangul Jamo jungseong
- #define NJUNG_ANCIENT 50

  ancient Hangul Jamo jungseong
- #define NJUNG_EXTB 23

  Hangul Extended-B jungseong.
- #define NJUNG_EXTB_RSRVD 4

Reserved at end of Extended-B junseong.

- #define NJONG_MODERN 27

  28 modern Hangul Jamo jongseong
- #define NJONG_ANCIENT 61

  ancient Hangul Jamo jongseong
- #define NJONG_EXTB 49

  Hangul Extended-B jongseong.
- #define NJONG_EXTB_RSRVD 4

  Reserved at end of Extended-B jonseong.
- #define CHO_VARIATIONS 6

  6 choseong variations
- #define JUNG_VARIATIONS 3

  3 jungseong variations
- #define JONG_VARIATIONS 1

  1 jongseong variation
- #define CHO_HEX 0x0001

  Location of first choseong (location 0x0000 is a blank glyph)
- #define CHO_ANCIENT_HEX (CHO_HEX + CHO_VARIATIONS ∗ NCHO_MODERN)

  Location of first ancient choseong.
- #define CHO_EXTA_HEX (CHO_ANCIENT_HEX + CHO_VARIATIONS ∗ NCHO_ANCIENT)

  U+A960 Extended-A choseong.
- #define CHO_LAST_HEX (CHO_EXTA_HEX + CHO_VARIATIONS ∗ (NCHO_EXTA + NCHO_EXTA_RSRVD) - 1)

  U+A97F Extended-A last location in .hex file, including reserved Unicode code points at end.
- #define JUNG_HEX (CHO_LAST_HEX + 1)

  Location of first jungseong (will be 0x2FB)
- #define JUNG_ANCIENT_HEX (JUNG_HEX + JUNG_VARIATIONS ∗ NJUNG_MODERN)

  Location of first ancient jungseong.
- #define JUNG_EXTB_HEX (JUNG_ANCIENT_HEX + JUNG_VARIATIONS ∗ NJUNG_ANCIENT)

  U+D7B0 Extended-B jungseong.
- #define JUNG_LAST_HEX (JUNG_EXTB_HEX + JUNG_VARIATIONS ∗ (NJUNG_EXTB + NJUNG_EXTB_RSRVD) - 1)

  U+D7CA Extended-B last location in .hex file, including reserved Unicode code points at end.
- #define JONG_HEX (JUNG_LAST_HEX + 1)

  Location of first jongseong (will be 0x421)
- #define JONG_ANCIENT_HEX (JONG_HEX + JONG_VARIATIONS ∗ NJONG_MODERN)

  Location of first ancient jongseong.
- #define JONG_EXTB_HEX (JONG_ANCIENT_HEX + JONG_VARIATIONS ∗ NJONG_ANCIENT)

  U+D7CB Extended-B jongseong.
- #define JONG_LAST_HEX (JONG_EXTB_HEX + JONG_VARIATIONS ∗ (NJONG_EXTB + NJONG_EXTB_RSRVD) - 1)

  U+D7FF Extended-B last location in .hex file, including reserved Unicode code points at end.
- #define JAMO_HEX 0x0500

  Start of U+1100..U+11FF glyphs.
- #define JAMO_END 0x05FF

  End of U+1100..U+11FF glyphs.
- #define JAMO_EXTA_HEX 0x0600

Start of U+A960..U+A97F glyphs.
- #define JAMO_EXTA_END 0x061F

  End of U+A960..U+A97F glyphs.
- #define JAMO_EXTB_HEX 0x0620

  Start of U+D7B0..U+D7FF glyphs.
- #define JAMO_EXTB_END 0x066F

  End of U+D7B0..U+D7FF glyphs.
- #define TOTAL_CHO (NCHO_MODERN + NCHO_ANCIENT + NCHO_EXTA)
- #define TOTAL_JUNG (NJUNG_MODERN + NJUNG_ANCIENT + NJUNG_EXTB)
- #define TOTAL_JONG (NJONG_MODERN + NJONG_ANCIENT + NJONG_EXTB)

Functions

- unsigned hangul_read_base8 (FILE ∗infp, unsigned char base[ ][32])

  Read hangul-base.hex file into a unsigned char array.
- unsigned hangul_read_base16 (FILE ∗infp, unsigned base[ ][16])

  Read hangul-base.hex file into a unsigned array.
- void hangul_decompose (unsigned codept, int ∗initial, int ∗medial, int ∗final)

  Decompose a Hangul Syllables code point into three letters.
- unsigned hangul_compose (int initial, int medial, int final)

  Compose a Hangul syllable into a code point, or 0 if none exists.
- void hangul_hex_indices (int choseong, int jungseong, int jongseong, int ∗cho_index, int ∗jung_index, int ∗jong_index)

  Determine index values to the bitmaps for a syllable's components.
- void hangul_variations (int choseong, int jungseong, int jongseong, int ∗cho_var, int ∗jung_var, int ∗jong_var)

  Determine the variations of each letter in a Hangul syllable.
- int is_wide_vowel (int vowel)

  Whether vowel has rightmost vertical stroke to the right.
- int cho_variation (int choseong, int jungseong, int jongseong)

  Return the Johab 6/3/1 choseong variation for a syllable.
- int jung_variation (int choseong, int jungseong, int jongseong)

  Return the Johab 6/3/1 jungseong variation.
- int jong_variation (int choseong, int jungseong, int jongseong)

  Return the Johab 6/3/1 jongseong variation.
- void hangul_syllable (int choseong, int jungseong, int jongseong, unsigned char hangul_base[ ][32], unsigned char ∗syllable)

  Given letters in a Hangul syllable, return a glyph.
- int glyph_overlap (unsigned ∗glyph1, unsigned ∗glyph2)

  See if two glyphs overlap.
- void combine_glyphs (unsigned ∗glyph1, unsigned ∗glyph2, unsigned ∗combined_glyph)

  Combine two glyphs into one glyph.
- void one_jamo (unsigned glyph_table[MAX_GLYPHS][16], unsigned jamo, unsigned ∗jamo_glyph)

  Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.
- void combined_jamo (unsigned glyph_table[MAX_GLYPHS][16], unsigned cho, unsigned jung, unsigned jong, unsigned ∗combined_glyph)

  Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.
- void print_glyph_txt (FILE ∗fp, unsigned codept, unsigned ∗this_glyph)

  Print one glyph in Unifont hexdraw plain text style.
- void print_glyph_hex (FILE ∗fp, unsigned codept, unsigned ∗this_glyph)

  Print one glyph in Unifont hexdraw hexadecimal string style.

5.1.1 Detailed Description

Define constants and function prototypes for using Hangul glyphs.

Author

Paul Hardy

Copyright

5.1.2 Function Documentation

5.1.2.1 cho_variation()

```
int cho_variation (
                    int choseong,
                    int jungseong,
                    int jongseong )
```

Return the Johab 6/3/1 choseong variation for a syllable.

This function takes the two or three (if jongseong is included) letters that comprise a syllable and determine the variation of the initial consonant (choseong).

Each choseong has 6 variations:

Variation Occurrence
0 Choseong with a vertical vowel such as "A". 1 Choseong with a horizontal vowel such as "O". 2 Choseong with a vertical and horizontal vowel such as "WA". 3 Same as variation 0, but with jongseong (final consonant). 4 Same as variation 1, but with jongseong (final consonant). Also a horizontal vowel pointing down, such as U and YU. 5 Same as variation 2, but with jongseong (final consonant). Also a horizontal vowel pointing down with vertical element, such as WEO, WE, and WI.
In addition, if the vowel is horizontal and a downward-pointing stroke as in the modern letters U, WEO, WE, WI, and YU, and in archaic letters YU-YEO, YU-YE, YU-I, araea, and araea-i, then 3 is added to the initial variation of 0 to 2, resulting in a choseong variation of 3 to 5, respectively.

Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable.

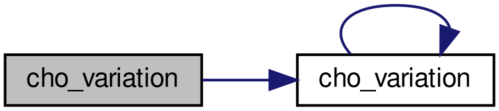
Returns

The choseong variation, 0 to 5.

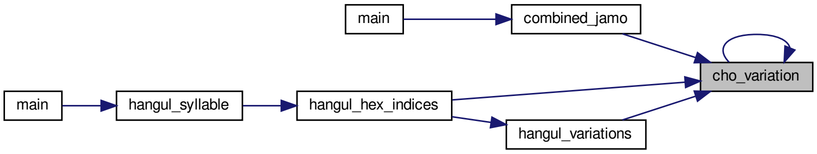
Definition at line 350 of file unihangul-support.c.

```
350                                                     {
351    int cho_variation;  /* Return value */
352 
353    /*
354 The Choseong cho_var is determined by the
355 21 modern + 50 ancient Jungseong, and whether
356 or not the syllable contains a final consonant
357 (Jongseong).
358 */
359    static int choseong_var [TOTAL_JUNG + 1] = {
360          /*
361 Modern Jungseong in positions 0..20.
362 */
363 /* Location Variations  Unicode Range    Vowel #    Vowel Names  */
364 /* -------- ----------  --------------   -------    -----------  */
365 /* 0x2FB */ 0, 0, 0, // U+1161..U+1163-->[ 0.. 2]  A,   AE,  YA
366 /* 0x304 */ 0, 0, 0, // U+1164..U+1166-->[ 3.. 5]  YAE, EO,  E
367 /* 0x30D */ 0, 0,    // U+1167..U+1168-->[ 6.. 7]  YEO, YE
368 /* 0x313 */ 1,       // U+1169         -->[ 8]      O
369 /* 0x316 */ 2, 2, 2, // U+116A..U+116C-->[ 9..11]  WA,  WAE, WE
370 /* 0x31F */ 1, 4,    // U+116D..U+116E-->[12..13]  YO,  U
371 /* 0x325 */ 5, 5, 5, // U+116F..U+1171-->[14..16]  WEO, WE,  WI
372 /* 0x32E */ 4, 1,    // U+1172..U+1173-->[17..18]  YU,  EU
373 /* 0x334 */ 2,       // U+1174         -->[19]      YI
374 /* 0x337 */ 0,       // U+1175         -->[20]      I
375          /*
376 Ancient Jungseong in positions 21..70.
377 */
378 /* Location  Variations  Unicode Range    Vowel #       Vowel Names  */
379 /* --------  ----------  --------------   -------       -----------  */
380 /* 0x33A: */ 2, 5, 2, // U+1176..U+1178-->[21..23]   A-O,    A-U,   YA-O
381 /* 0x343: */ 2, 2, 5, // U+1179..U+117B-->[24..26]  YA-YO,  EO-O,   EU-U
382 /* 0x34C: */ 2, 2, 5, // U+117C..U+117E-->[27..29]  EO-EU, YEO-O,  YEO-U
383 /* 0x355: */ 2, 5, 5, // U+117F..U+1181-->[30..32]   O-EO,   O-E,    O-YE,
384 /* 0x35E: */ 4, 4, 2, // U+1182..U+1184-->[33..35]   O-O,    O-U,   YO-YA,
385 /* 0x367: */ 2, 2, 5, // U+1185..U+1187-->[36..38]  YO-YAE, YO-YEO, YO-O,
386 /* 0x370: */ 2, 5, 5, // U+1188..U+118A-->[39..41]  YO-I,    U-A,    U-AE,
387 /* 0x379: */ 5, 5, 5, // U+118B..U+118D-->[42..44] U-EO-EU,  U-YE,   U-U,
388 /* 0x382: */ 5, 5, 5, // U+118E..U+1190-->[45..47]  YU-A,   YU-EO,  YU-E,
389 /* 0x38B: */ 5, 5, 2, // U+1191..U+1193-->[48..50]  YU-YEO, YU-YE,  YU-U,
390 /* 0x394: */ 5, 2, 2, // U+1194..U+1196-->[51..53]  YU-I,   EU-U,   EU-EU,
391 /* 0x39D: */ 2, 0, 0, // U+1197..U+1199-->[54..56]  YI-U,    I-A,    I-YA,
392 /* 0x3A6: */ 2, 5, 2, // U+119A..U+119C-->[57..59]   I-O,    I-U,    I-EU,
393 /* 0x3AF: */ 0, 1, 2, // U+119D..U+119F-->[60..62] I-ARAEA, ARAEA, ARAEA-EO,
394 /* 0x3B8: */ 1, 2, 1, // U+11A0..U+11A2-->[63..65] ARAEA-U, ARAEA-I,SSANGARAEA,
395 /* 0x3C1: */ 2, 5, 0, // U+11A3..U+11A5-->[66..68]   A-EU,  YA-U,  YEO-YA,
396 /* 0x3CA: */ 2, 2,    // U+11A6..U+11A7-->[69..70]   O-YA,   O-YAE,
397 #ifdef EXTENDED_HANGUL
398 /* 0x3D0: */ 2, 4, 5, // U+D7B0..U+D7B2-->[71..73]   O-YEO,  O-O-I, YO-A,
399 /* 0x3D9: */ 5, 2, 5, // U+D7B3..U+D7B5-->[74..76]  YO-AE,  YO-EO,   U-YEO,
400 /* 0x3E2: */ 5, 5, 4, // U+D7B6..U+D7B8-->[77..79]   U-I-I, YU-AE,  YU-O,
401 /* 0x3EB: */ 5, 2, 5, // U+D7B9..U+D7BB-->[80..82]  EU-A,   EU-EO,  EU-E,
402 /* 0x3F4: */ 4, 2, 3, // U+D7BC..U+D7BE-->[83..85]  EU-O,    I-YA-O, I-YAE,
403 /* 0x3FD: */ 3, 3, 2, // U+D7BF..U+D7C1-->[86..88]   I-YEO,  I-YE,   I-O-I,
404 /* 0x406: */ 2, 2, 0, // U+D7C2..U+D7C4-->[89..91]   I-YO,   I-YU,   I-I,
405 /* 0x40F: */ 2, 2,    // U+D7C5..U+D7C6-->[92..93] ARAEA-A, ARAEA-E,
406 /* 0x415: */ -1       // Mark end of list of vowels.
407 #else
408 /* 0x310: */ -1       // Mark end of list of vowels.
409 #endif
410    };
411 
412 
413    if (jungseong < 0 || jungseong >= TOTAL_JUNG) {
414       cho_variation = -1;
415    }
416    else {
417       cho_variation = choseong_var [jungseong];
418       if (choseong >= 0 && jongseong >= 0 && cho_variation < 3)
419          cho_variation += 3;
420    }
421 
422 
423    return cho_variation;
424 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.2 combine_glyphs()

```
void combine_glyphs (
                    unsigned * glyph1,
                    unsigned * glyph2,
                    unsigned * combined_glyph )
```

Combine two glyphs into one glyph.

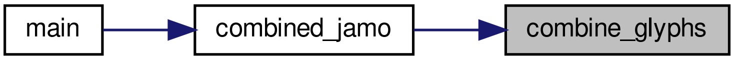
Parameters

in	glyph1	The first glyph to overlap.
in	glyph2	The second glyph to overlap.
out	combined_glyph	The returned combination glyph.

Definition at line 637 of file unihangul-support.c.

```
                                        {
   int i;

   for (i = 0; i < 16; i++)
      combined_glyph [i] = glyph1 [i] | glyph2 [i];

   return;
}
```

Here is the caller graph for this function:



5.1.2.3 combined_jamo()

```
void combined_jamo (
                unsigned glyph_table[MAX_GLYPHS][16],
                unsigned cho,
                unsigned jung,
                unsigned jong,
                unsigned * combined_glyph )
```

Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.

This function converts input Hangul choseong, jungseong, and jongseong Unicode code triplets into a Hangul syllable. Any of those with an out of range code point are assigned a blank glyph for combining.

This function performs the following steps:

```
 1) Determine the sequence number of choseong, jungseong,
    and jongseong, from 0 to the total number of choseong,
    jungseong, or jongseong, respectively, minus one.  The
    sequence for each is as follows:

    a) Choseong: Unicode code points of U+1100..U+115E
       and then U+A960..U+A97C.

    b) Jungseong: Unicode code points of U+1161..U+11A7
       and then U+D7B0..U+D7C6.

    c) Jongseong: Unicode code points of U+11A8..U+11FF
       and then U+D7CB..U+D7FB.

 2) From the choseong, jungseong, and jongseong sequence number,
    determine the variation of choseong and jungseong (there is
    only one jongseong variation, although it is shifted right
    by one column for some vowels with a pair of long vertical
    strokes on the right side).

 3) Convert the variation numbers for the three syllable
    components to index locations in the glyph array.

 4) Combine the glyph array glyphs into a syllable.
```

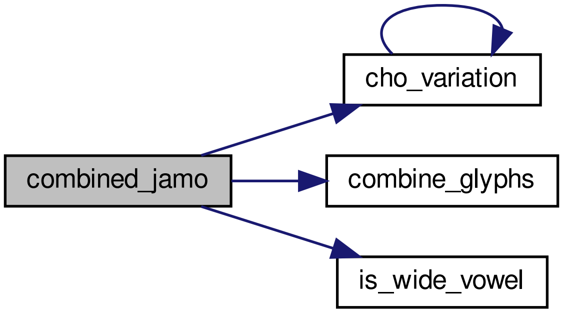
Parameters

in	glyph_table	The collection of all jamo glyphs.
in	cho	The choseong Unicode code point, 0 or 0x1100..0x115F.
in	jung	The jungseong Unicode code point, 0 or 0x1160..0x11A7.
in	jong	The jongseong Unicode code point, 0 or 0x11A8..0x11FF.
out	combined_glyph	The output glyph, 16 columns in each of 16 rows.

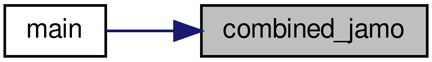
Definition at line 787 of file unihangul-support.c.

```
                           {

  int i;  /* Loop variable. */
  int cho_num,   jung_num,   jong_num;
  int cho_group, jung_group, jong_group;
  int cho_index, jung_index, jong_index;

  unsigned tmp_glyph[16]; /* Hold shifted jongsung for wide vertical vowel. */

  int  cho_variation (int choseong, int jungseong, int jongseong);

  void combine_glyphs (unsigned *glyph1, unsigned *glyph2,
                unsigned *combined_glyph);


  /* Choose a blank glyph for each syllalbe by default. */
  cho_index = jung_index = jong_index = 0x000;

  /*
Convert Unicode code points to jamo sequence number
of each letter, or -1 if letter is not in valid range.
*/
  if (cho >= 0x1100 && cho <= 0x115E)
    cho_num = cho - CHO_UNICODE_START;
  else if (cho >= CHO_EXTA_UNICODE_START &&
        cho < (CHO_EXTA_UNICODE_START + NCHO_EXTA))
    cho_num = cho - CHO_EXTA_UNICODE_START + NCHO_MODERN + NJONG_ANCIENT;
  else
    cho_num = -1;

  if (jung >= 0x1161 && jung <= 0x11A7)
    jung_num = jung - JUNG_UNICODE_START;
  else if (jung >= JUNG_EXTB_UNICODE_START &&
        jung < (JUNG_EXTB_UNICODE_START + NJUNG_EXTB))
    jung_num = jung - JUNG_EXTB_UNICODE_START + NJUNG_MODERN + NJUNG_ANCIENT;
  else
    jung_num = -1;

  if (jong >= 0x11A8 && jong <= 0x11FF)
    jong_num = jong - JONG_UNICODE_START;
  else if (jong >= JONG_EXTB_UNICODE_START &&
        jong < (JONG_EXTB_UNICODE_START + NJONG_EXTB))
    jong_num = jong - JONG_EXTB_UNICODE_START + NJONG_MODERN + NJONG_ANCIENT;
  else
    jong_num = -1;

  /*
Choose initial consonant (choseong) variation based upon
the vowel (jungseong) if both are specified.
*/
  if (cho_num < 0) {
    cho_index = cho_group = 0;  /* Use blank glyph for choseong. */
  }
  else {
    if (jung_num < 0 && jong_num < 0) {  /* Choseong is by itself. */
      cho_group = 0;
      if (cho_index < (NCHO_MODERN + NCHO_ANCIENT))
        cho_index = cho_num + JAMO_HEX;
      else  /* Choseong is in Hangul Jamo Extended-A range. */
        cho_index = cho_num - (NCHO_MODERN + NCHO_ANCIENT)
                     + JAMO_EXTA_HEX;
    }
    else {
      if (jung_num >= 0) {  /* Valid jungseong with choseong. */
        cho_group = cho_variation (cho_num, jung_num, jong_num);
      }
      else {  /* Invalid vowel; see if final consonant is valid. */
        /*
If initial consonant and final consonant are specified,
set cho_group to 4, which is the group tha would apply
to a horizontal-only vowel such as Hangul ”O”, so the
consonant appears full-width.
*/
        cho_group = 0;
        if (jong_num >= 0) {
          cho_group = 4;
        }
      }
      cho_index = CHO_HEX + CHO_VARIATIONS * cho_num +
              cho_group;
      }  /* Choseong combined with jungseong and/or jongseong. */
   }  /* Valid choseong. */

   /*
Choose vowel (jungseong) variation based upon the choseong
and jungseong.
*/
   jung_index = jung_group = 0;  /* Use blank glyph for jungseong. */

   if (jung_num >= 0) {
      if (cho_num < 0 && jong_num < 0) {  /* Jungseong is by itself. */
         jung_group = 0;
         jung_index = jung_num + JUNG_UNICODE_START;
      }
      else {
         if (jong_num >= 0) {  /* If there is a final consonant. */
            if (jong_num == 3) /* Nieun; choose variation 3. */
               jung_group = 2;
            else
               jung_group = 1;
         }  /* Valid jongseong. */
         /* If valid choseong but no jongseong, choose jungseong variation 0. */
         else if (cho_num >= 0)
            jung_group = 0;
      }
      jung_index = JUNG_HEX + JUNG_VARIATIONS * jung_num + jung_group;
   }

   /*
Choose final consonant (jongseong) based upon whether choseong
and/or jungseong are present.
*/
   if (jong_num < 0) {
      jong_index = jong_group = 0;  /* Use blank glyph for jongseong. */
   }
   else {  /* Valid jongseong. */
      if (cho_num < 0 && jung_num < 0) {  /* Jongseong is by itself. */
         jong_group = 0;
         jong_index = jung_num + 0x4A8;
      }
      else {  /* There is only one jongseong variation if combined. */
         jong_group = 0;
         jong_index = JONG_HEX + JONG_VARIATIONS * jong_num +
                      jong_group;
      }
   }

   /*
Now that we know the index locations for choseong, jungseong, and
jongseong glyphs, combine them into one glyph.
*/
   combine_glyphs (glyph_table [cho_index], glyph_table [jung_index],
                   combined_glyph);

   if (jong_index > 0) {
      /*
If the vowel has a vertical stroke that is one column
away from the right border, shift this jongseung right
by one column to line up with the rightmost vertical
stroke in the vowel.
*/
      if (is_wide_vowel (jung_num)) {
         for (i = 0; i < 16; i++) {
            tmp_glyph [i] = glyph_table [jong_index] [i] » 1;
         }
         combine_glyphs (combined_glyph, tmp_glyph,
                         combined_glyph);
      }
      else {
         combine_glyphs (combined_glyph, glyph_table [jong_index],
                         combined_glyph);
      }
   }

   return;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.4 glyph_overlap()

```
int glyph_overlap (
                    unsigned * glyph1,
                    unsigned * glyph2 )
```

See if two glyphs overlap.

Parameters

in	glyph1	The first glyph, as a 16-row bitmap.
in	glyph2	The second glyph, as a 16-row bitmap.

Returns

0 if no overlaps between glyphs, 1 otherwise.

Definition at line 613 of file unihangul-support.c.

```
613                                            {
614     int overlaps;  /* Return value; 0 if no overlaps, -1 if overlaps. */
615     int i;
616
```

```
   /* Check for overlaps between the two glyphs. */

   i = 0;
   do {
      overlaps = (glyph1[i] & glyph2[i]) != 0;
      i++;
   }  while (i < 16 && overlaps == 0);

   return overlaps;
}
```

5.1.2.5 hangul_compose()

```
unsigned hangul_compose (
                int initial,
                int medial,
                int final )
```

Compose a Hangul syllable into a code point, or 0 if none exists.

This function takes three letters that can form a modern Hangul syllable and returns the corresponding Unicode Hangul Syllables code point in the range 0xAC00 to 0xD7A3.

If a three-letter combination includes one or more archaic letters, it will not map into the Hangul Syllables range. In that case, the returned code point will be 0 to indicate that no valid Hangul Syllables code point exists.

Parameters

in	initial	The first letter (choseong), 0 to 18.
in	medial	The second letter (jungseong), 0 to 20.
in	final	The third letter (jongseong), 0 to 26 or -1 if none.

Returns

The Unicode Hangul Syllables code point, 0xAC00 to 0xD7A3.

Definition at line 201 of file unihangul-support.c.

```
201                                        {
202    unsigned codept;
203
204
205    if (initial >= 0 && initial <= 18 &&
206        medial  >= 0 && medial  <= 20 &&
207        final   >= 0 && final   <= 26) {
208
209       codept  = 0xAC00;
210       codept += initial * 21 * 28;
211       codept += medial  * 28;
212       codept += final + 1;
213    }
214    else {
215       codept = 0;
216    }
217
218    return codept;
219 }
```

5.1.2.6 hangul_decompose()

```
void hangul_decompose (
                unsigned codept,
                int * initial,
                int * medial,
                int * final )
```

Decompose a Hangul Syllables code point into three letters.
Decompose a Hangul Syllables code point (U+AC00..U+D7A3) into:

- Choseong 0-19
- Jungseong 0-20
- Jongseong 0-27 or -1 if no jongseong

All letter values are set to -1 if the letters do not form a syllable in the Hangul Syllables range. This function only handles modern Hangul, because that is all that is in the Hangul Syllables range.

Parameters

in	codept	The Unicode code point to decode, from 0xAC00 to 0xD7A3.
out	initial	The 1st letter (choseong) in the syllable.
out	initial	The 2nd letter (jungseong) in the syllable.
out	initial	The 3rd letter (jongseong) in the syllable.

Definition at line 167 of file unihangul-support.c.

```
167                                                   {
168
169    if (codept < 0xAC00 || codept > 0xD7A3) {
170       *initial = *medial = *final = -1;
171    }
172    else {
173       codept  -= 0xAC00;
174       *initial =  codept / (28  * 21);
175       *medial  = (codept /  28) % 21;
176       *final   =  codept %  28 - 1;
177    }
178
179    return;
180 }
```

Here is the caller graph for this function:

main
hangul_decompose

5.1.2.7 hangul_hex_indices()

void hangul_hex_indices (
int choseong,
int jungseong,
int jongseong,
int ∗ cho_index,
int ∗ jung_index,
int ∗ jong_index)

Determine index values to the bitmaps for a syllable's components.
This function reads these input values for modern and ancient Hangul letters:

- Choseong number (0 to the number of modern and archaic choseong - 1.
- Jungseong number (0 to the number of modern and archaic jungseong - 1.
- Jongseong number (0 to the number of modern and archaic jongseong - 1, or -1 if none.

It then determines the variation of each letter given the combination with the other two letters (or just choseong and jungseong if the jongseong value is -1).
These variations are then converted into index locations within the glyph array that was read in from the hangul-base.hex file. Those index locations can then be used to form a composite syllable.
There is no restriction to only use the modern Hangul letters.

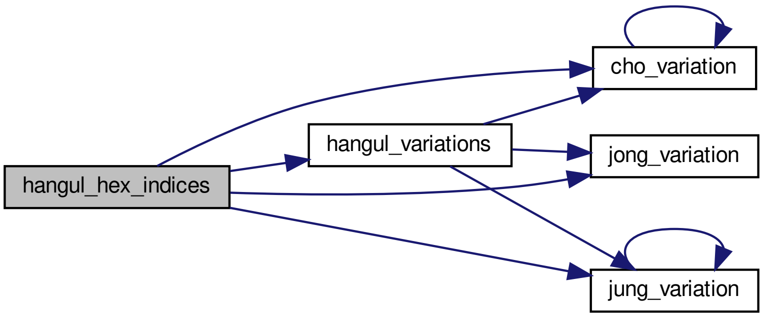
Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable, or -1 if none.
out	cho_index	Index location to the 1st letter variation from the hangul-base.hex file.
out	jung_index	Index location to the 2nd letter variation from the hangul-base.hex file.
out	jong_index	Index location to the 3rd letter variation from the hangul-base.hex file.

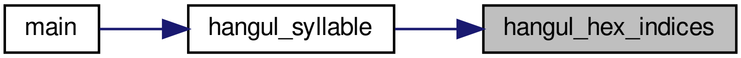
Definition at line 249 of file unihangul-support.c.

```
                                                            {

  int cho_variation, jung_variation, jong_variation;  /* Letter variations */

  void hangul_variations (int choseong, int jungseong, int jongseong,
          int *cho_variation, int *jung_variation, int *jong_variation);


  hangul_variations (choseong, jungseong, jongseong,
                     &cho_variation, &jung_variation, &jong_variation);

   *cho_index = CHO_HEX +  choseong * CHO_VARIATIONS +  cho_variation;
  *jung_index = JUNG_HEX      + jungseong * JUNG_VARIATIONS      + jung_variation;;
  *jong_index = jongseong < 0 ? 0x0000 :
                JONG_HEX + jongseong * JONG_VARIATIONS + jong_variation;

  return;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.8 hangul_read_base16()

```
unsigned hangul_read_base16 (
        FILE * infp,
        unsigned base[ ][16] )
```

Read hangul-base.hex file into a unsigned array.

Read a Hangul base .hex file with separate choseong, jungseong, and jongseong glyphs for syllable formation. The order is:

- Empty glyph in 0x0000 position.
- Initial consonants (choseong).
- Medial vowels and dipthongs (jungseong).
- Final consonants (jongseong).
- Individual letter forms in isolation, not for syllable formation.

The letters are arranged with all variations for one letter before continuing to the next letter. In the current encoding, there are 6 variations of choseong, 3 of jungseong, and 1 of jongseong per letter.

Parameters

in	Input	file pointer; can be stdin.
out	Array	of bit patterns, with 16 16-bit values per letter.

Returns

The maximum code point value read in the file.

Definition at line 116 of file unihangul-support.c.

```
                                                      {
   unsigned codept;
   unsigned max_codept;
   int      i, j;
   char     instring[MAXLINE];


   max_codept = 0;

   while (fgets (instring, MAXLINE, infp) != NULL) {
      sscanf (instring, "%X", &codept);
      codept -= PUA_START;
      /* If code point is within range, add it */
      if (codept < MAX_GLYPHS) {
         /* Find the start of the glyph bitmap. */
         for (i = 1; instring[i] != '\0' && instring[i] != ':'; i++);
         if (instring[i] == ':') {
            i++;  /* Skip over ':' to get to start of bitmap. */
            for (j = 0; j < 16; j++) {
               sscanf (&instring[i], "%4X", &base[codept][j]);
               i += 4;
            }
            if (codept > max_codept) max_codept = codept;
         }
      }
   }

   return max_codept;
}
```

Here is the caller graph for this function:

main
hangul_read_base16

5.1.2.9 hangul_read_base8()

unsigned hangul_read_base8 (
FILE ∗ infp,
unsigned char base[ ][32])

Read hangul-base.hex file into a unsigned char array.

Read a Hangul base .hex file with separate choseong, jungseong, and jongseong glyphs for syllable formation. The order is:

- Empty glyph in 0x0000 position.
- Initial consonants (choseong).
- Medial vowels and dipthongs (jungseong).
- Final consonants (jongseong).
- Individual letter forms in isolation, not for syllable formation.

The letters are arranged with all variations for one letter before continuing to the next letter. In the current encoding, there are 6 variations of choseong, 3 of jungseong, and 1 of jongseong per letter.

Parameters

in	Input	file pointer; can be stdin.
out	Array	of bit patterns, with 32 8-bit values per letter.

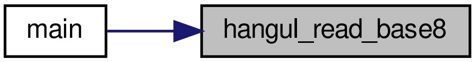
Returns

The maximum code point value read in the file.

Definition at line 63 of file unihangul-support.c.

```
                                                        {
   unsigned codept;
   unsigned max_codept;
   int      i, j;
   char     instring[MAXLINE];


   max_codept = 0;

   while (fgets (instring, MAXLINE, infp) != NULL) {
      sscanf (instring, "%X", &codept);
      codept -= PUA_START;
      /* If code point is within range, add it */
      if (codept < MAX_GLYPHS) {
         /* Find the start of the glyph bitmap. */
         for (i = 1; instring[i] != '\0' && instring[i] != ':'; i++);
         if (instring[i] == ':') {
            i++;  /* Skip over ':' to get to start of bitmap. */
            for (j = 0; j < 32; j++) {
               sscanf (&instring[i], "%2hhX", &base[codept][j]);
               i += 2;
            }
            if (codept > max_codept) max_codept = codept;
         }
      }
   }

   return max_codept;
}
```

Here is the caller graph for this function:



5.1.2.10 hangul_syllable()

void hangul_syllable (
int choseong,
int jungseong,
int jongseong,
unsigned char hangul_base[ ][32],
unsigned char ∗ syllable)

Given letters in a Hangul syllable, return a glyph.

This function returns a glyph bitmap comprising up to three Hangul letters that form a syllable. It reads the three component letters (choseong, jungseong, and jungseong), then calls a function that determines the appropriate variation of each letter, returning the letter bitmap locations in the glyph array. Then these letter bitmaps are combined with a logical OR operation to produce a final bitmap, which forms a 16 row by 16 column bitmap glyph.

Parameters

in	choseong	The 1st letter in the composite glyph.
in	jungseong	The 2nd letter in the composite glyph.
in	jongseong	The 3rd letter in the composite glyph.

Parameters

in	hangul_base	The glyphs read from the "hangul_base.hex" file.

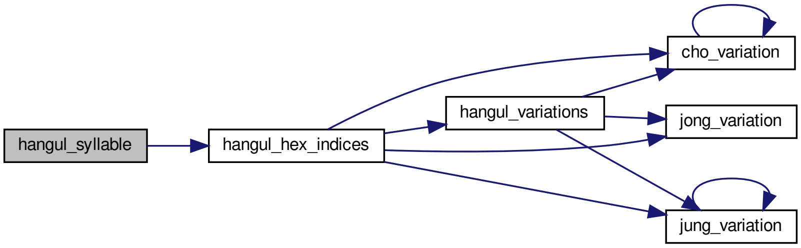
Returns

syllable The composite syllable, as a 16 by 16 pixel bitmap.

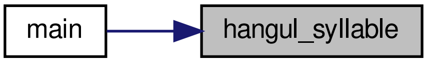
Definition at line 583 of file unihangul-support.c.

```
                                                            {

   int      i;  /* loop variable */
   int      cho_hex, jung_hex, jong_hex;
   unsigned char glyph_byte;


   hangul_hex_indices (choseong, jungseong, jongseong,
                       &cho_hex, &jung_hex, &jong_hex);

   for (i = 0; i < 32; i++) {
      glyph_byte  = hangul_base [cho_hex][i];
      glyph_byte |= hangul_base [jung_hex][i];
      if (jong_hex >= 0) glyph_byte |= hangul_base [jong_hex][i];
      syllable[i] = glyph_byte;
   }

   return;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.11 hangul_variations()

void hangul_variations (
int choseong,

int jungseong,
int jongseong,
int ∗ cho_var,
int ∗ jung_var,
int ∗ jong_var)

Determine the variations of each letter in a Hangul syllable.

Given the three letters that will form a syllable, return the variation of each letter used to form the composite glyph.

This function can determine variations for both modern and archaic Hangul letters; it is not limited to only the letters combinations that comprise the Unicode Hangul Syllables range.

This function reads these input values for modern and ancient Hangul letters:

- Choseong number (0 to the number of modern and archaic choseong - 1.
- Jungseong number (0 to the number of modern and archaic jungseong - 1.
- Jongseong number (0 to the number of modern and archaic jongseong - 1, or -1 if none.

It then determines the variation of each letter given the combination with the other two letters (or just choseong and jungseong if the jongseong value is -1).

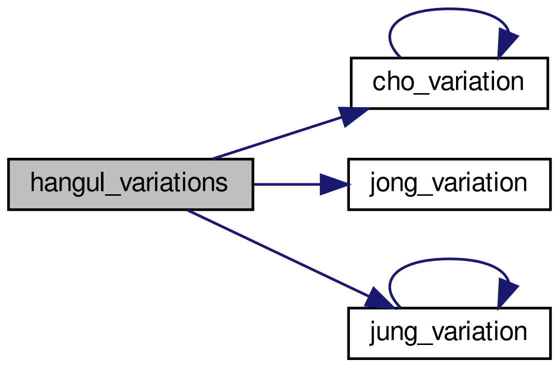
Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable, or -1 if none.
out	cho_var	Variation of the 1st letter from the hangul-base.hex file.
out	jung_var	Variation of the 2nd letter from the hangul-base.hex file.
out	jong_var	Variation of the 3rd letter from the hangul-base.hex file.

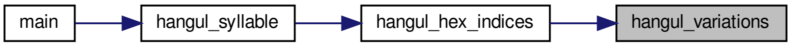
Definition at line 298 of file unihangul-support.c.

```
                                                    {

   int  cho_variation (int choseong, int jungseong, int jongseong);
   int jung_variation (int choseong, int jungseong, int jongseong);
   int jong_variation (int choseong, int jungseong, int jongseong);

   /*
Find the variation for each letter component.
*/
    *cho_var =  cho_variation (choseong, jungseong, jongseong);
   *jung_var = jung_variation (choseong, jungseong, jongseong);
   *jong_var = jong_variation (choseong, jungseong, jongseong);


   return;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.12 is_wide_vowel()

```
int is_wide_vowel (
                    int vowel )
```

Whether vowel has rightmost vertical stroke to the right.

Parameters

in	vowel	Vowel number, from 0 to TOTAL_JUNG - 1.

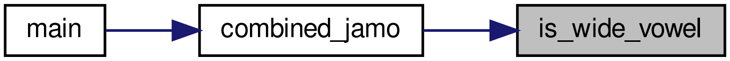
Returns

1 if this vowel's vertical stroke is wide on the right side; else 0.

Definition at line 434 of file unihangul-support.c.

```
434                           {
435    int retval;  /* Return value. */
436
437    static int wide_vowel [TOTAL_JUNG + 1] = {
438          /*
439 Modern Jungseong in positions 0..20.
440 */
441 /* Location Variations  Unicode Range    Vowel #     Vowel Names  */
442 /* -------- ----------  --------------    -------     -----------  */
/* 0x2FB */ 0, 1, 0, // U+1161..U+1163-->[ 0.. 2]  A,   AE,  YA
/* 0x304 */ 1, 0, 1, // U+1164..U+1166-->[ 3.. 5]  YAE, EO,  E
/* 0x30D */ 0, 1,    // U+1167..U+1168-->[ 6.. 7]  YEO, YE
/* 0x313 */ 0,       // U+1169        -->[ 8]      O
/* 0x316 */ 0, 1, 0, // U+116A..U+116C-->[ 9..11]  WA,  WAE, WE
/* 0x31F */ 0, 0,    // U+116D..U+116E-->[12..13]  YO,  U
/* 0x325 */ 0, 1, 0, // U+116F..U+1171-->[14..16]  WEO, WE,  WI
/* 0x32E */ 0, 0,    // U+1172..U+1173-->[17..18]  YU,  EU
/* 0x334 */ 0,       // U+1174        -->[19]      YI
/* 0x337 */ 0,       // U+1175        -->[20]      I
        /*
Ancient Jungseong in positions 21..70.
*/
/* Location  Variations  Unicode Range    Vowel #       Vowel Names  */
/* --------  ----------  --------------   -------       -----------  */
/* 0x33A: */ 0, 0, 0, // U+1176..U+1178-->[21..23]   A-O,    A-U,   YA-O
/* 0x343: */ 0, 0, 0, // U+1179..U+117B-->[24..26]  YA-YO,  EO-O,   EU-U
/* 0x34C: */ 0, 0, 0, // U+117C..U+117E-->[27..29]  EO-EU, YEO-O,  YEO-U
/* 0x355: */ 0, 1, 1, // U+117F..U+1181-->[30..32]   O-EO,   O-E,    O-YE,
/* 0x35E: */ 0, 0, 0, // U+1182..U+1184-->[33..35]   O-O,    O-U,   YO-YA,
/* 0x367: */ 1, 0, 0, // U+1185..U+1187-->[36..38]  YO-YAE, YO-YEO, YO-O,
/* 0x370: */ 0, 0, 1, // U+1188..U+118A-->[39..41]  YO-I,    U-A,    U-AE,
/* 0x379: */ 0, 1, 0, // U+118B..U+118D-->[42..44] U-EO-EU,  U-YE,   U-U,
/* 0x382: */ 0, 0, 1, // U+118E..U+1190-->[45..47]  YU-A,   YU-EO,  YU-E,
/* 0x38B: */ 0, 1, 0, // U+1191..U+1193-->[48..50]  YU-YEO, YU-YE,  YU-U,
/* 0x394: */ 0, 0, 0, // U+1194..U+1196-->[51..53]  YU-I,   EU-U,   EU-EU,
/* 0x39D: */ 0, 0, 0, // U+1197..U+1199-->[54..56]  YI-U,    I-A,    I-YA,
/* 0x3A6: */ 0, 0, 0, // U+119A..U+119C-->[57..59]   I-O,    I-U,    I-EU,
/* 0x3AF: */ 0, 0, 0, // U+119D..U+119F-->[60..62] I-ARAEA, ARAEA, ARAEA-EO,
/* 0x3B8: */ 0, 0, 0, // U+11A0..U+11A2-->[63..65] ARAEA-U, ARAEA-I,SSANGARAEA,
/* 0x3C1: */ 0, 0, 0, // U+11A3..U+11A5-->[66..68]   A-EU,  YA-U,  YEO-YA,
/* 0x3CA: */ 0, 1,    // U+11A6..U+11A7-->[69..70]   O-YA,   O-YAE
#ifdef EXTENDED_HANGUL
/* 0x3D0: */ 0, 0, 0, // U+D7B0..U+D7B2-->[71..73]   O-YEO,  O-O-I, YO-A,
/* 0x3D9: */ 1, 0, 0, // U+D7B3..U+D7B5-->[74..76]  YO-AE,  YO-EO,   U-YEO,
/* 0x3E2: */ 1, 1, 0, // U+D7B6..U+D7B8-->[77..79]   U-I-I, YU-AE,  YU-O,
/* 0x3EB: */ 0, 0, 1, // U+D7B9..U+D7BB-->[80..82]  EU-A,   EU-EO,  EU-E,
/* 0x3F4: */ 0, 0, 1, // U+D7BC..U+D7BE-->[83..85]  EU-O,    I-YA-O, I-YAE,
/* 0x3FD: */ 0, 1, 0, // U+D7BF..U+D7C1-->[86..88]   I-YEO,  I-YE,   I-O-I,
/* 0x406: */ 0, 0, 1, // U+D7C2..U+D7C4-->[89..91]   I-YO,   I-YU,   I-I,
/* 0x40F: */ 0, 1,    // U+D7C5..U+D7C6-->[92..93] ARAEA-A, ARAEA-E,
/* 0x415: */ -1       // Mark end of list of vowels.
#else
/* 0x310: */ -1       // Mark end of list of vowels.
#endif
   };


   if (vowel >= 0 && vowel < TOTAL_JUNG) {
      retval = wide_vowel [vowel];
   }
   else {
      retval = 0;
   }


   return retval;
}
```

Here is the caller graph for this function:



5.1.2.13 jong_variation()

```
int jong_variation (
                    int choseong,
                    int jungseong,
                    int jongseong )   [inline]
```

Return the Johab 6/3/1 jongseong variation.

There is only one jongseong variation, so this function always returns 0. It is a placeholder function for possible future adaptation to other johab encodings.

Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable.

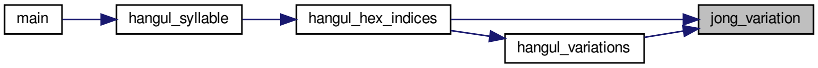
Returns

The jongseong variation, always 0.

Definition at line 558 of file unihangul-support.c.

```
558                                                    {
559
560    return 0;  /* There is only one Jongseong variation. */
561 }
```

Here is the caller graph for this function:



5.1.2.14 jung_variation()

```
int jung_variation (
                    int choseong,
                    int jungseong,
                    int jongseong )   [inline]
```

Return the Johab 6/3/1 jungseong variation.

This function takes the two or three (if jongseong is included) letters that comprise a syllable and determine the variation of the vowel (jungseong).

Each jungseong has 3 variations:

Variation Occurrence

0 Jungseong with only chungseong (no jungseong). 1 Jungseong with chungseong and jungseong (except nieun). 2 Jungseong with chungseong and jungseong nieun.

Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable.

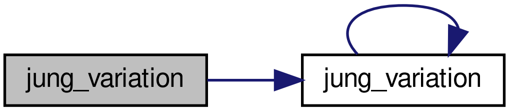
Returns

The jungseong variation, 0 to 2.

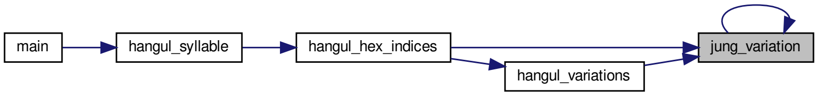
Definition at line 524 of file unihangul-support.c.

```
524                                                    {
525    int jung_variation;  /* Return value */
526
527    if (jungseong < 0) {
528       jung_variation = -1;
529    }
530    else {
531       jung_variation = 0;
532       if (jongseong >= 0) {
533          if (jongseong == 3)
534             jung_variation = 2;  /* Vowel for final Nieun. */
535          else
536             jung_variation = 1;
537       }
538    }
539
540
541    return jung_variation;
542 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.15 one_jamo()

void one_jamo (
unsigned glyph_table[MAX_GLYPHS][16],
unsigned jamo,
unsigned ∗ jamo_glyph)

Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.

Parameters

in	glyph_table	The collection of all jamo glyphs.

Parameters

in	jamo	The Unicode code point, 0 or 0x1100..0x115F.
out	jamo_glyph	The output glyph, 16 columns in each of 16 rows.

Definition at line 717 of file unihangul-support.c.

```
                                      {

   int i;  /* Loop variable */
   int glyph_index;  /* Location of glyph in "hangul-base.hex" array */


   /* If jamo is invalid range, use blank glyph, */
   if (jamo >= 0x1100 && jamo <= 0x11FF) {
      glyph_index = jamo - 0x1100 + JAMO_HEX;
   }
   else if (jamo >= 0xA960 && jamo <= 0xA97F) {
      glyph_index = jamo - 0xA960 + JAMO_EXTA_HEX;
   }
   else if (jamo >= 0xD7B0 && jamo <= 0xD7FF) {
      glyph_index = jamo - 0x1100 + JAMO_EXTB_HEX;
   }
   else {
      glyph_index = 0;
   }

   for (i = 0; i < 16; i++) {
      jamo_glyph [i] = glyph_table [glyph_index] [i];
   }

   return;
}
```

5.1.2.16 print_glyph_hex()

void print_glyph_hex (
FILE * fp,
unsigned codept,
unsigned * this_glyph)

Print one glyph in Unifont hexdraw hexadecimal string style.

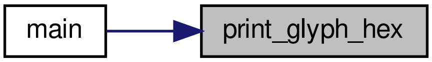
Parameters

in	fp	The file pointer for output.
in	codept	The Unicode code point to print with the glyph.
in	this_glyph	The 16-row by 16-column glyph to print.

Definition at line 692 of file unihangul-support.c.

```
                                                  {

   int i;


   fprintf (fp, "%04X:", codept);

   /* for each this_glyph row */
   for (i = 0; i < 16; i++) {
      fprintf (fp, "%04X", this_glyph[i]);
   }
   fputc ('\n', fp);

   return;
}
```

Here is the caller graph for this function:



5.1.2.17 print_glyph_txt()

```
void print_glyph_txt (
                FILE ∗ fp,
                unsigned codept,
                unsigned ∗ this_glyph )
```

Print one glyph in Unifont hexdraw plain text style.

Parameters

in	fp	The file pointer for output.
in	codept	The Unicode code point to print with the glyph.
in	this_glyph	The 16-row by 16-column glyph to print.

Definition at line 656 of file unihangul-support.c.

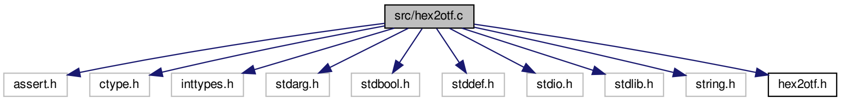
```
656                                                   {
657    int i;
658    unsigned mask;
659
660
661    fprintf (fp, ”%04X:”, codept);
662
663    /* for each this_glyph row */
664    for (i = 0; i < 16; i++) {
665       mask = 0x8000;
666       fputc (’\t’, fp);
667       while (mask != 0x0000) {
668          if (mask & this_glyph [i]) {
669             fputc (’#’, fp);
670          }
671          else {
672             fputc (’-’, fp);
673          }
674          mask »= 1;  /* shift to next bit in this_glyph row */
675       }
676       fputc (’\n’, fp);
677    }
678    fputc (’\n’, fp);
679
680    return;
681 }
```

5.2 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.

5.2.1 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)

5.2.2 Macro Definition Documentation

5.2.2.1 addByte

```
#define addByte(
                  shift )
```

Value:

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

5.2.2.2 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.

5.2.3 Typedef Documentation

5.2.3.1 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.

5.2.3.2 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.

5.2.3.3 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.

5.2.3.4 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.

5.2.4 Enumeration Type Documentation

5.2.4.1 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.

```
1136             {
1137     OP_CLOSE,    ///< Close the current contour path that was being drawn.
1138     OP_POINT     ///< Add one more (x,y) point to the contor being drawn.
1139 };
```

5.2.4.2 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.

```
1144            {
1145     FILL_LEFT,    ///< Draw outline counter-clockwise (CFF, PostScript).
1146     FILL_RIGHT    ///< Draw outline clockwise (TrueType).
1147 };
```

5.2.4.3 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.

```
658            {
659    LOCA_OFFSET16 = 0,    ///< Offset to location is a 16-bit Offset16 value
660    LOCA_OFFSET32 = 1     ///< Offset to location is a 32-bit Offset32 value
661 };
```

5.2.5 Function Documentation

5.2.5.1 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.

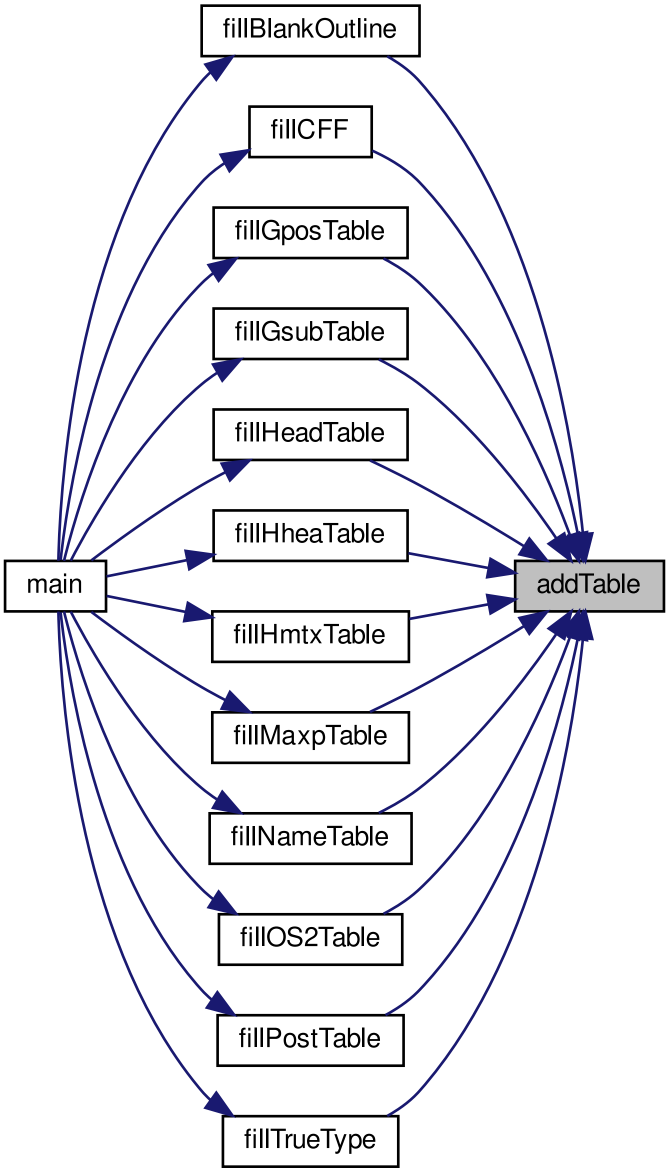
Parameters

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



5.2.5.2 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.

```
{
    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
}
```

5.2.5.3 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;
}
```

5.2.5.4 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;
}
```

5.2.5.5 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;
}
```

5.2.5.6 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.

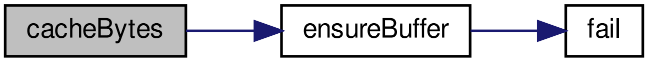
Parameters

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.

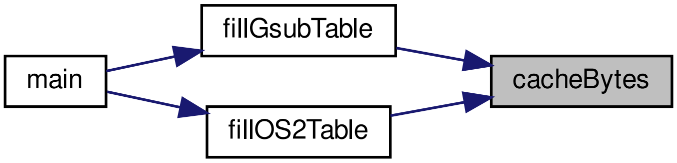
Definition at line 509 of file hex2otf.c.

```
510 {
511     ensureBuffer (buf, count);
512     memcpy (buf->next, src, count);
513     buf->next += count;
514 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.5.7 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

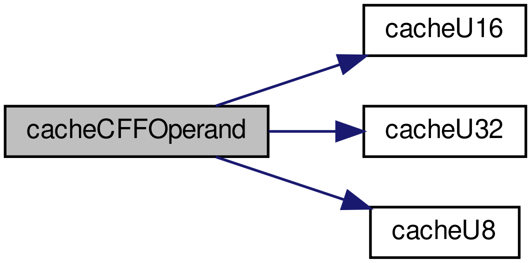
Parameters

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:



5.2.5.8 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.

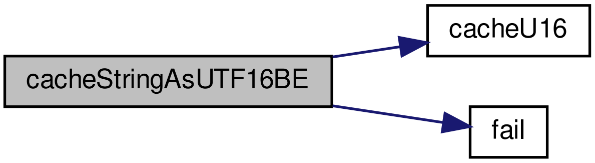
Parameters

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:



5.2.5.9 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.

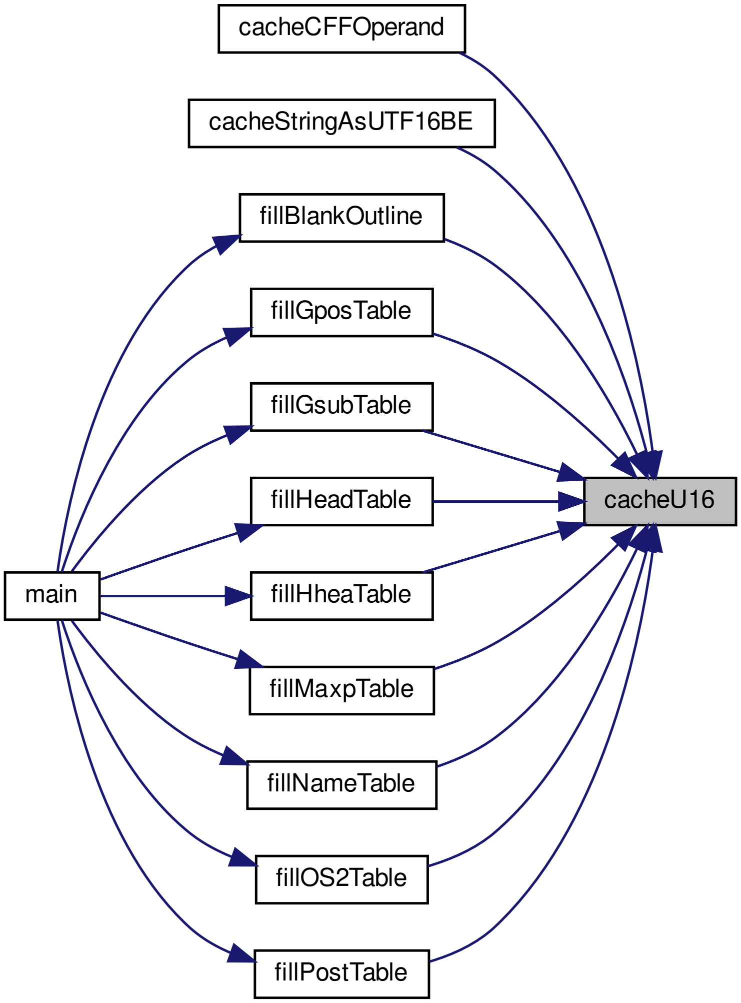
Parameters

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.

```
413 {
414     cacheU (buf, value, 2);
415 }
```

Here is the caller graph for this function:



5.2.5.10 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.

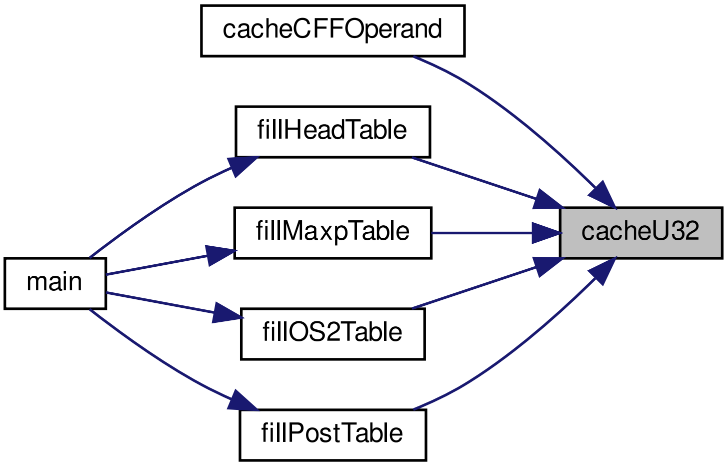
Parameters

in	value	The 32-bit unsigned value to append to the buf array.

Definition at line 427 of file hex2otf.c.

```
428 {
429     cacheU (buf, value, 4);
430 }
```

Here is the caller graph for this function:



5.2.5.11 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.

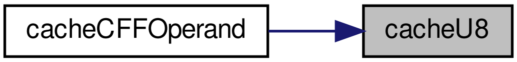
Parameters

in,out	buf	The array of bytes to which to append a new byte.
in	value	The 8-bit unsigned value to append to the buf array.

Definition at line 397 of file hex2otf.c.

```
398 {
399     storeU8 (buf, value & 0xff);
400 }
```

Here is the caller graph for this function:



5.2.5.12 cacheZeros()

void cacheZeros (
    Buffer ∗ buf,
    size_t count)

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

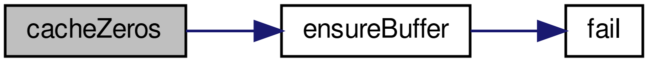
Parameters

in,out	buf	The buffer to which the operand value is appended.
in	count	The number of bytes containing zeroes to append.

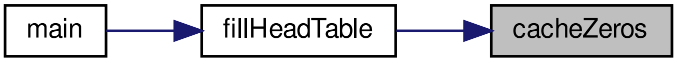
Definition at line 491 of file hex2otf.c.

```
492 {
493     ensureBuffer (buf, count);
494     memset (buf->next, 0, count);
495     buf->next += count;
496 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.5.13 cleanBuffers()

void cleanBuffers ( )

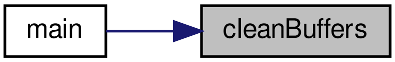
Free all allocated buffer pointers.

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:



5.2.5.14 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.

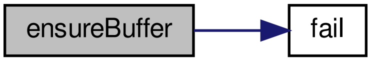
Parameters

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

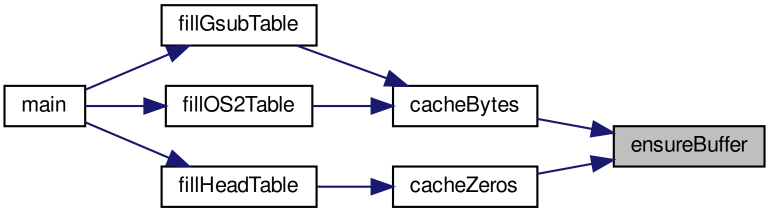
Definition at line 239 of file hex2otf.c.

```
{
    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:



5.2.5.15 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.

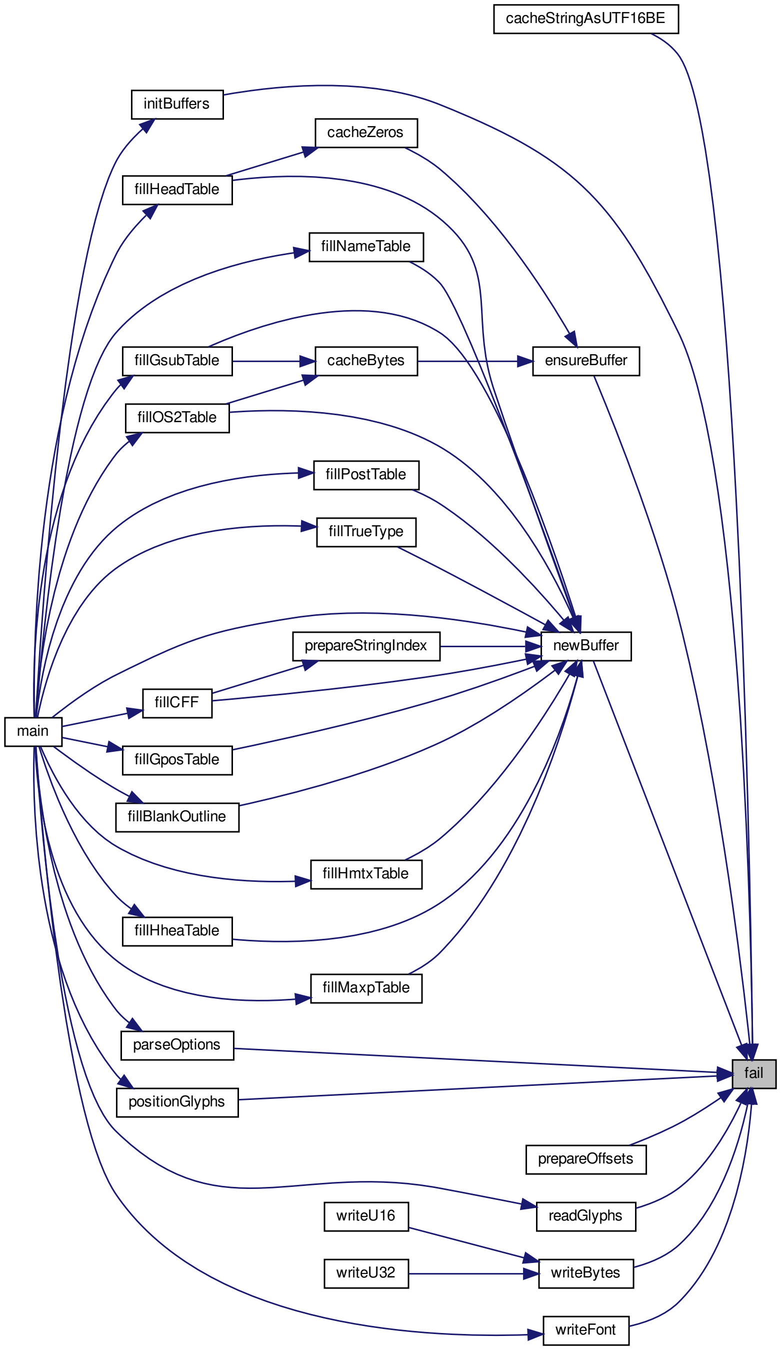
Parameters

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

Definition at line 113 of file hex2otf.c.

```
{
    fputs ("ERROR: ", stderr);
    va_list args;
    va_start (args, reason);
    vfprintf (stderr, reason, args);
    va_end (args);
    putc ('\n', stderr);
    exit (EXIT_FAILURE);
}
```

Here is the caller graph for this function:



5.2.5.16 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.

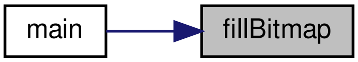
Parameters

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:



5.2.5.17 fillBlankOutline()

void fillBlankOutline (
Font ∗ font)

Create a dummy blank outline in a font table.

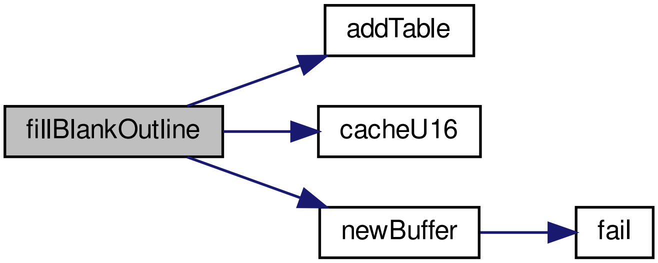
Parameters

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

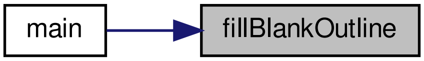
Definition at line 1697 of file hex2otf.c.

```
{
    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:



5.2.5.18 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.

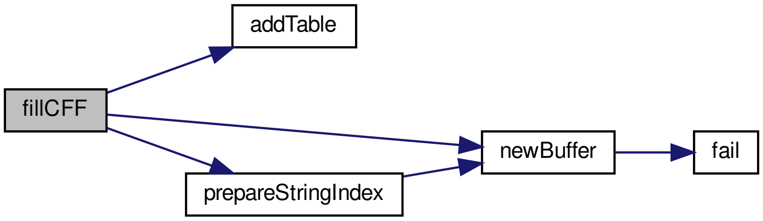
Use fixed width integer for variables to simplify offset calculation.

Definition at line 1329 of file hex2otf.c.

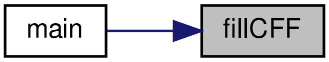
```
{
    // HACK: For convenience, CFF data structures are hard coded.
    assert (0 < version && version <= 2);
    Buffer *cff = newBuffer (65536);
    addTable (font, version == 1 ? ”CFF ” : ”CFF2”, cff);

/// 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:



5.2.5.19 fillCmapTable()

void fillCmapTable (
    Font ∗ font)

Fill a "cmap" font table.

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

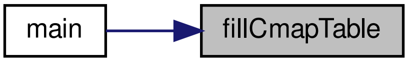
Parameters

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

Definition at line 2109 of file hex2otf.c.

```
{
    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);
}
```

Here is the caller graph for this function:



5.2.5.20 fillGposTable()

void fillGposTable (
Font ∗ font)

Fill a ”GPOS” font table.

The ”GPOS” table contains information for glyph positioning.

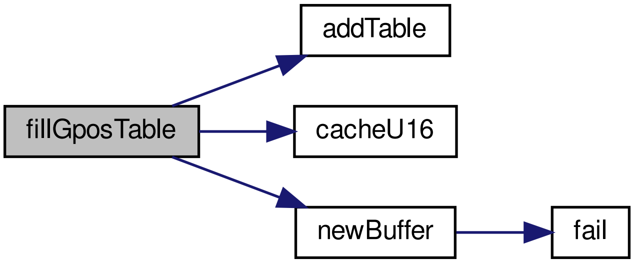
Parameters

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

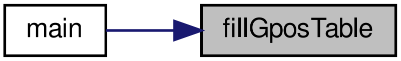
Definition at line 2241 of file hex2otf.c.

```
{
    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
    }
2256     { // Lookup List Table
2257         cacheU16 (gpos, 0); // lookupCount
2258     }
2259 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.5.21 fillGsubTable()

void fillGsubTable (
Font ∗ font)

Fill a ”GSUB” font table.

The ”GSUB” table contains information for glyph substitution.

Parameters

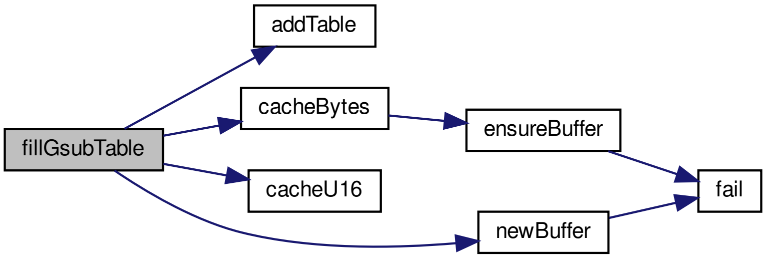
in,out	font	The Font struct to which to add the table.

Definition at line 2269 of file hex2otf.c.

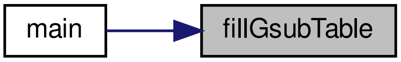
```
2270 {
2271     Buffer *gsub = newBuffer (38);
2272     addTable (font, "GSUB", gsub);
2273     cacheU16 (gsub, 1); // majorVersion
2274     cacheU16 (gsub, 0); // minorVersion
2275     cacheU16 (gsub, 10); // scriptListOffset
2276     cacheU16 (gsub, 34); // featureListOffset
2277     cacheU16 (gsub, 36); // lookupListOffset
2278     { // 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:



5.2.5.22 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.

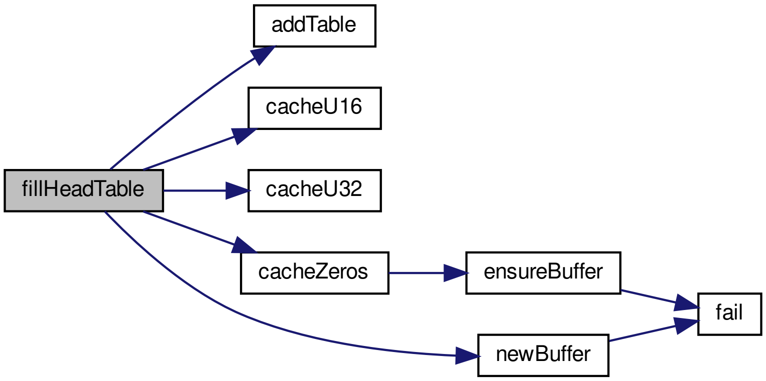
Parameters

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.

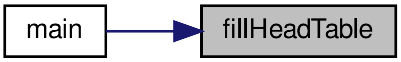
Definition at line 1853 of file hex2otf.c.

```
{
    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:



5.2.5.23 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.

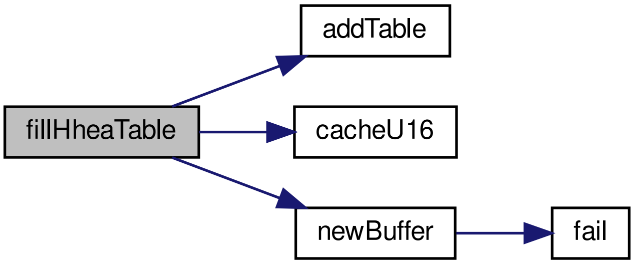
Parameters

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

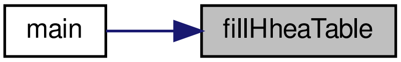
Definition at line 1918 of file hex2otf.c.

```
{
    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:



5.2.5.24 fillHmtxTable()

void fillHmtxTable (
Font ∗ font)

Fill an "hmtx" font table.

The "hmtx" table contains horizontal metrics information.

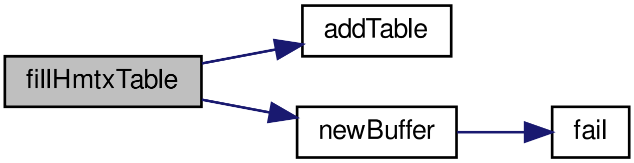
Parameters

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

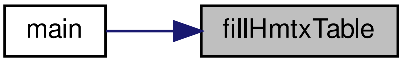
Definition at line 2087 of file hex2otf.c.

```
2088 {
2089     Buffer *hmtx = newBuffer (4 * font->glyphCount);
2090     addTable (font, ”hmtx”, hmtx);
2091     const Glyph *const glyphs = getBufferHead (font->glyphs);
2092     const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
2093     for (const Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
2094     {
2095         int_fast16_t aw = glyph->combining ? 0 : PW (glyph->byteCount);
2096         cacheU16 (hmtx, FU (aw)); // advanceWidth
2097         cacheU16 (hmtx, FU (glyph->lsb)); // lsb
2098     }
2099 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.5.25 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.

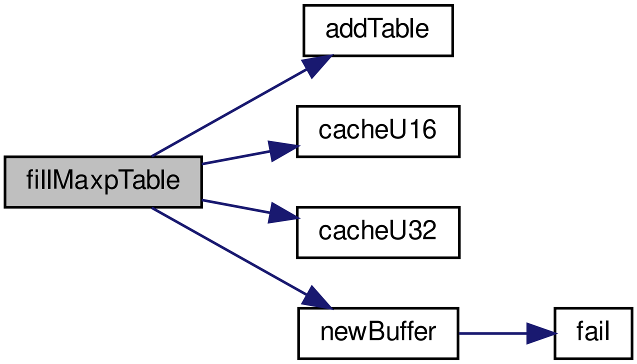
Parameters

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.

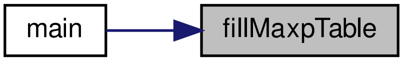
Definition at line 1954 of file hex2otf.c.

```
{
    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:



5.2.5.26 fillNameTable()

void fillNameTable (

Font ∗ font,
NameStrings nameStrings)

Fill a ”name” font table.

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

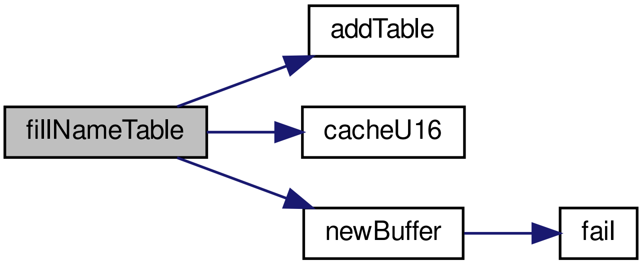
Parameters

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

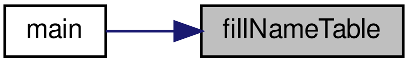
Definition at line 2366 of file hex2otf.c.

```
{
    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:



5.2.5.27 fillOS2Table()

void fillOS2Table (
    Font ∗ font)

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

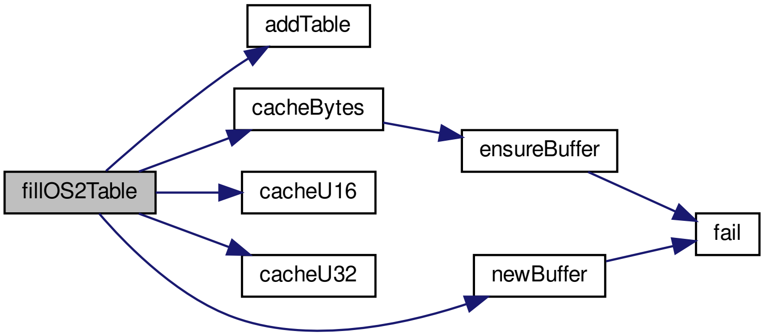
Parameters

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

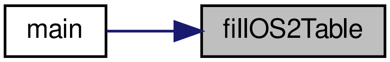
Definition at line 1986 of file hex2otf.c.

```
{
    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
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.5.28 fillPostTable()

void fillPostTable (
Font ∗ font)

Fill a "post" font table.
The "post" table contains information for PostScript printers.

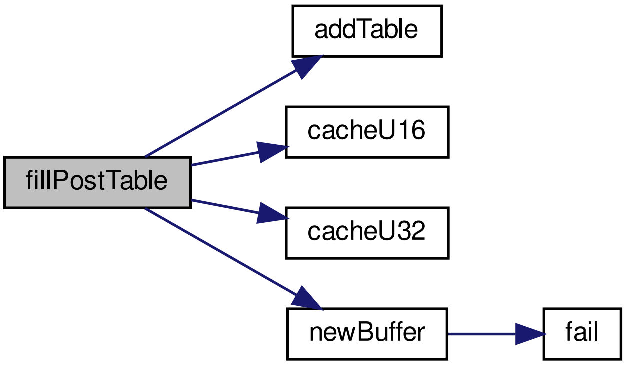
Parameters

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

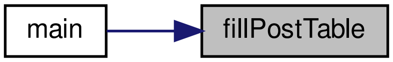
Definition at line 2218 of file hex2otf.c.

```
{
    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:



5.2.5.29 fillTrueType()

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

Add a TrueType table to a font.

Parameters

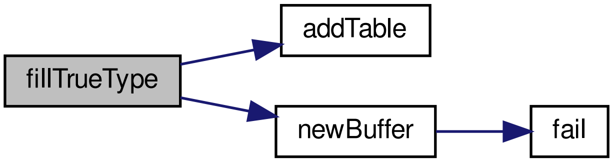
in,out	font	Pointer to a Font struct to contain the TrueType table.
in	format	The TrueType "loca" table format, Offset16 or Offset32.
in	names	List of NameStrings.

Definition at line 1597 of file hex2otf.c.

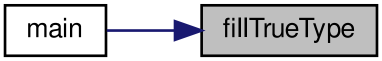
```
1599 {
1600     Buffer *glyf = newBuffer (65536);
```

```
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:



5.2.5.30 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.

```
{
    free (buf->begin);
    buf->capacity = 0;
}
```

5.2.5.31 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.

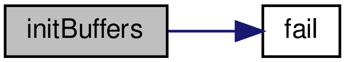
Parameters

in	count	The number of buffer array pointers to allocate.

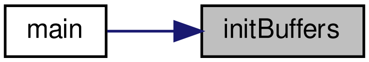
Definition at line 152 of file hex2otf.c.

```
{
    assert (count > 0);
    assert (bufferCount == 0); // uninitialized
    allBuffers = calloc (count, sizeof *allBuffers);
    if (!allBuffers)
        fail ("Failed to initialize buffers.");
    bufferCount = count;
    nextBufferIndex = 0;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.5.32 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.

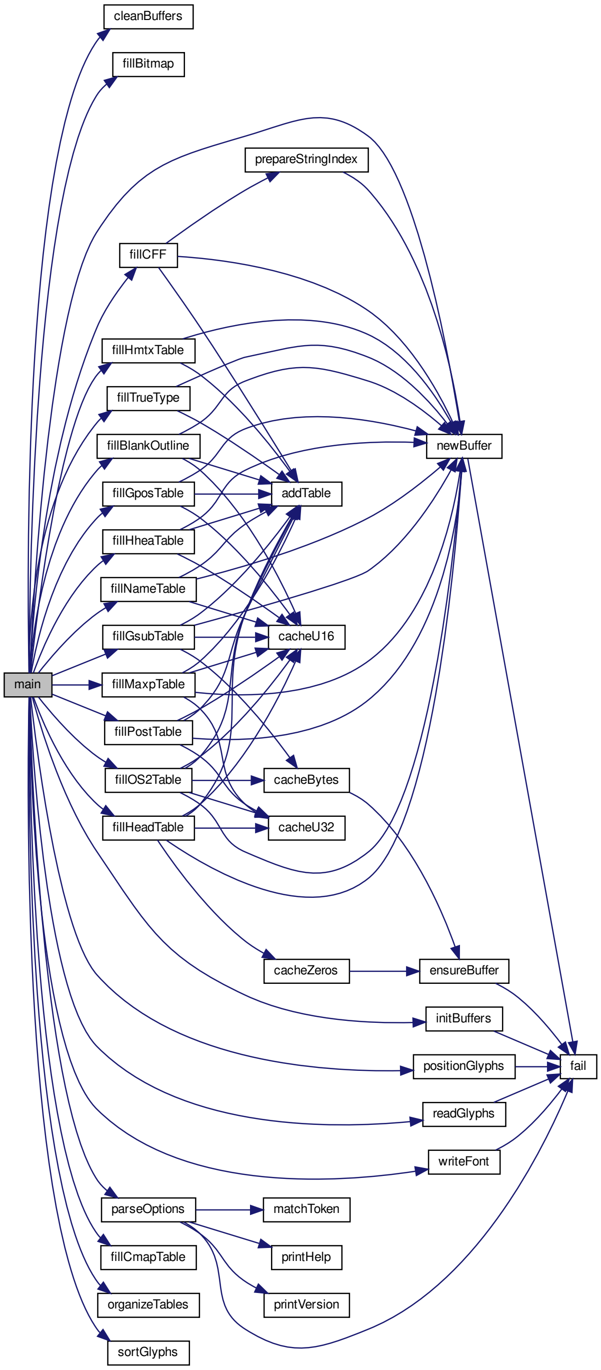
Returns

EXIT_FAILURE upon fatal error, EXIT_SUCCESS otherwise.

Definition at line 2603 of file hex2otf.c.

```
{
    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;
}
```

Here is the call graph for this function:



5.2.5.33 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.

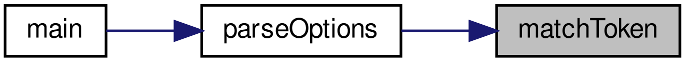
Returns

Pointer to the first character of the desired option.

Definition at line 2470 of file hex2otf.c.

```
{
    while (*key)
        if (*operand++ != *key++)
            return NULL;
    if (!*operand || *operand++ == delimiter)
        return operand;
    return NULL;
}
```

Here is the caller graph for this function:



5.2.5.34 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.

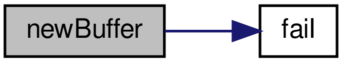
Parameters

in	initialCapacity	The initial number of elements in the buffer.

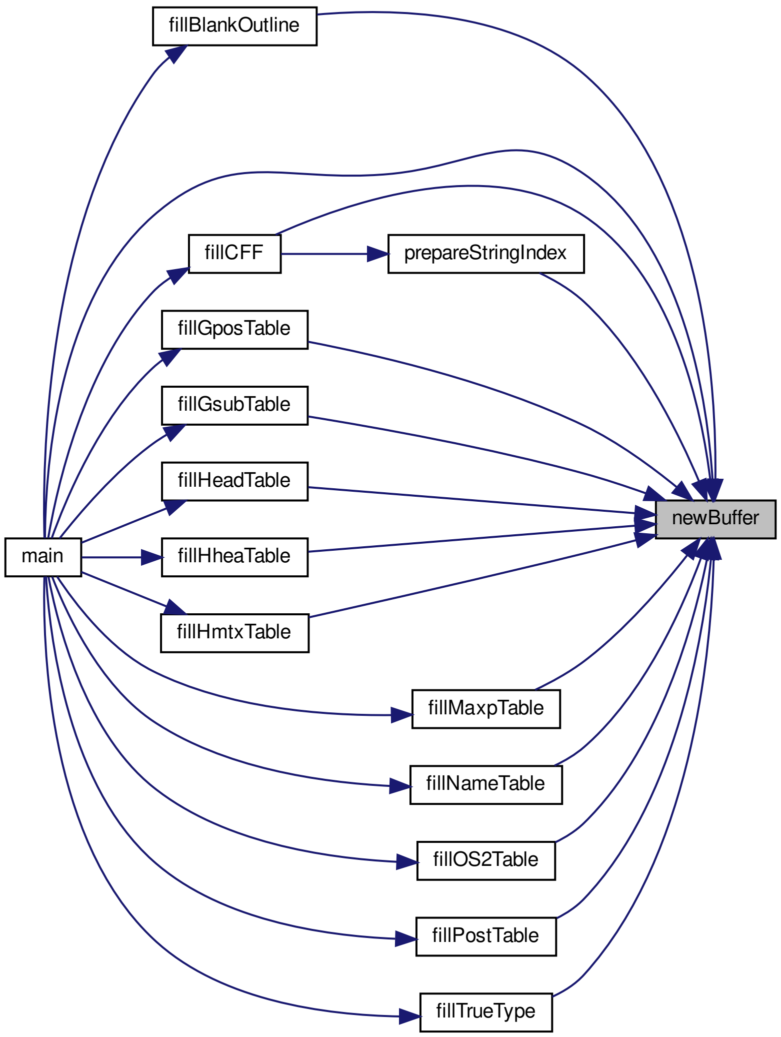
Definition at line 188 of file hex2otf.c.

```
{
    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:



5.2.5.35 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.

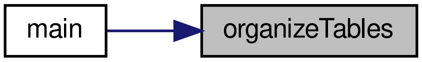
Parameters

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:



5.2.5.36 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.

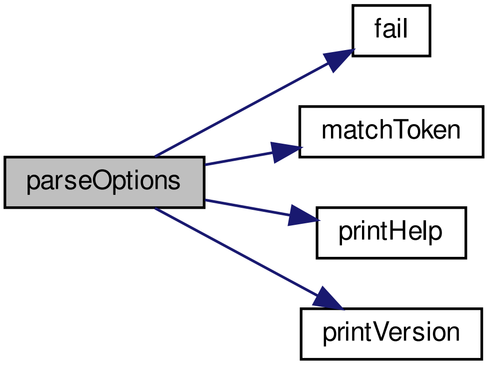
Returns

Data structure to hold requested command line options.

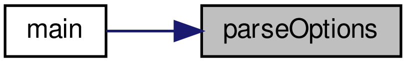
Definition at line 2500 of file hex2otf.c.

```
{
    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:



5.2.5.37 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.

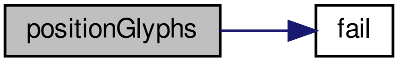
Parameters

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).

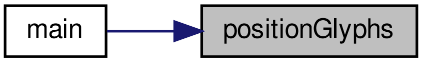
Definition at line 1061 of file hex2otf.c.

```
{
    *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);
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.5.38 prepareOffsets()

void prepareOffsets (
size_t ∗ sizes)

Prepare 32-bit glyph offsets in a font table.

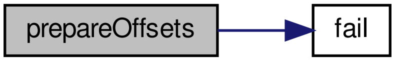
Parameters

in	sizes	Array of glyph sizes, for offset calculations.

Definition at line 1275 of file hex2otf.c.

```
1276 {
1277     size_t *p = sizes;
1278     for (size_t *i = sizes + 1; *i; i++)
1279         *i += *p++;
1280     if (*p > 2147483647U) // offset not representable
1281         fail (”CFF table is too large.”);
1282 }
```

Here is the call graph for this function:



5.2.5.39 prepareStringIndex()

Buffer∗ prepareStringIndex (
const NameStrings names)

Prepare a font name string index.

Parameters

in	names	List of name strings.

Returns

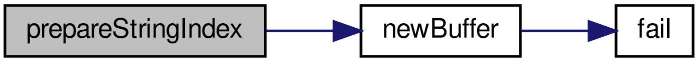
Pointer to a Buffer struct containing the string names.

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

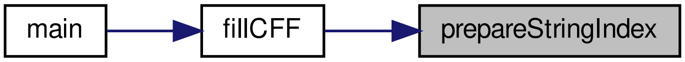
Definition at line 1291 of file hex2otf.c.

```
{
    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:



5.2.5.40 printHelp()

void printHelp ( )

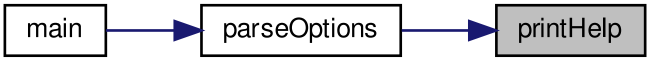
Print help message to stdout and then exit.

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

Definition at line 2426 of file hex2otf.c.

```
2426            {
2427    printf (”Synopsis: hex2otf <options>:\n\n”);
2428    printf (”    hex=<filename>         Specify Unifont .hex input file.\n”);
2429    printf (”    pos=<filename>         Specify combining file. (Optional)\n”);
2430    printf (”    out=<filename>         Specify output font file.\n”);
2431    printf (”    format=<f1>,<f2>,...   Specify font format(s); values:\n”);
2432    printf (”                           cff\n”);
2433    printf (”                           cff2\n”);
2434    printf (”                           truetype\n”);
2435    printf (”                           blank\n”);
2436    printf (”                           bitmap\n”);
2437    printf (”                           gpos\n”);
2438    printf (”                           gsub\n”);
2439    printf (”\nExample:\n\n”);
2440    printf (”    hex2otf hex=Myfont.hex out=Myfont.otf format=cff\n\n”);
2441    printf (”For more information, consult the hex2otf(1) man page.\n\n”);
2442
2443    exit (EXIT_SUCCESS);
2444 }
```

Here is the caller graph for this function:



5.2.5.41 printVersion()

void printVersion ( )

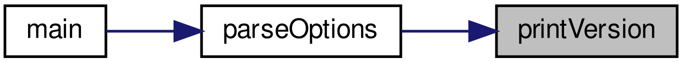
Print program version string on stdout.

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

Definition at line 2407 of file hex2otf.c.

```
2407               {
2408    printf (”hex2otf (GNU Unifont) %s\n”, VERSION);
2409    printf (”Copyright \u00A9 2022 \u4F55\u5FD7\u7FD4 (He Zhixiang)\n”);
2410    printf (”License GPLv2+: GNU GPL version 2 or later\n”);
2411    printf (”<https://gnu.org/licenses/gpl.html>\n”);
2412    printf (”This is free software: you are free to change and\n”);
2413    printf (”redistribute it.  There is NO WARRANTY, to the extent\n”);
2414    printf (”permitted by law.\n”);
2415
2416    exit (EXIT_SUCCESS);
2417 }
```

Here is the caller graph for this function:



5.2.5.42 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);
    }
}
```

5.2.5.43 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.

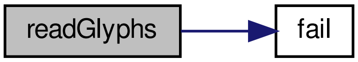
Parameters

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

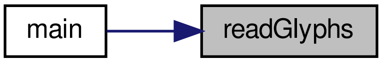
Definition at line 966 of file hex2otf.c.

```
{
    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:



5.2.5.44 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.

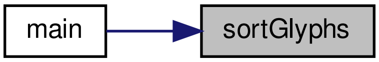
Parameters

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:



5.2.5.45 writeBytes()

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

Write an array of bytes to an output file.

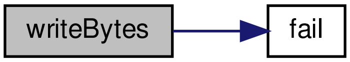
Parameters

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

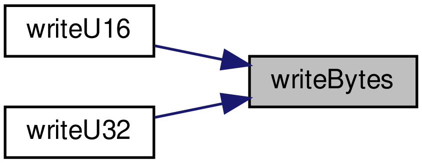
Definition at line 538 of file hex2otf.c.

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.5.46 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”.

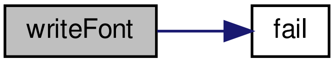
Parameters

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.

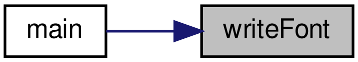
Add a byte shifted by 24, 16, 8, or 0 bits.
Definition at line 786 of file hex2otf.c.

```
{
    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:



5.2.5.47 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.

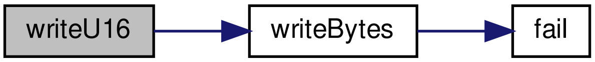
Parameters

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:



5.2.5.48 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.

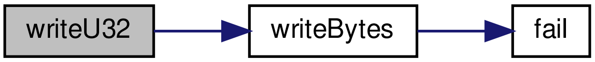
Parameters

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);
}
```

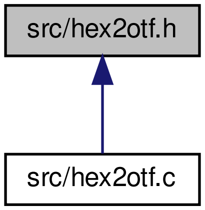
Here is the call graph for this function:



5.3 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 ”15.0.06”

  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)

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.

5.3.3 Variable Documentation

5.3.3.1 defaultNames

const NamePair defaultNames[]
Initial value:

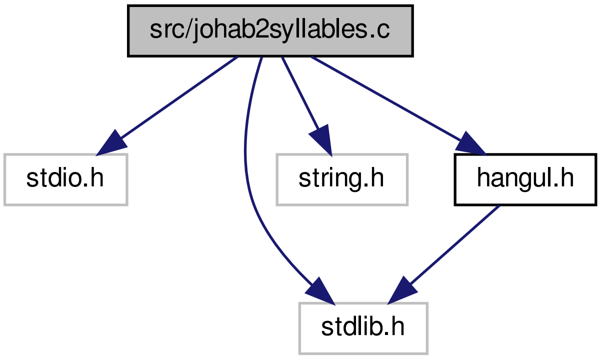
```
=
{
   NAMEPAIR (0),
   NAMEPAIR (1),
   NAMEPAIR (2),
   NAMEPAIR (5),
   NAMEPAIR (11),
   NAMEPAIR (13),
   NAMEPAIR (14),
   {0, NULL}
}
```

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 src/johab2syllables.c File Reference

Create the Unicode Hangul Syllables block from component letters.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hangul.h"
Include dependency graph for johab2syllables.c:



Functions

- int main (int argc, char ∗argv[])

  The main function.
- void print_help ()

  Print a help message.

5.4.1 Detailed Description

Create the Unicode Hangul Syllables block from component letters.
This program reads in a "hangul-base.hex" file containing Hangul letters in Johab 6/3/1 format and outputs a Unifont .hex format file covering the Unicode Hangul Syllables range of U+AC00..U+D7A3.

Author

Paul Hardy

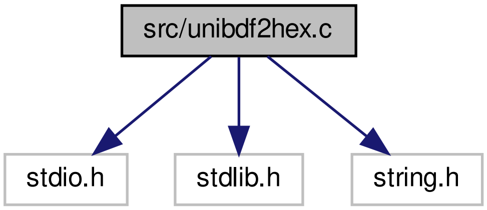
Copyright

Copyright © 2023 Paul Hardy

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”.

5.5.2 Function Documentation

5.5.2.1 main()

int main ( )

The main function.

Returns

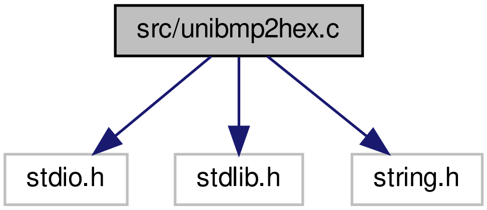
Exit status is always 0 (successful termination).

Definition at line 46 of file unibdf2hex.c.

```
47 {
48    int i;
49    int digitsout;  /* how many hex digits we output in a bitmap */
50    int thispoint;
51    char inbuf[MAXBUF];
52    int bbxx, bbxy, bbxxoff, bbxyoff;
53 
54    int descent=4; /* font descent wrt baseline */
55    int startrow;  /* row to start glyph         */
56    unsigned rowout;
57 
58    while (fgets (inbuf, MAXBUF - 1, stdin) != NULL) {
59       if (strncmp (inbuf, ”ENCODING ”, 9) == 0) {
60          sscanf (&inbuf[9], ”%d”, &thispoint); /* get code point */
61          /*
62 If we want this code point, get the BBX (bounding box) and
63 BITMAP information.
64 */
65          if ((thispoint >= 0x2E80 && thispoint <= 0x2EFF) || // CJK Radicals Supplement
66              (thispoint >= 0x2F00 && thispoint <= 0x2FDF) || // Kangxi Radicals
67              (thispoint >= 0x2FF0 && thispoint <= 0x2FFF) || // Ideographic Description Characters
68              (thispoint >= 0x3001 && thispoint <= 0x303F) || // CJK Symbols and Punctuation (U+3000 is a space)
69              (thispoint >= 0x3100 && thispoint <= 0x312F) || // Bopomofo
70              (thispoint >= 0x31A0 && thispoint <= 0x31BF) || // Bopomofo extend
71              (thispoint >= 0x31C0 && thispoint <= 0x31EF) || // CJK Strokes
72              (thispoint >= 0x3400 && thispoint <= 0x4DBF) || // CJK Unified Ideographs Extension A
73              (thispoint >= 0x4E00 && thispoint <= 0x9FCF) || // CJK Unified Ideographs
74              (thispoint >= 0xF900 && thispoint <= 0xFAFF))   // CJK Compatibility Ideographs
75             {
76             while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
77                    strncmp (inbuf, ”BBX ”, 4) != 0); /* find bounding box */
78 
79             sscanf (&inbuf[4], ”%d %d %d %d”, &bbxx, &bbxy, &bbxxoff, &bbxyoff);
80             while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
81                    strncmp (inbuf, ”BITMAP”, 6) != 0); /* find bitmap start */
82             fprintf (stdout, ”%04X:”, thispoint);
83             digitsout = 0;
84             /* Print initial blank rows */
85             startrow = descent + bbxyoff + bbxy;
86 
87             /* Force everything to 16 pixels wide */
88             for (i = 16; i > startrow; i--) {
89                fprintf (stdout,”0000”);
90                digitsout += 4;
91             }
92             while (fgets (inbuf, MAXBUF - 1, stdin) != NULL &&
93                    strncmp (inbuf, ”END”, 3) != 0) { /* copy bitmap until END */
94                sscanf (inbuf, ”%X”, &rowout);
95                /* Now force glyph to a 16x16 grid even if they’d fit in 8x16 */
96                if (bbxx <= 8) rowout «= 8;  /* shift left for 16x16 glyph */
97                rowout »= bbxxoff;
98                fprintf (stdout, ”%04X”, rowout);
99                digitsout += 4;
100             }
101 
102             /* Pad for 16x16 glyph */
103             while (digitsout < 64) {
104                fprintf (stdout,”0000”);
105                digitsout += 4;
106             }
107             fprintf (stdout,”\n”);
108          }
109       }
110    }
111    exit (0);
112 }
```

5.6 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]

5.6.1 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]

5.6.2 Function Documentation

5.6.2.1 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.

```
150 {
151
152   int i, j, k;              /* loop variables                         */
153   unsigned char inchar;       /* temporary input character */
154   char header[MAXBUF];        /* input buffer for bitmap file header */
155   int wbmp=0; /* =0 for Windows Bitmap (.bmp); 1 for Wireless Bitmap (.wbmp) */
156   int fatal; /* =1 if a fatal error occurred */
157   int match; /* =1 if we're still matching a pattern, 0 if no match */
158   int empty1, empty2; /* =1 if bytes tested are all zeroes */
159   unsigned char thischar1[16], thischar2[16]; /* bytes of hex char */
160   unsigned char thischar0[16], thischar3[16]; /* bytes for quadruple-width */
161   int thisrow; /* index to point into thischar1[] and thischar2[] */
162   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);
}
```

5.6.3 Variable Documentation

5.6.3.1

struct { ... } bmp_header
Bitmap Header parameters

5.6.3.2 color_table

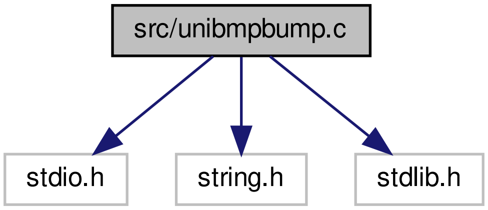
unsigned char color_table[256][4]
Bitmap Color Table – maximum of 256 colors in a BMP file
Definition at line 137 of file unibmp2hex.c.

5.6.3.3 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 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.

5.7.1 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

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]

5.7.2 Function Documentation

5.7.2.1 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.

```
487                                     {
488    int i;
489    unsigned char inchar[4];
490    unsigned inword;
491
492    for (i = 0; i < nbytes; i++) {
493       if (fread (&inchar[i], 1, 1, infp) != 1) {
494          inchar[i] = 0;
495       }
496    }
497    for (i = nbytes; i < 4; i++) inchar[i] = 0;
498
499    inword = ((inchar[3] & 0xFF) « 24) | ((inchar[2] & 0xFF) « 16) |
500             ((inchar[1] & 0xFF) «  8) |  (inchar[0] & 0xFF);
501
502    return inword;
503 }
```

5.7.2.2 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.

```
50                                 {
51
52    /*
53 Values preserved from file header (first 14 bytes).
54 */
55    char file_format[3];        /* "BM" for original Windows format             */
56    unsigned filesize;          /* size of file in bytes                        */
57    unsigned char rsvd_hdr[4]; /* 4 reserved bytes                              */
58    unsigned image_start;       /* byte offset of image in file                  */
59
60    /*
61 Values preserved from Device Independent Bitmap (DIB) Header.
62
63 The DIB fields below are in the standard 40-byte header.  Version
64 4 and version 5 headers have more information, mainly for color
65 information.  That is skipped over, because a valid glyph image
66 is just monochrome.
67 */
68    int dib_length;             /* in bytes, for parsing by header version    */
69    int image_width = 0;        /* Signed image width                          */
70    int image_height = 0;       /* Signed image height                         */
71    int num_planes = 0;         /* number of planes; must be 1                  */
72    int bits_per_pixel = 0;     /* for palletized color maps (< 2^16 colors)  */
73    /*
74 The following fields are not in the original spec, so initialize
75 them to 0 so we can correctly parse an original file format.
76 */
77    int compression_method=0;  /* 0 --> uncompressed RGB/monochrome           */
78    int image_size = 0;         /* 0 is a valid size if no compression         */
79    int hres = 0;               /* image horizontal resolution                 */
80    int vres = 0;               /* image vertical resolution                   */
81    int num_colors = 0;         /* Number of colors for pallettized images    */
82    int important_colors = 0;  /* Number of significant colors (0 or 2)       */
83
84    int true_colors = 0;        /* interpret num_colors, which can equal 0    */
85
86    /*
87 Color map.  This should be a monochrome file, so only two
88 colors are stored.
89 */
90    unsigned char color_map[2][4]; /* two of R, G, B, and possibly alpha  */
91
92    /*
93 The monochrome image bitmap, stored as a vector 544 rows by
94 72*8 columns.
95 */
96    unsigned image_bytes[544*72];
97
98    /*
99 Flags for conversion & I/O.
100 */
101    int verbose       = 0;       /* Whether to print file info on stderr      */
102    unsigned image_xor = 0x00; /* Invert (= 0xFF) if color 0 is not black  */
103
104    /*
105 Temporary variables.
106 */
```

```
   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);
}
```

5.7.2.3 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.

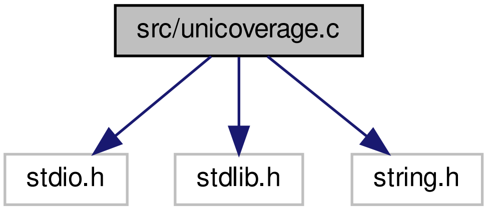
```
514                                {
515    int i, j, k;  /* loop variables */
516    int offset;
517    unsigned glyph_row; /* one grid row of 32 pixels */
518    unsigned last_pixel; /* last pixel in a byte, to preserve */
519
520    /* To insert "00" after "U+" at top of image */
521    char zero_pattern[16] = {
522        0x00, 0x00, 0x00, 0x00, 0x18, 0x24, 0x42, 0x42,
523        0x42, 0x42, 0x42, 0x42, 0x24, 0x18, 0x00, 0x00
524    };
525
526    /* 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;
}
```

5.8 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.

5.8.1 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.

5.8.2 Function Documentation

5.8.2.1 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

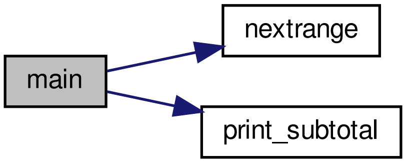
This program exits with status 0.

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:



5.8.2.2 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.
in	cend	Ending code point in current Unicode script range.
out	coverstring	String containing <cstart>-<cend> substring.

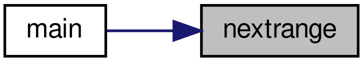
Returns

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:



5.8.2.3 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.

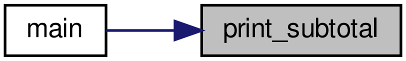
Parameters

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.
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:

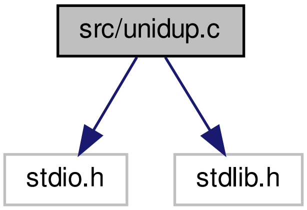


5.9 src/unidup.c File Reference

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.

5.9.1 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!]

5.9.2 Function Documentation

5.9.2.1 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.

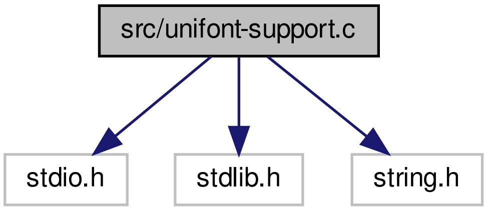
```
{

   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);
}
```

5.10 src/unifont-support.c File Reference

: Support functions for Unifont .hex files.

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

Include dependency graph for unifont-support.c:



Functions

- void parse_hex (char ∗hexstring, int ∗width, unsigned ∗codept, unsigned char glyph[16][2])

  Decode a Unifont .hex file into Uniocde code point and glyph.
- void glyph2bits (int width, unsigned char glyph[16][2], unsigned char glyphbits[16][16])

  Convert a Unifont binary glyph into a binary glyph array of bits.
- void hexpose (int width, unsigned char glyphbits[16][16], unsigned char transpose[2][16])

  Transpose a Unifont .hex format glyph into 2 column-major sub-arrays.
- void glyph2string (int width, unsigned codept, unsigned char glyph[16][2], char ∗outstring)

  Convert a glyph code point and byte array into a Unifont .hex string.
- void xglyph2string (int width, unsigned codept, unsigned char transpose[2][16], char ∗outstring)

  Convert a code point and transposed glyph into a Unifont .hex string.

5.10.1 Detailed Description

: Support functions for Unifont .hex files.

Author

Paul Hardy

Copyright

Copyright © 2023 Paul Hardy

5.10.2 Function Documentation

5.10.2.1 glyph2bits()

```
void glyph2bits (
                int width,
                unsigned char glyph[16][2],
                unsigned char glyphbits[16][16] )
```

Convert a Unifont binary glyph into a binary glyph array of bits.

This function takes a Unifont 16-row by 1- or 2-byte wide binary glyph and returns an array of 16 rows by 16 columns. For each output array element, a 1 indicates the corresponding bit was set in the binary glyph, and a 0 indicates the corresponding bit was not set.

Parameters

in	width	The number of columns in the glyph.
in	glyph	The binary glyph, as a 16-row by 2-byte array.
out	glyphbits	The converted glyph, as a 16-row, 16-column array.

Definition at line 91 of file unifont-support.c.

```
                                    {

   unsigned char tmp_byte;
   unsigned char mask;
   int row, column;

   for (row = 0; row < 16; row++) {
      tmp_byte = glyph [row][0];
      mask = 0x80;
      for (column = 0; column < 8; column++) {
            glyphbits [row][column] = tmp__byte & mask ? 1 : 0;
            mask »= 1;
        }

        if (width > 8)
            tmp__byte = glyph [row][1];
        else
            tmp__byte = 0x00;

        mask = 0x80;
        for (column = 8; column < 16; column++) {
            glyphbits [row][column] = tmp__byte & mask ? 1 : 0;
            mask »= 1;
        }
    }


    return;
}
```

5.10.2.2 glyph2string()

void glyph2string (
int width,
unsigned codept,
unsigned char glyph[16][2],
char ∗ outstring)

Convert a glyph code point and byte array into a Unifont .hex string.

This function takes a code point and a 16-row by 1- or 2-byte binary glyph, and converts it into a Unifont .hex format character array.

Parameters

in	width	The number of columns in the glyph.
in	codept	The code point to appear in the output .hex string.
in	glyph	The glyph, with each of 16 rows 1 or 2 bytes wide.
out	outstring	The output string, in Unifont .hex format.

Definition at line 221 of file unifont-support.c.

```
                                {

    int i;            /* index into outstring array */
    int row;

    if (codept <= 0xFFFF) {
        sprintf (outstring, ”%04X:”, codept);
        i = 5;
    }
    else {
        sprintf (outstring, ”%06X:”, codept);
        i = 7;
    }

    for (row = 0; row < 16; row++) {
        sprintf (&outstring[i], ”%02X”, glyph [row][0]);
        i += 2;

        if (width > 8) {
            sprintf (&outstring[i], ”%02X”, glyph [row][1]);
            i += 2;
        }
    }

    outstring[i] = ’\0’;  /* terminate output string */


    return;
}
```

5.10.2.3 hexpose()

```
void hexpose (
        int width,
        unsigned char glyphbits[16][16],
        unsigned char transpose[2][16] )
```

Transpose a Unifont .hex format glyph into 2 column-major sub-arrays.

This function takes a 16-by-16 cell bit array made from a Unifont glyph (as created by the glyph2bits function) and outputs a transposed array of 2 sets of 8 or 16 columns, depending on the glyph width. This format simplifies outputting these bit patterns on a graphics display with a controller chip designed to output a column of 8 pixels at a time.

For a line of text with Unifont output, first all glyphs can have their first 8 rows of pixels displayed on a line. Then the second 8 rows of all glyphs on the line can be displayed. This simplifies code for such controller chips that are designed to automatically increment input bytes of column data by one column at a time for each successive byte.

The glyphbits array contains a '1' in each cell where the corresponding non-transposed glyph has a pixel set, and 0 in each cell where a pixel is not set.

Parameters

in	width	The number of columns in the glyph.
in	glyphbits	The 16-by-16 pixel glyph bits.
out	transpose	The array of 2 sets of 8 ot 16 columns of 8 pixels.

Definition at line 150 of file unifont-support.c.

```
                                  {

   int column;


   for (column = 0; column < 8; column++) {
      transpose [0][column] =
            (glyphbits [ 0][column] « 7) |
            (glyphbits [ 1][column] « 6) |
            (glyphbits [ 2][column] « 5) |
            (glyphbits [ 3][column] « 4) |
            (glyphbits [ 4][column] « 3) |
            (glyphbits [ 5][column] « 2) |
            (glyphbits [ 6][column] « 1) |
            (glyphbits [ 7][column]     );
      transpose [1][column] =
            (glyphbits [ 8][column] « 7) |
            (glyphbits [ 9][column] « 6) |
            (glyphbits [10][column] « 5) |
            (glyphbits [11][column] « 4) |
            (glyphbits [12][column] « 3) |
            (glyphbits [13][column] « 2) |
            (glyphbits [14][column] « 1) |
            (glyphbits [15][column]     );
   }
   if (width > 8) {
      for (column = 8; column < width; column++) {
         transpose [0][column] =
               (glyphbits [0][column] « 7) |
               (glyphbits [1][column] « 6) |
               (glyphbits [2][column] « 5) |
               (glyphbits [3][column] « 4) |
               (glyphbits [4][column] « 3) |
               (glyphbits [5][column] « 2) |
               (glyphbits [6][column] « 1) |
               (glyphbits [7][column]     );
         transpose [1][column] =
               (glyphbits [ 8][column] « 7) |
               (glyphbits [ 9][column] « 6) |
               (glyphbits [10][column] « 5) |
                (glyphbits [11][column] « 4) |
                (glyphbits [12][column] « 3) |
                (glyphbits [13][column] « 2) |
                (glyphbits [14][column] « 1) |
                (glyphbits [15][column]      );
      }
   }
   else {
      for (column = 8; column < width; column++)
         transpose [0][column] = transpose [1][column] = 0x00;
   }


   return;
}
```

5.10.2.4 parse_hex()

void parse_hex (
char ∗ hexstring,
int ∗ width,
unsigned ∗ codept,
unsigned char glyph[16][2])

Decode a Unifont .hex file into Uniocde code point and glyph.

This function takes one line from a Unifont .hex file and decodes it into a code point followed by a 16-row glyph array. The glyph array can be one byte (8 columns) or two bytes (16 columns).

Parameters

in	hexstring	The Unicode .hex string for one code point.
out	width	The number of columns in a glyph with 16 rows.
out	codept	The code point, contained in the first .hex file field.
out	glyph	The Unifont glyph, as 16 rows by 1 or 2 bytes wide.

Definition at line 44 of file unifont-support.c.

```
                                          {

   int i;
   int row;
   int length;

   sscanf (hexstring, ”%X”, codept);
   length = strlen (hexstring);
   for (i = length - 1; i > 0 && hexstring[i] != ’\n’; i--);
   hexstring[i] = ’\0’;
   for (i = 0; i < 9 && hexstring[i] != ’:’; i++);
   i++;  /* Skip over ’:’ */
   *width = (length - i) * 4 / 16;  /* 16 rows per glyphbits */

   for (row = 0; row < 16; row++) {
      sscanf (&hexstring[i], ”%2hhX”, &glyph [row][0]);
      i += 2;
      if (*width > 8) {
        sscanf (&hexstring[i], ”%2hhX”, &glyph [row][1]);
        i += 2;
      }
      else {
         glyph [row][1] = 0x00;
      }
   }


   return;
}
```

5.10.2.5 xglyph2string()

void xglyph2string (
    int width,
    unsigned codept,
    unsigned char transpose[2][16],
    char ∗ outstring)

Convert a code point and transposed glyph into a Unifont .hex string.

This function takes a code point and a transposed Unifont glyph of 2 rows of 8 pixels in a column, and converts it into a Unifont .hex format character array.

Parameters

in	width	The number of columns in the glyph.
in	codept	The code point to appear in the output .hex string.
in	transpose	The transposed glyph, with 2 sets of 8-row data.
out	outstring	The output string, in Unifont .hex format.

Definition at line 267 of file unifont-support.c.

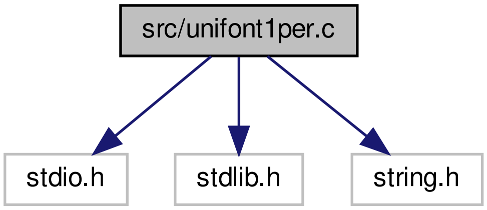
```
                                {

   int i;            /* index into outstring array */
   int  column;

   if (codept <= 0xFFFF) {
      sprintf (outstring, ”%04X:”, codept);
      i = 5;
   }
   else {
      sprintf (outstring, ”%06X:”, codept);
      i = 7;
   }

   for (column = 0; column < 8; column++) {
      sprintf (&outstring[i], ”%02X”, transpose [0][column]);
      i += 2;
   }
   if (width > 8) {
      for (column = 8; column < 16; column++) {
         sprintf (&outstring[i], ”%02X”, transpose [0][column]);
         i += 2;
      }
   }
   for (column = 0; column < 8; column++) {
      sprintf (&outstring[i], ”%02X”, transpose [1][column]);
      i += 2;
   }
   if (width > 8) {
      for (column = 8; column < 16; column++) {
         sprintf (&outstring[i], ”%02X”, transpose [1][column]);
         i += 2;
      }
   }

   outstring[i] = ’\0’;  /* terminate output string */


   return;
}
```

5.11 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.

5.11.1 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

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

5.11.2 Macro Definition Documentation

5.11.2.1 MAXFILENAME

#define MAXFILENAME 20
Maximum size of a filename of the form "U+%06X.bmp".
Definition at line 60 of file unifont1per.c.

5.11.2.2 MAXSTRING

#define MAXSTRING 266

Maximum size of an input line in a Unifont .hex file - 1.

Definition at line 57 of file unifont1per.c.

5.11.3 Function Documentation

5.11.3.1 main()

int main ( )

The main function.

Returns

> This program exits with status EXIT_SUCCESS.

Definition at line 69 of file unifont1per.c.

```
69        {
70
71    int i; /* loop variable */
72
73    /*
74 Define bitmap header bytes
75 */
76    unsigned char header [62] = {
77       /*
78 Bitmap File Header -- 14 bytes
79 */
80       ’B’, ’M’,       /* Signature          */
81       0x7E, 0, 0, 0,  /* File Size          */
82          0, 0, 0, 0,  /* Reserved           */
83       0x3E, 0, 0, 0,  /* Pixel Array Offset */
84
85       /*
86 Device Independent Bitmap Header -- 40 bytes
87
88 Image Width and Image Height are assigned final values
89 based on the dimensions of each glyph.
90 */
91       0x28,    0,    0,    0,  /* DIB Header Size                */
92       0x10,    0,    0,    0,  /* Image Width = 16 pixels         */
93       0xF0, 0xFF, 0xFF, 0xFF,  /* Image Height = -16 pixels       */
94       0x01,    0,              /* Planes                        */
95       0x01,    0,              /* Bits Per Pixel                 */
96          0,    0,    0,    0,  /* Compression                   */
97       0x40,    0,    0,    0,  /* Image Size                    */
98       0x14, 0x0B,    0,    0,  /* X Pixels Per Meter = 72 dpi     */
99       0x14, 0x0B,    0,    0,  /* Y Pixels Per Meter = 72 dpi     */
100       0x02,    0,    0,    0,  /* Colors In Color Table           */
101          0,    0,    0,    0,  /* Important Colors               */
102
103       /*
104 Color Palette -- 8 bytes
105 */
106       0xFF, 0xFF, 0xFF, 0,  /* White */
107          0,    0,    0, 0   /* Black */
108    };
109
110    char instring[MAXSTRING]; /* input string                         */
111    int  code_point;           /* current Unicode code point          */
112    char glyph[MAXSTRING];     /* bitmap string for this glyph         */
113    int  glyph_height=16;      /* for now, fixed at 16 pixels high    */
114    int  glyph_width;          /* 8, 16, 24, or 32 pixels wide        */
115    char filename[MAXFILENAME];/* name of current output file         */
116    FILE *outfp;               /* file pointer to current output file */
117
118    int string_index;  /* pointer into hexadecimal glyph string */
119    int nextbyte;      /* next set of 8 bits to print out       */
120
121    /* Repeat for each line in the input stream */
122    while (fgets (instring, MAXSTRING - 1, stdin) != NULL) {
123       /* Read next Unifont ASCII hexadecimal format glyph description */
```

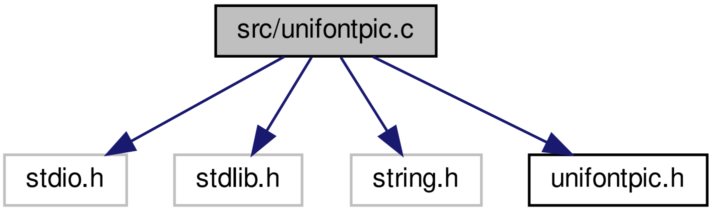
```
      sscanf (instring, ”%X:%s”, &code_point, glyph);
      /* Calculate width of a glyph in pixels; 4 bits per ASCII hex digit */
      glyph_width = strlen (glyph) / (glyph_height / 4);
      snprintf (filename, MAXFILENAME, ”U+%06X.bmp”, code_point);
      header [18] =  glyph_width;  /* bitmap 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);
}
```

5.12 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.

5.12.1 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

5.12.2 Macro Definition Documentation

5.12.2.1 HDR_LEN

```
#define HDR_LEN 33
```

Define length of header string for top of chart.

Definition at line 67 of file unifontpic.c.

5.12.3 Function Documentation

5.12.3.1 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

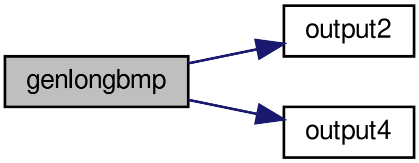
in	plane_array	The array of glyph bitmaps for a plane.
in	dpi	Dots per inch, for encoding in the BMP output file header.
in	tinynum	Whether to generate tiny numbers in wide grid (unused).
in	plane	The Unicode plane, 0..17.

Definition at line 294 of file unifontpic.c.

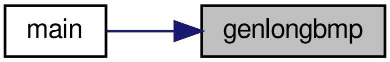
```
{

   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:



5.12.3.2 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.

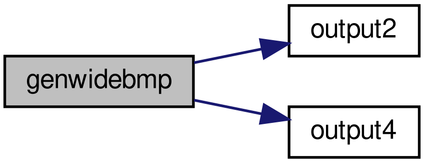
Parameters

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.

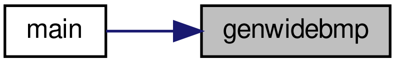
Definition at line 581 of file unifontpic.c.

```
{

   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:



5.12.3.3 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

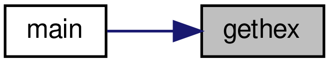
in	instring	One line from a Unifont .hex-format file.
in,out	plane_array	Bitmap for this plane, one bitmap row per element.
in	plane	The Unicode plane, 0..17.

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:



5.12.3.4 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

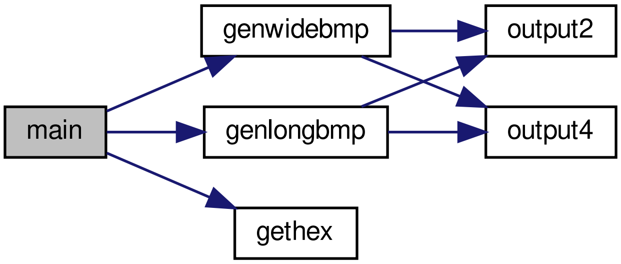
This program exits with status EXIT_SUCCESS.

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:



5.12.3.5 output2()

void output2 (
int thisword)

Output a 2-byte integer in little-endian order.

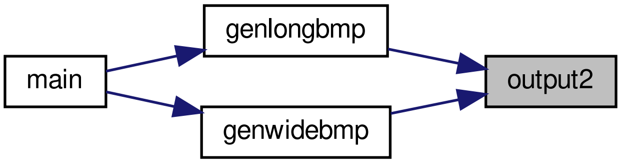
Parameters

in	thisword	The 2-byte integer to output as binary data.

Definition at line 194 of file unifontpic.c.

```
{

   putchar ( thisword       & 0xFF);
   putchar ((thisword » 8) & 0xFF);

   return;
}
```

Here is the caller graph for this function:



5.12.3.6 output4()

```
void output4 (
                    int thisword )
```

Output a 4-byte integer in little-endian order.

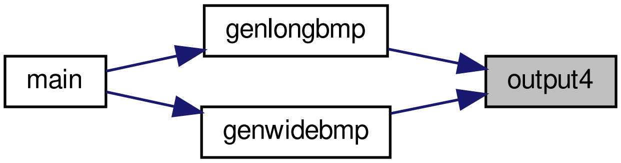
Parameters

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;
}
```

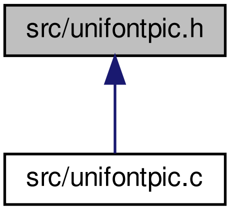
Here is the caller graph for this function:



5.13 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 15.1.01"

  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.

5.13.1 Detailed Description

unifontpic.h - Header file for unifontpic.c

Author

Paul Hardy, July 2017

Copyright

Copyright (C) 2017 Paul Hardy

5.13.2 Variable Documentation

5.13.2.1 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 179 of file unifontpic.h.

5.13.2.2 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 42 of file unifontpic.h.

5.13.2.3 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 188 of file unifontpic.h.

5.14 src/unigen-hangul.c File Reference

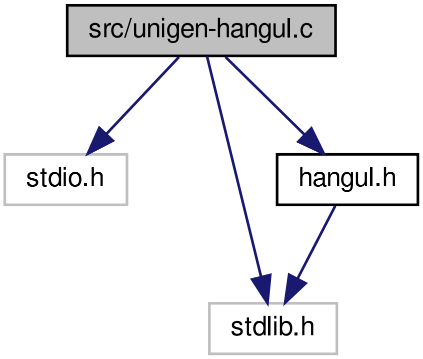
Generate arbitrary hangul syllables.

#include <stdio.h>

#include <stdlib.h>

#include ”hangul.h”

Include dependency graph for unigen-hangul.c:



Data Structures

- struct PARAMS

Functions

- int main (int argc, char ∗argv[])
  Program entry point.
- void parse_args (int argc, char ∗argv[], struct PARAMS ∗params)
  Parse command line arguments.
- void get_hex_range (char ∗instring, unsigned ∗start, unsigned ∗end)
  Scan a hexadecimal range from a character string.

5.14.1 Detailed Description

Generate arbitrary hangul syllables.

Input is a Unifont .hex file such as the ”hangul-base.hex” file that is included in the Unifont package.

The default program parameters will generate the Unicode Hangul Syllables range of U+AC00..U+D7A3. The syllables will appear in this order:

```
For each modern choseong {
   For each modern jungseong {
      Output syllable of choseong and jungseong
      For each modern jongseong {
         Output syllable of choseong + jungseong + jongseong
      }
   }
}
```

By starting the jongseong code point at one before the first valid jongseong, the first inner loop iteration will add a blank glyph for the jongseong portion of the syllable, so only the current choseong and jungseong will be output first.

Author

Paul Hardy

Copyright

5.14.2 Function Documentation

5.14.2.1 main()

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

Program entry point.

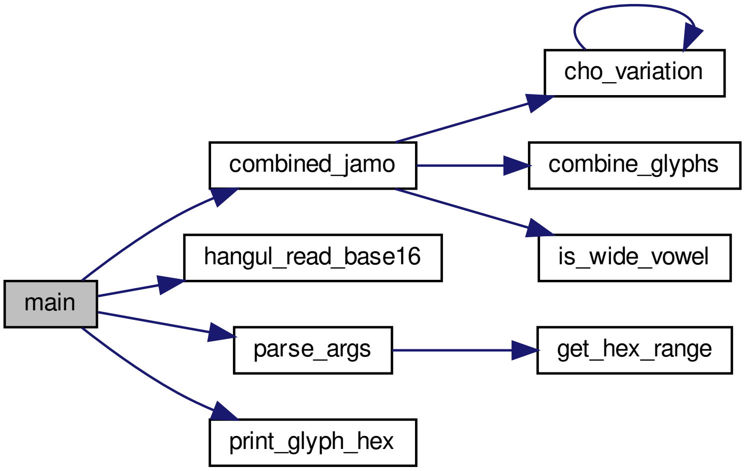
Default parameters for Hangul syllable generation.

Definition at line 69 of file unigen-hangul.c.

```
69                    {
70
71    int i;  /* loop variable */
72    unsigned codept;
73    unsigned max_codept;
74    unsigned glyph[MAX_GLYPHS][16];
75    unsigned tmp_glyph [16];  /* To build one combined glyph at a time. */
76    int cho, jung, jong;      /* The 3 components in a Hangul syllable. */
77
78 /// Default parameters for Hangul syllable generation.
79    struct PARAMS params = { 0xAC00,  /* Starting output Unicode code point */
80                             0x1100,  /* First modern choseong              */
81                             0x1112,  /* Last modern choseong               */
82                             0x1161,  /* First modern jungseong             */
```

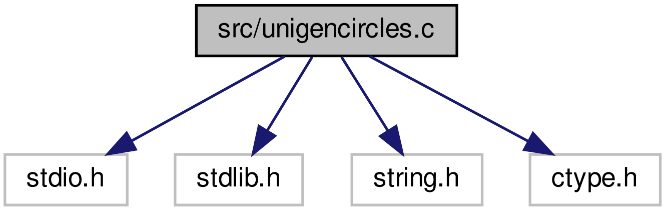
```
                  0x1175,  /* Last modern jungseong              */
                  0x11A7,  /* One before first modern jongseong  */
                  0x11C2,  /* Last modern jongseong              */
                  stdin,   /* Default input file pointer         */
                  stdout   /* Default output file pointer        */
                 };

 void parse_args (int argc, char *argv[], struct PARAMS *params);

 unsigned hangul_read_base16 (FILE *infp, unsigned glyph[][16]);

 void print_glyph_hex (FILE *fp, unsigned codept, unsigned *this_glyph);

 void combined_jamo (unsigned glyph [MAX_GLYPHS][16],
               unsigned cho, unsigned jung, unsigned jong,
               unsigned *combined_glyph);


  if (argc > 1) {
    parse_args (argc, argv, &params);

#ifdef DEBUG
    fprintf (stderr,
          "Range: (U+%04X, U+%04X, U+%04X) to (U+%04X, U+%04X, U+%04X)\n",
          params.cho_start, params.jung_start, params.jong_start,
          params.cho_end,   params.jung_end,   params.jong_end);
#endif
  }

  /*
Initialize glyph array to all zeroes.
*/
  for (codept = 0; codept < MAX_GLYPHS; codept++) {
    for (i = 0; i < 16; i++) glyph[codept][i] = 0x0000;
  }

  /*
Read Hangul base glyph file.
*/
  max_codept = hangul_read_base16 (params.infp, glyph);
  if (max_codept > 0x8FF) {
    fprintf (stderr, "\nWARNING: Hangul glyph range exceeds PUA space.\n\n");
  }

  codept = params.starting_codept;  /* First code point to output */

  for (cho = params.cho_start; cho <= params.cho_end; cho++) {
    for (jung = params.jung_start; jung <= params.jung_end; jung++) {
      for (jong = params.jong_start; jong <= params.jong_end; jong++) {

#ifdef DEBUG
        fprintf (params.outfp,
              "(U+%04X, U+%04X, U+%04X)\n",
              cho, jung, jong);
#endif
        combined_jamo (glyph, cho, jung, jong, tmp_glyph);
        print_glyph_hex (params.outfp, codept, tmp_glyph);
        codept++;
        if (jong == JONG_UNICODE_END)
          jong = JONG_EXTB_UNICODE_START - 1; /* Start Extended-B range */
      }
      if (jung == JUNG_UNICODE_END)
        jung = JUNG_EXTB_UNICODE_START - 1;  /* Start Extended-B range */
    }
    if (cho == CHO_UNICODE_END)
      cho = CHO_EXTA_UNICODE_START - 1;  /* Start Extended-A range */
  }

  if (params.infp  != stdin)  fclose (params.infp);
  if (params.outfp != stdout) fclose (params.outfp);

  exit (EXIT_SUCCESS);
}
```

Here is the call graph for this function:



5.15 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.

5.15.1 Detailed Description

unigencircles - Superimpose dashed combining circles on combining glyphs

Author

Paul Hardy

Copyright

Copyright (C) 2013, Paul Hardy.

5.15.2 Function Documentation

5.15.2.1 add_double_circle()

void add_double_circle (
char ∗ glyphstring,
int offset)

Superimpose a double-width dashed combining circle on a glyph bitmap.

Parameters

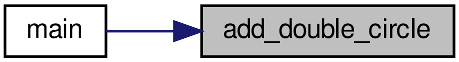
in,out	glyphstring	A double-width glyph, 16x16 pixels.

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 */
      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:



5.15.2.2 add_single_circle()

void add_single_circle (
char ∗ glyphstring)

Superimpose a single-width dashed combining circle on a glyph bitmap.

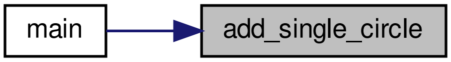
Parameters

in,out	glyphstring	A single-width glyph, 8x16 pixels.

Definition at line 163 of file unigencircles.c.

```
{

   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;
}
```

Here is the caller graph for this function:



5.15.2.3 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.

Returns

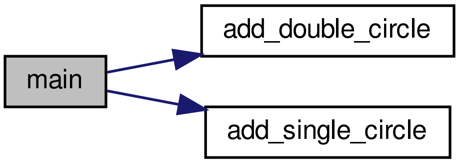
This program exits with status EXIT_SUCCESS.

Definition at line 73 of file unigencircles.c.

```
74 {
75
76    char teststring[MAXSTRING];  /* current input line                       */
77    int  loc;                    /* Unicode code point of current input line */
78    int  offset;                 /* offset value of a combining character    */
79    char *gstart;                  /* glyph start, pointing into teststring    */
80
81    char combining[0x110000];     /* 1 --> combining glyph; 0 --> non-combining */
82    char x_offset [0x110000];     /* second value in *combining.txt files        */
83
84    void add_single_circle(char *);       /* add a single-width dashed circle */
85    void add_double_circle(char *, int); /* add a double-width dashed circle */
86
87    FILE *infilefp;
88
89    /*
90 if (argc != 3) {
91 fprintf (stderr,
92 ”\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n”);
93 exit (EXIT_FAILURE);
94 }
95 */
96
97    /*
98 Read the combining characters list.
99 */
100    /* Start with no combining code points flagged */
101    memset (combining, 0, 0x110000 * sizeof (char));
102    memset (x_offset , 0, 0x110000 * sizeof (char));
103
104    if ((infilefp = fopen (argv[1],”r”)) == NULL) {
105       fprintf (stderr,”ERROR - combining characters file %s not found.\n\n”,
106               argv[1]);
107       exit (EXIT_FAILURE);
108    }
109
110    /* Flag list of combining characters to add a dashed circle. */
111    while (fscanf (infilefp, ”%X:%d”, &loc, &offset) != EOF) {
112       /*
113 U+01107F and U+01D1A0 are not defined as combining characters
114 in Unicode; they were added in a combining.txt file as the
115 only way to make them look acceptable in proximity to other
116 glyphs in their script.
117 */
118       if (loc != 0x01107F && loc != 0x01D1A0) {
119          combining[loc] = 1;
120          x_offset [loc] = offset;
121       }
122    }
123    fclose (infilefp); /* all done reading combining.txt */
124
125    /* Now read the non-printing glyphs; they never have dashed circles */
126    if ((infilefp = fopen (argv[2],”r”)) == NULL) {
127       fprintf (stderr,”ERROR - nonprinting characters file %s not found.\n\n”,
128               argv[1]);
129       exit (EXIT_FAILURE);
130    }
131
132    /* Reset list of nonprinting characters to avoid adding a dashed circle. */
133    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);
}
```

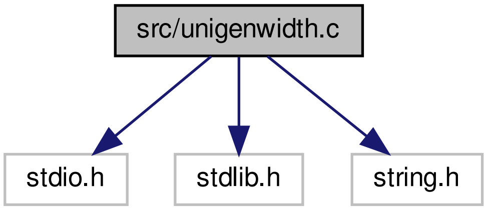
Here is the call graph for this function:



5.16 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.

5.16.1 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).

5.16.2 Macro Definition Documentation

5.16.2.1 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.

5.16.3 Function Documentation

5.16.3.1 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.

```
64 {
65
66    int i; /* loop variable */
67
68    char teststring[MAXSTRING];
69    int  loc;
70    char *gstart;
71
72    char glyph_width[0x20000];
73    char pikto_width[PIKTO_SIZE];
74
75    FILE *infilefp;
76
77    if (argc != 3) {
78       fprintf (stderr, ”\n\nUsage: %s <unifont.hex> <combining.txt>\n\n”, argv[0]);
79       exit (EXIT_FAILURE);
80    }
81
82    /*
83 Read the collection of hex glyphs.
84 */
85    if ((infilefp = fopen (argv[1],”r”)) == NULL) {
86       fprintf (stderr,”ERROR - hex input file %s not found.\n\n”, argv[1]);
87       exit (EXIT_FAILURE);
88    }
89
90    /* Flag glyph as non-existent until found. */
91    memset (glyph_width, -1, 0x20000 * sizeof (char));
92    memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));
93
94    teststring[MAXSTRING-1] = ’\0’;
95    while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
96       sscanf (teststring, ”%X:%*s”, &loc);
97       if (loc < 0x20000) {
98          gstart = strchr (teststring,’:’) + 1;
99          /*
100 16 rows per glyph, 2 ASCII hexadecimal digits per byte,
101 so divide number of digits by 32 (shift right 5 bits).
102 */
103          glyph_width[loc] = (strlen (gstart) - 1) » 5;
104       }
105       else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
106          gstart = strchr (teststring,’:’) + 1;
107          pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
108       }
109    }
110
111    fclose (infilefp);
112
113    /*
114 Now read the combining character code points.  These have width of 0.
115 */
116    if ((infilefp = fopen (argv[2],”r”)) == NULL) {
117       fprintf (stderr,”ERROR - combining characters file %s not found.\n\n”, argv[2]);
118       exit (EXIT_FAILURE);
119    }
120
121    while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
122       sscanf (teststring, ”%X:%*s”, &loc);
123       if (loc < 0x20000) glyph_width[loc] = 0;
124    }
125
126    fclose (infilefp);
127
128    /*
129 Code Points with Unusual Properties (Unicode Standard, Chapter 4).
130
131 As of Unifont 10.0.04, use the widths in the ”*-nonprinting.hex”
132 files.  If an application is smart enough to know how to handle
133 these special cases, it will not render the ”nonprinting” glyph
134 and will treat the code point as being zero-width.
135 */
136 // glyph_width[0]=0; /* NULL character */
137 // for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
138 // for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */
139
```

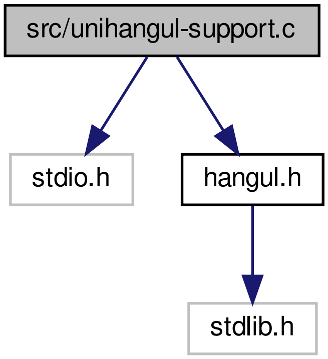
```
// glyph_width[0x034F]=0; /* combining grapheme joiner               */
// glyph_width[0x180B]=0; /* Mongolian free variation selector one   */
// glyph_width[0x180C]=0; /* Mongolian free variation selector two   */
// glyph_width[0x180D]=0; /* Mongolian free variation selector three */
// glyph_width[0x180E]=0; /* 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);
}
```

5.17 src/unihangul-support.c File Reference

Functions for converting Hangul letters into syllables.

#include <stdio.h>
#include ”hangul.h”

Include dependency graph for unihangul-support.c:



Functions

- unsigned hangul_read_base8 (FILE ∗infp, unsigned char base[ ][32])

  Read hangul-base.hex file into a unsigned char array.
- unsigned hangul_read_base16 (FILE ∗infp, unsigned base[ ][16])

  Read hangul-base.hex file into a unsigned array.
- void hangul_decompose (unsigned codept, int ∗initial, int ∗medial, int ∗final)

  Decompose a Hangul Syllables code point into three letters.
- unsigned hangul_compose (int initial, int medial, int final)

  Compose a Hangul syllable into a code point, or 0 if none exists.
- void hangul_hex_indices (int choseong, int jungseong, int jongseong, int ∗cho_index, int ∗jung_index, int ∗jong_index)

  Determine index values to the bitmaps for a syllable's components.
- void hangul_variations (int choseong, int jungseong, int jongseong, int ∗cho_var, int ∗jung_var, int ∗jong_var)

  Determine the variations of each letter in a Hangul syllable.
- int cho_variation (int choseong, int jungseong, int jongseong)

  Return the Johab 6/3/1 choseong variation for a syllable.
- int is_wide_vowel (int vowel)

  Whether vowel has rightmost vertical stroke to the right.
- int jung_variation (int choseong, int jungseong, int jongseong)

  Return the Johab 6/3/1 jungseong variation.
- int jong_variation (int choseong, int jungseong, int jongseong)

  Return the Johab 6/3/1 jongseong variation.
- void hangul_syllable (int choseong, int jungseong, int jongseong, unsigned char hangul_base[ ][32], unsigned char ∗syllable)

  Given letters in a Hangul syllable, return a glyph.
- int glyph_overlap (unsigned ∗glyph1, unsigned ∗glyph2)

  See if two glyphs overlap.

- void combine_glyphs (unsigned ∗glyph1, unsigned ∗glyph2, unsigned ∗combined_glyph)

  Combine two glyphs into one glyph.
- void print_glyph_txt (FILE ∗fp, unsigned codept, unsigned ∗this_glyph)

  Print one glyph in Unifont hexdraw plain text style.
- void print_glyph_hex (FILE ∗fp, unsigned codept, unsigned ∗this_glyph)

  Print one glyph in Unifont hexdraw hexadecimal string style.
- void one_jamo (unsigned glyph_table[MAX_GLYPHS][16], unsigned jamo, unsigned ∗jamo_glyph)

  Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.
- void combined_jamo (unsigned glyph_table[MAX_GLYPHS][16], unsigned cho, unsigned jung, unsigned jong, unsigned ∗combined_glyph)

  Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.

5.17.1 Detailed Description

Functions for converting Hangul letters into syllables.

This file contains functions for reading in Hangul letters arranged in a Johab 6/3/1 pattern and composing syllables with them. One function maps an iniital letter (choseong), medial letter (jungseong), and final letter (jongseong) into the Hangul Syllables Unicode block, U+AC00..U+D7A3. Other functions allow formation of glyphs that include the ancient Hangul letters that Hanterm supported. More can be added if desired, with appropriate changes to start positions and lengths defined in "hangul.h".

Author

Paul Hardy

Copyright

5.17.2 Function Documentation

5.17.2.1 cho_variation()

```
int cho_variation (
        int choseong,
        int jungseong,
        int jongseong )
```

Return the Johab 6/3/1 choseong variation for a syllable.

This function takes the two or three (if jongseong is included) letters that comprise a syllable and determine the variation of the initial consonant (choseong).

Each choseong has 6 variations:

Variation Occurrence

0 Choseong with a vertical vowel such as "A". 1 Choseong with a horizontal vowel such as "O". 2 Choseong with a vertical and horizontal vowel such as "WA". 3 Same as variation 0, but with jongseong (final consonant). 4 Same as variation 1, but with jongseong (final consonant). Also a horizontal vowel pointing down, such as U and YU. 5 Same as variation 2, but with jongseong (final consonant). Also a horizontal vowel pointing down with vertical element, such as WEO, WE, and WI.

In addition, if the vowel is horizontal and a downward-pointing stroke as in the modern letters U, WEO, WE, WI, and YU, and in archaic letters YU-YEO, YU-YE, YU-I, araea, and araea-i, then 3 is added to the initial variation of 0 to 2, resulting in a choseong variation of 3 to 5, respectively.

Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable.

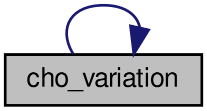
Returns

The choseong variation, 0 to 5.

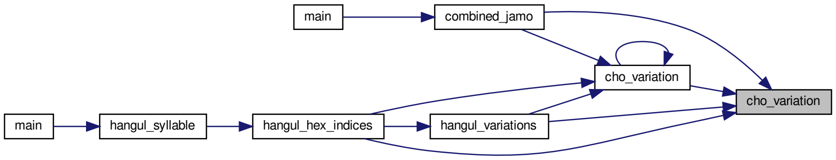
Definition at line 350 of file unihangul-support.c.

```
350                                                        {
351    int cho_variation;  /* Return value */
352
353    /*
354 The Choseong cho_var is determined by the
355 21 modern + 50 ancient Jungseong, and whether
356 or not the syllable contains a final consonant
357 (Jongseong).
358 */
359    static int choseong_var [TOTAL_JUNG + 1] = {
360           /*
361 Modern Jungseong in positions 0..20.
362 */
363 /* Location Variations  Unicode Range    Vowel #    Vowel Names  */
364 /* -------- ----------  --------------   -------    -----------  */
365 /* 0x2FB */ 0, 0, 0, // U+1161..U+1163-->[ 0.. 2]  A,   AE,  YA
366 /* 0x304 */ 0, 0, 0, // U+1164..U+1166-->[ 3.. 5]  YAE, EO,  E
367 /* 0x30D */ 0, 0,    // U+1167..U+1168-->[ 6.. 7]  YEO, YE
368 /* 0x313 */ 1,       // U+1169         -->[ 8]      O
369 /* 0x316 */ 2, 2, 2, // U+116A..U+116C-->[ 9..11]  WA,  WAE, WE
370 /* 0x31F */ 1, 4,    // U+116D..U+116E-->[12..13]  YO,  U
371 /* 0x325 */ 5, 5, 5, // U+116F..U+1171-->[14..16]  WEO, WE,  WI
372 /* 0x32E */ 4, 1,    // U+1172..U+1173-->[17..18]  YU,  EU
373 /* 0x334 */ 2,       // U+1174         -->[19]      YI
374 /* 0x337 */ 0,       // U+1175         -->[20]      I
375           /*
376 Ancient Jungseong in positions 21..70.
377 */
378 /* Location  Variations  Unicode Range    Vowel #       Vowel Names  */
379 /* --------  ----------  --------------   -------       -----------  */
380 /* 0x33A: */ 2, 5, 2, // U+1176..U+1178-->[21..23]   A-O,    A-U,   YA-O
381 /* 0x343: */ 2, 2, 5, // U+1179..U+117B-->[24..26]  YA-YO,  EO-O,   EU-U
382 /* 0x34C: */ 2, 2, 5, // U+117C..U+117E-->[27..29]  EO-EU, YEO-O,  YEO-U
383 /* 0x355: */ 2, 5, 5, // U+117F..U+1181-->[30..32]   O-EO,   O-E,    O-YE,
384 /* 0x35E: */ 4, 4, 2, // U+1182..U+1184-->[33..35]   O-O,    O-U,   YO-YA,
385 /* 0x367: */ 2, 2, 5, // U+1185..U+1187-->[36..38]  YO-YAE, YO-YEO, YO-O,
386 /* 0x370: */ 2, 5, 5, // U+1188..U+118A-->[39..41]  YO-I,    U-A,    U-AE,
387 /* 0x379: */ 5, 5, 5, // U+118B..U+118D-->[42..44] U-EO-EU,  U-YE,   U-U,
388 /* 0x382: */ 5, 5, 5, // U+118E..U+1190-->[45..47]  YU-A,   YU-EO,  YU-E,
389 /* 0x38B: */ 5, 5, 2, // U+1191..U+1193-->[48..50]  YU-YEO, YU-YE,  YU-U,
390 /* 0x394: */ 5, 2, 2, // U+1194..U+1196-->[51..53]  YU-I,   EU-U,   EU-EU,
391 /* 0x39D: */ 2, 0, 0, // U+1197..U+1199-->[54..56]  YI-U,    I-A,    I-YA,
392 /* 0x3A6: */ 2, 5, 2, // U+119A..U+119C-->[57..59]   I-O,    I-U,    I-EU,
393 /* 0x3AF: */ 0, 1, 2, // U+119D..U+119F-->[60..62] I-ARAEA, ARAEA, ARAEA-EO,
394 /* 0x3B8: */ 1, 2, 1, // U+11A0..U+11A2-->[63..65] ARAEA-U, ARAEA-I,SSANGARAEA,
395 /* 0x3C1: */ 2, 5, 0, // U+11A3..U+11A5-->[66..68]   A-EU,  YA-U,  YEO-YA,
396 /* 0x3CA: */ 2, 2,    // U+11A6..U+11A7-->[69..70]   O-YA,   O-YAE,
397 #ifdef EXTENDED_HANGUL
398 /* 0x3D0: */ 2, 4, 5, // U+D7B0..U+D7B2-->[71..73]   O-YEO,  O-O-I, YO-A,
399 /* 0x3D9: */ 5, 2, 5, // U+D7B3..U+D7B5-->[74..76]  YO-AE,  YO-EO,   U-YEO,
400 /* 0x3E2: */ 5, 5, 4, // U+D7B6..U+D7B8-->[77..79]   U-I-I, YU-AE,  YU-O,
401 /* 0x3EB: */ 5, 2, 5, // U+D7B9..U+D7BB-->[80..82]  EU-A,   EU-EO,  EU-E,
402 /* 0x3F4: */ 4, 2, 3, // U+D7BC..U+D7BE-->[83..85]  EU-O,    I-YA-O, I-YAE,
403 /* 0x3FD: */ 3, 3, 2, // U+D7BF..U+D7C1-->[86..88]   I-YEO,  I-YE,   I-O-I,
404 /* 0x406: */ 2, 2, 0, // U+D7C2..U+D7C4-->[89..91]   I-YO,   I-YU,   I-I,
405 /* 0x40F: */ 2, 2,    // U+D7C5..U+D7C6-->[92..93] ARAEA-A, ARAEA-E,
406 /* 0x415: */ -1       // Mark end of list of vowels.
407 #else
408 /* 0x310: */ -1       // Mark end of list of vowels.
409 #endif
410    };
411
412
413    if (jungseong < 0 || jungseong >= TOTAL_JUNG) {
      cho_variation = -1;
   }
   else {
      cho_variation = choseong_var [jungseong];
      if (choseong >= 0 && jongseong >= 0 && cho_variation < 3)
         cho_variation += 3;
   }


   return cho_variation;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.2.2 combine_glyphs()

void combine_glyphs (
unsigned ∗ glyph1,
unsigned ∗ glyph2,
unsigned ∗ combined_glyph)

Combine two glyphs into one glyph.

Parameters

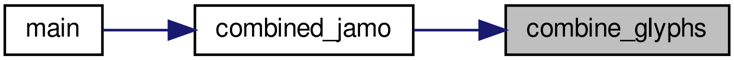
in	glyph1	The first glyph to overlap.
in	glyph2	The second glyph to overlap.
out	combined_glyph	The returned combination glyph.

Definition at line 637 of file unihangul-support.c.

```
                                        {
   int i;

   for (i = 0; i < 16; i++)
      combined_glyph [i] = glyph1 [i] | glyph2 [i];
```

```

    return;
}
```

Here is the caller graph for this function:



5.17.2.3 combined_jamo()

```
void combined_jamo (
                    unsigned glyph_table[MAX_GLYPHS][16],
                    unsigned cho,
                    unsigned jung,
                    unsigned jong,
                    unsigned * combined_glyph )
```

Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.

This function converts input Hangul choseong, jungseong, and jongseong Unicode code triplets into a Hangul syllable. Any of those with an out of range code point are assigned a blank glyph for combining.

This function performs the following steps:

```
 1) Determine the sequence number of choseong, jungseong,
    and jongseong, from 0 to the total number of choseong,
    jungseong, or jongseong, respectively, minus one.  The
    sequence for each is as follows:

    a) Choseong: Unicode code points of U+1100..U+115E
       and then U+A960..U+A97C.

    b) Jungseong: Unicode code points of U+1161..U+11A7
       and then U+D7B0..U+D7C6.

    c) Jongseong: Unicode code points of U+11A8..U+11FF
       and then U+D7CB..U+D7FB.

 2) From the choseong, jungseong, and jongseong sequence number,
    determine the variation of choseong and jungseong (there is
    only one jongseong variation, although it is shifted right
    by one column for some vowels with a pair of long vertical
    strokes on the right side).

 3) Convert the variation numbers for the three syllable
    components to index locations in the glyph array.

 4) Combine the glyph array glyphs into a syllable.
```

Parameters

in	glyph_table	The collection of all jamo glyphs.
in	cho	The choseong Unicode code point, 0 or 0x1100..0x115F.
in	jung	The jungseong Unicode code point, 0 or 0x1160..0x11A7.

Parameters

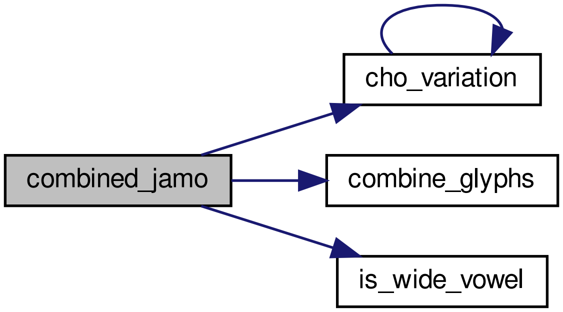
in	jong	The jongseong Unicode code point, 0 or 0x11A8..0x11FF.
out	combined_glyph	The output glyph, 16 columns in each of 16 rows.

Definition at line 787 of file unihangul-support.c.

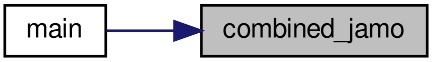
```
                            {

   int i;  /* Loop variable. */
   int cho_num,   jung_num,   jong_num;
   int cho_group, jung_group, jong_group;
   int cho_index, jung_index, jong_index;

   unsigned tmp_glyph[16]; /* Hold shifted jongsung for wide vertical vowel. */

   int  cho_variation (int choseong, int jungseong, int jongseong);

   void combine_glyphs (unsigned *glyph1, unsigned *glyph2,
                 unsigned *combined_glyph);


   /* Choose a blank glyph for each syllalbe by default. */
   cho_index = jung_index = jong_index = 0x000;

   /*
Convert Unicode code points to jamo sequence number
of each letter, or -1 if letter is not in valid range.
*/
   if (cho >= 0x1100 && cho <= 0x115E)
     cho_num = cho - CHO_UNICODE_START;
   else if (cho >= CHO_EXTA_UNICODE_START &&
         cho < (CHO_EXTA_UNICODE_START + NCHO_EXTA))
     cho_num = cho - CHO_EXTA_UNICODE_START + NCHO_MODERN + NJONG_ANCIENT;
   else
     cho_num = -1;

   if (jung >= 0x1161 && jung <= 0x11A7)
     jung_num = jung - JUNG_UNICODE_START;
   else if (jung >= JUNG_EXTB_UNICODE_START &&
         jung < (JUNG_EXTB_UNICODE_START + NJUNG_EXTB))
     jung_num = jung - JUNG_EXTB_UNICODE_START + NJUNG_MODERN + NJUNG_ANCIENT;
   else
     jung_num = -1;

   if (jong >= 0x11A8 && jong <= 0x11FF)
     jong_num = jong - JONG_UNICODE_START;
   else if (jong >= JONG_EXTB_UNICODE_START &&
         jong < (JONG_EXTB_UNICODE_START + NJONG_EXTB))
     jong_num = jong - JONG_EXTB_UNICODE_START + NJONG_MODERN + NJONG_ANCIENT;
   else
     jong_num = -1;

   /*
Choose initial consonant (choseong) variation based upon
the vowel (jungseong) if both are specified.
*/
   if (cho_num < 0) {
     cho_index = cho_group = 0;  /* Use blank glyph for choseong. */
   }
   else {
     if (jung_num < 0 && jong_num < 0) {  /* Choseong is by itself. */
       cho_group = 0;
       if (cho_index < (NCHO_MODERN + NCHO_ANCIENT))
         cho_index = cho_num + JAMO_HEX;
       else  /* Choseong is in Hangul Jamo Extended-A range. */
         cho_index = cho_num - (NCHO_MODERN + NCHO_ANCIENT)
                      + JAMO_EXTA_HEX;
     }
     else {
       if (jung_num >= 0) {  /* Valid jungseong with choseong. */
         cho_group = cho_variation (cho_num, jung_num, jong_num);
       }
       else {  /* Invalid vowel; see if final consonant is valid. */
         /*
If initial consonant and final consonant are specified,
set cho_group to 4, which is the group tha would apply
to a horizontal-only vowel such as Hangul "O", so the
consonant appears full-width.
*/
            cho_group = 0;
            if (jong_num >= 0) {
               cho_group = 4;
            }
         }
         cho_index = CHO_HEX + CHO_VARIATIONS * cho_num +
                     cho_group;
      }  /* Choseong combined with jungseong and/or jongseong. */
   }  /* Valid choseong. */

   /*
Choose vowel (jungseong) variation based upon the choseong
and jungseong.
*/
   jung_index = jung_group = 0;  /* Use blank glyph for jungseong. */

   if (jung_num >= 0) {
      if (cho_num < 0 && jong_num < 0) {  /* Jungseong is by itself. */
         jung_group = 0;
         jung_index = jung_num + JUNG_UNICODE_START;
      }
      else {
         if (jong_num >= 0) {  /* If there is a final consonant. */
            if (jong_num == 3) /* Nieun; choose variation 3. */
               jung_group = 2;
            else
               jung_group = 1;
         }  /* Valid jongseong. */
         /* If valid choseong but no jongseong, choose jungseong variation 0. */
         else if (cho_num >= 0)
            jung_group = 0;
      }
      jung_index = JUNG_HEX + JUNG_VARIATIONS * jung_num + jung_group;
   }

   /*
Choose final consonant (jongseong) based upon whether choseong
and/or jungseong are present.
*/
   if (jong_num < 0) {
      jong_index = jong_group = 0;  /* Use blank glyph for jongseong. */
   }
   else {  /* Valid jongseong. */
      if (cho_num < 0 && jung_num < 0) {  /* Jongseong is by itself. */
         jong_group = 0;
         jong_index = jung_num + 0x4A8;
      }
      else {  /* There is only one jongseong variation if combined. */
         jong_group = 0;
         jong_index = JONG_HEX + JONG_VARIATIONS * jong_num +
                      jong_group;
      }
   }

   /*
Now that we know the index locations for choseong, jungseong, and
jongseong glyphs, combine them into one glyph.
*/
   combine_glyphs (glyph_table [cho_index], glyph_table [jung_index],
                   combined_glyph);

   if (jong_index > 0) {
      /*
If the vowel has a vertical stroke that is one column
away from the right border, shift this jongseung right
by one column to line up with the rightmost vertical
stroke in the vowel.
*/
      if (is_wide_vowel (jung_num)) {
         for (i = 0; i < 16; i++) {
            tmp_glyph [i] = glyph_table [jong_index] [i] » 1;
         }
         combine_glyphs (combined_glyph, tmp_glyph,
                         combined_glyph);
      }
      else {
         combine_glyphs (combined_glyph, glyph_table [jong_index],
                         combined_glyph);
      }
   }
```

```

    return;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.2.4 glyph_overlap()

```
int glyph_overlap (
            unsigned * glyph1,
            unsigned * glyph2 )
```

See if two glyphs overlap.

Parameters

in	glyph1	The first glyph, as a 16-row bitmap.
in	glyph2	The second glyph, as a 16-row bitmap.

Returns

0 if no overlaps between glyphs, 1 otherwise.

Definition at line 613 of file unihangul-support.c.

```
613                                             {
    int overlaps;  /* Return value; 0 if no overlaps, -1 if overlaps. */
    int i;

    /* Check for overlaps between the two glyphs. */

    i = 0;
    do {
       overlaps = (glyph1[i] & glyph2[i]) != 0;
       i++;
    }  while (i < 16 && overlaps == 0);

    return overlaps;
}
```

5.17.2.5 hangul_compose()

```
unsigned hangul_compose (
              int initial,
              int medial,
              int final )
```

Compose a Hangul syllable into a code point, or 0 if none exists.

This function takes three letters that can form a modern Hangul syllable and returns the corresponding Unicode Hangul Syllables code point in the range 0xAC00 to 0xD7A3.

If a three-letter combination includes one or more archaic letters, it will not map into the Hangul Syllables range. In that case, the returned code point will be 0 to indicate that no valid Hangul Syllables code point exists.

Parameters

in	initial	The first letter (choseong), 0 to 18.
in	medial	The second letter (jungseong), 0 to 20.
in	final	The third letter (jongseong), 0 to 26 or -1 if none.

Returns

The Unicode Hangul Syllables code point, 0xAC00 to 0xD7A3.

Definition at line 201 of file unihangul-support.c.

```
201                                                {
202    unsigned codept;
203
204
205    if (initial >= 0 && initial <= 18 &&
206        medial  >= 0 && medial  <= 20 &&
207        final   >= 0 && final   <= 26) {
208
209       codept  = 0xAC00;
210       codept += initial * 21 * 28;
211       codept += medial  * 28;
212       codept += final + 1;
213    }
214    else {
215       codept = 0;
216    }
217
218    return codept;
219 }
```

5.17.2.6 hangul_decompose()

```
void hangul_decompose (
              unsigned codept,
              int * initial,
```

int ∗ medial,
int ∗ final)

Decompose a Hangul Syllables code point into three letters.

Decompose a Hangul Syllables code point (U+AC00..U+D7A3) into:

- Choseong 0-19
- Jungseong 0-20
- Jongseong 0-27 or -1 if no jongseong

All letter values are set to -1 if the letters do not form a syllable in the Hangul Syllables range. This function only handles modern Hangul, because that is all that is in the Hangul Syllables range.

Parameters

in	codept	The Unicode code point to decode, from 0xAC00 to 0xD7A3.
out	initial	The 1st letter (choseong) in the syllable.
out	initial	The 2nd letter (jungseong) in the syllable.
out	initial	The 3rd letter (jongseong) in the syllable.

Definition at line 167 of file unihangul-support.c.

```
167                                                       {
168 
169    if (codept < 0xAC00 || codept > 0xD7A3) {
170       *initial = *medial = *final = -1;
171    }
172    else {
173       codept  -= 0xAC00;
174       *initial =  codept / (28  * 21);
175       *medial  = (codept /  28) % 21;
176       *final   =  codept %  28 - 1;
177    }
178 
179    return;
180 }
```

Here is the caller graph for this function:

main
hangul_decompose

5.17.2.7 hangul_hex_indices()

void hangul_hex_indices (
int choseong,
int jungseong,
int jongseong,
int ∗ cho_index,
int ∗ jung_index,
int ∗ jong_index)

Determine index values to the bitmaps for a syllable's components.
This function reads these input values for modern and ancient Hangul letters:

- Choseong number (0 to the number of modern and archaic choseong - 1.
- Jungseong number (0 to the number of modern and archaic jungseong - 1.
- Jongseong number (0 to the number of modern and archaic jongseong - 1, or -1 if none.

It then determines the variation of each letter given the combination with the other two letters (or just choseong and jungseong if the jongseong value is -1).
These variations are then converted into index locations within the glyph array that was read in from the hangul-base.hex file. Those index locations can then be used to form a composite syllable.
There is no restriction to only use the modern Hangul letters.

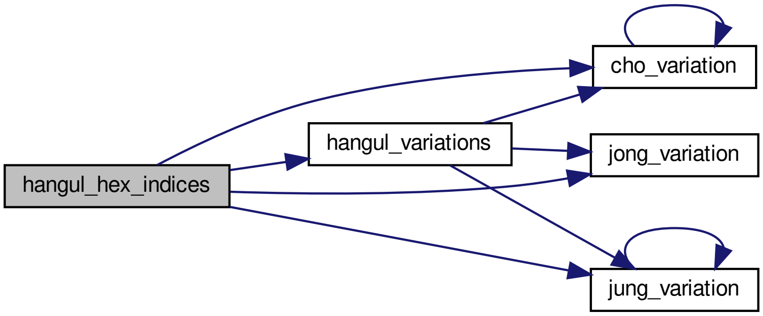
Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable, or -1 if none.
out	cho_index	Index location to the 1st letter variation from the hangul-base.hex file.
out	jung_index	Index location to the 2nd letter variation from the hangul-base.hex file.
out	jong_index	Index location to the 3rd letter variation from the hangul-base.hex file.

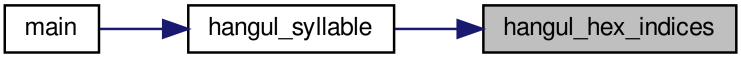
Definition at line 249 of file unihangul-support.c.

```
                                                        {

   int cho_variation, jung_variation, jong_variation;  /* Letter variations */

   void hangul_variations (int choseong, int jungseong, int jongseong,
        int *cho_variation, int *jung_variation, int *jong_variation);


   hangul_variations (choseong, jungseong, jongseong,
                      &cho_variation, &jung_variation, &jong_variation);

    *cho_index = CHO_HEX +  choseong * CHO_VARIATIONS +  cho_variation;
   *jung_index = JUNG_HEX      + jungseong * JUNG_VARIATIONS      + jung_variation;;
   *jong_index = jongseong < 0 ? 0x0000 :
                 JONG_HEX + jongseong * JONG_VARIATIONS + jong_variation;

   return;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.2.8 hangul_read_base16()

```
unsigned hangul_read_base16 (
                FILE * infp,
                unsigned base[][16] )
```

Read hangul-base.hex file into a unsigned array.
Read a Hangul base .hex file with separate choseong, jungseong, and jongseong glyphs for syllable formation. The order is:

- Empty glyph in 0x0000 position.
- Initial consonants (choseong).
- Medial vowels and dipthongs (jungseong).
- Final consonants (jongseong).
- Individual letter forms in isolation, not for syllable formation.

The letters are arranged with all variations for one letter before continuing to the next letter. In the current encoding, there are 6 variations of choseong, 3 of jungseong, and 1 of jongseong per letter.

Parameters

in	Input	file pointer; can be stdin.
out	Array	of bit patterns, with 16 16-bit values per letter.

Returns

The maximum code point value read in the file.

Definition at line 116 of file unihangul-support.c.

```
                                                {
   unsigned codept;
   unsigned max_codept;
   int       i, j;
   char      instring[MAXLINE];


   max_codept = 0;

   while (fgets (instring, MAXLINE, infp) != NULL) {
      sscanf (instring, "%X", &codept);
      codept -= PUA_START;
      /* If code point is within range, add it */
      if (codept < MAX_GLYPHS) {
         /* Find the start of the glyph bitmap. */
         for (i = 1; instring[i] != '\0' && instring[i] != ':'; i++);
         if (instring[i] == ':') {
            i++;  /* Skip over ':' to get to start of bitmap. */
            for (j = 0; j < 16; j++) {
               sscanf (&instring[i], "%4X", &base[codept][j]);
               i += 4;
            }
            if (codept > max_codept) max_codept = codept;
         }
      }
   }

   return max_codept;
}
```

Here is the caller graph for this function:

main
hangul_read_base16

5.17.2.9 hangul_read_base8()

unsigned hangul_read_base8 (
FILE ∗ infp,
unsigned char base[ ][32])

Read hangul-base.hex file into a unsigned char array.

Read a Hangul base .hex file with separate choseong, jungseong, and jongseong glyphs for syllable formation. The order is:

- Empty glyph in 0x0000 position.

- Initial consonants (choseong).
- Medial vowels and dipthongs (jungseong).
- Final consonants (jongseong).
- Individual letter forms in isolation, not for syllable formation.

The letters are arranged with all variations for one letter before continuing to the next letter. In the current encoding, there are 6 variations of choseong, 3 of jungseong, and 1 of jongseong per letter.

Parameters

in	Input	file pointer; can be stdin.
out	Array	of bit patterns, with 32 8-bit values per letter.

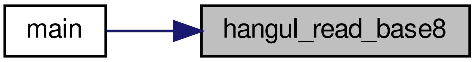
Returns

The maximum code point value read in the file.

Definition at line 63 of file unihangul-support.c.

```
                                            {
   unsigned codept;
   unsigned max_codept;
   int      i, j;
   char     instring[MAXLINE];


   max_codept = 0;

   while (fgets (instring, MAXLINE, infp) != NULL) {
      sscanf (instring, ”%X”, &codept);
      codept -= PUA_START;
      /* If code point is within range, add it */
      if (codept < MAX_GLYPHS) {
         /* Find the start of the glyph bitmap. */
         for (i = 1; instring[i] != ’\0’ && instring[i] != ’:’; i++);
         if (instring[i] == ’:’) {
            i++;  /* Skip over ’:’ to get to start of bitmap. */
            for (j = 0; j < 32; j++) {
               sscanf (&instring[i], ”%2hhX”, &base[codept][j]);
               i += 2;
            }
            if (codept > max_codept) max_codept = codept;
         }
      }
   }

   return max_codept;
}
```

Here is the caller graph for this function:



5.17.2.10 hangul_syllable()

void hangul_syllable (
    int choseong,
    int jungseong,
    int jongseong,
    unsigned char hangul_base[ ][32],
    unsigned char ∗ syllable)

Given letters in a Hangul syllable, return a glyph.

This function returns a glyph bitmap comprising up to three Hangul letters that form a syllable. It reads the three component letters (choseong, jungseong, and jungseong), then calls a function that determines the appropriate variation of each letter, returning the letter bitmap locations in the glyph array. Then these letter bitmaps are combined with a logical OR operation to produce a final bitmap, which forms a 16 row by 16 column bitmap glyph.

Parameters

in	choseong	The 1st letter in the composite glyph.
in	jungseong	The 2nd letter in the composite glyph.
in	jongseong	The 3rd letter in the composite glyph.
in	hangul_base	The glyphs read from the ”hangul_base.hex” file.

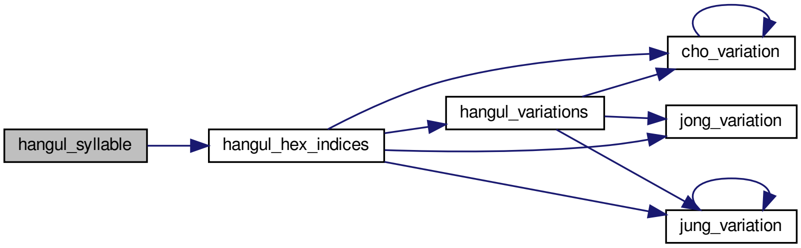
Returns

syllable The composite syllable, as a 16 by 16 pixel bitmap.

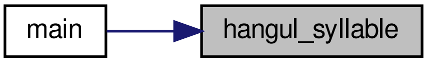
Definition at line 583 of file unihangul-support.c.

```
                                                                {

   int      i;  /* loop variable */
   int      cho_hex, jung_hex, jong_hex;
   unsigned char glyph_byte;


   hangul_hex_indices (choseong, jungseong, jongseong,
                      &cho_hex, &jung_hex, &jong_hex);

   for (i = 0; i < 32; i++) {
      glyph_byte  = hangul_base [cho_hex][i];
      glyph_byte |= hangul_base [jung_hex][i];
      if (jong_hex >= 0) glyph_byte |= hangul_base [jong_hex][i];
      syllable[i] = glyph_byte;
   }

   return;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.2.11 hangul_variations()

void hangul_variations (
int choseong,
int jungseong,
int jongseong,
int ∗ cho_var,
int ∗ jung_var,
int ∗ jong_var)

Determine the variations of each letter in a Hangul syllable.

Given the three letters that will form a syllable, return the variation of each letter used to form the composite glyph.

This function can determine variations for both modern and archaic Hangul letters; it is not limited to only the letters combinations that comprise the Unicode Hangul Syllables range.

This function reads these input values for modern and ancient Hangul letters:

- Choseong number (0 to the number of modern and archaic choseong - 1.
- Jungseong number (0 to the number of modern and archaic jungseong - 1.
- Jongseong number (0 to the number of modern and archaic jongseong - 1, or -1 if none.

It then determines the variation of each letter given the combination with the other two letters (or just choseong and jungseong if the jongseong value is -1).

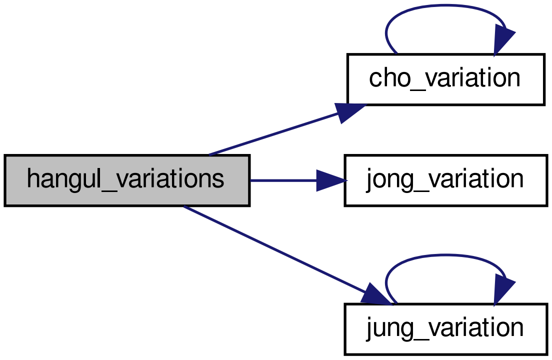
Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable, or -1 if none.
out	cho_var	Variation of the 1st letter from the hangul-base.hex file.
out	jung_var	Variation of the 2nd letter from the hangul-base.hex file.
out	jong_var	Variation of the 3rd letter from the hangul-base.hex file.

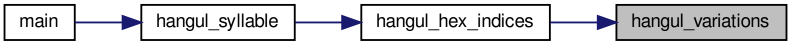
Definition at line 298 of file unihangul-support.c.

```
{

int  cho_variation (int choseong, int jungseong, int jongseong);
int jung_variation (int choseong, int jungseong, int jongseong);
int jong_variation (int choseong, int jungseong, int jongseong);

/*
Find the variation for each letter component.
*/
    *cho_var =  cho_variation (choseong, jungseong, jongseong);
   *jung_var = jung_variation (choseong, jungseong, jongseong);
   *jong_var = jong_variation (choseong, jungseong, jongseong);


   return;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.2.12 is_wide_vowel()

int is_wide_vowel (
int vowel)

Whether vowel has rightmost vertical stroke to the right.

Parameters

in	vowel	Vowel number, from 0 to TOTAL_JUNG - 1.

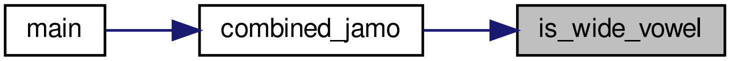
Returns

1 if this vowel's vertical stroke is wide on the right side; else 0.

Definition at line 434 of file unihangul-support.c.

```
                          {
   int retval;  /* Return value. */

   static int wide_vowel [TOTAL_JUNG + 1] = {
         /*
Modern Jungseong in positions 0..20.
*/
/* Location Variations  Unicode Range    Vowel #    Vowel Names  */
/* -------- ----------  --------------   -------    -----------  */
/* 0x2FB */ 0, 1, 0, // U+1161..U+1163-->[ 0.. 2]  A,   AE,  YA
/* 0x304 */ 1, 0, 1, // U+1164..U+1166-->[ 3.. 5]  YAE, EO,  E
/* 0x30D */ 0, 1,    // U+1167..U+1168-->[ 6.. 7]  YEO, YE
/* 0x313 */ 0,       // U+1169         -->[ 8]      O
/* 0x316 */ 0, 1, 0, // U+116A..U+116C-->[ 9..11]  WA,  WAE, WE
/* 0x31F */ 0, 0,    // U+116D..U+116E-->[12..13]  YO,  U
/* 0x325 */ 0, 1, 0, // U+116F..U+1171-->[14..16]  WEO, WE,  WI
/* 0x32E */ 0, 0,    // U+1172..U+1173-->[17..18]  YU,  EU
/* 0x334 */ 0,       // U+1174         -->[19]      YI
/* 0x337 */ 0,       // U+1175         -->[20]      I
         /*
Ancient Jungseong in positions 21..70.
*/
/* Location  Variations  Unicode Range    Vowel #       Vowel Names  */
/* --------  ----------  --------------   -------       -----------  */
/* 0x33A: */ 0, 0, 0, // U+1176..U+1178-->[21..23]   A-O,    A-U,   YA-O
/* 0x343: */ 0, 0, 0, // U+1179..U+117B-->[24..26]  YA-YO,  EO-O,   EU-U
/* 0x34C: */ 0, 0, 0, // U+117C..U+117E-->[27..29]  EO-EU, YEO-O,  YEO-U
/* 0x355: */ 0, 1, 1, // U+117F..U+1181-->[30..32]   O-EO,   O-E,    O-YE,
/* 0x35E: */ 0, 0, 0, // U+1182..U+1184-->[33..35]   O-O,    O-U,   YO-YA,
/* 0x367: */ 1, 0, 0, // U+1185..U+1187-->[36..38]  YO-YAE, YO-YEO, YO-O,
/* 0x370: */ 0, 0, 1, // U+1188..U+118A-->[39..41]  YO-I,    U-A,    U-AE,
/* 0x379: */ 0, 1, 0, // U+118B..U+118D-->[42..44] U-EO-EU,  U-YE,   U-U,
/* 0x382: */ 0, 0, 1, // U+118E..U+1190-->[45..47]  YU-A,   YU-EO,  YU-E,
/* 0x38B: */ 0, 1, 0, // U+1191..U+1193-->[48..50]  YU-YEO, YU-YE,  YU-U,
/* 0x394: */ 0, 0, 0, // U+1194..U+1196-->[51..53]  YU-I,   EU-U,   EU-EU,
/* 0x39D: */ 0, 0, 0, // U+1197..U+1199-->[54..56]  YI-U,    I-A,    I-YA,
/* 0x3A6: */ 0, 0, 0, // U+119A..U+119C-->[57..59]   I-O,    I-U,    I-EU,
/* 0x3AF: */ 0, 0, 0, // U+119D..U+119F-->[60..62] I-ARAEA, ARAEA, ARAEA-EO,
/* 0x3B8: */ 0, 0, 0, // U+11A0..U+11A2-->[63..65] ARAEA-U, ARAEA-I,SSANGARAEA,
/* 0x3C1: */ 0, 0, 0, // U+11A3..U+11A5-->[66..68]   A-EU,  YA-U,  YEO-YA,
/* 0x3CA: */ 0, 1,    // U+11A6..U+11A7-->[69..70]   O-YA,   O-YAE
#ifdef EXTENDED_HANGUL
/* 0x3D0: */ 0, 0, 0, // U+D7B0..U+D7B2-->[71..73]   O-YEO,  O-O-I, YO-A,
/* 0x3D9: */ 1, 0, 0, // U+D7B3..U+D7B5-->[74..76]  YO-AE,  YO-EO,   U-YEO,
/* 0x3E2: */ 1, 1, 0, // U+D7B6..U+D7B8-->[77..79]   U-I-I, YU-AE,  YU-O,
/* 0x3EB: */ 0, 0, 1, // U+D7B9..U+D7BB-->[80..82]  EU-A,   EU-EO,  EU-E,
/* 0x3F4: */ 0, 0, 1, // U+D7BC..U+D7BE-->[83..85]  EU-O,    I-YA-O, I-YAE,
/* 0x3FD: */ 0, 1, 0, // U+D7BF..U+D7C1-->[86..88]   I-YEO,  I-YE,   I-O-I,
/* 0x406: */ 0, 0, 1, // U+D7C2..U+D7C4-->[89..91]   I-YO,   I-YU,   I-I,
/* 0x40F: */ 0, 1,    // U+D7C5..U+D7C6-->[92..93] ARAEA-A, ARAEA-E,
/* 0x415: */ -1       // Mark end of list of vowels.
#else
/* 0x310: */ -1       // Mark end of list of vowels.
#endif
   };


   if (vowel >= 0 && vowel < TOTAL_JUNG) {
      retval = wide_vowel [vowel];
   }
   else {
      retval = 0;
   }


   return retval;
}
```

Here is the caller graph for this function:



5.17.2.13 jong_variation()

```
int jong_variation (
                    int choseong,
                    int jungseong,
                    int jongseong )   [inline]
```

Return the Johab 6/3/1 jongseong variation.

There is only one jongseong variation, so this function always returns 0. It is a placeholder function for possible future adaptation to other johab encodings.

Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable.

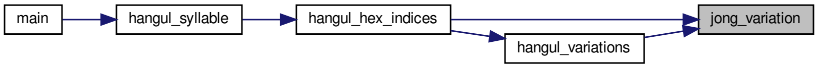
Returns

The jongseong variation, always 0.

Definition at line 558 of file unihangul-support.c.

```
558                                                {
559
560    return 0;  /* There is only one Jongseong variation. */
561 }
```

Here is the caller graph for this function:



5.17.2.14 jung_variation()

```
int jung_variation (
                    int choseong,
```

int jungseong,
int jongseong) [inline]

Return the Johab 6/3/1 jungseong variation.

This function takes the two or three (if jongseong is included) letters that comprise a syllable and determine the variation of the vowel (jungseong).

Each jungseong has 3 variations:

Variation Occurrence

---

0 Jungseong with only chungseong (no jungseong). 1 Jungseong with chungseong and jungseong (except nieun). 2 Jungseong with chungseong and jungseong nieun.

Parameters

in	choseong	The 1st letter in the syllable.
in	jungseong	The 2nd letter in the syllable.
in	jongseong	The 3rd letter in the syllable.

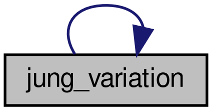
Returns

The jungseong variation, 0 to 2.

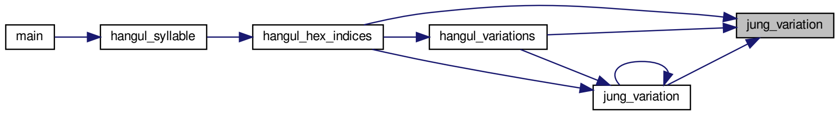
Definition at line 524 of file unihangul-support.c.

```
524                                                     {
525    int jung_variation;  /* Return value */
526
527    if (jungseong < 0) {
528       jung_variation = -1;
529    }
530    else {
531       jung_variation = 0;
532       if (jongseong >= 0) {
533          if (jongseong == 3)
534             jung_variation = 2;  /* Vowel for final Nieun. */
535          else
536             jung_variation = 1;
537       }
538    }
539
540
541    return jung_variation;
542 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.2.15 one_jamo()

void one_jamo (
unsigned glyph_table[MAX_GLYPHS][16],
unsigned jamo,
unsigned ∗ jamo_glyph)

Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.

Parameters

in	glyph_table	The collection of all jamo glyphs.
in	jamo	The Unicode code point, 0 or 0x1100..0x115F.
out	jamo_glyph	The output glyph, 16 columns in each of 16 rows.

Definition at line 717 of file unihangul-support.c.

```
                                        {

   int i;  /* Loop variable */
   int glyph_index;  /* Location of glyph in "hangul-base.hex" array */


   /* If jamo is invalid range, use blank glyph, */
   if (jamo >= 0x1100 && jamo <= 0x11FF) {
      glyph_index = jamo - 0x1100 + JAMO_HEX;
   }
   else if (jamo >= 0xA960 && jamo <= 0xA97F) {
      glyph_index = jamo - 0xA960 + JAMO_EXTA_HEX;
   }
   else if (jamo >= 0xD7B0 && jamo <= 0xD7FF) {
      glyph_index = jamo - 0x1100 + JAMO_EXTB_HEX;
   }
   else {
      glyph_index = 0;
   }

   for (i = 0; i < 16; i++) {
      jamo_glyph [i] = glyph_table [glyph_index] [i];
   }

   return;
}
```

5.17.2.16 print_glyph_hex()

void print_glyph_hex (
FILE ∗ fp,
unsigned codept,
unsigned ∗ this_glyph)

Print one glyph in Unifont hexdraw hexadecimal string style.

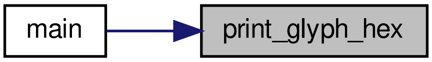
Parameters

in	fp	The file pointer for output.
in	codept	The Unicode code point to print with the glyph.
in	this_glyph	The 16-row by 16-column glyph to print.

Definition at line 692 of file unihangul-support.c.

```
692                                                    {
693
694    int i;
695
696
697    fprintf (fp, ”%04X:”, codept);
698
699    /* for each this_glyph row */
700    for (i = 0; i < 16; i++) {
701       fprintf (fp, ”%04X”, this_glyph[i]);
702    }
703    fputc (’\n’, fp);
704
705    return;
706 }
```

Here is the caller graph for this function:



5.17.2.17 print_glyph_txt()

void print_glyph_txt (
FILE ∗ fp,
unsigned codept,
unsigned ∗ this_glyph)

Print one glyph in Unifont hexdraw plain text style.

Parameters

in	fp	The file pointer for output.
in	codept	The Unicode code point to print with the glyph.
in	this_glyph	The 16-row by 16-column glyph to print.

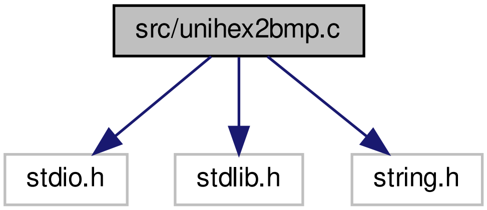
Definition at line 656 of file unihangul-support.c.

```
656                                                    {
657    int i;
658    unsigned mask;
659
660
661    fprintf (fp, ”%04X:”, codept);
662
663    /* for each this_glyph row */
664    for (i = 0; i < 16; i++) {
665       mask = 0x8000;
```

```
      fputc ('\t', fp);
      while (mask != 0x0000) {
         if (mask & this_glyph [i]) {
            fputc ('#', fp);
         }
         else {
            fputc ('-', fp);
         }
         mask »= 1;  /* shift to next bit in this_glyph row */
      }
      fputc ('\n', fp);
   }
   fputc ('\n', fp);

   return;
}
```

5.18 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.

5.18.1 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]

5.18.2 Function Documentation

5.18.2.1 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

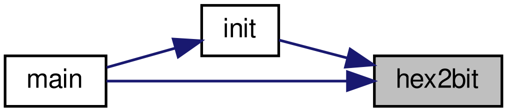
Always returns 0.

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);
}
```

Here is the caller graph for this function:



5.18.2.2 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.

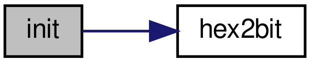
Definition at line 412 of file unihex2bmp.c.

```
{
    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;

```

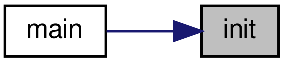
```
   for (i=0; i<18; i++) { /* bitmaps for ’0’..’9’, ’A’-’F’, ’u’, ’+’ */

      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);
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.18.2.3 main()

int main (
int argc,
char ∗ argv[])

The main function.

Parameters

in	argc	The count of command line arguments.

Parameters

in	argv	Pointer to array of command line arguments.

Returns

This program exits with status 0.

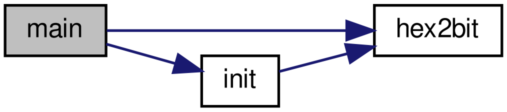
Definition at line 96 of file unihex2bmp.c.

```
{

 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);
}
```

Here is the call graph for this function:



5.18.3 Variable Documentation

5.18.3.1 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.

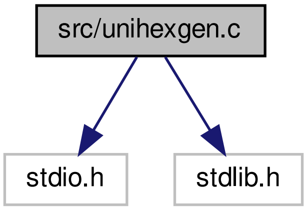
These are the GNU Unifont hex strings for '0'-'9' and 'A'-'F', for encoding as bit strings in row and column headers.

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.

5.19 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.

5.19.1 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.

5.19.2 Function Documentation

5.19.2.1 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.

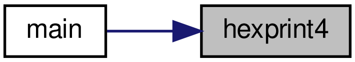
Parameters

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:



5.19.2.2 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

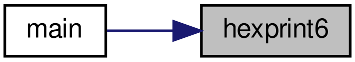
in	thiscp	The current code point for which to generate a glyph.

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] ˆ= 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:



5.19.2.3 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

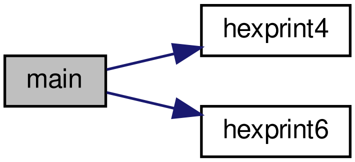
This program exits with status EXIT_SUCCESS.

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:



5.19.3 Variable Documentation

5.19.3.1 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:

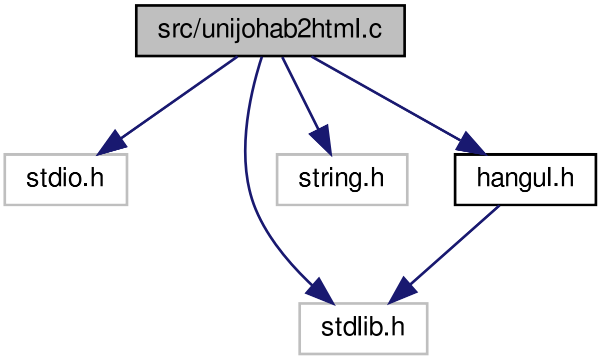
{0x2,0x6,0x2,0x2,0x7},

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.

5.20 src/unijohab2html.c File Reference

Display overalpped Hangul letter combinations in a grid.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hangul.h"
Include dependency graph for unijohab2html.c:



Macros

- #define MAXFILENAME 1024
- #define START_JUNG 0

  Vowel index of first vowel with which to begin.
- #define RED 0xCC0000

  Color code for slightly unsaturated HTML red.
- #define GREEN 0x00CC00

  Color code for slightly unsaturated HTML green.
- #define BLUE 0x0000CC

  Color code for slightly unsaturated HTML blue.
- #define BLACK 0x000000

  Color code for HTML black.
- #define WHITE 0xFFFFFF

  Color code for HTML white.

Functions

- int main (int argc, char ∗argv[ ])

  The main function.
- void parse_args (int argc, char ∗argv[ ], int ∗inindex, int ∗outindex, int ∗modern_only)

  Parse command line arguments.

5.20.1 Detailed Description

Display overalpped Hangul letter combinations in a grid.

This displays overlapped letters that form Unicode Hangul Syllables combinations, as a tool to determine bounding boxes for all combinations. It works with both modern and archaic Hangul letters.

Input is a Unifont .hex file such as the "hangul-base.hex" file that is part of the Unifont package. Glyphs are all processed as being 16 pixels wide and 16 pixels tall.

Output is an HTML file containing 16 by 16 pixel grids shwoing overlaps in table format, arranged by variation of the initial consonant (choseong).

Initial consonants (choseong) have 6 variations. In general, the first three are for combining with vowels (jungseong) that are vertical, horizontal, or vertical and horizontal, respectively; the second set of three variations are for combinations with a final consonant.

The output HTML file can be viewed in a web browser.

Author

Paul Hardy

Copyright

Copyright © 2023 Paul Hardy

5.20.2 Function Documentation

5.20.2.1 parse_args()

```
void parse_args (
                    int argc,
                    char ∗ argv[ ],
                    int ∗ inindex,
                    int ∗ outindex,
                    int ∗ modern_only )
```

Parse command line arguments.

Parameters

in	argc	The argc parameter to the main function.
in	argv	The argv command line arguments to the main function.
in,out	infile	The input filename; defaults to NULL.
in,out	outfile	The output filename; defaults to NULL.

Definition at line 608 of file unijohab2html.c.

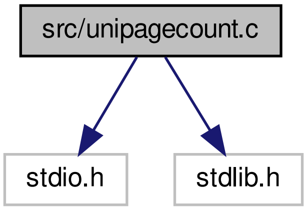
```
{
   int arg_count;  /* Current index into argv[]. */

   int strncmp (const char *s1, const char *s2, size_t n);


   arg_count = 1;

   while (arg_count < argc) {
      /* If input file is specified, open it for read access. */
      if (strncmp (argv [arg_count], ”-i”, 2) == 0) {
         arg_count++;
         if (arg_count < argc) {
            *inindex = arg_count;
         }
      }
      /* If only modern Hangul is desired, set modern_only flag. */
      else if (strncmp (argv [arg_count], ”-m”, 2) == 0 ||
               strncmp (argv [arg_count], ”--modern”, 8) == 0) {
         *modern_only = 1;
      }
      /* If output file is specified, open it for write access. */
      else if (strncmp (argv [arg_count], ”-o”, 2) == 0) {
         arg_count++;
         if (arg_count < argc) {
            *outindex = arg_count;
         }
      }
      /* If help is requested, print help message and exit. */
      else if (strncmp (argv [arg_count], ”-h”,     2) == 0 ||
               strncmp (argv [arg_count], ”--help”, 6) == 0) {
         printf (”\nunijohab2html [options]\n\n”);
         printf (”    Generates an HTML page of overlapping Hangul letters from an input\n”);
         printf (”    Unifont .hex file encoded in Johab 6/3/1 format.\n\n”);

         printf (”    Option      Parameters  Function\n”);
         printf (”    ------      ----------  --------\n”);
         printf (”    -h, --help              Print this message and exit.\n\n”);
         printf (”    -i          input_file  Unifont hangul-base.hex formatted input file.\n\n”);
         printf (”    -o          output_file HTML output file showing overlapping letters.\n\n”);
         printf (”    -m, --modern            Only examine modern Hangul letters.\n\n”);
         printf (”    Example:\n\n”);
         printf (”        unijohab2html -i hangul-base.hex -o hangul-syllables.html\n\n”);

         exit (EXIT_SUCCESS);
      }

      arg_count++;
   }

   return;
}
```

5.21 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.

5.21.1 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
```

5.21.2 Function Documentation

5.21.2.1 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

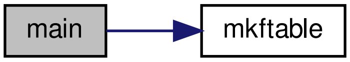
This program exits with status 0.

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:



5.21.2.2 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).

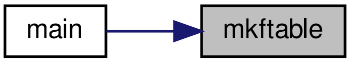
Parameters

in	plane	The Unicode plane, 0..17.
in	pagecount	Array with count of glyphs in each 256 code point range.
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:



Index