GNU Unifont

16.0.01

Generated by Doxygen 1.9.6

Chapter 1

GNU Unifont

1.1 GNU Unifont C Utilities

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

1.2 LICENSE

1.3 Introduction

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

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

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

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

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

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

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

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

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.

4.1.2 Field Documentation

4.1.2.1 begin

byte∗ Buffer::begin

Definition at line 136 of file hex2otf.c.

4.1.2.2 capacity

size_t Buffer::capacity

Definition at line 135 of file hex2otf.c.

4.1.2.3 end

byte ∗ Buffer::end

Definition at line 136 of file hex2otf.c.

4.1.2.4 next

byte ∗ Buffer::next

Definition at line 136 of file hex2otf.c.

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

- src/hex2otf.c

4.2 Font Struct Reference

Data structure to hold information for one font.

Collaboration diagram for Font:

Data Fields

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

4.2.1 Detailed Description

Data structure to hold information for one font.

Definition at line 628 of file hex2otf.c.

4.2.2 Field Documentation

4.2.2.1 glyphCount

uint_fast32_t Font::glyphCount

Definition at line 632 of file hex2otf.c.

4.2.2.2 glyphs

Buffer∗ Font::glyphs

Definition at line 631 of file hex2otf.c.

4.2.2.3 maxWidth

pixels_t Font::maxWidth

Definition at line 633 of file hex2otf.c.

4.2.2.4 tables

Buffer∗ Font::tables

Definition at line 630 of file hex2otf.c.

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

- src/hex2otf.c

4.3 Glyph Struct Reference

Data structure to hold data for one bitmap glyph.

Data Fields

- uint_least32_t codePoint

  undefined for glyph 0
- byte bitmap [GLYPH_MAX_BYTE_COUNT]

  hexadecimal bitmap character array
- uint_least8_t byteCount

  length of bitmap data
- bool combining

  whether this is a combining glyph
- pixels_t pos
- pixels_t lsb

  left side bearing (x position of leftmost contour point)

4.3.1 Detailed Description

Data structure to hold data for one bitmap glyph.

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

Definition at line 614 of file hex2otf.c.

4.3.2 Field Documentation

4.3.2.1 bitmap

byte Glyph::bitmap[GLYPH_MAX_BYTE_COUNT]

hexadecimal bitmap character array

Definition at line 617 of file hex2otf.c.

4.3.2.2 byteCount

uint_least8_t Glyph::byteCount

length of bitmap data

Definition at line 618 of file hex2otf.c.

4.3.2.3 codePoint

uint_least32_t Glyph::codePoint

undefined for glyph 0

Definition at line 616 of file hex2otf.c.

4.3.2.4 combining

bool Glyph::combining

whether this is a combining glyph

Definition at line 619 of file hex2otf.c.

4.3.2.5 lsb

pixels_t Glyph::lsb

left side bearing (x position of leftmost contour point)

Definition at line 622 of file hex2otf.c.

4.3.2.6 pos

pixels_t Glyph::pos

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

Definition at line 620 of file hex2otf.c.

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

- src/hex2otf.c

4.4 NamePair Struct Reference

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

#include <hex2otf.h>

Data Fields

- int id
- const char ∗ str

4.4.1 Detailed Description

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

Definition at line 77 of file hex2otf.h.

4.4.2 Field Documentation

4.4.2.1 id

int NamePair::id

Definition at line 79 of file hex2otf.h.

4.4.2.2 str

const char∗ NamePair::str

Definition at line 80 of file hex2otf.h.

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

- src/hex2otf.h

4.5 Options Struct Reference

Data structure to hold options for OpenType font output.

Data Fields

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

4.5.1 Detailed Description

Data structure to hold options for OpenType font output.

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

Definition at line 2453 of file hex2otf.c.

4.5.2 Field Documentation

4.5.2.1 bitmap

bool Options::bitmap

Definition at line 2455 of file hex2otf.c.

4.5.2.2 blankOutline

bool Options::blankOutline

Definition at line 2455 of file hex2otf.c.

4.5.2.3 cff

int Options::cff

Definition at line 2456 of file hex2otf.c.

4.5.2.4 gpos

bool Options::gpos

Definition at line 2455 of file hex2otf.c.

4.5.2.5 gsub

bool Options::gsub

Definition at line 2455 of file hex2otf.c.

4.5.2.6 hex

const char* Options::hex

Definition at line 2457 of file hex2otf.c.

4.5.2.7 nameStrings

NameStrings Options::nameStrings

Definition at line 2458 of file hex2otf.c.

4.5.2.8 out

const char * Options::out

Definition at line 2457 of file hex2otf.c.

4.5.2.9 pos

const char * Options::pos

Definition at line 2457 of file hex2otf.c.

4.5.2.10 truetype

bool Options::truetype

Definition at line 2455 of file hex2otf.c.

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

- src/hex2otf.c

4.6 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.

4.6.2 Field Documentation

4.6.2.1 cho_end

unsigned PARAMS::cho_end

Definition at line 57 of file unigen-hangul.c.

4.6.2.2 cho_start

unsigned PARAMS::cho_start

Definition at line 57 of file unigen-hangul.c.

4.6.2.3 infp

FILE∗ PARAMS::infp

Definition at line 60 of file unigen-hangul.c.

4.6.2.4 jong_end

unsigned PARAMS::jong_end

Definition at line 59 of file unigen-hangul.c.

4.6.2.5 jong_start

unsigned PARAMS::jong_start

Definition at line 59 of file unigen-hangul.c.

4.6.2.6 jung_end

unsigned PARAMS::jung_end

Definition at line 58 of file unigen-hangul.c.

4.6.2.7 jung_start

unsigned PARAMS::jung_start

Definition at line 58 of file unigen-hangul.c.

4.6.2.8 outfp

FILE∗ PARAMS::outfp

Definition at line 61 of file unigen-hangul.c.

4.6.2.9 starting_codept

unsigned PARAMS::starting_codept

Definition at line 56 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.

4.7.2 Field Documentation

4.7.2.1 content

Buffer* Table::content

Definition at line 648 of file hex2otf.c.

4.7.2.2 tag

uint_fast32_t Table::tag

Definition at line 647 of file hex2otf.c.

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

- src/hex2otf.c

4.8 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.

4.8.2 Field Documentation

4.8.2.1 checksum

uint_least32_t TableRecord::checksum

Definition at line 749 of file hex2otf.c.

4.8.2.2 length

uint_least32_t TableRecord::length

Definition at line 749 of file hex2otf.c.

4.8.2.3 offset

uint_least32_t TableRecord::offset

Definition at line 749 of file hex2otf.c.

4.8.2.4 tag

uint_least32_t TableRecord::tag

Definition at line 749 of file hex2otf.c.

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

- src/hex2otf.c

Chapter 5

File Documentation

5.1 src/hangul.h File Reference

Define constants and function prototypes for using Hangul glyphs.

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

5.2 hangul.h

Go to the documentation of this file.

```
/**
   @file hangul.h

   @brief Define constants and function prototypes for using Hangul glyphs.

   @author Paul Hardy

   @copyright Copyright © 2023 Paul Hardy
*/
/*
   LICENSE:

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

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

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

#ifndef __HANGUL_H__
#define __HANGUL_H__

#include <stdlib.h>


#define MAXLINE     256 ///< Length of maximum file input line.

```

```
#define EXTENDED_HANGUL /* Use rare Hangul code points beyond U+1100 */

/* Definitions to move Hangul .hex file contents into the Private Use Area. */
#define PUA_START   0xE000
#define PUA_END     0xE8FF
#define MAX_GLYPHS  (PUA_END - PUA_START + 1) /* Maximum .hex file glyphs */

/*
   Unicode ranges for Hangul choseong, jungseong, and jongseong.

   U+1100..U+11FF is the main range of modern and ancient Hangul jamo.
   U+A960..U+A97C is the range for extended Hangul choseong.
   U+D7B0..U+D7C6 is the range for extended Hangul jungseong.
   U+D7CB..U+D7FB is the range for extended Hangul jongseong.
*/
#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


/*
   Number of modern and ancient letters in hangul-base.hex file.
*/
#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


/*
   Number of variations of each component in a Johab 6/3/1 arrangement.
*/
#define CHO_VARIATIONS  6   ///< 6 choseong variations
#define JUNG_VARIATIONS 3   ///< 3 jungseong variations
#define JONG_VARIATIONS 1   ///< 1 jongseong variation

/*
   Starting positions in the hangul-base.hex file for each component.
*/
/// Location of first choseong (location 0x0000 is a blank glyph)
#define CHO_HEX      0x0001

/// Location of first ancient choseong
#define CHO_ANCIENT_HEX (CHO_HEX         + CHO_VARIATIONS * NCHO_MODERN)

/// U+A960 Extended-A choseong
#define CHO_EXTA_HEX    (CHO_ANCIENT_HEX + CHO_VARIATIONS * NCHO_ANCIENT)

/// U+A97F Extended-A last location in .hex file, including reserved Unicode code points at end
#define CHO_LAST_HEX    (CHO_EXTA_HEX    + CHO_VARIATIONS * (NCHO_EXTA + NCHO_EXTA_RSRVD) - 1)

/// Location of first jungseong (will be 0x2FB)
#define JUNG_HEX    (CHO_LAST_HEX + 1)

/// Location of first ancient jungseong
#define JUNG_ANCIENT_HEX (JUNG_HEX         + JUNG_VARIATIONS * NJUNG_MODERN)

/// U+D7B0 Extended-B jungseong
#define JUNG_EXTB_HEX    (JUNG_ANCIENT_HEX + JUNG_VARIATIONS * NJUNG_ANCIENT)

```

```
/// U+D7CA Extended-B last location in .hex file, including reserved Unicode code points at end
#define JUNG_LAST_HEX   (JUNG_EXTB_HEX     + JUNG_VARIATIONS * (NJUNG_EXTB +
    NJUNG_EXTB_RSRVD) - 1)

/// Location of first jongseong (will be 0x421)
#define JONG_HEX     (JUNG_LAST_HEX + 1)

/// Location of first ancient jongseong
#define JONG_ANCIENT_HEX (JONG_HEX         + JONG_VARIATIONS * NJONG_MODERN)

/// U+D7CB Extended-B jongseong
#define JONG_EXTB_HEX    (JONG_ANCIENT_HEX + JONG_VARIATIONS * NJONG_ANCIENT)

/// U+D7FF Extended-B last location in .hex file, including reserved Unicode code points at end
#define JONG_LAST_HEX   (JONG_EXTB_HEX     + JONG_VARIATIONS * (NJONG_EXTB +
    NJONG_EXTB_RSRVD) - 1)

/* Common modern and ancient Hangul Jamo range */
#define JAMO_HEX    0x0500  ///< Start of U+1100..U+11FF glyphs
#define JAMO_END    0x05FF  ///< End   of U+1100..U+11FF glyphs

/* Hangul Jamo Extended-A range */
#define JAMO_EXTA_HEX   0x0600  ///< Start of U+A960..U+A97F glyphs
#define JAMO_EXTA_END   0x061F  ///< End   of U+A960..U+A97F glyphs

/* Hangul Jamo Extended-B range */
#define JAMO_EXTB_HEX   0x0620  ///< Start of U+D7B0..U+D7FF glyphs
#define JAMO_EXTB_END   0x066F  ///< End   of U+D7B0..U+D7FF glyphs

/*
   These values allow enumeration of all modern and ancient letters.

   If RARE_HANGUL is defined, include Hangul code points above U+11FF.
*/
#ifdef EXTENDED_HANGUL

#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)

#else

#define TOTAL_CHO   (NCHO_MODERN  + NCHO_ANCIENT )
#define TOTAL_JUNG      (NJUNG_MODERN + NJUNG_ANCIENT)
#define TOTAL_JONG  (NJONG_MODERN + NJONG_ANCIENT)

#endif


/*
   Function Prototypes.
*/

unsigned hangul_read_base8  (FILE *infp,  unsigned char base[][32]);
unsigned hangul_read_base16 (FILE *infp, unsigned base[][16]);

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

void hangul_hex_indices (int choseong, int jungseong, int jongseong,
                         int *cho_index, int *jung_index, int *jong_index);
void hangul_variations (int choseong, int jungseong, int jongseong,
                        int *cho_var, int *jung_var, int *jong_var);
int is_wide_vowel  (int vowel);
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);

void hangul_syllable (int choseong, int jungseong, int jongseong,
                      unsigned char hangul_base[][32], unsigned char *syllable);
int glyph_overlap (unsigned *glyph1, unsigned *glyph2);
void combine_glyphs (unsigned *glyph1, unsigned *glyph2,
                     unsigned *combined_glyph);
void one_jamo (unsigned glyph_table [MAX_GLYPHS][16],
               unsigned jamo, unsigned *jamo_glyph);
void combined_jamo (unsigned glyph_table [MAX_GLYPHS][16],
                    unsigned cho, unsigned jung, unsigned jong,
                    unsigned *combined_glyph);
void print_glyph_txt (FILE *fp, unsigned codept, unsigned *this_glyph);
void print_glyph_hex (FILE *fp, unsigned codept, unsigned *this_glyph);
```

```
#endif
```

5.3 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.

5.3.1 Detailed Description

hex2otf - Convert GNU Unifont .hex file to OpenType font

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

Copyright

Author

何志翔 (He Zhixiang)

Definition in file hex2otf.c.

5.3.2 Macro Definition Documentation

5.3.2.1 addByte

#define addByte(
shift)

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

5.3.2.2 ASCENDER

#define ASCENDER (GLYPH_HEIGHT - DESCENDER)

Count of pixels above baseline.

Definition at line 79 of file hex2otf.c.

5.3.2.3 B0

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

Clear a given bit in a word.

Definition at line 66 of file hex2otf.c.

5.3.2.4 B1

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

Set a given bit in a word.

Definition at line 67 of file hex2otf.c.

5.3.2.5 BX

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

Truncate & shift word.

Definition at line 65 of file hex2otf.c.

5.3.2.6 defineStore

```
#define defineStore(
        name,
        type )
```

Value:

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

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

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

Definition at line 350 of file hex2otf.c.

5.3.2.7 DESCENDER

```
#define DESCENDER 2
```

Count of pixels below baseline.

Definition at line 76 of file hex2otf.c.

5.3.2.8 FU

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

Convert pixels to font units.

Definition at line 91 of file hex2otf.c.

5.3.2.9 FUPEM

#define FUPEM 64

Font units per em.

Definition at line 82 of file hex2otf.c.

5.3.2.10 GLYPH_HEIGHT

#define GLYPH_HEIGHT 16

Maximum glyph height, in pixels.

Definition at line 70 of file hex2otf.c.

5.3.2.11 GLYPH_MAX_BYTE_COUNT

#define GLYPH_MAX_BYTE_COUNT (GLYPH_HEIGHT * GLYPH_MAX_WIDTH / 8)

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

Definition at line 73 of file hex2otf.c.

5.3.2.12 GLYPH_MAX_WIDTH

#define GLYPH_MAX_WIDTH 16

Maximum glyph width, in pixels.

Definition at line 69 of file hex2otf.c.

5.3.2.13 MAX_GLYPHS

#define MAX_GLYPHS 65536

An OpenType font has at most 65536 glyphs.

Definition at line 85 of file hex2otf.c.

5.3.2.14 MAX_NAME_IDS

#define MAX_NAME_IDS 256

Name IDs 0-255 are used for standard names.

Definition at line 88 of file hex2otf.c.

5.3.2.15 PRI_CP

#define PRI_CP "U+%.4"PRIXFAST32

Format string to print Unicode code point.

Definition at line 58 of file hex2otf.c.

5.3.2.16 PW

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

Convert glyph byte count to pixel width.

Definition at line 94 of file hex2otf.c.

5.3.2.17 static_assert

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

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

Definition at line 61 of file hex2otf.c.

5.3.2.18 U16MAX

#define U16MAX 0xffff

Maximum UTF-16 code point value.

Definition at line 55 of file hex2otf.c.

5.3.2.19 U32MAX

#define U32MAX 0xffffffff

Maximum UTF-32 code point value.

Definition at line 56 of file hex2otf.c.

5.3.2.20 VERSION

#define VERSION "1.0.1"

Program version, for "--version" option.

Definition at line 51 of file hex2otf.c.

5.3.3 Typedef Documentation

5.3.3.1 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.

5.3.3.2 byte

typedef unsigned char byte

Definition of "byte" type as an unsigned char.

Definition at line 97 of file hex2otf.c.

5.3.3.3 Glyph

typedef struct Glyph Glyph

Data structure to hold data for one bitmap glyph.

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

5.3.3.4 NameStrings

typedef const char* NameStrings[MAX_NAME_IDS]

Array of OpenType names indexed directly by Name IDs.

Definition at line 604 of file hex2otf.c.

5.3.3.5 Options

typedef struct Options Options

Data structure to hold options for OpenType font output.

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

5.3.3.6 pixels_t

typedef int_least8_t pixels_t

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

Definition at line 100 of file hex2otf.c.

5.3.3.7 Table

typedef struct Table Table

Data structure for an OpenType table.

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

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

5.3.4 Enumeration Type Documentation

5.3.4.1 ContourOp

enum ContourOp

Specify the current contour drawing operation.

Enumerator

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

Definition at line 1136 of file hex2otf.c.

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

5.3.4.2 FillSide

enum FillSide

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

Enumerator

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

Definition at line 1144 of file hex2otf.c.

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

5.3.4.3 LocaFormat

enum LocaFormat

Index to Location ("loca") offset information.

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

Enumerator

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

Definition at line 658 of file hex2otf.c.

```
00658 {
```

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

5.3.5 Function Documentation

5.3.5.1 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:

5.3.5.2 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
}
```

Here is the caller graph for this function:

5.3.5.3 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;
}
```

Here is the caller graph for this function:

5.3.5.4 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;
}
```

Here is the caller graph for this function:

5.3.5.5 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.

```
00524 {
00525     size_t length = countBufferedBytes (bufSrc);
00526     ensureBuffer (bufDest, length);
00527     memcpy (bufDest->next, bufSrc->begin, length);
00528     bufDest->next += length;
00529 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.6 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.

```
00510 {
00511     ensureBuffer (buf, count);
00512     memcpy (buf->next, src, count);
00513     buf->next += count;
00514 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.7 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.

```
00461 {
00462     if (-107 <= value && value <= 107)
00463         cacheU8 (buf, value + 139);
00464     else if (108 <= value && value <= 1131)
00465     {
00466         cacheU8 (buf, (value - 108) / 256 + 247);
00467         cacheU8 (buf, (value - 108) % 256);
00468     }
00469     else if (-32768 <= value && value <= 32767)
00470     {
00471         cacheU8 (buf, 28);
00472         cacheU16 (buf, value);
00473     }
00474     else if (-2147483647 <= value && value <= 2147483647)
00475     {
00476         cacheU8 (buf, 29);
00477         cacheU32 (buf, value);
00478     }
00479     else
00480         assert (false); // other encodings are not used and omitted
00481     static_assert (GLYPH_MAX_WIDTH <= 107, "More encodings are needed.");
00482 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.8 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: Here is the caller graph for this function:

5.3.5.9 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.

```
{
    cacheU (buf, value, 2);
}
```

Here is the caller graph for this function:

5.3.5.10 cacheU32()

void cacheU32 (
    Buffer ∗ buf,
    uint_fast32_t value)

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

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

Parameters

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

Definition at line 427 of file hex2otf.c.

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

Here is the caller graph for this function:

5.3.5.11 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.

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

Here is the caller graph for this function:

5.3.5.12 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.

```
{
    ensureBuffer (buf, count);
    memset (buf->next, 0, count);
    buf->next += count;
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.13 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:

5.3.5.14 defineStore()

defineStore (
    storeU8 ,
    uint_least8_t)

Definition at line 356 of file hex2otf.c.

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

5.3.5.15 ensureBuffer()

void ensureBuffer (
 Buffer * buf,
 size_t needed)

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

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

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

Parameters

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

Definition at line 239 of file hex2otf.c.

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

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.16 fail()

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

Print an error message on stderr, then exit.

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

Parameters

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

Definition at line 113 of file hex2otf.c.

```
{
    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:

5.3.5.17 fillBitmap()

void fillBitmap (
    Font * font)

Fill OpenType bitmap data and location tables.

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

Parameters

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

Definition at line 1728 of file hex2otf.c.

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

```
    { // bitmapSizes[0]
        cacheU32 (eblc, 56); // indexSubTableArrayOffset
        cacheU32 (eblc, (8 + 20) * rangeCount); // indexTablesSize
        cacheU32 (eblc, rangeCount); // numberOfIndexSubTables
        cacheU32 (eblc, 0); // colorRef
        { // hori
            cacheU8 (eblc, ASCENDER); // ascender
            cacheU8 (eblc, -DESCENDER); // descender
            cacheU8 (eblc, font->maxWidth); // widthMax
            cacheU8 (eblc, 1); // caretSlopeNumerator
            cacheU8 (eblc, 0); // caretSlopeDenominator
            cacheU8 (eblc, 0); // caretOffset
            cacheU8 (eblc, 0); // minOriginSB
            cacheU8 (eblc, 0); // minAdvanceSB
            cacheU8 (eblc, ASCENDER); // maxBeforeBL
            cacheU8 (eblc, -DESCENDER); // minAfterBL
            cacheU8 (eblc, 0); // pad1
            cacheU8 (eblc, 0); // pad2
        }
        { // vert
            cacheU8 (eblc, ASCENDER); // ascender
            cacheU8 (eblc, -DESCENDER); // descender
            cacheU8 (eblc, font->maxWidth); // widthMax
            cacheU8 (eblc, 1); // caretSlopeNumerator
            cacheU8 (eblc, 0); // caretSlopeDenominator
            cacheU8 (eblc, 0); // caretOffset
            cacheU8 (eblc, 0); // minOriginSB
            cacheU8 (eblc, 0); // minAdvanceSB
            cacheU8 (eblc, ASCENDER); // maxBeforeBL
            cacheU8 (eblc, -DESCENDER); // minAfterBL
            cacheU8 (eblc, 0); // pad1
            cacheU8 (eblc, 0); // pad2
        }
        cacheU16 (eblc, 0); // startGlyphIndex
        cacheU16 (eblc, font->glyphCount - 1); // endGlyphIndex
        cacheU8 (eblc, 16); // ppemX
        cacheU8 (eblc, 16); // ppemY
        cacheU8 (eblc, 1); // bitDepth
        cacheU8 (eblc, 1); // flags = Horizontal
    }
    { // IndexSubTableArray
        uint_fast32_t offset = rangeCount * 8;
        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
        {
            cacheU16 (eblc, *p); // firstGlyphIndex
            cacheU16 (eblc, p[1] - 1); // lastGlyphIndex
            cacheU32 (eblc, offset); // additionalOffsetToIndexSubtable
            offset += 20;
        }
    }
    { // IndexSubTables
        const uint_least32_t *offset = getBufferHead (offsets);
        for (const uint_least16_t *p = ranges; p < rangesEnd; p++)
        {
            const Glyph *glyph = &glyphs[*p];
            cacheU16 (eblc, 2); // indexFormat
            cacheU16 (eblc, 5); // imageFormat
            cacheU32 (eblc, *offset++); // imageDataOffset
            cacheU32 (eblc, glyph->byteCount); // imageSize
            { // bigMetrics
                cacheU8 (eblc, GLYPH_HEIGHT); // height
                const uint_fast8_t width = PW (glyph->byteCount);
                cacheU8 (eblc, width); // width
                cacheU8 (eblc, glyph->pos); // horiBearingX
                cacheU8 (eblc, ASCENDER); // horiBearingY
                cacheU8 (eblc, glyph->combining ? 0 : width); // horiAdvance
                cacheU8 (eblc, 0); // vertBearingX
                cacheU8 (eblc, 0); // vertBearingY
                cacheU8 (eblc, GLYPH_HEIGHT); // vertAdvance
            }
        }
    }
    freeBuffer (rangeHeads);
    freeBuffer (offsets);
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.18 fillBlankOutline()

void fillBlankOutline (
Font ∗ font)

Create a dummy blank outline in a font table.

Parameters

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

Definition at line 1697 of file hex2otf.c.

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

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.19 fillCFF()

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

Add a CFF table to a font.

Parameters

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

Use fixed width integer for variables to simplify offset calculation.

Definition at line 1329 of file hex2otf.c.

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

```

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

// In Unifont, 16px glyphs are more common. This is used by CFF1 only.
const pixels_t defaultWidth = 16, nominalWidth = 8;
if (version == 1)
{
    Buffer *strings = prepareStringIndex (names);
    size_t stringsSize = countBufferedBytes (strings);
    const char *cffName = names[6];
    assert (cffName);
    size_t nameLength = strlen (cffName);
    size_t namesSize = nameLength + 5;
    // These sizes must be updated together with the data below.
    size_t offsets[] = {4, namesSize, 45, stringsSize, 2, 5, 8, 32, 4, 0};
    prepareOffsets (offsets);
    { // Header
        cacheU8 (cff, 1); // major
        cacheU8 (cff, 0); // minor
        cacheU8 (cff, 4); // hdrSize
        cacheU8 (cff, 1); // offSize
    }
    assert (countBufferedBytes (cff) == offsets[0]);
    { // Name INDEX (should not be used by OpenType readers)
        cacheU16 (cff, 1); // count
        cacheU8 (cff, 1); // offSize
        cacheU8 (cff, 1); // offset[0]
        if (nameLength + 1 > 255) // must be too long; spec limit is 63
            fail ("PostScript name is too long.");
        cacheU8 (cff, nameLength + 1); // offset[1]
        cacheBytes (cff, cffName, nameLength);
    }
    assert (countBufferedBytes (cff) == offsets[1]);
    { // Top DICT INDEX
        cacheU16 (cff, 1); // count
        cacheU8 (cff, 1); // offSize
        cacheU8 (cff, 1); // offset[0]
        cacheU8 (cff, 41); // offset[1]
        cacheCFFOperand (cff, 391); // "Adobe"
        cacheCFFOperand (cff, 392); // "Identity"
        cacheCFFOperand (cff, 0);
        cacheBytes (cff, (byte[]){12, 30}, 2); // ROS
        cacheCFF32 (cff, font->glyphCount);
        cacheBytes (cff, (byte[]){12, 34}, 2); // CIDCount
        cacheCFF32 (cff, offsets[6]);
        cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
        cacheCFF32 (cff, offsets[5]);
        cacheBytes (cff, (byte[]){12, 37}, 2); // FDSelect
        cacheCFF32 (cff, offsets[4]);
        cacheU8 (cff, 15); // charset
        cacheCFF32 (cff, offsets[8]);
        cacheU8 (cff, 17); // CharStrings
    }
    assert (countBufferedBytes (cff) == offsets[2]);
    { // String INDEX
        cacheBuffer (cff, strings);
        freeBuffer (strings);
    }
    assert (countBufferedBytes (cff) == offsets[3]);
    cacheU16 (cff, 0); // Global Subr INDEX
    assert (countBufferedBytes (cff) == offsets[4]);
    { // Charsets
        cacheU8 (cff, 2); // format
        { // Range2[0]
            cacheU16 (cff, 1); // first
            cacheU16 (cff, font->glyphCount - 2); // nLeft
        }
    }
    assert (countBufferedBytes (cff) == offsets[5]);
    { // FDSelect
        cacheU8 (cff, 3); // format
        cacheU16 (cff, 1); // nRanges
        cacheU16 (cff, 0); // first
        cacheU8 (cff, 0); // fd
        cacheU16 (cff, font->glyphCount); // sentinel
    }
    assert (countBufferedBytes (cff) == offsets[6]);
    { // FDArray
        cacheU16 (cff, 1); // count
        cacheU8 (cff, 1); // offSize
        cacheU8 (cff, 1); // offset[0]
```

```
            cacheU8 (cff, 28); // offset[1]
            cacheCFFOperand (cff, 393);
            cacheBytes (cff, (byte[]){12, 38}, 2); // FontName
            // Windows requires FontMatrix in Font DICT.
            const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
            cacheCFFOperand (cff, offsets[8] - offsets[7]); // size
            cacheCFF32 (cff, offsets[7]); // offset
            cacheU8 (cff, 18); // Private
        }
        assert (countBufferedBytes (cff) == offsets[7]);
        { // Private
            cacheCFFOperand (cff, FU (defaultWidth));
            cacheU8 (cff, 20); // defaultWidthX
            cacheCFFOperand (cff, FU (nominalWidth));
            cacheU8 (cff, 21); // nominalWidthX
        }
        assert (countBufferedBytes (cff) == offsets[8]);
    }
    else
    {
        assert (version == 2);
        // These sizes must be updated together with the data below.
        size_t offsets[] = {5, 21, 4, 10, 0};
        prepareOffsets (offsets);
        { // Header
            cacheU8 (cff, 2); // majorVersion
            cacheU8 (cff, 0); // minorVersion
            cacheU8 (cff, 5); // headerSize
            cacheU16 (cff, offsets[1] - offsets[0]); // topDictLength
        }
        assert (countBufferedBytes (cff) == offsets[0]);
        { // Top DICT
            const byte unit[] = {0x1e,0x15,0x62,0x5c,0x6f}; // 1/64 (0.015625)
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, unit, sizeof unit);
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheBytes (cff, (byte[]){12, 7}, 2); // FontMatrix
            cacheCFFOperand (cff, offsets[2]);
            cacheBytes (cff, (byte[]){12, 36}, 2); // FDArray
            cacheCFFOperand (cff, offsets[3]);
            cacheU8 (cff, 17); // CharStrings
        }
        assert (countBufferedBytes (cff) == offsets[1]);
        cacheU32 (cff, 0); // Global Subr INDEX
        assert (countBufferedBytes (cff) == offsets[2]);
        { // Font DICT INDEX
            cacheU32 (cff, 1); // count
            cacheU8 (cff, 1); // offSize
            cacheU8 (cff, 1); // offset[0]
            cacheU8 (cff, 4); // offset[1]
            cacheCFFOperand (cff, 0);
            cacheCFFOperand (cff, 0);
            cacheU8 (cff, 18); // Private
        }
        assert (countBufferedBytes (cff) == offsets[3]);
    }
    { // CharStrings INDEX
        Buffer *offsets = newBuffer (4096);
        Buffer *charstrings = newBuffer (4096);
        Buffer *outline = newBuffer (1024);
        const Glyph *glyph = getBufferHead (font->glyphs);
        const Glyph *const endGlyph = glyph + font->glyphCount;
        for (; glyph < endGlyph; glyph++)
        {
            // CFF offsets start at 1
            storeU32 (offsets, countBufferedBytes (charstrings) + 1);

            pixels_t rx = -glyph->pos;
            pixels_t ry = DESCENDER;
            resetBuffer (outline);
            buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_LEFT);
```

```
        enum CFFOp {rmoveto=21, hmoveto=22, vmoveto=4, hlineto=6,
            vlineto=7, endchar=14};
        enum CFFOp pendingOp = 0;
        const int STACK_LIMIT = version == 1 ? 48 : 513;
        int stackSize = 0;
        bool isDrawing = false;
        pixels_t width = glyph->combining ? 0 : PW (glyph->byteCount);
        if (version == 1 && width != defaultWidth)
        {
            cacheCFFOperand (charstrings, FU (width - nominalWidth));
            stackSize++;
        }
        for (const pixels_t *p = getBufferHead (outline),
             *const end = getBufferTail (outline); p < end;)
        {
            int s = 0;
            const enum ContourOp op = *p++;
            if (op == OP_POINT)
            {
                const pixels_t x = *p++, y = *p++;
                if (x != rx)
                {
                    cacheCFFOperand (charstrings, FU (x - rx));
                    rx = x;
                    stackSize++;
                    s |= 1;
                }
                if (y != ry)
                {
                    cacheCFFOperand (charstrings, FU (y - ry));
                    ry = y;
                    stackSize++;
                    s |= 2;
                }
                assert (!(isDrawing && s == 3));
            }
            if (s)
            {
                if (!isDrawing)
                {
                    const enum CFFOp moves[] = {0, hmoveto, vmoveto,
                        rmoveto};
                    cacheU8 (charstrings, moves[s]);
                    stackSize = 0;
                }
                else if (!pendingOp)
                    pendingOp = (enum CFFOp[]){0, hlineto, vlineto}[s];
            }
            else if (!isDrawing)
            {
                // only when the first point happens to be (0, 0)
                cacheCFFOperand (charstrings, FU (0));
                cacheU8 (charstrings, hmoveto);
                stackSize = 0;
            }
            if (op == OP_CLOSE || stackSize >= STACK_LIMIT)
            {
                assert (stackSize <= STACK_LIMIT);
                cacheU8 (charstrings, pendingOp);
                pendingOp = 0;
                stackSize = 0;
            }
            isDrawing = op != OP_CLOSE;
        }
        if (version == 1)
            cacheU8 (charstrings, endchar);
    }
    size_t lastOffset = countBufferedBytes (charstrings) + 1;
    #if SIZE_MAX > U32MAX
        if (lastOffset > U32MAX)
            fail ("CFF data exceeded size limit.");
    #endif
    storeU32 (offsets, lastOffset);
    int offsetSize = 1 + (lastOffset > 0xff)
                       + (lastOffset > 0xffff)
                       + (lastOffset > 0xffffff);
    // count (must match 'numGlyphs' in 'maxp' table)
    cacheU (cff, font->glyphCount, version * 2);
    cacheU8 (cff, offsetSize); // offSize
    const uint_least32_t *p = getBufferHead (offsets);
    const uint_least32_t *const end = getBufferTail (offsets);
```

```
        for (; p < end; p++)
            cacheU (cff, *p, offsetSize); // offsets
        cacheBuffer (cff, charstrings); // data
        freeBuffer (offsets);
        freeBuffer (charstrings);
        freeBuffer (outline);
    }
    #undef cacheCFF32
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.20 fillCmapTable()

void fillCmapTable (
    Font ∗ font)

Fill a ”cmap” font table.

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

Parameters

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

Definition at line 2109 of file hex2otf.c.

```
{
    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 call graph for this function: Here is the caller graph for this function:

5.3.5.21 fillGposTable()

void fillGposTable (
    Font * font)

Fill a "GPOS" font table.

The "GPOS" table contains information for glyph positioning.

Parameters

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

Definition at line 2241 of file hex2otf.c.

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

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.22 fillGsubTable()

void fillGsubTable (
    Font ∗ font)

Fill a "GSUB" font table.

The "GSUB" table contains information for glyph substitution.

Parameters

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

Definition at line 2269 of file hex2otf.c.

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

```
            cacheU16 (gsub, 0); // requiredFeatureIndex
            cacheU16 (gsub, 0); // featureIndexCount
        }
    }
}
{ // Feature List table
    cacheU16 (gsub, 0); // featureCount
}
{ // Lookup List Table
    cacheU16 (gsub, 0); // lookupCount
}
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.23 fillHeadTable()

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

Fill a "head" font table.

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

Parameters

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

Definition at line 1853 of file hex2otf.c.

```
{
    Buffer *head = newBuffer (56);
    addTable (font, "head", head);
    cacheU16 (head, 1); // majorVersion
    cacheU16 (head, 0); // minorVersion
    cacheZeros (head, 4); // fontRevision (unused)
    // The 'checksumAdjustment' field is a checksum of the entire file.
    // It is later calculated and written directly in the 'writeFont' function.
    cacheU32 (head, 0); // checksumAdjustment (placeholder)
    cacheU32 (head, 0x5f0f3cf5); // magicNumber
    const uint_fast16_t flags =
        + B1 ( 0) // baseline at y=0
        + B1 ( 1) // LSB at x=0 (doubtful; probably should be LSB=xMin)
        + B0 ( 2) // instructions may depend on point size
        + B0 ( 3) // force internal ppem to integers
        + B0 ( 4) // instructions may alter advance width
        + B0 ( 5) // not used in OpenType
        + B0 ( 6) // not used in OpenType
        + B0 ( 7) // not used in OpenType
        + B0 ( 8) // not used in OpenType
        + B0 ( 9) // not used in OpenType
        + B0 (10) // not used in OpenType
        + B0 (11) // font transformed
        + B0 (12) // font converted
        + B0 (13) // font optimized for ClearType
        + B0 (14) // last resort font
        + B0 (15) // reserved
    ;
    cacheU16 (head, flags); // flags
    cacheU16 (head, FUPEM); // unitsPerEm
    cacheZeros (head, 8); // created (unused)
    cacheZeros (head, 8); // modified (unused)
```

```
01886     cacheU16 (head, FU (xMin)); // xMin
01887     cacheU16 (head, FU (-DESCENDER)); // yMin
01888     cacheU16 (head, FU (font->maxWidth)); // xMax
01889     cacheU16 (head, FU (ASCENDER)); // yMax
01890     // macStyle (must agree with 'fsSelection' in 'OS/2' table)
01891     const uint_fast16_t macStyle =
01892         + B0 (0) // bold
01893         + B0 (1) // italic
01894         + B0 (2) // underline
01895         + B0 (3) // outline
01896         + B0 (4) // shadow
01897         + B0 (5) // condensed
01898         + B0 (6) // extended
01899         //    7-15  reserved
01900     ;
01901     cacheU16 (head, macStyle);
01902     cacheU16 (head, GLYPH_HEIGHT); // lowestRecPPEM
01903     cacheU16 (head, 2); // fontDirectionHint
01904     cacheU16 (head, locaFormat); // indexToLocFormat
01905     cacheU16 (head, 0); // glyphDataFormat
01906 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.24 fillHheaTable()

void fillHheaTable (
  Font * font,
  pixels_t xMin)

Fill a "hhea" font table.

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

Parameters

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

Definition at line 1918 of file hex2otf.c.

```
01919 {
01920     Buffer *hhea = newBuffer (36);
01921     addTable (font, "hhea", hhea);
01922     cacheU16 (hhea, 1); // majorVersion
01923     cacheU16 (hhea, 0); // minorVersion
01924     cacheU16 (hhea, FU (ASCENDER)); // ascender
01925     cacheU16 (hhea, FU (-DESCENDER)); // descender
01926     cacheU16 (hhea, FU (0)); // lineGap
01927     cacheU16 (hhea, FU (font->maxWidth)); // advanceWidthMax
01928     cacheU16 (hhea, FU (xMin)); // minLeftSideBearing
01929     cacheU16 (hhea, FU (0)); // minRightSideBearing (unused)
01930     cacheU16 (hhea, FU (font->maxWidth)); // xMaxExtent
01931     cacheU16 (hhea, 1); // caretSlopeRise
01932     cacheU16 (hhea, 0); // caretSlopeRun
01933     cacheU16 (hhea, 0); // caretOffset
01934     cacheU16 (hhea, 0); // reserved
01935     cacheU16 (hhea, 0); // reserved
01936     cacheU16 (hhea, 0); // reserved
01937     cacheU16 (hhea, 0); // reserved
01938     cacheU16 (hhea, 0); // metricDataFormat
01939     cacheU16 (hhea, font->glyphCount); // numberOfHMetrics
01940 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.25 fillHmtxTable()

void fillHmtxTable (
    Font ∗ font)

Fill an "hmtx" font table.

The "hmtx" table contains horizontal metrics information.

Parameters

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

Definition at line 2087 of file hex2otf.c.

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

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.26 fillMaxpTable()

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

Fill a "maxp" font table.

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

Parameters

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

Definition at line 1954 of file hex2otf.c.

```
{
    Buffer *maxp = newBuffer (32);
    addTable (font, "maxp", maxp);
    cacheU32 (maxp, isCFF ? 0x00005000 : 0x00010000); // version
    cacheU16 (maxp, font->glyphCount); // numGlyphs
```

```
01961     if (isCFF)
01962         return;
01963     cacheU16 (maxp, maxPoints); // maxPoints
01964     cacheU16 (maxp, maxContours); // maxContours
01965     cacheU16 (maxp, 0); // maxCompositePoints
01966     cacheU16 (maxp, 0); // maxCompositeContours
01967     cacheU16 (maxp, 0); // maxZones
01968     cacheU16 (maxp, 0); // maxTwilightPoints
01969     cacheU16 (maxp, 0); // maxStorage
01970     cacheU16 (maxp, 0); // maxFunctionDefs
01971     cacheU16 (maxp, 0); // maxInstructionDefs
01972     cacheU16 (maxp, 0); // maxStackElements
01973     cacheU16 (maxp, 0); // maxSizeOfInstructions
01974     cacheU16 (maxp, 0); // maxComponentElements
01975     cacheU16 (maxp, 0); // maxComponentDepth
01976 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.27 fillNameTable()

void fillNameTable (
    Font ∗ font,
    NameStrings nameStrings)

Fill a "name" font table.

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

Parameters

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

Definition at line 2366 of file hex2otf.c.

```
02367 {
02368     Buffer *name = newBuffer (2048);
02369     addTable (font, "name", name);
02370     size_t nameStringCount = 0;
02371     for (size_t i = 0; i < MAX_NAME_IDS; i++)
02372         nameStringCount += !!nameStrings[i];
02373     cacheU16 (name, 0); // version
02374     cacheU16 (name, nameStringCount); // count
02375     cacheU16 (name, 2 * 3 + 12 * nameStringCount); // storageOffset
02376     Buffer *stringData = newBuffer (1024);
02377     // nameRecord[]
02378     for (size_t i = 0; i < MAX_NAME_IDS; i++)
02379     {
02380         if (!nameStrings[i])
02381             continue;
02382         size_t offset = countBufferedBytes (stringData);
02383         cacheStringAsUTF16BE (stringData, nameStrings[i]);
02384         size_t length = countBufferedBytes (stringData) - offset;
02385         if (offset > U16MAX || length > U16MAX)
02386             fail ("Name strings are too long.");
02387         // Platform ID 0 (Unicode) is not well supported.
02388         // ID 3 (Windows) seems to be the best for compatibility.
02389         cacheU16 (name, 3); // platformID = Windows
02390         cacheU16 (name, 1); // encodingID = Unicode BMP
02391         cacheU16 (name, 0x0409); // languageID = en-US
02392         cacheU16 (name, i); // nameID
02393         cacheU16 (name, length); // length
02394         cacheU16 (name, offset); // stringOffset
02395     }
02396     cacheBuffer (name, stringData);
02397     freeBuffer (stringData);
```

```
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.28 fillOS2Table()

void fillOS2Table (
    Font ∗ font)

Fill an "OS/2" font table.

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

Parameters

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

Definition at line 1986 of file hex2otf.c.

```
{
    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:

5.3.5.29 fillPostTable()

```
void fillPostTable (
        Font * font )
```

Fill a "post" font table.

The "post" table contains information for PostScript printers.

Parameters

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

Definition at line 2218 of file hex2otf.c.

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

```
    cacheU32 (post, 0); // minMemType42
    cacheU32 (post, 0); // maxMemType42
    cacheU32 (post, 0); // minMemType1
    cacheU32 (post, 0); // maxMemType1
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.30 fillTrueType()

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

Add a TrueType table to a font.

Parameters

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

Definition at line 1597 of file hex2otf.c.

```
{
    Buffer *glyf = newBuffer (65536);
    addTable (font, "glyf", glyf);
    Buffer *loca = newBuffer (4 * (font->glyphCount + 1));
    addTable (font, "loca", loca);
    *format = LOCA_OFFSET32;
    Buffer *endPoints = newBuffer (256);
    Buffer *flags = newBuffer (256);
    Buffer *xs = newBuffer (256);
    Buffer *ys = newBuffer (256);
    Buffer *outline = newBuffer (1024);
    Glyph *const glyphs = getBufferHead (font->glyphs);
    const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
    for (Glyph *glyph = glyphs; glyph < glyphsEnd; glyph++)
    {
        cacheU32 (loca, countBufferedBytes (glyf));
        pixels_t rx = -glyph->pos;
        pixels_t ry = DESCENDER;
        pixels_t xMin = GLYPH_MAX_WIDTH, xMax = 0;
        pixels_t yMin = ASCENDER, yMax = -DESCENDER;
        resetBuffer (endPoints);
        resetBuffer (flags);
        resetBuffer (xs);
        resetBuffer (ys);
        resetBuffer (outline);
        buildOutline (outline, glyph->bitmap, glyph->byteCount, FILL_RIGHT);
        uint_fast32_t pointCount = 0, contourCount = 0;
        for (const pixels_t *p = getBufferHead (outline),
             *const end = getBufferTail (outline); p < end;)
        {
            const enum ContourOp op = *p++;
            if (op == OP_CLOSE)
            {
                contourCount++;
                assert (contourCount <= U16MAX);
                cacheU16 (endPoints, pointCount - 1);
                continue;
            }
            assert (op == OP_POINT);
            pointCount++;
```

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

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.31 freeBuffer()

void freeBuffer (
    Buffer ∗ buf)

Free the memory previously allocated for a buffer.

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

Parameters

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

Definition at line 337 of file hex2otf.c.

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

Here is the caller graph for this function:

5.3.5.32 initBuffers()

void initBuffers (
    size_t count)

Initialize an array of buffer pointers to all zeroes.

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

Parameters

in	count	The number of buffer array pointers to allocate.

Definition at line 152 of file hex2otf.c.

```
{
    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:

5.3.5.33 main()

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

The main function.

Parameters

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

Returns

EXIT_FAILURE upon fatal error, EXIT_SUCCESS otherwise.

Definition at line 2603 of file hex2otf.c.

```
{
    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:

5.3.5.34 matchToken()

```
const char * matchToken (
        const char * operand,
        const char * key,
        char delimiter )
```

Match a command line option with its key for enabling.

Parameters

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

Returns

Pointer to the first character of the desired option.

Definition at line 2470 of file hex2otf.c.

```
{
    while (*key)
```

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

Here is the caller graph for this function:

5.3.5.35 newBuffer()

Buffer ∗ newBuffer (
    size_t initialCapacity)

Create a new buffer.

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

Parameters

in	initialCapacity	The initial number of elements in the buffer.

Definition at line 188 of file hex2otf.c.

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

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.36 organizeTables()

void organizeTables (
    Font ∗ font,
    bool isCFF)

Sort tables according to OpenType recommendations.

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

Parameters

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

Definition at line 711 of file hex2otf.c.

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

Here is the caller graph for this function:

5.3.5.37 parseOptions()

Options parseOptions (
    char ∗const argv[const])

Parse command line options.

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

Parameters

in	argv	Pointer to array of command line options.

Returns

Data structure to hold requested command line options.

Definition at line 2500 of file hex2otf.c.

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

```
        {"cff", &cff},
        {"cff2", &cff2},
        {"truetype", &opt.truetype},
        {"blank", &opt.blankOutline},
        {"bitmap", &opt.bitmap},
        {"gpos", &opt.gpos},
        {"gsub", &opt.gsub},
        {NULL, NULL} // sentinel
    };
    while (*format)
    {
        const struct Symbol *p;
        const char *next = NULL;
        for (p = symbols; p->key; p++)
            if ((next = matchToken (format, p->key, ',')))
                break;
        if (!p->key)
            fail ("Invalid format.");
        *p->found = true;
        format = next;
    }
    if (cff + cff2 + opt.truetype + opt.blankOutline > 1)
        fail ("At most one outline format can be accepted.");
    if (!(cff || cff2 || opt.truetype || opt.bitmap))
        fail ("Invalid format.");
    opt.cff = cff + cff2 * 2;
    return opt;
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.38 positionGlyphs()

void positionGlyphs (
Font ∗ font,
const char ∗ fileName,
pixels_t ∗ xMin)

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

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

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

Parameters

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

Definition at line 1061 of file hex2otf.c.

```
{
    *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:

5.3.5.39 prepareOffsets()

void prepareOffsets (
size_t ∗ sizes)

Prepare 32-bit glyph offsets in a font table.

Parameters

in	sizes	Array of glyph sizes, for offset calculations.

Definition at line 1275 of file hex2otf.c.

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

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.40 prepareStringIndex()

Buffer ∗ prepareStringIndex (
const NameStrings names)

Prepare a font name string index.

Parameters

in	names	List of name strings.

Returns

Pointer to a Buffer struct containing the string names.

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

Definition at line 1291 of file hex2otf.c.

```
01292 {
01293     Buffer *buf = newBuffer (256);
01294     assert (names[6]);
01295     const char *strings[] = {"Adobe", "Identity", names[6]};
01296     /// Get the number of elements in array char *strings[].
01297     #define stringCount (sizeof strings / sizeof *strings)
01298     static_assert (stringCount <= U16MAX, "too many strings");
01299     size_t offset = 1;
01300     size_t lengths[stringCount];
01301     for (size_t i = 0; i < stringCount; i++)
01302     {
01303         assert (strings[i]);
01304         lengths[i] = strlen (strings[i]);
01305         offset += lengths[i];
01306     }
01307     int offsetSize = 1 + (offset > 0xff)
01308                        + (offset > 0xffff)
01309                        + (offset > 0xffffff);
01310     cacheU16 (buf, stringCount); // count
01311     cacheU8 (buf, offsetSize); // offSize
01312     cacheU (buf, offset = 1, offsetSize); // offset[0]
01313     for (size_t i = 0; i < stringCount; i++)
01314         cacheU (buf, offset += lengths[i], offsetSize); // offset[i + 1]
01315     for (size_t i = 0; i < stringCount; i++)
01316         cacheBytes (buf, strings[i], lengths[i]);
01317     #undef stringCount
01318     return buf;
01319 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.41 printHelp()

void printHelp ( )

Print help message to stdout and then exit.

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

Definition at line 2426 of file hex2otf.c.

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

Here is the caller graph for this function:

5.3.5.42 printVersion()

void printVersion ( )

Print program version string on stdout.

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

Definition at line 2407 of file hex2otf.c.

```
02407                {
02408     printf ("hex2otf (GNU Unifont) %s\n", VERSION);
02409     printf ("Copyright \u00A9 2022 \u4F55\u5FD7\u7FD4 (He Zhixiang)\n");
02410     printf ("License GPLv2+: GNU GPL version 2 or later\n");
02411     printf ("<https://gnu.org/licenses/gpl.html>\n");
02412     printf ("This is free software: you are free to change and\n");
02413     printf ("redistribute it.  There is NO WARRANTY, to the extent\n");
02414     printf ("permitted by law.\n");
02415
02416     exit (EXIT_SUCCESS);
02417 }
```

Here is the caller graph for this function:

5.3.5.43 readCodePoint()

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

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

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

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

Parameters

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

Returns

true if at end of file, false otherwise.

Definition at line 919 of file hex2otf.c.

```
00920 {
00921     *codePoint = 0;
00922     uint_fast8_t digitCount = 0;
00923     for (;;)
00924     {
00925         int c = getc (file);
00926         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);
    }
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.44 readGlyphs()

void readGlyphs (
    Font ∗ font,
    const char ∗ fileName)

Read glyph definitions from a Unifont .hex format file.

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

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

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

Parameters

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

Definition at line 966 of file hex2otf.c.

```
{
    FILE *file = fopen (fileName, "r");
    if (!file)
        fail ("Failed to open file '%s'.", fileName);
    uint_fast32_t glyphCount = 1; // for glyph 0
    uint_fast8_t maxByteCount = 0;
    { // Hard code the .notdef glyph.
        const byte bitmap[] = "\0\0\0~fZZzvv~vv~\0\0"; // same as U+FFFD
        const size_t byteCount = sizeof bitmap - 1;
        assert (byteCount <= GLYPH_MAX_BYTE_COUNT);
        assert (byteCount % GLYPH_HEIGHT == 0);
        Glyph *notdef = getBufferSlot (font->glyphs, sizeof (Glyph));
        memcpy (notdef->bitmap, bitmap, byteCount);
        notdef->byteCount = maxByteCount = byteCount;
        notdef->combining = false;
```

```
        notdef->pos = 0;
        notdef->lsb = 0;
    }
    for (;;)
    {
        uint_fast32_t codePoint;
        if (readCodePoint (&codePoint, fileName, file))
            break;
        if (++glyphCount > MAX_GLYPHS)
            fail ("OpenType does not support more than %lu glyphs.",
                MAX_GLYPHS);
        Glyph *glyph = getBufferSlot (font->glyphs, sizeof (Glyph));
        glyph->codePoint = codePoint;
        glyph->byteCount = 0;
        glyph->combining = false;
        glyph->pos = 0;
        glyph->lsb = 0;
        for (byte *p = glyph->bitmap;; p++)
        {
            int h, l;
            if (isxdigit (h = getc (file)) && isxdigit (l = getc (file)))
            {
                if (++glyph->byteCount > GLYPH_MAX_BYTE_COUNT)
                    fail ("Hex stream of "PRI_CP" is too long.", codePoint);
                *p = nibbleValue (h) << 4 | nibbleValue (l);
            }
            else if (h == '\n' || (h == EOF && feof (file)))
                break;
            else if (ferror (file))
                fail ("%s: Read error.", fileName);
            else
                fail ("Hex stream of "PRI_CP" is invalid.", codePoint);
        }
        if (glyph->byteCount % GLYPH_HEIGHT != 0)
            fail ("Hex length of "PRI_CP" is indivisible by glyph height %d.",
                codePoint, GLYPH_HEIGHT);
        if (glyph->byteCount > maxByteCount)
            maxByteCount = glyph->byteCount;
    }
    if (glyphCount == 1)
        fail ("No glyph is specified.");
    font->glyphCount = glyphCount;
    font->maxWidth = PW (maxByteCount);
    fclose (file);
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.45 sortGlyphs()

void sortGlyphs (
    Font ∗ font)

Sort the glyphs in a font by Unicode code point.

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

Parameters

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

Definition at line 1119 of file hex2otf.c.

```
{
    Glyph *glyphs = getBufferHead (font->glyphs);
    const Glyph *const glyphsEnd = getBufferTail (font->glyphs);
    glyphs++; // glyph 0 does not need sorting
```

```
qsort (glyphs, glyphsEnd - glyphs, sizeof *glyphs, byCodePoint);
for (const Glyph *glyph = glyphs; glyph < glyphsEnd - 1; glyph++)
{
    if (glyph[0].codePoint == glyph[1].codePoint)
        fail ("Duplicate code point: "PRI_CP".", glyph[0].codePoint);
    assert (glyph[0].codePoint < glyph[1].codePoint);
}
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.46 writeBytes()

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

Write an array of bytes to an output file.

Parameters

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

Definition at line 538 of file hex2otf.c.

```
{
    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:

5.3.5.47 writeFont()

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

Write OpenType font to output file.

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

Parameters

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

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

Definition at line 786 of file hex2otf.c.

```
00787 {
00788     FILE *file = fopen (fileName, "wb");
00789     if (!file)
00790         fail ("Failed to open file '%s'.", fileName);
00791     const Table *const tables = getBufferHead (font->tables);
00792     const Table *const tablesEnd = getBufferTail (font->tables);
00793     size_t tableCount = tablesEnd - tables;
00794     assert (0 < tableCount && tableCount <= U16MAX);
00795     size_t offset = 12 + 16 * tableCount;
00796     uint_fast32_t totalChecksum = 0;
00797     Buffer *tableRecords =
00798         newBuffer (sizeof (struct TableRecord) * tableCount);
00799     for (size_t i = 0; i < tableCount; i++)
00800     {
00801         struct TableRecord *record =
00802             getBufferSlot (tableRecords, sizeof *record);
00803         record->tag = tables[i].tag;
00804         size_t length = countBufferedBytes (tables[i].content);
00805         #if SIZE_MAX > U32MAX
00806             if (offset > U32MAX)
00807                 fail ("Table offset exceeded 4 GiB.");
00808             if (length > U32MAX)
00809                 fail ("Table size exceeded 4 GiB.");
00810         #endif
00811         record->length = length;
00812         record->checksum = 0;
00813         const byte *p = getBufferHead (tables[i].content);
00814         const byte *const end = getBufferTail (tables[i].content);
00815 
00816         /// Add a byte shifted by 24, 16, 8, or 0 bits.
00817         #define addByte(shift) \
00818             if (p == end) \
00819                 break; \
00820             record->checksum += (uint_fast32_t)*p++ « (shift);
00821 
00822         for (;;)
00823         {
00824             addByte (24)
00825             addByte (16)
00826             addByte (8)
00827             addByte (0)
00828         }
00829         #undef addByte
00830         cacheZeros (tables[i].content, (~length + 1U) & 3U);
00831         record->offset = offset;
00832         offset += countBufferedBytes (tables[i].content);
00833         totalChecksum += record->checksum;
00834     }
00835     struct TableRecord *records = getBufferHead (tableRecords);
00836     qsort (records, tableCount, sizeof *records, byTableTag);
00837     // Offset Table
00838     uint_fast32_t sfntVersion = isCFF ? 0x4f54544f : 0x00010000;
00839     writeU32 (sfntVersion, file); // sfntVersion
00840     totalChecksum += sfntVersion;
00841     uint_fast16_t entrySelector = 0;
00842     for (size_t k = tableCount; k != 1; k »= 1)
00843         entrySelector++;
00844     uint_fast16_t searchRange = 1 « (entrySelector + 4);
00845     uint_fast16_t rangeShift = (tableCount - (1 « entrySelector)) « 4;
00846     writeU16 (tableCount, file); // numTables
00847     writeU16 (searchRange, file); // searchRange
00848     writeU16 (entrySelector, file); // entrySelector
00849     writeU16 (rangeShift, file); // rangeShift
00850     totalChecksum += (uint_fast32_t)tableCount « 16;
00851     totalChecksum += searchRange;
00852     totalChecksum += (uint_fast32_t)entrySelector « 16;
00853     totalChecksum += rangeShift;
00854     // Table Records (always sorted by table tags)
00855     for (size_t i = 0; i < tableCount; i++)
00856     {
00857         // Table Record
00858         writeU32 (records[i].tag, file); // tableTag
00859         writeU32 (records[i].checksum, file); // checkSum
00860         writeU32 (records[i].offset, file); // offset
00861         writeU32 (records[i].length, file); // length
00862         totalChecksum += records[i].tag;
00863         totalChecksum += records[i].checksum;
00864         totalChecksum += records[i].offset;
00865         totalChecksum += records[i].length;
00866     }
```

```
    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:

5.3.5.48 writeU16()

void writeU16 (
uint_fast16_t value,
FILE ∗ file)

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

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

Parameters

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

Definition at line 554 of file hex2otf.c.

```
00555 {
00556     byte bytes[] =
00557     {
00558         (value » 8) & 0xff,
00559         (value     ) & 0xff,
00560     };
00561     writeBytes (bytes, sizeof bytes, file);
00562 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.5.49 writeU32()

void writeU32 (
uint_fast32_t value,
FILE ∗ file)

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

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

Parameters

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

Definition at line 574 of file hex2otf.c.

```
00575 {
00576     byte bytes[] =
00577     {
00578         (value » 24) & 0xff,
00579         (value » 16) & 0xff,
00580         (value »  8) & 0xff,
00581         (value     ) & 0xff,
00582     };
00583     writeBytes (bytes, sizeof bytes, file);
00584 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.3.6 Variable Documentation

5.3.6.1 allBuffers

Buffer∗ allBuffers

Initial allocation of empty array of buffer pointers.

Definition at line 139 of file hex2otf.c.

5.3.6.2 bufferCount

size_t bufferCount

Number of buffers in a Buffer ∗ array.

Definition at line 140 of file hex2otf.c.

5.3.6.3 nextBufferIndex

size_t nextBufferIndex

Index number to tail element of Buffer ∗ array.

Definition at line 141 of file hex2otf.c.

5.4 hex2otf.c

Go to the documentation of this file.

```c
/**
   @file hex2otf.c

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

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

   @copyright Copyright © 2022 何志翔 (He Zhixiang)

   @author 何志翔 (He Zhixiang)
*/

/*
   LICENSE:

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

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

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

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

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

#include "hex2otf.h"

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

/**
    @brief Free all allocated buffer pointers.

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

/**
    @brief Create a new buffer.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

```
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

/**
```

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

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

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

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

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

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

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

```
    }
}

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

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

/**
    @brief Build a glyph outline.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```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;
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
}

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

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

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

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

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

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

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

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

    The "hmtx" table contains horizontal metrics information.

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

/**
```

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

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

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

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

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

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

    The "post" table contains information for PostScript printers.

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

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

    The "GPOS" table contains information for glyph positioning.

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

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

```

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

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

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

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

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

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

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

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

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

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

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

    exit (EXIT_SUCCESS);
}

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

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

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

    exit (EXIT_SUCCESS);
}

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

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

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

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

/**
    @brief Parse command line options.

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

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

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

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

/**
   @brief The main function.

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

5.5 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 "16.0.01"

  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.

5.5.1 Detailed Description

hex2otf.h - Header file for hex2otf.c

Copyright

Author

何志翔 (He Zhixiang)

Definition in file hex2otf.h.

5.5.2 Macro Definition Documentation

5.5.2.1 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.

5.5.2.2 DEFAULT_ID1

#define DEFAULT_ID1 "Unifont"

Default NameID 1 string (Font Family)

Definition at line 57 of file hex2otf.h.

5.5.2.3 DEFAULT_ID11

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

Default NameID 11 string (Font Vendor URL)

Definition at line 64 of file hex2otf.h.

5.5.2.4 DEFAULT_ID13

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

Default NameID 13 string (License Description)

Definition at line 67 of file hex2otf.h.

5.5.2.5 DEFAULT_ID14

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

Default NameID 14 string (License Information URLs)

Definition at line 71 of file hex2otf.h.

5.5.2.6 DEFAULT_ID2

#define DEFAULT_ID2 "Regular"

Default NameID 2 string (Font Subfamily)

Definition at line 58 of file hex2otf.h.

5.5.2.7 DEFAULT_ID5

#define DEFAULT_ID5 "Version "UNIFONT_VERSION

Default NameID 5 string (Version of the Name Table)

Definition at line 61 of file hex2otf.h.

5.5.2.8 NAMEPAIR

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

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

Definition at line 84 of file hex2otf.h.

5.5.2.9 UNIFONT_VERSION

#define UNIFONT_VERSION "16.0.01"

Current Unifont version.

Definition at line 36 of file hex2otf.h.

5.5.3 Variable Documentation

5.5.3.1 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.

5.6 hex2otf.h

Go to the documentation of this file.

```
/**
   @file hex2otf.h

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

   @copyright Copyright © 2022 何志翔 (He Zhixiang)

   @author 何志翔 (He Zhixiang)
*/

/*
   LICENSE:

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

   This program is distributed in the hope that it will be useful,
```

```
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

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

   NOTE: It is a violation of the license terms of this software
   to delete license and copyright information below if creating
   a font derived from Unifont glyphs.
*/
#ifndef __HEX2OTF_H__
#define __HEX2OTF_H__

#define UNIFONT_VERSION "16.0.01"   ///< Current Unifont version.

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

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

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

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

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

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

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

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

/**
   @brief Data structure for a font ID number and name character string.
*/
typedef struct NamePair
{
   int id;
   const char *str;
} NamePair;

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

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

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

```
    NAMEPAIR (14), // License information URL
    {0, NULL}      // Sentinel
};

#undef NAMEPAIR

#endif
```

5.7 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.

5.7.1 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

Definition in file johab2syllables.c.

5.7.2 Function Documentation

5.7.2.1 main()

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

The main function.

Definition at line 42 of file johab2syllables.c.

```
00042 {
00043    int      i;           /* Loop variables     */
00044    int      arg_count;  /* index into *argv[] */
00045    unsigned codept;
00046    unsigned max_codept;
00047    unsigned char  hangul_base[MAX_GLYPHS][32];
00048    int      initial, medial, final;          /* Base glyphs for a syllable. */
00049    unsigned char  syllable[32];          /* Syllable glyph built for output. */
00050 
00051    FILE *infp  = stdin;   /* Input Hangul Johab 6/3/1 file */
00052    FILE *outfp = stdout;  /* Output Hangul Syllables file  */
00053 
00054    /* Print a help message */
00055    void print_help ();
00056 
00057    /* Read the file containing Hangul base glyphs. */
00058    unsigned hangul_read_base8 (FILE *infp, unsigned char hangul_base[][32]);
00059 
00060    /* Given a Hangul Syllables code point, determine component glyphs. */
00061    void hangul_decompose (unsigned codept, int *, int *, int *);
00062 
00063    /* Given letters in a Hangul syllable, return a glyph. */
00064    void hangul_syllable (int choseong, int jungseong, int jongseong,
00065                          unsigned char hangul_base[][32],
00066                          unsigned char *syllable);
00067 
00068 
00069    /*
00070       If there are command line arguments, parse them.
00071    */
00072    arg_count = 1;
00073 
00074    while (arg_count < argc) {
00075       /* If input file is specified, open it for read access. */
00076       if (strncmp (argv [arg_count], "-i", 2) == 0) {
00077          arg_count++;
00078          if (arg_count < argc) {
00079             infp = fopen (argv [arg_count], "r");
00080             if (infp == NULL) {
00081                fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",
00082                         argv [arg_count]);
00083                exit (EXIT_FAILURE);
00084             }
00085          }
00086       }
00087       /* If output file is specified, open it for write access. */
00088       else if (strncmp (argv [arg_count], "-o", 2) == 0) {
00089          arg_count++;
00090          if (arg_count < argc) {
00091             outfp = fopen (argv [arg_count], "w");
00092             if (outfp == NULL) {
00093                fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
00094                         argv [arg_count]);
00095                exit (EXIT_FAILURE);
00096             }
00097          }
00098       }
00099       /* If help is requested, print help message and exit. */
00100       else if (strncmp (argv [arg_count], "-h",     2) == 0 ||
00101                strncmp (argv [arg_count], "--help", 6) == 0) {
00102          print_help ();
00103          exit (EXIT_SUCCESS);
00104       }
00105 
00106       arg_count++;
00107    }
00108 
```

```
   /*
      Initialize entire glyph array to zeroes in case the input
      file skips over some code points.
   */
   for (codept = 0; codept < MAX_GLYPHS; codept++) {
      for (i = 0; i < 32; i++) hangul_base[codept][i] = 0;
   }

   /*
      Read the entire "hangul-base.hex" file into an array
      organized as hangul_base [code_point][glyph_byte].
      The Hangul glyphs are 16 columns wide, which is
      two bytes, by 16 rows, for a total of 2 * 16 = 32 bytes.
   */
   max_codept = hangul_read_base8 (infp, hangul_base);
   if (max_codept > 0x8FF) {
      fprintf (stderr, "\nWARNING: Hangul glyph range exceeds PUA space.\n\n");
   }

   /*
      For each glyph in the Unicode Hangul Syllables block,
      form a composite glyph of choseong + jungseong +
      optional jongseong and output it in Unifont .hex format.
   */
   for (codept = 0xAC00; codept < 0xAC00 + 19 * 21 * 28; codept++) {
      hangul_decompose (codept, &initial, &medial, &final);

      hangul_syllable (initial, medial, final, hangul_base, syllable);

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

      for (i = 0; i < 32; i++) {
         fprintf (outfp, "%02X", syllable[i]);
      }
      fputc ('\n', outfp);
   }

   exit (EXIT_SUCCESS);
}
```

Here is the call graph for this function:

5.7.2.2 print_help()

void print_help ( )

Print a help message.

Definition at line 155 of file johab2syllables.c.

```
00155            {
00156
00157    printf ("\ngen-hangul [options]\n\n");
00158    printf ("     Generates Hangul syllables from an input Unifont .hex file encoded\n");
00159    printf ("     in Johab 6/3/1 format.  The output is the Unicode Hangul Syllables\n");
00160    printf ("     range, U+AC00..U+D7A3.\n\n");
00161    printf ("     This program demonstrates forming Hangul syllables without shifting\n");
00162    printf ("     the final consonant (jongseong) when combined with a vowel having\n");
00163    printf ("     a long double vertical stroke.  For a program that demonstrtes\n");
00164    printf ("     shifting jongseong in those cases, see unigen-hangul, which is what\n");
00165    printf ("     creates the Unifont Hangul Syllables block.\n\n");
00166
00167    printf ("     This program may be invoked with the following command line options:\n\n");
00168
00169    printf ("     Option     Parameters     Function\n");
00170    printf ("     ------     ----------     --------\n");
00171    printf ("     -h, --help                Print this message and exit.\n\n");
00172    printf ("     -i         input_file     Unifont hangul-base.hex formatted input file.\n\n");
00173    printf ("     -o         output_file    Unifont .hex format output file.\n\n");
00174    printf ("      Example:\n\n");
00175    printf ("         johab2syllables -i hangul-base.hex -o hangul-syllables.hex\n\n");
00176
00177    return;
00178 }
```

Here is the caller graph for this function:

5.8 johab2syllables.c

Go to the documentation of this file.

```
/**
  @file johab2syllables.c

  @brief 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
*/
/*
  LICENSE:

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

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

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

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

#include "hangul.h"


/**
  @brief The main function.
*/
int
main (int argc, char *argv[]) {
  int     i;          /* Loop variables     */
  int     arg_count;  /* index into *argv[] */
  unsigned codept;
  unsigned max_codept;
  unsigned char  hangul_base[MAX_GLYPHS][32];
  int     initial, medial, final;          /* Base glyphs for a syllable. */
  unsigned char  syllable[32];         /* Syllable glyph built for output. */

  FILE *infp  = stdin;   /* Input Hangul Johab 6/3/1 file */
  FILE *outfp = stdout;  /* Output Hangul Syllables file  */

  /* Print a help message */
  void print_help ();

  /* Read the file containing Hangul base glyphs. */
  unsigned hangul_read_base8 (FILE *infp, unsigned char hangul_base[][32]);

  /* Given a Hangul Syllables code point, determine component glyphs. */
  void hangul_decompose (unsigned codept, int *, int *, int *);

  /* Given letters in a Hangul syllable, return a glyph. */
  void hangul_syllable (int choseong, int jungseong, int jongseong,
                        unsigned char hangul_base[][32],
                        unsigned char *syllable);


  /*
    If there are command line arguments, parse them.
  */
  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) {
               infp = fopen (argv [arg_count], "r");
               if (infp == NULL) {
                  fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",
                           argv [arg_count]);
                  exit (EXIT_FAILURE);
               }
            }
         }
         /* 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) {
               outfp = fopen (argv [arg_count], "w");
               if (outfp == NULL) {
                  fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
                           argv [arg_count]);
                  exit (EXIT_FAILURE);
               }
            }
         }
         /* 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) {
            print_help ();
            exit (EXIT_SUCCESS);
         }

         arg_count++;
      }


   /*
      Initialize entire glyph array to zeroes in case the input
      file skips over some code points.
   */
   for (codept = 0; codept < MAX_GLYPHS; codept++) {
      for (i = 0; i < 32; i++) hangul_base[codept][i] = 0;
   }

   /*
      Read the entire "hangul-base.hex" file into an array
      organized as hangul_base [code_point][glyph_byte].
      The Hangul glyphs are 16 columns wide, which is
      two bytes, by 16 rows, for a total of 2 * 16 = 32 bytes.
   */
   max_codept = hangul_read_base8 (infp, hangul_base);
   if (max_codept > 0x8FF) {
      fprintf (stderr, "\nWARNING: Hangul glyph range exceeds PUA space.\n\n");
   }

   /*
      For each glyph in the Unicode Hangul Syllables block,
      form a composite glyph of choseong + jungseong +
      optional jongseong and output it in Unifont .hex format.
   */
   for (codept = 0xAC00; codept < 0xAC00 + 19 * 21 * 28; codept++) {
      hangul_decompose (codept, &initial, &medial, &final);

      hangul_syllable (initial, medial, final, hangul_base, syllable);

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

      for (i = 0; i < 32; i++) {
         fprintf (outfp, "%02X", syllable[i]);
      }
      fputc ('\n', outfp);
   }

   exit (EXIT_SUCCESS);
}


/**
   @brief Print a help message.
*/
void
print_help () {

   printf ("\ngen-hangul [options]\n\n");
```

```
    printf ("     Generates Hangul syllables from an input Unifont .hex file encoded\n");
    printf ("     in Johab 6/3/1 format.  The output is the Unicode Hangul Syllables\n");
    printf ("     range, U+AC00..U+D7A3.\n\n");
    printf ("     This program demonstrates forming Hangul syllables without shifting\n");
    printf ("     the final consonant (jongseong) when combined with a vowel having\n");
    printf ("     a long double vertical stroke.  For a program that demonstrtes\n");
    printf ("     shifting jongseong in those cases, see unigen-hangul, which is what\n");
    printf ("     creates the Unifont Hangul Syllables block.\n\n");

    printf ("     This program may be invoked with the following command line options:\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    Unifont .hex format output file.\n\n");
    printf ("      Example:\n\n");
    printf ("         johab2syllables -i hangul-base.hex -o hangul-syllables.hex\n\n");

    return;
}
```

5.9 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.

5.9.1 Detailed Description

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

Author

Paul Hardy, January 2008

Copyright

Copyright (C) 2008, 2013 Paul Hardy

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

Definition in file unibdf2hex.c.

5.9.2 Macro Definition Documentation

5.9.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum allowable input file line length - 1.

Definition at line 37 of file unibdf2hex.c.

5.9.2.2 UNISTART

```
#define UNISTART 0x3400
```

First Unicode code point to examine.

Definition at line 34 of file unibdf2hex.c.

5.9.2.3 UNISTOP

```
#define UNISTOP 0x4DBF
```

Last Unicode code point to examine.

Definition at line 35 of file unibdf2hex.c.

5.9.3 Function Documentation

5.9.3.1 main()

int main ( )

The main function.

Returns

Exit status is always 0 (successful termination).

Definition at line 46 of file unibdf2hex.c.

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

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

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

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

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

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

5.10 unibdf2hex.c

Go to the documentation of this file.

```
/**
  @file unibdf2hex.c

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

  @author Paul Hardy, January 2008

  @copyright Copyright (C) 2008, 2013 Paul Hardy

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

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

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

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

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

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

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


/**
  @brief The main function.

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

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

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

```
                strncmp (inbuf, ”BBX ”, 4) != 0); /* find bounding box */

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

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

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

5.11 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]

5.11.1 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

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

Definition in file unibmp2hex.c.

5.11.2 Macro Definition Documentation

5.11.2.1 MAXBUF

#define MAXBUF 256

Maximum input file line length - 1.

Definition at line 114 of file unibmp2hex.c.

5.11.3 Function Documentation

5.11.3.1 main()

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

The main function.

Parameters

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

Returns

This program exits with status 0.

Definition at line 159 of file unibmp2hex.c.

```
{

   int i, j, k;               /* loop variables                       */
   unsigned char inchar;       /* temporary input character */
   char header[MAXBUF];        /* input buffer for bitmap file header */
   int wbmp=0; /* =0 for Windows Bitmap (.bmp); 1 for Wireless Bitmap (.wbmp) */
   int fatal; /* =1 if a fatal error occurred */
   int match; /* =1 if we're still matching a pattern, 0 if no match */
   int empty1, empty2; /* =1 if bytes tested are all zeroes */
   unsigned char thischar1[16], thischar2[16]; /* bytes of hex char */
   unsigned char thischar0[16], thischar3[16]; /* bytes for quadruple-width */
   int thisrow; /* index to point into thischar1[] and thischar2[] */
   int tmpsum;  /* temporary sum to see if a character is blank */
   unsigned this_pixel;  /* color of one pixel, if > 1 bit per pixel */
   unsigned next_pixels; /* pending group of 8 pixels being read */
   unsigned color_mask = 0x00;  /* to invert monochrome bitmap, set to 0xFF */

```

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

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

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

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

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

/* Supplemental Multilingual Plane (Plane 01) */
for (i = 0x0105C0; i <= 0x0105FF; i++) wide[i] = 1; /* Todhri             */
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 = 0x011380; i <= 0x0113FF; i++) wide[i] = 1; /* Tulu-Tigalari      */
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 = 0x0116D0; i <= 0x0116FF; i++) wide[i] = 1; /* Myanmar Extended-C */
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 = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi               */
for (i = 0x012F90; i <= 0x012FFF; i++) wide[i] = 1; /* Cypro-Minoan       */
/* Make Bassa Vah all single width or all double width */
for (i = 0x016100; i <= 0x01613F; i++) wide[i] = 1; /* Gurung Khema       */
for (i = 0x016AD0; i <= 0x016AFF; i++) wide[i] = 1; /* Bassa Vah          */
for (i = 0x016B00; i <= 0x016B8F; i++) wide[i] = 1; /* Pahawh Hmong       */
for (i = 0x016D40; i <= 0x016D7F; i++) wide[i] = 1; /* Kirat Rai          */
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 = 0x018B00; i <= 0x018CFF; i++) wide[i] = 1; /* Khitan Small Script*/
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 = 0x01E500; i <= 0x01E5FF; i++) wide[i] = 1; /* Ol Onal            */
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);
}
```

5.11.4 Variable Documentation

5.11.4.1 bits_per_pixel

int bits_per_pixel

Definition at line 137 of file unibmp2hex.c.

5.11.4.2

struct { ... } bmp_header

Bitmap Header parameters

5.11.4.3 color_table

unsigned char color_table[256][4]

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

Definition at line 147 of file unibmp2hex.c.

5.11.4.4 compression

int compression

Definition at line 138 of file unibmp2hex.c.

5.11.4.5 file_size

int file_size

Definition at line 131 of file unibmp2hex.c.

5.11.4.6 filetype

char filetype[2]

Definition at line 130 of file unibmp2hex.c.

5.11.4.7 flip

unsigned flip =0

=1 if we're transposing glyph matrix

Definition at line 121 of file unibmp2hex.c.

5.11.4.8 forcewide

unsigned forcewide =0

=1 to set each glyph to 16 pixels wide

Definition at line 122 of file unibmp2hex.c.

5.11.4.9 height

int height

Definition at line 135 of file unibmp2hex.c.

5.11.4.10 hexdigit

unsigned hexdigit[16][4]

32 bit representation of 16x8 0..F bitmap

Definition at line 117 of file unibmp2hex.c.

5.11.4.11 image_offset

int image_offset

Definition at line 132 of file unibmp2hex.c.

5.11.4.12 image_size

int image_size

Definition at line 139 of file unibmp2hex.c.

5.11.4.13 important_colors

int important_colors

Definition at line 143 of file unibmp2hex.c.

5.11.4.14 info_size

int info_size

Definition at line 133 of file unibmp2hex.c.

5.11.4.15 ncolors

int ncolors

Definition at line 142 of file unibmp2hex.c.

5.11.4.16 nplanes

int nplanes

Definition at line 136 of file unibmp2hex.c.

5.11.4.17 planeset

unsigned planeset =0

=1: use plane specified with -p parameter

Definition at line 120 of file unibmp2hex.c.

5.11.4.18 unidigit

unsigned unidigit[6][4]

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

Definition at line 125 of file unibmp2hex.c.

5.11.4.19 uniplane

unsigned uniplane =0

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

Definition at line 119 of file unibmp2hex.c.

5.11.4.20 width

int width

Definition at line 134 of file unibmp2hex.c.

5.11.4.21 x_ppm

int x_ppm

Definition at line 140 of file unibmp2hex.c.

5.11.4.22 y__ppm

int y__ppm

Definition at line 141 of file unibmp2hex.c.

5.12 unibmp2hex.c

Go to the documentation of this file.

```
/**
   @file unibmp2hex.c

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

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

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

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

   LICENSE:

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

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

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

/*
   2 September 2024 [Paul Hardy] - Set these scripts to double width:
      - U+10D40..U+10D8F (Garay)
      - U+11380..U+113FF (Tulu-Tigalari)
      - U+116D0..U+116FF (Myanmar Extended-C)
      - U+11F00..U+11F5F (Kawi)
      - U+16100..U+1613F (Gurung Khema)
      - U+16D40..U+16D7F (Kirat Rai)
      - U+18B00..U+18CFF (Khitan Small Script)
      - U+1E5D0..U+1E5FF (Ol Onal)

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

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

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

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

```

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

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

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

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

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

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

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

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


*/

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

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


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

unsigned uniplane=0;       ///< Unicode plane number, 0..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

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


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

```c

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

// #define DEBUG

/**
   @brief The main function.

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

   int i, j, k;                  /* loop variables                          */
   unsigned char inchar;         /* temporary input character */
   char header[MAXBUF];          /* input buffer for bitmap file header */
   int wbmp=0; /* =0 for Windows Bitmap (.bmp); 1 for Wireless Bitmap (.wbmp) */
   int fatal; /* =1 if a fatal error occurred */
   int match; /* =1 if we're still matching a pattern, 0 if no match */
   int empty1, empty2; /* =1 if bytes tested are all zeroes */
   unsigned char thischar1[16], thischar2[16]; /* bytes of hex char */
   unsigned char thischar0[16], thischar3[16]; /* bytes for quadruple-width */
   int thisrow; /* index to point into thischar1[] and thischar2[] */
   int tmpsum;  /* temporary sum to see if a character is blank */
   unsigned this_pixel;  /* color of one pixel, if > 1 bit per pixel */
   unsigned next_pixels; /* pending group of 8 pixels being read */
   unsigned color_mask = 0x00;  /* to invert monochrome bitmap, set to 0xFF */

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

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

   if (argc > 1) {
      for (i = 1; i < argc; i++) {
         if (argv[i][0] == '-') {  /* this is an option argument */
            switch (argv[i][1]) {
               case 'i':  /* name of input file */
                  infile = &argv[i][2];
                  break;
               case 'o':  /* name of output file */
                  outfile = &argv[i][2];
                  break;
               case 'p':  /* specify a Unicode plane */
                  sscanf (&argv[i][2], "%x", &uniplane); /* Get Unicode plane */
                  planeset = 1; /* Use specified range, not what's in bitmap */
                  break;
               case 'w': /* force wide (16 pixels) for each glyph */
                  forcewide = 1;
                  break;
               default:   /* if unrecognized option, print list and exit */
                  fprintf (stderr, "\nSyntax:\n\n");
                  fprintf (stderr, "   %s -p<Unicode_Page> ", argv[0]);
                  fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
                  fprintf (stderr, "   -w specifies .wbmp output instead of ");
                  fprintf (stderr, "default Windows .bmp output.\n\n");
                  fprintf (stderr, "   -p is followed by 1 to 6 ");
                  fprintf (stderr, "Unicode plane hex digits ");
                  fprintf (stderr, "(default is Page 0).\n\n");
                  fprintf (stderr, "\nExample:\n\n");
                  fprintf (stderr, "   %s -p83 -iunifont.hex -ou83.bmp\n\n\n",
                         argv[0]);
                  exit (1);
            }
         }
      }
   }
   /*
      Make sure we can open any I/O files that were specified before
      doing anything else.
   */
```

```
  if (strlen (infile) > 0) {
    if ((infp = fopen (infile, "r")) == NULL) {
      fprintf (stderr, "Error: can't open %s for input.\n", infile);
      exit (1);
    }
  }
  else {
    infp = stdin;
  }
  if (strlen (outfile) > 0) {
    if ((outfp = fopen (outfile, "w")) == NULL) {
      fprintf (stderr, "Error: can't open %s for output.\n", outfile);
      exit (1);
    }
  }
  else {
    outfp = stdout;
  }
  /*
    Initialize selected code points for double width (16x16).
    Double-width is forced in cases where a glyph (usually a combining
    glyph) only occupies the left-hand side of a 16x16 grid, but must
    be rendered as double-width to appear properly with other glyphs
    in a given script.  If additions were made to a script after
    Unicode 5.0, the Unicode version is given in parentheses after
    the script name.
  */
  for (i = 0x0700; i <= 0x074F; i++) wide[i] = 1; /* Syriac                 */
  for (i = 0x0800; i <= 0x083F; i++) wide[i] = 1; /* Samaritan (5.2)        */
  for (i = 0x0900; i <= 0x0DFF; i++) wide[i] = 1; /* Indic                  */
  for (i = 0x1000; i <= 0x109F; i++) wide[i] = 1; /* Myanmar                */
  for (i = 0x1100; i <= 0x11FF; i++) wide[i] = 1; /* Hangul Jamo            */
  for (i = 0x1400; i <= 0x167F; i++) wide[i] = 1; /* Canadian Aboriginal    */
  for (i = 0x1700; i <= 0x171F; i++) wide[i] = 1; /* Tagalog                */
  for (i = 0x1720; i <= 0x173F; i++) wide[i] = 1; /* Hanunoo                */
  for (i = 0x1740; i <= 0x175F; i++) wide[i] = 1; /* Buhid                  */
  for (i = 0x1760; i <= 0x177F; i++) wide[i] = 1; /* Tagbanwa               */
  for (i = 0x1780; i <= 0x17FF; i++) wide[i] = 1; /* Khmer                  */
  for (i = 0x18B0; i <= 0x18FF; i++) wide[i] = 1; /* Ext. Can. Aboriginal   */
  for (i = 0x1800; i <= 0x18AF; i++) wide[i] = 1; /* Mongolian              */
  for (i = 0x1900; i <= 0x194F; i++) wide[i] = 1; /* Limbu                  */
// for (i = 0x1980; i <= 0x19DF; i++) wide[i] = 1; /* New Tai Lue            */
  for (i = 0x1A00; i <= 0x1A1F; i++) wide[i] = 1; /* Buginese               */
  for (i = 0x1A20; i <= 0x1AAF; i++) wide[i] = 1; /* Tai Tham (5.2)         */
  for (i = 0x1B00; i <= 0x1B7F; i++) wide[i] = 1; /* Balinese               */
  for (i = 0x1B80; i <= 0x1BBF; i++) wide[i] = 1; /* Sundanese (5.1)        */
  for (i = 0x1BC0; i <= 0x1BFF; i++) wide[i] = 1; /* Batak (6.0)            */
  for (i = 0x1C00; i <= 0x1C4F; i++) wide[i] = 1; /* Lepcha (5.1)           */
  for (i = 0x1CC0; i <= 0x1CCF; i++) wide[i] = 1; /* Sundanese Supplement   */
  for (i = 0x1CD0; i <= 0x1CFF; i++) wide[i] = 1; /* Vedic Extensions (5.2) */
  wide[0x2329] = wide[0x232A] = 1; /* Left- & Right-pointing Angle Brackets */
  for (i = 0x2E80; i <= 0xA4CF; i++) wide[i] = 1; /* CJK                    */
// for (i = 0x9FD8; i <= 0x9FE9; i++) wide[i] = 4; /* CJK quadruple-width    */
  for (i = 0xA900; i <= 0xA92F; i++) wide[i] = 1; /* Kayah Li (5.1)         */
  for (i = 0xA930; i <= 0xA95F; i++) wide[i] = 1; /* Rejang (5.1)           */
  for (i = 0xA960; i <= 0xA97F; i++) wide[i] = 1; /* Hangul Jamo Extended-A */
  for (i = 0xA980; i <= 0xA9DF; i++) wide[i] = 1; /* Javanese (5.2)         */
  for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham (5.1)             */
  for (i = 0xA9E0; i <= 0xA9FF; i++) wide[i] = 1; /* Myanmar Extended-B     */
  for (i = 0xAA00; i <= 0xAA5F; i++) wide[i] = 1; /* Cham                   */
  for (i = 0xAA60; i <= 0xAA7F; i++) wide[i] = 1; /* Myanmar Extended-A     */
  for (i = 0xAAE0; i <= 0xAAFF; i++) wide[i] = 1; /* Meetei Mayek Ext (6.0) */
  for (i = 0xABC0; i <= 0xABFF; i++) wide[i] = 1; /* Meetei Mayek (5.2)     */
  for (i = 0xAC00; i <= 0xD7AF; i++) wide[i] = 1; /* Hangul Syllables       */
  for (i = 0xD7B0; i <= 0xD7FF; i++) wide[i] = 1; /* Hangul Jamo Extended-B */
  for (i = 0xF900; i <= 0xFAFF; i++) wide[i] = 1; /* CJK Compatibility      */
  for (i = 0xFE10; i <= 0xFE1F; i++) wide[i] = 1; /* Vertical Forms         */
  for (i = 0xFE30; i <= 0xFE60; i++) wide[i] = 1; /* CJK Compatibility Forms*/
  for (i = 0xFFE0; i <= 0xFFE6; i++) wide[i] = 1; /* CJK Compatibility Forms*/

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

  /* Supplemental Multilingual Plane (Plane 01) */
  for (i = 0x0105C0; i <= 0x0105FF; i++) wide[i] = 1; /* Todhri             */
  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 = 0x011380; i <= 0x0113FF; i++) wide[i] = 1; /* Tulu-Tigalari      */
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 = 0x0116D0; i <= 0x0116FF; i++) wide[i] = 1; /* Myanmar Extended-C */
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 = 0x011F00; i <= 0x011F5F; i++) wide[i] = 1; /* Kawi               */
for (i = 0x012F90; i <= 0x012FFF; i++) wide[i] = 1; /* Cypro-Minoan       */
/* Make Bassa Vah all single width or all double width */
for (i = 0x016100; i <= 0x01613F; i++) wide[i] = 1; /* Gurung Khema       */
for (i = 0x016AD0; i <= 0x016AFF; i++) wide[i] = 1; /* Bassa Vah          */
for (i = 0x016B00; i <= 0x016B8F; i++) wide[i] = 1; /* Pahawh Hmong       */
for (i = 0x016D40; i <= 0x016D7F; i++) wide[i] = 1; /* Kirat Rai          */
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 = 0x018B00; i <= 0x018CFF; i++) wide[i] = 1; /* Khitan Small Script*/
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 = 0x01E500; i <= 0x01E5FF; i++) wide[i] = 1; /* Ol Onal            */
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);
}
```

5.13 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.

5.13.1 Detailed Description

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

Author

Paul Hardy, unifoundry <at> unifoundry.com

Copyright

Copyright (C) 2019 Paul Hardy

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

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

Definition in file unibmpbump.c.

5.13.2 Macro Definition Documentation

5.13.2.1 MAX_COMPRESSION_METHOD

```
#define MAX_COMPRESSION_METHOD 13
```

Maximum supported compression method.

Definition at line 40 of file unibmpbump.c.

5.13.2.2 VERSION

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

Version of this program.

Definition at line 38 of file unibmpbump.c.

5.13.3 Function Documentation

5.13.3.1 get_bytes()

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

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

Parameters

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

Returns

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

Definition at line 487 of file unibmpbump.c.

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

Here is the caller graph for this function:

5.13.3.2 main()

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

The main function.

Parameters

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

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 50 of file unibmpbump.c.

```
00050 {
00051
00052    /*
00053       Values preserved from file header (first 14 bytes).
00054    */
00055    char file_format[3];       /* "BM" for original Windows format          */
00056    unsigned filesize;         /* size of file in bytes                     */
00057    unsigned char rsvd_hdr[4]; /* 4 reserved bytes                          */
```

```
unsigned image_start;       /* byte offset of image in file                  */

/*
   Values preserved from Device Independent Bitmap (DIB) Header.

   The DIB fields below are in the standard 40-byte header.  Version
   4 and version 5 headers have more information, mainly for color
   information.  That is skipped over, because a valid glyph image
   is just monochrome.
*/
int dib_length;             /* in bytes, for parsing by header version     */
int image_width = 0;        /* Signed image width                           */
int image_height = 0;       /* Signed image height                         */
int num_planes = 0;         /* number of planes; must be 1                  */
int bits_per_pixel = 0;     /* for palletized color maps (< 2^16 colors)  */
/*
   The following fields are not in the original spec, so initialize
   them to 0 so we can correctly parse an original file format.
*/
int compression_method=0;  /* 0 --> uncompressed RGB/monochrome            */
int image_size = 0;         /* 0 is a valid size if no compression         */
int hres = 0;               /* image horizontal resolution                 */
int vres = 0;               /* image vertical resolution                   */
int num_colors = 0;         /* Number of colors for pallettized images     */
int important_colors = 0;  /* Number of significant colors (0 or 2)       */

int true_colors = 0;        /* interpret num_colors, which can equal 0     */

/*
   Color map.  This should be a monochrome file, so only two
   colors are stored.
*/
unsigned char color_map[2][4]; /* two of R, G, B, and possibly alpha  */

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

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

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

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

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

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

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

/*
   Process command line arguments.
*/
if (argc > 1) {
   for (i = 1; i < argc; i++) {
      if (argv[i][0] == '-') {  /* this is an option argument */
         switch (argv[i][1]) {
            case 'i':  /* name of input file */
               infile = &argv[i][2];
               break;
            case 'o':  /* name of output file */
               outfile = &argv[i][2];
               break;
            case 'v':  /* verbose output */
               verbose = 1;
               break;
            case 'V':  /* print version & quit */
               fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
               exit (EXIT_SUCCESS);
               break;
            case '-':  /* see if "--verbose" */
               if (strcmp (argv[i], "--verbose") == 0) {
                  verbose = 1;
               }
               else if (strcmp (argv[i], "--version") == 0) {
                  fprintf (stderr, "unibmpbump version %s\n\n", VERSION);
                  exit (EXIT_SUCCESS);
               }
               break;
            default:   /* if unrecognized option, print list and exit */
               fprintf (stderr, "\nSyntax:\n\n");
               fprintf (stderr, "   unibmpbump ");
               fprintf (stderr, "-i<Input_File> -o<Output_File>\n\n");
               fprintf (stderr, "-v or --verbose gives verbose output");
               fprintf (stderr, " on stderr\n\n");
               fprintf (stderr, "-V or --version prints version");
               fprintf (stderr, " on stderr and exits\n\n");
               fprintf (stderr, "\nExample:\n\n");
               fprintf (stderr, "   unibmpbump -iuni0101.bmp");
               fprintf (stderr, " -onew-uni0101.bmp\n\n");
               exit (EXIT_SUCCESS);
         }
      }
   }
}

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


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

/* Read file size */
filesize = get_bytes (infp, 4);

```

```
    /* Read Reserved bytes */
    rsvd_hdr[0] = get_bytes (infp, 1);
    rsvd_hdr[1] = get_bytes (infp, 1);
    rsvd_hdr[2] = get_bytes (infp, 1);
    rsvd_hdr[3] = get_bytes (infp, 1);

    /* Read Image Offset Address within file */
    image_start = get_bytes (infp, 4);

    /*
       See if this looks like a valid image file based on
       the file header first two bytes.
    */
    if (strncmp (file_format, "BM", 2) != 0) {
       fprintf (stderr, "\nInvalid file format: not file type \"BM\".\n\n");
       exit (EXIT_FAILURE);
    }

    if (verbose) {
       fprintf (stderr, "\nFile Header:\n");
       fprintf (stderr, "    File Type:    \"%s\"\n", file_format);
       fprintf (stderr, "    File Size:    %d bytes\n", filesize);
       fprintf (stderr, "    Reserved:    ");
       for (i = 0; i < 4; i++) fprintf (stderr, " 0x%02X", rsvd_hdr[i]);
       fputc ('\n', stderr);
       fprintf (stderr, "    Image Start: %d. = 0x%02X = 0%05o\n\n",
                image_start, image_start, image_start);
    }  /* if (verbose) */

    /*
       Device Independent Bitmap (DIB) Header: bitmap information header
       ("BM" format file DIB Header is 12 bytes long).
    */
    dib_length = get_bytes (infp, 4);

    /*
       Parse one of three versions of Device Independent Bitmap (DIB) format:

            Length  Format
            ------  ------
               12   BITMAPCOREHEADER
               40   BITMAPINFOHEADER
              108   BITMAPV4HEADER
              124   BITMAPV5HEADER
    */
    if (dib_length == 12) { /* BITMAPCOREHEADER format -- UNTESTED */
       image_width    = get_bytes (infp, 2);
       image_height   = get_bytes (infp, 2);
       num_planes     = get_bytes (infp, 2);
       bits_per_pixel = get_bytes (infp, 2);
    }
    else if (dib_length >= 40) { /* BITMAPINFOHEADER format or later */
       image_width = get_bytes (infp, 4);
       image_height       = get_bytes (infp, 4);
       num_planes         = get_bytes (infp, 2);
       bits_per_pixel     = get_bytes (infp, 2);
       compression_method = get_bytes (infp, 4);  /* BI_BITFIELDS */
       image_size         = get_bytes (infp, 4);
       hres               = get_bytes (infp, 4);
       vres               = get_bytes (infp, 4);
       num_colors         = get_bytes (infp, 4);
       important_colors   = get_bytes (infp, 4);

       /* true_colors is true number of colors in image */
       if (num_colors == 0)
          true_colors = 1 << bits_per_pixel;
       else
          true_colors = num_colors;

       /*
          If dib_length > 40, the format is BITMAPV4HEADER or
          BITMAPV5HEADER.  As this program is only designed
          to handle a monochrome image, we can ignore the rest
          of the header but must read past the remaining bytes.
       */
       for (i = 40; i < dib_length; i++) (void)get_bytes (infp, 1);
    }

    if (verbose) {
       fprintf (stderr, "Device Independent Bitmap (DIB) Header:\n");
       fprintf (stderr, "    DIB Length:  %9d bytes (version = ", dib_length);
```

```

      if      (dib_length ==  12) fprintf (stderr, "\"BITMAPCOREHEADER\")\n");
      else if (dib_length ==  40) fprintf (stderr, "\"BITMAPINFOHEADER\")\n");
      else if (dib_length == 108) fprintf (stderr, "\"BITMAPV4HEADER\")\n");
      else if (dib_length == 124) fprintf (stderr, "\"BITMAPV5HEADER\")\n");
      else fprintf (stderr, "unknown)");
      fprintf (stderr, "    Bitmap Width:    %6d pixels\n", image_width);
      fprintf (stderr, "    Bitmap Height:   %6d pixels\n", image_height);
      fprintf (stderr, "    Color Planes:    %6d\n",        num_planes);
      fprintf (stderr, "    Bits per Pixel:  %6d\n",        bits_per_pixel);
      fprintf (stderr, "    Compression Method: %2d --> ", compression_method);
      if (compression_method <= MAX_COMPRESSION_METHOD) {
         fprintf (stderr, "%s", compression_type [compression_method]);
      }
      /*
         Supported compression method values:
              0 --> uncompressed RGB
             11 --> uncompressed CMYK
      */
      if (compression_method == 0 || compression_method == 11) {
         fprintf (stderr, " (no compression)");
      }
      else {
         fprintf (stderr, "Image uses compression; this is unsupported.\n\n");
         exit (EXIT_FAILURE);
      }
      fprintf (stderr, "\n");
      fprintf (stderr, "    Image Size:             %5d bytes\n", image_size);
      fprintf (stderr, "    Horizontal Resolution: %5d pixels/meter\n", hres);
      fprintf (stderr, "    Vertical Resolution:   %5d pixels/meter\n", vres);
      fprintf (stderr, "    Number of Colors:      %5d", num_colors);
      if (num_colors != true_colors) {
         fprintf (stderr, " --> %d", true_colors);
      }
      fputc ('\n', stderr);
      fprintf (stderr, "    Important Colors:      %5d", important_colors);
      if (important_colors == 0)
         fprintf (stderr, " (all colors are important)");
      fprintf (stderr, "\n\n");
   }  /* if (verbose) */

   /*
      Print Color Table information for images with pallettized colors.
   */
   if (bits_per_pixel <= 8) {
      for (i = 0; i < 2; i++) {
         color_map [i][0] = get_bytes (infp, 1);
         color_map [i][1] = get_bytes (infp, 1);
         color_map [i][2] = get_bytes (infp, 1);
         color_map [i][3] = get_bytes (infp, 1);
      }
      /* Skip remaining color table entries if more than 2 */
      while (i < true_colors) {
         (void) get_bytes (infp, 4);
         i++;
      }

      if (color_map [0][0] >= 128) image_xor = 0xFF;  /* Invert colors */
   }

   if (verbose) {
      fprintf (stderr, "Color Palette [R, G, B, %s] Values:\n",
               (dib_length <= 40) ? "reserved" : "Alpha");
      for (i = 0; i < 2; i++) {
         fprintf (stderr, "%7d: [", i);
         fprintf (stderr, "%3d,",    color_map [i][0] & 0xFF);
         fprintf (stderr, "%3d,",    color_map [i][1] & 0xFF);
         fprintf (stderr, "%3d,",    color_map [i][2] & 0xFF);
         fprintf (stderr, "%3d]\n", color_map [i][3] & 0xFF);
      }
      if (image_xor == 0xFF) fprintf (stderr, "Will Invert Colors.\n");
      fputc ('\n', stderr);

   }  /* if (verbose) */


   /*
      Check format before writing output file.
   */
   if (image_width != 560 && image_width != 576) {
      fprintf (stderr, "\nUnsupported image width: %d\n", image_width);
```

```
      fprintf (stderr, "Width should be 560 or 576 pixels.\n\n");
      exit (EXIT_FAILURE);
   }

   if (image_height != 544) {
      fprintf (stderr, "\nUnsupported image height: %d\n", image_height);
      fprintf (stderr, "Height should be 544 pixels.\n\n");
      exit (EXIT_FAILURE);
   }

   if (num_planes != 1) {
      fprintf (stderr, "\nUnsupported number of planes: %d\n", num_planes);
      fprintf (stderr, "Number of planes should be 1.\n\n");
      exit (EXIT_FAILURE);
   }

   if (bits_per_pixel != 1) {
      fprintf (stderr, "\nUnsupported number of bits per pixel: %d\n",
               bits_per_pixel);
      fprintf (stderr, "Bits per pixel should be 1.\n\n");
      exit (EXIT_FAILURE);
   }

   if (compression_method != 0 && compression_method != 11) {
      fprintf (stderr, "\nUnsupported compression method: %d\n",
               compression_method);
      fprintf (stderr, "Compression method should be 1 or 11.\n\n");
      exit (EXIT_FAILURE);
   }

   if (true_colors != 2) {
      fprintf (stderr, "\nUnsupported number of colors: %d\n", true_colors);
      fprintf (stderr, "Number of colors should be 2.\n\n");
      exit (EXIT_FAILURE);
   }


   /*
      If we made it this far, things look okay, so write out
      the standard header for image conversion.
   */
   for (i = 0; i < 62; i++) fputc (standard_header[i], outfp);


   /*
      Image Data.  Each row must be a multiple of 4 bytes, with
      padding at the end of each row if necessary.
   */
   k = 0;  /* byte number within the binary image */
   for (i = 0; i < 544; i++) {
      /*
         If original image is 560 pixels wide (not 576), add
         2 white bytes at beginning of row.
      */
      if (image_width == 560) {  /* Insert 2 white bytes */
         image_bytes[k++] = 0xFF;
         image_bytes[k++] = 0xFF;
      }
      for (j = 0; j < 70; j++) {  /* Copy next 70 bytes */
         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
      }
      /*
         If original image is 560 pixels wide (not 576), skip
         2 padding bytes at end of row in file because we inserted
         2 white bytes at the beginning of the row.
      */
      if (image_width == 560) {
         (void) get_bytes (infp, 2);
      }
      else {  /* otherwise, next 2 bytes are part of the image so copy them */
         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ^ image_xor;
      }
   }


   /*
      Change the image to match the unihex2bmp.c format if original wasn't
   */
   if (image_width == 560) {
      regrid (image_bytes);
```

```
    }

    for (i = 0; i < 544 * 576 / 8; i++) {
       fputc (image_bytes[i], outfp);
    }


    /*
       Wrap up.
    */
    fclose (infp);
    fclose (outfp);

    exit (EXIT_SUCCESS);
}
```

Here is the call graph for this function:

5.13.3.3 regrid()

void regrid (
    unsigned ∗ image_bytes)

After reading in the image, shift it.

This function adjusts the input image from an original PNG file to match unihex2bmp.c format.

Parameters

in,out	image_bytes	The pixels in an image.

Definition at line 514 of file unibmpbump.c.

```
00514                                   {
00515    int i, j, k;  /* loop variables */
00516    int offset;
00517    unsigned glyph_row; /* one grid row of 32 pixels */
00518    unsigned last_pixel; /* last pixel in a byte, to preserve */
00519
00520    /* To insert "00" after "U+" at top of image */
00521    char zero_pattern[16] = {
00522        0x00, 0x00, 0x00, 0x00, 0x18, 0x24, 0x42, 0x42,
00523        0x42, 0x42, 0x42, 0x42, 0x24, 0x18, 0x00, 0x00
00524    };
00525
00526    /* This is the horizontal grid pattern on glyph boundaries */
00527    unsigned hgrid[72] = {
00528       /*  0 */ 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xfe,
00529       /*  8 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00530       /* 16 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00531       /* 24 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00532       /* 32 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00533       /* 40 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00534       /* 48 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00535       /* 56 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
00536       /* 64 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00
00537    };
00538
00539
00540    /*
00541       First move "U+" left and insert "00" after it.
00542    */
00543    j = 15; /* rows are written bottom to top, so we'll decrement j */
00544    for (i = 543 - 8; i > 544 - 24; i--) {
00545       offset = 72 * i;
00546       image_bytes [offset + 0] = image_bytes [offset + 2];
00547       image_bytes [offset + 1] = image_bytes [offset + 3];
00548       image_bytes [offset + 2] = image_bytes [offset + 4];
```

```
      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;
}
```

Here is the caller graph for this function:

5.14 unibmpbump.c

Go to the documentation of this file.

```
/**
   @file unibmpbump.c

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

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

   @copyright Copyright (C) 2019 Paul Hardy

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

   Synopsis: unibmpbump [-iin_file.bmp] [-oout_file.bmp]
*/
/*
   LICENSE:

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

```

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

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

#define VERSION "1.0"   ///< Version of this program

#define MAX_COMPRESSION_METHOD 13   ///< Maximum supported compression method


/**
  @brief The main function.

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

  /*
     Values preserved from file header (first 14 bytes).
  */
  char file_format[3];        /* "BM" for original Windows format           */
  unsigned filesize;          /* size of file in bytes                       */
  unsigned char rsvd_hdr[4]; /* 4 reserved bytes                            */
  unsigned image_start;       /* byte offset of image in file                */

  /*
     Values preserved from Device Independent Bitmap (DIB) Header.

     The DIB fields below are in the standard 40-byte header.  Version
     4 and version 5 headers have more information, mainly for color
     information.  That is skipped over, because a valid glyph image
     is just monochrome.
  */
  int dib_length;             /* in bytes, for parsing by header version    */
  int image_width = 0;        /* Signed image width                          */
  int image_height = 0;       /* Signed image height                         */
  int num_planes = 0;         /* number of planes; must be 1                 */
  int bits_per_pixel = 0;     /* for palletized color maps (< 2^16 colors)  */
  /*
     The following fields are not in the original spec, so initialize
     them to 0 so we can correctly parse an original file format.
  */
  int compression_method=0;  /* 0 --> uncompressed RGB/monochrome           */
  int image_size = 0;         /* 0 is a valid size if no compression         */
  int hres = 0;               /* image horizontal resolution                 */
  int vres = 0;               /* image vertical resolution                   */
  int num_colors = 0;         /* Number of colors for pallettized images     */
  int important_colors = 0;  /* Number of significant colors (0 or 2)       */

  int true_colors = 0;        /* interpret num_colors, which can equal 0     */

  /*
     Color map.  This should be a monochrome file, so only two
     colors are stored.
  */
  unsigned char color_map[2][4]; /* two of R, G, B, and possibly alpha  */

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

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

  /*
     Temporary variables.
  */
```

```c
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) ˆ 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) ˆ image_xor;
         image_bytes[k++] = (get_bytes (infp, 1) & 0xFF) ˆ image_xor;
      }
   }


   /*
      Change the image to match the unihex2bmp.c format if original wasn't
   */
   if (image_width == 560) {
      regrid (image_bytes);
   }

   for (i = 0; i < 544 * 576 / 8; i++) {
      fputc (image_bytes[i], outfp);
   }


   /*
      Wrap up.
   */
   fclose (infp);
   fclose (outfp);

   exit (EXIT_SUCCESS);
}


/**
   @brief Get from 1 to 4 bytes, inclusive, from input file.

   @param[in] infp Pointer to input file.
   @param[in] nbytes Number of bytes to read, from 1 to 4, inclusive.
   @return The unsigned 1 to 4 bytes in machine native endian format.
*/
unsigned get_bytes (FILE *infp, int nbytes) {
   int i;
   unsigned char inchar[4];
   unsigned inword;

   for (i = 0; i < nbytes; i++) {
      if (fread (&inchar[i], 1, 1, infp) != 1) {
         inchar[i] = 0;
      }
   }
   for (i = nbytes; i < 4; i++) inchar[i] = 0;

   inword = ((inchar[3] & 0xFF) « 24) | ((inchar[2] & 0xFF) « 16) |
            ((inchar[1] & 0xFF) «  8) |  (inchar[0] & 0xFF);

   return inword;
}


/**
   @brief After reading in the image, shift it.

   This function adjusts the input image from an original PNG file
   to match unihex2bmp.c format.

```

```
   @param[in,out] image_bytes The pixels in an image.
*/
void regrid (unsigned *image_bytes) {
   int i, j, k;  /* loop variables */
   int offset;
   unsigned glyph_row; /* one grid row of 32 pixels */
   unsigned last_pixel; /* last pixel in a byte, to preserve */

   /* To insert ”00” after ”U+” at top of image */
   char zero_pattern[16] = {
       0x00, 0x00, 0x00, 0x00, 0x18, 0x24, 0x42, 0x42,
       0x42, 0x42, 0x42, 0x42, 0x24, 0x18, 0x00, 0x00
   };

   /* This is the horizontal grid pattern on glyph boundaries */
   unsigned hgrid[72] = {
      /*  0 */ 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xfe,
      /*  8 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 16 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 24 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 32 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 40 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 48 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 56 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00,
      /* 64 */ 0x00, 0x81, 0x81, 0x00, 0x00, 0x81, 0x81, 0x00
   };


   /*
      First move ”U+” left and insert ”00” after it.
   */
   j = 15; /* rows are written bottom to top, so we’ll decrement j */
   for (i = 543 - 8; i > 544 - 24; i--) {
      offset = 72 * i;
      image_bytes [offset + 0] = image_bytes [offset + 2];
      image_bytes [offset + 1] = image_bytes [offset + 3];
      image_bytes [offset + 2] = image_bytes [offset + 4];
      image_bytes [offset + 3] = image_bytes [offset + 4] =
         ~zero_pattern[15 - j--] & 0xFF;
   }

   /*
      Now move glyph bitmaps to the right by 8 pixels.
   */
   for (i = 0; i < 16; i++) { /* for each glyph row */
      for (j = 0; j < 16; j++) { /* for each glyph column */
         /* set offset to lower left-hand byte of next glyph */
         offset = (32 * 72 * i) + (9 * 72) + (4 * j) + 8;
         for (k = 0; k < 16; k++) { /* for each glyph row */
            glyph_row = (image_bytes [offset + 0] « 24) |
                        (image_bytes [offset + 1] « 16) |
                        (image_bytes [offset + 2] «  8) |
                        (image_bytes [offset + 3]);
            last_pixel = glyph_row & 1;  /* preserve border */
            glyph_row »= 4;
            glyph_row &=  0x0FFFFFFE;
            /* Set left 4 pixels to white and preserve last pixel */
            glyph_row |=  0xF0000000 | last_pixel;
            image_bytes [offset + 3] = glyph_row & 0xFF;
            glyph_row »= 8;
            image_bytes [offset + 2] = glyph_row & 0xFF;
            glyph_row »= 8;
            image_bytes [offset + 1] = glyph_row & 0xFF;
            glyph_row »= 8;
            image_bytes [offset + 0] = glyph_row & 0xFF;
            offset += 72;  /* move up to next row in current glyph */
         }
      }
   }

   /* Replace horizontal grid with unihex2bmp.c grid */
   for (i = 0; i <= 16; i++) {
      offset = 32 * 72 * i;
      for (j = 0; j < 72; j++) {
         image_bytes [offset + j] = hgrid [j];
      }
   }

   return;
}
```

5.15 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.

5.15.1 Detailed Description

unicoverage - Show the coverage of Unicode plane scripts for a GNU Unifont hex glyph file

Author

Paul Hardy, unifoundry <at> unifoundry.com, 6 January 2008

Copyright

Copyright (C) 2008, 2013 Paul Hardy

Synopsis: unicoverage [-ifont_file.hex] [-ocoverage_file.txt]

This program requires the file "coverage.dat" to be present in the directory from which it is run.

Definition in file unicoverage.c.

5.15.2 Macro Definition Documentation

5.15.2.1 MAXBUF

```
#define MAXBUF 256
```

Maximum input line length - 1.

Definition at line 60 of file unicoverage.c.

5.15.3 Function Documentation

5.15.3.1 main()

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

The main function.

Parameters

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

Returns

This program exits with status 0.

Definition at line 71 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         */

   /* to get next range & name of Unicode glyphs */
   int nextrange (FILE *coveragefp, int *cstart, int *cend, char *coverstring);

   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:

5.15.3.2 nextrange()

```
int nextrange (
        FILE * coveragefp,
        int * cstart,
        int * cend,
        char * coverstring )
```

Get next Unicode range.

This function reads the next Unicode script range to count its glyph coverage.

Parameters

in	coveragefp	File pointer to Unicode script range data file.
in	cstart	Starting code point in current Unicode script range.
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 192 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:

5.15.3.3 print_subtotal()

```
void print_subtotal (
        FILE * outfp,
        int print_n,
        int nglyphs,
        int cstart,
        int cend,
        char * coverstring )
```

Print the subtotal for one Unicode script range.

Parameters

in	outfp	Pointer to output file.
in	print_n	1 = print number of glyphs, 0 = print percentage.
in	nglyphs	Number of glyphs in current range.
in	cstart	Starting code point for current range.
in	cend	Ending code point for current range.
in	coverstring	Character string of "<cstart>-<cend>".

Definition at line 233 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:

5.16 unicoverage.c

Go to the documentation of this file.

```
/**
   @file unicoverage.c

   @brief unicoverage - Show the coverage of Unicode plane scripts
                        for a GNU Unifont hex glyph file

   @author Paul Hardy, unifoundry <at> unifoundry.com, 6 January 2008

   @copyright Copyright (C) 2008, 2013 Paul Hardy

   Synopsis: unicoverage [-ifont_file.hex] [-ocoverage_file.txt]

```

```
   This program requires the file "coverage.dat" to be present
   in the directory from which it is run.
*/
/*
   LICENSE:

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

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

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

/*
   2016 (Paul Hardy): Modified in Unifont 9.0.01 release to remove non-existent
   "-p" option and empty example from help printout.

   2018 (Paul Hardy): Modified to cover entire Unicode range, not just Plane 0.

   11 May 2019: [Paul Hardy] changed strcpy function call to strlcpy
   for better error handling.

   31 May 2019: [Paul Hardy] replaced strlcpy call with strncpy
   for compilation on more systems.

   4 June 2022: [Paul Hardy] Adjusted column spacing for better alignment
   of Unicode Plane 1-15 scripts.  Added "-n" option to print number of
   glyphs in each range instead of percent coverage.

   18 September 2022: [Paul Hardy] in nextrange function, initialize retval.

   21 October 2023: [Paul Hardy]
   Added full function prototype for nextrange function in main function.
*/

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


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


/**
   @brief The main function.

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

   int      print_n=0;        /* print # of glyphs, not percentage   */
   unsigned i;                /* loop variable                       */
   unsigned slen;             /* string length of coverage file line */
   char     inbuf[256];       /* input buffer                        */
   unsigned thischar;         /* the current character               */

   char *infile="", *outfile="";  /* names of input and output files        */
   FILE *infp, *outfp;        /* file pointers of input and output files    */
   FILE *coveragefp;          /* file pointer to coverage.dat file          */
   int cstart, cend;          /* current coverage start and end code points */
   char coverstring[MAXBUF];  /* description of current coverage range      */
   int nglyphs;               /* number of glyphs in this section           */

   /* to get next range & name of Unicode glyphs */
   int nextrange (FILE *coveragefp, int *cstart, int *cend, char *coverstring);

   void print_subtotal (FILE *outfp, int print_n, int nglyphs,
                        int cstart, int cend, char *coverstring);

   if ((coveragefp = fopen ("coverage.dat", "r")) == NULL) {
```

```c
      fprintf (stderr, "\nError: data file \"coverage.dat\" not found.\n\n");
      exit (0);
   }

   if (argc > 1) {
      for (i = 1; i < argc; i++) {
         if (argv[i][0] == '-') {  /* this is an option argument */
            switch (argv[i][1]) {
               case 'i':  /* name of input file */
                  infile = &argv[i][2];
                  break;
               case 'n':  /* print number of glyphs instead of percentage */
                  print_n = 1;
               case 'o':  /* name of output file */
                  outfile = &argv[i][2];
                  break;
               default:   /* if unrecognized option, print list and exit */
                  fprintf (stderr, "\nSyntax:\n\n");
                  fprintf (stderr, "   %s -p<Unicode_Page> ", argv[0]);
                  fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
                  exit (1);
            }
         }
      }
   }
   /*
      Make sure we can open any I/O files that were specified before
      doing anything else.
   */
   if (strlen (infile) > 0) {
      if ((infp = fopen (infile, "r")) == NULL) {
         fprintf (stderr, "Error: can't open %s for input.\n", infile);
         exit (1);
      }
   }
   else {
      infp = stdin;
   }
   if (strlen (outfile) > 0) {
      if ((outfp = fopen (outfile, "w")) == NULL) {
         fprintf (stderr, "Error: can't open %s for output.\n", outfile);
         exit (1);
      }
   }
   else {
      outfp = stdout;
   }

   /*
      Print header row.
   */
   if (print_n) {
      fprintf (outfp, "# Glyphs      Range          Script\n");
      fprintf (outfp, "--------      -----          ------\n");
   }
   else {
      fprintf (outfp, "Covered      Range          Script\n");
      fprintf (outfp, "-------      -----          ------\n\n");
   }

   slen = nextrange (coveragefp, &cstart, &cend, coverstring);
   nglyphs = 0;

   /*
      Read in the glyphs in the file
   */
   while (slen != 0 && fgets (inbuf, MAXBUF-1, infp) != NULL) {
      sscanf (inbuf, "%x", &thischar);

      /* Read a character beyond end of current script. */
      while (cend < thischar && slen != 0) {
         print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);

         /* start new range total */
         slen = nextrange (coveragefp, &cstart, &cend, coverstring);
         nglyphs = 0;
      }
      nglyphs++;
   }

   print_subtotal (outfp, print_n, nglyphs, cstart, cend, coverstring);
```

```
    exit (0);
}

/**
    @brief Get next Unicode range.

    This function reads the next Unicode script range to count its
    glyph coverage.

    @param[in] coveragefp File pointer to Unicode script range data file.
    @param[in] cstart Starting code point in current Unicode script range.
    @param[in] cend Ending code point in current Unicode script range.
    @param[out] coverstring String containing <cstart>-<cend> substring.
    @return Length of the last string read, or 0 for end of file.
*/
int
nextrange (FILE *coveragefp,
           int *cstart, int *cend,
           char *coverstring)
{
    int i;
    static char inbuf[MAXBUF];
    int retval;         /* the return value */

    retval = 0;

    do {
        if (fgets (inbuf, MAXBUF-1, coveragefp) != NULL) {
            retval = strlen (inbuf);
            if ((inbuf[0] >= '0' && inbuf[0] <= '9') ||
                (inbuf[0] >= 'A' && inbuf[0] <= 'F') ||
                (inbuf[0] >= 'a' && inbuf[0] <= 'f')) {
                sscanf (inbuf, "%x-%x", cstart, cend);
                i = 0;
                while (inbuf[i] != ' ') i++;  /* find first blank */
                while (inbuf[i] == ' ') i++;  /* find next non-blank */
                strncpy (coverstring, &inbuf[i], MAXBUF);
            }
            else retval = 0;
        }
        else retval = 0;
    } while (retval == 0 && !feof (coveragefp));

    return (retval);
}


/**
    @brief Print the subtotal for one Unicode script range.

    @param[in] outfp Pointer to output file.
    @param[in] print_n 1 = print number of glyphs, 0 = print percentage.
    @param[in] nglyphs Number of glyphs in current range.
    @param[in] cstart Starting code point for current range.
    @param[in] cend Ending code point for current range.
    @param[in] coverstring Character string of "<cstart>-<cend>".
*/
void print_subtotal (FILE *outfp, int print_n, int nglyphs,
                     int cstart, int cend, char *coverstring) {

    /* print old range total */
    if (print_n) {  /* Print number of glyphs, not percentage */
        fprintf (outfp, " %6d ", nglyphs);
    }
    else {
        fprintf (outfp, " %5.1f%%", 100.0*nglyphs/(1+cend-cstart));
    }

    if (cend < 0x10000)
        fprintf (outfp, "  U+%04X..U+%04X   %s",
                 cstart, cend, coverstring);
    else
        fprintf (outfp, " U+%05X..U+%05X  %s",
                 cstart, cend, coverstring);

    return;
}
```

5.17 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.

5.17.1 Detailed Description

unidup - Check for duplicate code points in sorted unifont.hex file

Author

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

Copyright

This program reads a sorted list of glyphs in Unifont .hex format and prints duplicate code points on stderr if any were detected.

Synopsis: unidup < unifont_file.hex

[Hopefully there won't be any output!]

Definition in file unidup.c.

5.17.2 Macro Definition Documentation

5.17.2.1 MAXBUF

#define MAXBUF 256

Maximum input line length - 1.

Definition at line 37 of file unidup.c.

5.17.3 Function Documentation

5.17.3.1 main()

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

The main function.

Parameters

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

Returns

This program exits with status 0.

Definition at line 48 of file unidup.c.

```
00049 {
00050
00051    int ix, iy;
00052    char inbuf[MAXBUF];
00053    char *infile;   /* the input file name */
00054    FILE *infilefp; /* file pointer to input file */
00055
00056    if (argc > 1) {
00057       infile = argv[1];
00058       if ((infilefp = fopen (infile, "r")) == NULL) {
00059          fprintf (stderr, "\nERROR: Can't open file %s\n\n", infile);
00060          exit (EXIT_FAILURE);
00061       }
00062    }
00063    else {
00064       infilefp = stdin;
00065    }
00066
00067    ix = -1;
00068
00069    while (fgets (inbuf, MAXBUF-1, infilefp) != NULL) {
00070       sscanf (inbuf, "%X", &iy);
00071       if (ix == iy) fprintf (stderr, "Duplicate code point: %04X\n", ix);
00072       else ix = iy;
00073    }
00074    exit (0);
00075 }
```

5.18 unidup.c

Go to the documentation of this file.

```c
/**
   @file unidup.c

   @brief unidup - Check for duplicate code points in sorted unifont.hex file

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

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

   This program reads a sorted list of glyphs in Unifont .hex format
   and prints duplicate code points on stderr if any were detected.

   Synopsis: unidup < unifont_file.hex

             [Hopefully there won't be any output!]
*/
/*
   LICENSE:

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

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

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

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

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


/**
   @brief The main function.

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

   int ix, iy;
   char inbuf[MAXBUF];
   char *infile;   /* the input file name */
   FILE *infilefp; /* file pointer to input file */

   if (argc > 1) {
      infile = argv[1];
      if ((infilefp = fopen (infile, "r")) == NULL) {
         fprintf (stderr, "\nERROR: Can't open file %s\n\n", infile);
         exit (EXIT_FAILURE);
      }
   }
   else {
      infilefp = stdin;
   }

   ix = -1;

   while (fgets (inbuf, MAXBUF-1, infilefp) != NULL) {
      sscanf (inbuf, "%X", &iy);
      if (ix == iy) fprintf (stderr, "Duplicate code point: %04X\n", ix);
      else ix = iy;
   }
   exit (0);
}
```

5.19 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.

5.19.1 Detailed Description

: Support functions for Unifont .hex files.

Author

Paul Hardy

Copyright

Definition in file unifont-support.c.

5.19.2 Function Documentation

5.19.2.1 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;
}
```

5.19.2.2 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;
}
```

5.19.2.3 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;
}
```

5.19.2.4 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;
}
```

5.19.2.5 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;
}
```

5.20 unifont-support.c

Go to the documentation of this file.

```
/**
   @file: unifont-support.c

   @brief: Support functions for Unifont .hex files.

   @author Paul Hardy

   @copyright Copyright © 2023 Paul Hardy
*/
/*
   LICENSE:

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

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

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


```

```
/**
  @brief 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).

  @param[in] hexstring The Unicode .hex string for one code point.
  @param[out] width The number of columns in a glyph with 16 rows.
  @param[out] codept The code point, contained in the first .hex file field.
  @param[out] glyph The Unifont glyph, as 16 rows by 1 or 2 bytes wide.
*/
void
parse_hex (char *hexstring,
           int *width,
           unsigned *codept,
           unsigned char glyph[16][2]) {

  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;
}


/**
  @brief 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.

  @param[in] width The number of columns in the glyph.
  @param[in] glyph The binary glyph, as a 16-row by 2-byte array.
  @param[out] glyphbits The converted glyph, as a 16-row, 16-column array.
*/
void
glyph2bits (int width,
            unsigned char glyph[16][2],
            unsigned char glyphbits [16][16]) {

  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;
}


/**
   @brief 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.

   @param[in] width The number of columns in the glyph.
   @param[in] glyphbits The 16-by-16 pixel glyph bits.
   @param[out] transpose The array of 2 sets of 8 ot 16 columns of 8 pixels.
*/
void
hexpose (int width,
         unsigned char glyphbits [16][16],
         unsigned char transpose [2][16]) {

   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;
}


/**
   @brief 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.

   @param[in] width The number of columns in the glyph.
   @param[in] codept The code point to appear in the output .hex string.
   @param[in] glyph The glyph, with each of 16 rows 1 or 2 bytes wide.
   @param[out] outstring The output string, in Unifont .hex format.
*/
void
glyph2string (int width, unsigned codept,
              unsigned char glyph [16][2],
              char *outstring) {

   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;
}


/**
   @brief 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.

   @param[in] width The number of columns in the glyph.
   @param[in] codept The code point to appear in the output .hex string.
   @param[in] transpose The transposed glyph, with 2 sets of 8-row data.
   @param[out] outstring The output string, in Unifont .hex format.
*/
void
xglyph2string (int width, unsigned codept,
               unsigned char transpose [2][16],
               char *outstring) {

   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;
}

```

5.21 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.

5.21.1 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

Definition in file unifont1per.c.

5.21.2 Macro Definition Documentation

5.21.2.1 MAXFILENAME

#define MAXFILENAME 20

Maximum size of a filename of the form "U+%06X.bmp".

Definition at line 60 of file unifont1per.c.

5.21.2.2 MAXSTRING

#define MAXSTRING 266

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

Definition at line 57 of file unifont1per.c.

5.21.3 Function Documentation

5.21.3.1 main()

int main ( )

The main function.

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 69 of file unifont1per.c.

```
00069 {
00070
00071    int i; /* loop variable */
00072
00073    /*
00074       Define bitmap header bytes
00075    */
00076    unsigned char header [62] = {
00077       /*
00078          Bitmap File Header -- 14 bytes
00079       */
00080       'B', 'M',        /* Signature          */
00081       0x7E, 0, 0, 0,   /* File Size          */
00082          0, 0, 0, 0,   /* Reserved           */
00083       0x3E, 0, 0, 0,   /* Pixel Array Offset */
00084
00085       /*
00086          Device Independent Bitmap Header -- 40 bytes
00087
00088          Image Width and Image Height are assigned final values
00089          based on the dimensions of each glyph.
00090       */
00091       0x28,    0,    0,    0,  /* DIB Header Size                 */
00092       0x10,    0,    0,    0,  /* Image Width = 16 pixels         */
00093       0xF0, 0xFF, 0xFF, 0xFF,  /* Image Height = -16 pixels       */
00094       0x01,    0,              /* Planes                          */
00095       0x01,    0,              /* Bits Per Pixel                  */
00096          0,    0,    0,    0,  /* Compression                     */
00097       0x40,    0,    0,    0,  /* Image Size                      */
00098       0x14, 0x0B,    0,    0,  /* X Pixels Per Meter = 72 dpi     */
00099       0x14, 0x0B,    0,    0,  /* Y Pixels Per Meter = 72 dpi     */
00100       0x02,    0,    0,    0,  /* Colors In Color Table           */
00101          0,    0,    0,    0,  /* Important Colors                */
00102
00103       /*
00104          Color Palette -- 8 bytes
00105       */
00106       0xFF, 0xFF, 0xFF, 0,  /* White */
00107          0,    0,    0, 0   /* Black */
00108    };
00109
00110    char instring[MAXSTRING]; /* input string                      */
00111    int  code_point;          /* current Unicode code point        */
00112    char glyph[MAXSTRING];    /* bitmap string for this glyph      */
00113    int  glyph_height=16;     /* for now, fixed at 16 pixels high  */
00114    int  glyph_width;         /* 8, 16, 24, or 32 pixels wide      */
00115    char filename[MAXFILENAME];/* name of current output file      */
00116    FILE *outfp;              /* file pointer to current output file */
00117
00118    int string_index;  /* pointer into hexadecimal glyph string */
00119    int nextbyte;      /* next set of 8 bits to print out       */
00120
00121    /* Repeat for each line in the input stream */
00122    while (fgets (instring, MAXSTRING - 1, stdin) != NULL) {
00123       /* Read next 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);
}
```

5.22 unifont1per.c

Go to the documentation of this file.

```
/**
   @file unifont1per.c

   @brief unifont1per - Read a Unifont .hex file from standard input and
          produce one glyph per ".bmp" bitmap file as output

   @author Paul Hardy, unifoundry <at> unifoundry.com, December 2016

   @copyright Copyright (C) 2016, 2017 Paul Hardy

   Each glyph is 16 pixels tall, and can be 8, 16, 24,
   or 32 pixels wide.  The width of each output graphic
   file is determined automatically by the width of each
   Unifont hex representation.

   This program creates files of the form "U+<codepoint>.bmp", 1 per glyph.

   Synopsis: unifont1per < unifont.hex
*/
```

```
/*
  LICENSE:

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

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

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

  Example:

    mkdir my-bmp
    cd my-bmp
    unifont1per < ../glyphs.hex

*/

/*
  11 May 2019 [Paul Hardy]:
    - Changed sprintf function call to snprintf for writing
   "filename" character string.
    - Defined MAXFILENAME to hold size of "filename" array
   for snprintf function call.
*/

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

/** Maximum size of an input line in a Unifont .hex file - 1. */
#define MAXSTRING 266

/** Maximum size of a filename of the form "U+%06X.bmp". */
#define MAXFILENAME 20


/**
  @brief The main function.

  @return This program exits with status EXIT_SUCCESS.
*/
int
main () {

  int i; /* loop variable */

  /*
    Define bitmap header bytes
  */
  unsigned char header [62] = {
    /*
      Bitmap File Header -- 14 bytes
    */
    'B', 'M',       /* Signature          */
    0x7E, 0, 0, 0,  /* File Size          */
      0, 0, 0, 0,   /* Reserved           */
    0x3E, 0, 0, 0,  /* Pixel Array Offset */

    /*
      Device Independent Bitmap Header -- 40 bytes

      Image Width and Image Height are assigned final values
      based on the dimensions of each glyph.
    */
    0x28,    0,    0,    0,  /* DIB Header Size              */
    0x10,    0,    0,    0,  /* Image Width = 16 pixels      */
    0xF0, 0xFF, 0xFF, 0xFF,  /* Image Height = -16 pixels    */
    0x01,    0,              /* Planes                       */
    0x01,    0,              /* Bits Per Pixel               */
       0,    0,    0,    0,  /* Compression                  */
    0x40,    0,    0,    0,  /* Image Size                   */
    0x14, 0x0B,    0,    0,  /* X Pixels Per Meter = 72 dpi  */
    0x14, 0x0B,    0,    0,  /* Y Pixels Per Meter = 72 dpi  */
    0x02,    0,    0,    0,  /* Colors In Color Table        */
```

```
      0,    0,    0,    0,  /* Important Colors                */

    /*
      Color Palette -- 8 bytes
    */
    0xFF, 0xFF, 0xFF, 0,  /* White */
      0,    0,    0, 0   /* Black */
  };

  char instring[MAXSTRING]; /* input string                          */
  int  code_point;          /* current Unicode code point            */
  char glyph[MAXSTRING];    /* bitmap string for this glyph          */
  int  glyph_height=16;     /* for now, fixed at 16 pixels high     */
  int  glyph_width;         /* 8, 16, 24, or 32 pixels wide         */
  char filename[MAXFILENAME];/* name of current output file          */
  FILE *outfp;              /* file pointer to current output file */

  int string_index;  /* pointer into hexadecimal glyph string */
  int nextbyte;      /* next set of 8 bits to print out        */

  /* Repeat for each line in the input stream */
  while (fgets (instring, MAXSTRING - 1, stdin) != NULL) {
    /* Read next Unifont ASCII hexadecimal format glyph description */
    sscanf (instring, "%X:%s", &code_point, glyph);
    /* Calculate width of a glyph in pixels; 4 bits per ASCII hex digit */
    glyph_width = strlen (glyph) / (glyph_height / 4);
    snprintf (filename, MAXFILENAME, "U+%06X.bmp", code_point);
    header [18] =  glyph_width;  /* bitmap width */
    header [22] = -glyph_height; /* negative height --> draw top to bottom */
    if ((outfp = fopen (filename, "w")) != NULL) {
      for (i = 0; i < 62; i++) fputc (header[i], outfp);
      /*
        Bitmap, with each row padded with zeroes if necessary
        so each row is four bytes wide.  (Each row must end
        on a four-byte boundary, and four bytes is the maximum
        possible row length for up to 32 pixels in a row.)
      */
      string_index = 0;
      for (i = 0; i < glyph_height; i++) {
        /* Read 2 ASCII hexadecimal digits (1 byte of output pixels) */
        sscanf (&glyph[string_index], "%2X", &nextbyte);
        string_index += 2;
        fputc (nextbyte, outfp);  /* write out the 8 pixels    */
        if (glyph_width <= 8) {   /* pad row with 3 zero bytes */
          fputc (0x00, outfp); fputc (0x00, outfp); fputc (0x00, outfp);
        }
        else { /* get 8 more pixels */
          sscanf (&glyph[string_index], "%2X", &nextbyte);
          string_index += 2;
          fputc (nextbyte, outfp);  /* write out the 8 pixels    */
          if (glyph_width <= 16) {  /* pad row with 2 zero bytes */
            fputc (0x00, outfp); fputc (0x00, outfp);
          }
          else { /* get 8 more pixels */
            sscanf (&glyph[string_index], "%2X", &nextbyte);
            string_index += 2;
            fputc (nextbyte, outfp);  /* write out the 8 pixels   */
            if (glyph_width <= 24) {  /* pad row with 1 zero byte */
              fputc (0x00, outfp);
            }
            else { /* get 8 more pixels */
              sscanf (&glyph[string_index], "%2X", &nextbyte);
              string_index += 2;
              fputc (nextbyte, outfp); /* write out the 8 pixels */
            }  /* glyph is 32 pixels wide */
          }  /* glyph is 24 pixels wide */
        }  /* glyph is 16 pixels wide */
      }  /* glyph is 8 pixels wide */

      fclose (outfp);
    }
  }

  exit (EXIT_SUCCESS);
}
```

5.23 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.

5.23.1 Detailed Description

unifontpic - See the "Big Picture": the entire Unifont in one BMP bitmap

Author

Paul Hardy, 2013

Copyright

Copyright (C) 2013, 2017 Paul Hardy

Definition in file unifontpic.c.

5.23.2 Macro Definition Documentation

5.23.2.1 HDR_LEN

#define HDR_LEN 33

Define length of header string for top of chart.

Definition at line 73 of file unifontpic.c.

5.23.3 Function Documentation

5.23.3.1 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 303 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:

5.23.3.2 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 590 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 */
```

```c
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:

5.23.3.3 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 224 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:

5.23.3.4 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 93 of file unifontpic.c.

```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     (char *instring, int plane_array[0x10000][16], int plane);
   void genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum,
                    int plane);
   void genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum,
                    int plane);

   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++) {
      /* convert Unifont hexadecimal string to bitmap */
      gethex ((char *)ascii_hex[i], plane_array, 0);
      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:

5.23.3.5 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 203 of file unifontpic.c.

```
{

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

    return;
}
```

Here is the caller graph for this function:

5.23.3.6 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 185 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:

5.24 unifontpic.c

Go to the documentation of this file.

```
/**
   @file unifontpic.c

   @brief unifontpic - See the "Big Picture": the entire Unifont
                      in one BMP bitmap

   @author Paul Hardy, 2013

   @copyright Copyright (C) 2013, 2017 Paul Hardy
*/
/*
   LICENSE:

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

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

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

/*
     11 June 2017 [Paul Hardy]:
        -  Modified to take glyphs that are 24 or 32 pixels wide and
           compress them horizontally by 50%.

     8 July 2017 [Paul Hardy]:
        - Modified to print Unifont charts above Unicode Plane 0.
        - Adds "-P" option to specify Unicode plane in decimal,
          as "-P0" through "-P17".  Omitting this argument uses
          plane 0 as the default.
        - Appends Unicode plane number to chart title.
        - Reads in "unifontpic.h", which was added mainly to
          store ASCII chart title glyphs in an embedded array
          rather than requiring these ASCII glyphs to be in
          the ".hex" file that is read in for the chart body
          (which was the case previously, when all that was
          able to print was Unicode place 0).
        - Fixes truncated header in long bitmap format, making
          the long chart title glyphs single-spaced.  This leaves
          room for the Unicode plane to appear even in the narrow
          chart title of the "long" format chart.  The wide chart
          title still has double-spaced ASCII glyphs.
        - Adjusts centering of title on long and wide charts.

     11 May 2019 [Paul Hardy]:
        - Changed strncpy calls to memcpy.
        - Added "HDR_LEN" to define length of header string
      for use in snprintf function call.
    - Changed sprintf function calls to snprintf function
      calls for writing chart header string.

     21 October 2023 [Paul Hardy]:
    - Added full function prototypes in main function for
      functions gethex, genlongbmp, and genwidebmp.
    - Typecast ascii_hex[i] to char * in gethex function call
      to avoid warning about const char * conversion.
*/


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

/** Define length of header string for top of chart. */
#define HDR_LEN 33


/*
```

```
  Stylistic Note:

  Many variables in this program use multiple words scrunched
  together, with each word starting with an upper-case letter.
  This is only done to match the canonical field names in the
  Windows Bitmap Graphics spec.
*/

/**
  @brief The main function.

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

  /* long and dpi are set from command-line options */
  int wide=1; /* =1 for a 256x256 grid, =0 for a 16x4096 grid */
  int dpi=96; /* change for 256x256 grid to fit paper if desired */
  int tinynum=0; /* whether to use tiny labels for 256x256 grid */

  int i, j; /* loop variables */

  int plane=0;      /* Unicode plane, 0..17; Plane 0 is default */
  /* 16 pixel rows for each of 65,536 glyphs in a Unicode plane */
  int plane_array[0x10000][16];

  void gethex     (char *instring, int plane_array[0x10000][16], int plane);
  void genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum,
                   int plane);
  void genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum,
                   int plane);

  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++) {
    /* convert Unifont hexadecimal string to bitmap */
    gethex ((char *)ascii_hex[i], plane_array, 0);
    for (j = 0; j < 16; j++) ascii_bits[i][j] = plane_array[i][j];
  }


  /*
    Read in the Unifont hex file to render from standard input
```

```
   */
   memset ((void *)plane_array, 0, 0x10000 * 16 * sizeof (int));
   while (fgets (instring, MAXSTRING, stdin) != NULL) {
      gethex (instring, plane_array, plane); /* read .hex input file and fill plane_array with glyph data */
   }  /* while not EOF */


   /*
      Write plane_array glyph data to BMP file as wide or long bitmap.
   */
   if (wide) {
      genwidebmp (plane_array, dpi, tinynum, plane);
   }
   else {
      genlongbmp (plane_array, dpi, tinynum, plane);
   }

   exit (EXIT_SUCCESS);
}


/**
   @brief Output a 4-byte integer in little-endian order.

   @param[in] thisword The 4-byte integer to output as binary data.
*/
void
output4 (int thisword)
{

   putchar ( thisword        & 0xFF);
   putchar ((thisword »  8) & 0xFF);
   putchar ((thisword » 16) & 0xFF);
   putchar ((thisword » 24) & 0xFF);

   return;
}


/**
   @brief Output a 2-byte integer in little-endian order.

   @param[in] thisword The 2-byte integer to output as binary data.
*/
void
output2 (int thisword)
{

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

   return;
}


/**
   @brief Read a Unifont .hex-format input file from stdin.

   Each glyph can be 2, 4, 6, or 8 ASCII hexadecimal digits wide.
   Glyph height is fixed at 16 pixels.

   @param[in] instring One line from a Unifont .hex-format file.
   @param[in,out] plane_array Bitmap for this plane, one bitmap row per element.
   @param[in] plane The Unicode plane, 0..17.
*/
void
gethex (char *instring, int plane_array[0x10000][16], int plane)
{
   char *bitstring;  /* pointer into instring for glyph bitmap */
   int i;        /* loop variable                              */
   int codept;  /* the Unicode code point of the current glyph */
   int glyph_plane; /* Unicode plane of current glyph          */
   int ndigits; /* number of ASCII hexadecimal digits in glyph */
   int bytespl; /* bytes per line of pixels in a glyph         */
   int temprow; /* 1 row of a quadruple-width glyph            */
   int newrow;  /* 1 row of double-width output pixels         */
   unsigned bitmask; /* to mask off 2 bits of long width glyph */

   /*
      Read each input line and place its glyph into the bit array.
   */
```

```
   sscanf (instring, ”%X”, &codept);
   glyph_plane = codept » 16;
   if (glyph_plane == plane) {
      codept &= 0xFFFF;  /* array index will only have 16 bit address */
      /* find the colon separator */
      for (i = 0; (i < 9) && (instring[i] != ':'); i++);
      i++; /* position past it */
      bitstring = &instring[i];
      ndigits = strlen (bitstring);
      /* don't count '\n' at end of line if present */
      if (bitstring[ndigits - 1] == '\n') ndigits--;
      bytespl = ndigits » 5;  /* 16 rows per line, 2 digits per byte */

      if (bytespl >= 1 && bytespl <= 4) {
         for (i = 0; i < 16; i++) { /* 16 rows per glyph */
            /* Read correct number of hexadecimal digits given glyph width */
            switch (bytespl) {
               case 1: sscanf (bitstring, ”%2X”, &temprow);
                       bitstring += 2;
                       temprow «= 8; /* left-justify single-width glyph */
                       break;
               case 2: sscanf (bitstring, ”%4X”, &temprow);
                       bitstring += 4;
                       break;
               /* cases 3 and 4 widths will be compressed by 50% (see below) */
               case 3: sscanf (bitstring, ”%6X”, &temprow);
                       bitstring += 6;
                       temprow «= 8; /* left-justify */
                       break;
               case 4: sscanf (bitstring, ”%8X”, &temprow);
                       bitstring += 8;
                       break;
            }  /* switch on number of bytes per row */
            /* compress glyph width by 50% if greater than double-width */
            if (bytespl > 2) {
               newrow = 0x0000;
               /* mask off 2 bits at a time to convert each pair to 1 bit out */
               for (bitmask = 0xC0000000; bitmask != 0; bitmask »= 2) {
                  newrow «= 1;
                  if ((temprow & bitmask) != 0) newrow |= 1;
               }
               temprow = newrow;
            }  /* done conditioning glyphs beyond double-width */
            plane_array[codept][i] = temprow;  /* store glyph bitmap for output */
         }  /* for each row */
      }  /* if 1 to 4 bytes per row/line */
   }  /* if this is the plane we are seeking */

   return;
}


/**
   @brief Generate the BMP output file in long format.

   This function generates the BMP output file from a bitmap parameter.
   This is a long bitmap, 16 glyphs wide by 4,096 glyphs tall.

   @param[in] plane_array The array of glyph bitmaps for a plane.
   @param[in] dpi Dots per inch, for encoding in the BMP output file header.
   @param[in] tinynum Whether to generate tiny numbers in wide grid (unused).
   @param[in] plane The Unicode plane, 0..17.
*/
void
genlongbmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
{

   char header_string[HDR_LEN]; /* centered header              */
   char raw_header[HDR_LEN];    /* left-aligned header          */
   int header[16][16];      /* header row, for chart title */
   int hdrlen;              /* length of HEADER_STRING      */
   int startcol;            /* column to start printing header, for centering */

   unsigned leftcol[0x1000][16]; /* code point legend on left side of chart */
   int d1, d2, d3, d4;            /* digits for filling leftcol[][] legend   */
   int codept;                    /* current starting code point for legend  */
   int thisrow;                   /* glyph row currently being rendered       */
   unsigned toprow[16][16];       /* code point legend on top of chart        */
   int digitrow;        /* row we're in (0..4) for the above hexdigit digits */

   /*
```

```
   DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
*/
int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
int ImageSize;
int FileSize;
int Width, Height; /* bitmap image width and height in pixels */
int ppm;      /* integer pixels per meter */

int i, j, k;

unsigned bytesout;

void output4(int), output2(int);

/*
   Image width and height, in pixels.

      N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
*/
Width  =   18 * 16;  /* (2 legend +   16 glyphs) * 16 pixels/glyph */
Height = 4099 * 16;  /* (1 header + 4096 glyphs) * 16 rows/glyph   */

ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */

FileSize = DataOffset + ImageSize;

/* convert dots/inch to pixels/meter */
if (dpi == 0) dpi = 96;
ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);

/*
   Generate the BMP Header
*/
putchar ('B');
putchar ('M');

/*
   Calculate file size:

      BMP Header + InfoHeader + Color Table + Raster Data
*/
output4 (FileSize);  /* FileSize */
output4 (0x0000); /* reserved */

/* Calculate DataOffset */
output4 (DataOffset);

/*
   InfoHeader
*/
output4 (40);         /* Size of InfoHeader                       */
output4 (Width);      /* Width of bitmap in pixels                */
output4 (Height);     /* Height of bitmap in pixels               */
output2 (1);          /* Planes (1 plane)                         */
output2 (1);          /* BitCount (1 = monochrome)                */
output4 (0);          /* Compression (0 = none)                   */
output4 (ImageSize);  /* ImageSize, in bytes                      */
output4 (ppm);        /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (ppm);        /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (2);          /* ColorsUsed (= 2)                         */
output4 (2);          /* ColorsImportant (= 2)                    */
output4 (0x00000000); /* black (reserved, B, G, R)                */
output4 (0x00FFFFFF); /* white (reserved, B, G, R)                */

/*
   Create header row bits.
*/
snprintf (raw_header, HDR_LEN, "%s Plane %d", HEADER_STRING, plane);
memset ((void *)header, 0, 16 * 16 * sizeof (int)); /* fill with white */
memset ((void *)header_string, ' ', 32 * sizeof (char)); /* 32 spaces */
header_string[32] = '\0';  /* null-terminated */

hdrlen = strlen (raw_header);
if (hdrlen > 32) hdrlen = 32;        /* only 32 columns to print header */
startcol = 16 - ((hdrlen + 1) >> 1); /* to center header                */
/* center up to 32 chars */
memcpy (&header_string[startcol], raw_header, hdrlen);

/* Copy each letter's bitmap from the plane_array[][] we constructed. */
/* Each glyph must be single-width, to fit two glyphs in 16 pixels */
for (j = 0; j < 16; j++) {
```

```
      for (i = 0; i < 16; i++) {
         header[i][j] =
            (ascii_bits[header_string[j+j  ] & 0x7F][i] & 0xFF00) |
            (ascii_bits[header_string[j+j+1] & 0x7F][i] » 8);
      }
   }

   /*
      Create the left column legend.
   */
   memset ((void *)leftcol, 0, 4096 * 16 * sizeof (unsigned));

   for (codept = 0x0000; codept < 0x10000; codept += 0x10) {
      d1 = (codept » 12) & 0xF; /* most significant hex digit */
      d2 = (codept »  8) & 0xF;
      d3 = (codept »  4) & 0xF;

      thisrow = codept » 4; /* rows of 16 glyphs */

      /* fill in first and second digits */
      for (digitrow = 0; digitrow < 5; digitrow++) {
         leftcol[thisrow][2 + digitrow] =
            (hexdigit[d1][digitrow] « 10) |
            (hexdigit[d2][digitrow] «  4);
      }

      /* fill in third digit */
      for (digitrow = 0; digitrow < 5; digitrow++) {
         leftcol[thisrow][9 + digitrow] = hexdigit[d3][digitrow] « 10;
      }
      leftcol[thisrow][9 + 4] |= 0xF « 4; /* underscore as 4th digit */

      for (i = 0; i < 15; i ++) {
         leftcol[thisrow][i] |= 0x00000002;      /* right border */
      }

      leftcol[thisrow][15] = 0x0000FFFE;        /* bottom border */

      if (d3 == 0xF) {                     /* 256-point boundary */
         leftcol[thisrow][15] |= 0x00FF0000;  /* longer tic mark */
      }

      if ((thisrow % 0x40) == 0x3F) {    /* 1024-point boundary */
         leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
      }
   }

   /*
      Create the top row legend.
   */
   memset ((void *)toprow, 0, 16 * 16 * sizeof (unsigned));

   for (codept = 0x0; codept <= 0xF; codept++) {
      d1 = (codept » 12) & 0xF; /* most significant hex digit */
      d2 = (codept »  8) & 0xF;
      d3 = (codept »  4) & 0xF;
      d4 =  codept        & 0xF; /* least significant hex digit */

      /* fill in last digit */
      for (digitrow = 0; digitrow < 5; digitrow++) {
         toprow[6 + digitrow][codept] = hexdigit[d4][digitrow] « 6;
      }
   }

   for (j = 0; j < 16; j++) {
      /* force bottom pixel row to be white, for separation from glyphs */
      toprow[15][j] = 0x0000;
   }

   /* 1 pixel row with left-hand legend line */
   for (j = 0; j < 16; j++) {
      toprow[14][j] |= 0xFFFF;
   }

   /* 14 rows with line on left to fill out this character row */
   for (i = 13; i >= 0; i--) {
      for (j = 0; j < 16; j++) {
         toprow[i][j] |= 0x0001;
      }
   }

```

```
/*
   Now write the raster image.

   XOR each byte with 0xFF because black = 0, white = 1 in BMP.
*/

/* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
for (i = 0xFFF0; i >= 0; i -= 0x10) {
   thisrow = i >> 4; /* 16 glyphs per row */
   for (j = 15; j >= 0; j--) {
      /* left-hand legend */
      putchar ((~leftcol[thisrow][j] >> 24) & 0xFF);
      putchar ((~leftcol[thisrow][j] >> 16) & 0xFF);
      putchar ((~leftcol[thisrow][j] >>  8) & 0xFF);
      putchar ( ~leftcol[thisrow][j]        & 0xFF);
      /* Unifont glyph */
      for (k = 0; k < 16; k++) {
         bytesout = ~plane_array[i+k][j] & 0xFFFF;
         putchar ((bytesout >> 8) & 0xFF);
         putchar ( bytesout       & 0xFF);
      }
   }
}

/*
   Write the top legend.
*/
/* i == 15: bottom pixel row of header is output here */
/* left-hand legend: solid black line except for right-most pixel */
putchar (0x00);
putchar (0x00);
putchar (0x00);
putchar (0x01);
for (j = 0; j < 16; j++) {
   putchar ((~toprow[15][j] >> 8) & 0xFF);
   putchar ( ~toprow[15][j]       & 0xFF);
}

putchar (0xFF);
putchar (0xFF);
putchar (0xFF);
putchar (0xFC);
for (j = 0; j < 16; j++) {
   putchar ((~toprow[14][j] >> 8) & 0xFF);
   putchar ( ~toprow[14][j]       & 0xFF);
}

for (i = 13; i >= 0; i--) {
   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFD);
   for (j = 0; j < 16; j++) {
      putchar ((~toprow[i][j] >> 8) & 0xFF);
      putchar ( ~toprow[i][j]       & 0xFF);
   }
}

/*
   Write the header.
*/

/* 7 completely white rows */
for (i = 7; i >= 0; i--) {
   for (j = 0; j < 18; j++) {
      putchar (0xFF);
      putchar (0xFF);
   }
}

for (i = 15; i >= 0; i--) {
   /* left-hand legend */
   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFF);
   putchar (0xFF);
   /* header glyph */
   for (j = 0; j < 16; j++) {
      bytesout = ~header[i][j] & 0xFFFF;
      putchar ((bytesout >> 8) & 0xFF);
      putchar ( bytesout       & 0xFF);
```

```
      }
   }

   /* 8 completely white rows at very top */
   for (i = 7; i >= 0; i--) {
      for (j = 0; j < 18; j++) {
      putchar (0xFF);
      putchar (0xFF);
      }
   }

   return;
}


/**
   @brief Generate the BMP output file in wide format.

   This function generates the BMP output file from a bitmap parameter.
   This is a wide bitmap, 256 glyphs wide by 256 glyphs tall.

   @param[in] plane_array The array of glyph bitmaps for a plane.
   @param[in] dpi Dots per inch, for encoding in the BMP output file header.
   @param[in] tinynum Whether to generate tiny numbers in 256x256 grid.
   @param[in] plane The Unicode plane, 0..17.
*/
void
genwidebmp (int plane_array[0x10000][16], int dpi, int tinynum, int plane)
{

   char header_string[257];
   char raw_header[HDR_LEN];
   int header[16][256]; /* header row, for chart title */
   int hdrlen;          /* length of HEADER_STRING */
   int startcol;        /* column to start printing header, for centering */

   unsigned leftcol[0x100][16]; /* code point legend on left side of chart */
   int d1, d2, d3, d4;          /* digits for filling leftcol[][] legend   */
   int codept;                  /* current starting code point for legend  */
   int thisrow;                 /* glyph row currently being rendered      */
   unsigned toprow[32][256];    /* code point legend on top of chart       */
   int digitrow;        /* row we're in (0..4) for the above hexdigit digits */
   int hexalpha1, hexalpha2;    /* to convert hex digits to ASCII          */

   /*
      DataOffset = BMP Header bytes + InfoHeader bytes + ColorTable bytes.
   */
   int DataOffset = 14 + 40 + 8; /* fixed size for monochrome BMP */
   int ImageSize;
   int FileSize;
   int Width, Height; /* bitmap image width and height in pixels */
   int ppm;      /* integer pixels per meter */

   int i, j, k;

   unsigned bytesout;

   void output4(int), output2(int);

   /*
      Image width and height, in pixels.

         N.B.: Width must be an even multiple of 32 pixels, or 4 bytes.
   */
   Width  = 258 * 16;  /* (           2 legend + 256 glyphs) * 16 pixels/glyph */
   Height = 260 * 16;  /* (2 header + 2 legend + 256 glyphs) * 16 rows/glyph   */

   ImageSize = Height * (Width / 8); /* in bytes, calculated from pixels */

   FileSize = DataOffset + ImageSize;

   /* convert dots/inch to pixels/meter */
   if (dpi == 0) dpi = 96;
   ppm = (int)((double)dpi * 100.0 / 2.54 + 0.5);

   /*
      Generate the BMP Header
   */
   putchar ('B');
   putchar ('M');
   /*
```

```
   Calculate file size:

      BMP Header + InfoHeader + Color Table + Raster Data
*/
output4 (FileSize);  /* FileSize */
output4 (0x0000); /* reserved */
/* Calculate DataOffset */
output4 (DataOffset);

/*
   InfoHeader
*/
output4 (40);         /* Size of InfoHeader                       */
output4 (Width);      /* Width of bitmap in pixels                */
output4 (Height);     /* Height of bitmap in pixels               */
output2 (1);          /* Planes (1 plane)                         */
output2 (1);          /* BitCount (1 = monochrome)                */
output4 (0);          /* Compression (0 = none)                   */
output4 (ImageSize);  /* ImageSize, in bytes                      */
output4 (ppm);        /* XpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (ppm);        /* YpixelsPerM (96 dpi = 3780 pixels/meter) */
output4 (2);          /* ColorsUsed (= 2)                         */
output4 (2);          /* ColorsImportant (= 2)                    */
output4 (0x00000000); /* black (reserved, B, G, R)                */
output4 (0x00FFFFFF); /* white (reserved, B, G, R)                */

/*
   Create header row bits.
*/
snprintf (raw_header, HDR_LEN, ”%s Plane %d”, HEADER_STRING, plane);
memset ((void *)header, 0, 256 * 16 * sizeof (int)); /* fill with white */
memset ((void *)header_string, ’ ’, 256 * sizeof (char)); /* 256 spaces */
header_string[256] = ’\0’;  /* null-terminated */

hdrlen = strlen (raw_header);
/* Wide bitmap can print 256 columns, but limit to 32 columns for long bitmap. */
if (hdrlen > 32) hdrlen = 32;
startcol = 127 - ((hdrlen - 1) » 1);  /* to center header */
/* center up to 32 chars */
memcpy (&header_string[startcol], raw_header, hdrlen);

/* Copy each letter’s bitmap from the plane_array[][] we constructed. */
for (j = 0; j < 256; j++) {
   for (i = 0; i < 16; i++) {
      header[i][j] = ascii_bits[header_string[j] & 0x7F][i];
   }
}

/*
   Create the left column legend.
*/
memset ((void *)leftcol, 0, 256 * 16 * sizeof (unsigned));

for (codept = 0x0000; codept < 0x10000; codept += 0x100) {
   d1 = (codept » 12) & 0xF; /* most significant hex digit */
   d2 = (codept »  8) & 0xF;

   thisrow = codept » 8; /* rows of 256 glyphs */

   /* fill in first and second digits */

   if (tinynum) { /* use 4x5 pixel glyphs */
      for (digitrow = 0; digitrow < 5; digitrow++) {
         leftcol[thisrow][6 + digitrow] =
            (hexdigit[d1][digitrow] « 10) |
            (hexdigit[d2][digitrow] «  4);
      }
   }
   else { /* bigger numbers -- use glyphs from Unifont itself */
      /* convert hexadecimal digits to ASCII equivalent */
      hexalpha1 = d1 < 0xA ? ’0’ + d1 : ’A’ + d1 - 0xA;
      hexalpha2 = d2 < 0xA ? ’0’ + d2 : ’A’ + d2 - 0xA;

      for (i = 0 ; i < 16; i++) {
         leftcol[thisrow][i] =
            (ascii_bits[hexalpha1][i] « 2) |
            (ascii_bits[hexalpha2][i] » 6);
      }
   }

   for (i = 0; i < 15; i ++) {
```

```
        leftcol[thisrow][i] |= 0x00000002;      /* right border */
    }

    leftcol[thisrow][15] = 0x0000FFFE;         /* bottom border */

    if (d2 == 0xF) {                           /* 4096-point boundary */
        leftcol[thisrow][15] |= 0x00FF0000;  /* longer tic mark */
    }

    if ((thisrow % 0x40) == 0x3F) {    /* 16,384-point boundary */
        leftcol[thisrow][15] |= 0xFFFF0000; /* longest tic mark */
    }
}

/*
    Create the top row legend.
*/
memset ((void *)toprow, 0, 32 * 256 * sizeof (unsigned));

for (codept = 0x00; codept <= 0xFF; codept++) {
    d3 = (codept »  4) & 0xF;
    d4 =  codept        & 0xF; /* least significant hex digit */

    if (tinynum) {
        for (digitrow = 0; digitrow < 5; digitrow++) {
            toprow[16 + 6 + digitrow][codept] =
                (hexdigit[d3][digitrow] « 10) |
                (hexdigit[d4][digitrow] «  4);
        }
    }
    else {
        /* convert hexadecimal digits to ASCII equivalent */
        hexalpha1 = d3 < 0xA ? '0' + d3 : 'A' + d3 - 0xA;
        hexalpha2 = d4 < 0xA ? '0' + d4 : 'A' + d4 - 0xA;
        for (i = 0 ; i < 16; i++) {
            toprow[14 + i][codept] =
                (ascii_bits[hexalpha1][i]     ) |
                (ascii_bits[hexalpha2][i] » 7);
        }
    }
}

for (j = 0; j < 256; j++) {
    /* force bottom pixel row to be white, for separation from glyphs */
    toprow[16 + 15][j] = 0x0000;
}

/* 1 pixel row with left-hand legend line */
for (j = 0; j < 256; j++) {
    toprow[16 + 14][j] |= 0xFFFF;
}

/* 14 rows with line on left to fill out this character row */
for (i = 13; i >= 0; i--) {
    for (j = 0; j < 256; j++) {
        toprow[16 + i][j] |= 0x0001;
    }
}

/* Form the longer tic marks in top legend */
for (i = 8; i < 16; i++) {
    for (j = 0x0F; j < 0x100; j += 0x10) {
        toprow[i][j] |= 0x0001;
    }
}

/*
    Now write the raster image.

    XOR each byte with 0xFF because black = 0, white = 1 in BMP.
*/

/* Write the glyphs, bottom-up, left-to-right, in rows of 16 (i.e., 0x10) */
for (i = 0xFF00; i >= 0; i -= 0x100) {
    thisrow = i » 8; /* 256 glyphs per row */
    for (j = 15; j >= 0; j--) {
        /* left-hand legend */
        putchar ((~leftcol[thisrow][j] » 24) & 0xFF);
        putchar ((~leftcol[thisrow][j] » 16) & 0xFF);
        putchar ((~leftcol[thisrow][j] »  8) & 0xFF);
        putchar ( ~leftcol[thisrow][j]        & 0xFF);
```

```
        /* Unifont glyph */
        for (k = 0x00; k < 0x100; k++) {
           bytesout = ~plane_array[i+k][j] & 0xFFFF;
           putchar ((bytesout » 8) & 0xFF);
           putchar ( bytesout       & 0xFF);
        }
     }
  }

  /*
     Write the top legend.
  */
  /* i == 15: bottom pixel row of header is output here */
  /* left-hand legend: solid black line except for right-most pixel */
  putchar (0x00);
  putchar (0x00);
  putchar (0x00);
  putchar (0x01);
  for (j = 0; j < 256; j++) {
     putchar ((~toprow[16 + 15][j] » 8) & 0xFF);
     putchar ( ~toprow[16 + 15][j]       & 0xFF);
  }

  putchar (0xFF);
  putchar (0xFF);
  putchar (0xFF);
  putchar (0xFC);
  for (j = 0; j < 256; j++) {
     putchar ((~toprow[16 + 14][j] » 8) & 0xFF);
     putchar ( ~toprow[16 + 14][j]       & 0xFF);
  }

  for (i = 16 + 13; i >= 0; i--) {
     if (i >= 8) { /* make vertical stroke on right */
        putchar (0xFF);
        putchar (0xFF);
        putchar (0xFF);
        putchar (0xFD);
     }
     else { /* all white */
        putchar (0xFF);
        putchar (0xFF);
        putchar (0xFF);
        putchar (0xFF);
     }
     for (j = 0; j < 256; j++) {
        putchar ((~toprow[i][j] » 8) & 0xFF);
        putchar ( ~toprow[i][j]       & 0xFF);
     }
  }

  /*
     Write the header.
  */

  /* 8 completely white rows */
  for (i = 7; i >= 0; i--) {
     for (j = 0; j < 258; j++) {
        putchar (0xFF);
        putchar (0xFF);
     }
  }

  for (i = 15; i >= 0; i--) {
     /* left-hand legend */
     putchar (0xFF);
     putchar (0xFF);
     putchar (0xFF);
     putchar (0xFF);
     /* header glyph */
     for (j = 0; j < 256; j++) {
        bytesout = ~header[i][j] & 0xFFFF;
        putchar ((bytesout » 8) & 0xFF);
        putchar ( bytesout       & 0xFF);
     }
  }

  /* 8 completely white rows at very top */
  for (i = 7; i >= 0; i--) {
     for (j = 0; j < 258; j++) {
     putchar (0xFF);
```

```
    putchar (0xFF);
    }
  }

  return;
}

```

5.25 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 16.0.01"

  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.

5.25.1 Detailed Description

unifontpic.h - Header file for unifontpic.c

Author

  Paul Hardy, July 2017

Copyright

  Copyright (C) 2017 Paul Hardy

Definition in file unifontpic.h.

5.25.2 Macro Definition Documentation

5.25.2.1 HEADER_STRING

#define HEADER_STRING "GNU Unifont 16.0.01"

To be printed as chart title.

Definition at line 32 of file unifontpic.h.

5.25.2.2 MAXSTRING

#define MAXSTRING 256

Maximum input string allowed.

Definition at line 30 of file unifontpic.h.

5.25.3 Variable Documentation

5.25.3.1 ascii_bits

int ascii_bits[128][16]

Array to hold ASCII bitmaps for chart title.

This array will be created from the strings in ascii_hex[] above.

Definition at line 179 of file unifontpic.h.

5.25.3.2 ascii_hex

const char∗ ascii_hex[128]

Array of Unifont ASCII glyphs for chart row & column headings.

Define the array of Unifont ASCII glyphs, code points 0 through 127. This allows using unifontpic to print charts of glyphs above Unicode Plane 0. These were copied from font/plane00/unifont-base.hex, plus U+0020 (ASCII space character).

Definition at line 42 of file unifontpic.h.

5.25.3.3 hexdigit

char hexdigit[16][5]

Initial value:
```
= {
  {0x6,0x9,0x9,0x9,0x6},
  {0x2,0x6,0x2,0x2,0x7},
  {0xF,0x1,0xF,0x8,0xF},
  {0xE,0x1,0x7,0x1,0xE},
  {0x9,0x9,0xF,0x1,0x1},
  {0xF,0x8,0xF,0x1,0xF},
  {0x6,0x8,0xE,0x9,0x6},
  {0xF,0x1,0x2,0x4,0x4},
  {0x6,0x9,0x6,0x9,0x6},
  {0x6,0x9,0x7,0x1,0x6},
  {0xF,0x9,0xF,0x9,0x9},
  {0xE,0x9,0xE,0x9,0xE},
  {0x7,0x8,0x8,0x8,0x7},
  {0xE,0x9,0x9,0x9,0xE},
  {0xF,0x8,0xE,0x8,0xF},
  {0xF,0x8,0xE,0x8,0x8}
}
```

Array of 4x5 hexadecimal digits for legend.

hexdigit contains 4x5 pixel arrays of tiny digits for the legend. See unihexgen.c for a more detailed description in the comments.

Definition at line 188 of file unifontpic.h.

5.26 unifontpic.h

Go to the documentation of this file.
```
00001 /**
00002    @file unifontpic.h
00003
00004    @brief unifontpic.h - Header file for unifontpic.c
00005
00006    @author Paul Hardy, July 2017
00007
00008    @copyright Copyright (C) 2017 Paul Hardy
00009 */
00010 /*
00011    LICENSE:
00012
00013       This program is free software: you can redistribute it and/or modify
00014       it under the terms of the GNU General Public License as published by
00015       the Free Software Foundation, either version 2 of the License, or
00016       (at your option) any later version.
00017
00018       This program is distributed in the hope that it will be useful,
00019       but WITHOUT ANY WARRANTY; without even the implied warranty of
00020       MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00021       GNU General Public License for more details.
00022
00023       You should have received a copy of the GNU General Public License
00024       along with this program.  If not, see <http://www.gnu.org/licenses/>.
00025 */
00026
00027 #ifndef _UNIFONTPIC_H_
00028 #define _UNIFONTPIC_H_
00029
00030 #define MAXSTRING 256  ///< Maximum input string allowed.
00031
00032 #define HEADER_STRING "GNU Unifont 16.0.01" ///< To be printed as chart title.
00033
00034 /**
00035    @brief Array of Unifont ASCII glyphs for chart row & column headings.
```

```

   Define the array of Unifont ASCII glyphs, code points 0 through 127.
   This allows using unifontpic to print charts of glyphs above Unicode
   Plane 0.  These were copied from font/plane00/unifont-base.hex, plus
   U+0020 (ASCII space character).
*/
const char *ascii_hex [128] = {
   "0000:AAAA00018000000180004A51EA505A51C99E0001800000018000000180005555",
   "0001:AAAA0001800000018000399 3C252325F8A527193800000018000000180005555",
   "0002:AAAA000180000001800 03BA5C124311989247125800000018000000180005555",
   "0003:AAAA00018000000180007BA5C1247919C1247925800000018000000180005555",
   "0004:AAAA00018000000180007 9BFC2487A49C2487989800000018000000180005555",
   "0005:AAAA00018000000180007A4DC2527B53C2D67A4F800000018000000180005555",
   "0006:AAAA000180000001800031A5CA287A31CA2849A5800000018000000180005555",
   "0007:AAAA000180000001800073D1CA1073D1CA1073DF800000018000000180005555",
   "0008:AAAA00018000000180001E3991401E3191081E71800000018000000180005555",
   "0009:AAAA000180000001800022F9A2203E21A2202221800000018000000180005555",
   "000A:AAAA000180000001800020F9A08020F9A0803E81800000018000000180005555",
   "000B:AAAA000180000001800022F9A2202221942008218000000180000000180005555",
   "000C:AAAA00018000000180003EF9A0803EF9A0802081800000018000000180005555",
   "000D:AAAA00018000000180001EF1A08820F1A0901E89800000018000000180005555",
   "000E:AAAA00018000000180001E71A0881C8982883C71800000018000000180005555",
   "000F:AAAA00018000000180001EF9A0201C2182203CF9800000018000000180005555",
   "0010:AAAA0001800000018000391DA510251DA51039DD800000018000000180005555",
   "0011:AAAA00018000000180007189CA184A09CA08719D800000018000000180005555",
   "0012:AAAA00018000000180007199CA044A09CA10719D800000018000000180005555",
   "0013:AAAA00018000000180007199CA044A19CA047199800000018000000180005555",
   "0014:AAAA00018000000180007185CA0C4A15CA1C7185800000018000000180005555",
   "0015:AAAA00018000000180004993EA546A59DBD44A53800000018000000180005555",
   "0016:AAAA00018000000180003453C29A311789127113800000018000000180005555",
   "0017:AAAA00018000000180007BB9C1247939C1247939800000018000000180005555",
   "0018:AAAA00018000000180003325C4B447ADC4A434A5800000018000000180005555",
   "0019:AAAA00018000000180003E89A0D83EA9A0883E89800000018000000180005555",
   "001A:AAAA00018000000180003A5DC252325D8A52719D800000018000000180005555",
   "001B:AAAA000180000001800079CFC2107991C0507B8F800000018000000180005555",
   "001C:AAAA00018000000180001E7190801E61901010E1800000018000000180005555",
   "001D:AAAA00018000000180000E719080166192100EE1800000018000000180005555",
   "001E:AAAA00018000000180001C7192801C61941012E1800000018000000180005555",
   "001F:AAAA000180000001800012719280126192100CE1800000018000000180005555",
   "0020:00000000000000000000000000000000",
   "0021:00000000080808080808000808000000",
   "0022:00002222222200000000000000000000",
   "0023:000000001212127E24247E4848480000",
   "0024:00000000083E4948380E09493E080000",
   "0025:00000000314A4A340808162929460000",
   "0026:000000001C2222141829454246390000",
   "0027:00000808080800000000000000000000",
   "0028:00000004080810101010101008080400",
   "0029:00000020101008080808080810102000",
   "002A:00000000000008492A1C2A4908000000",
   "002B:0000000000000808087F080808000000",
   "002C:00000000000000000000000018080810",
   "002D:0000000000000000003C000000000000",
   "002E:00000000000000000000000018180000",
   "002F:00000000020204080810102040400000",
   "0030:00000000182442464A52624224180000",
   "0031:000000000818280808080808083E0000",
   "0032:000000003C4242020C102040407E0000",
   "0033:000000003C4242021C020242423C0000",
   "0034:00000000040C142444447E0404040000",
   "0035:000000007E4040407C020202423C0000",
   "0036:000000001C2040407C424242423C0000",
   "0037:000000007E0202040404080808080000",
   "0038:000000003C4242423C424242423C0000",
   "0039:000000003C4242423E02020204380000",
   "003A:00000000000018180000001818000000",
   "003B:00000000000018180000001808081000",
   "003C:00000000000204081020100804020000",
   "003D:000000000000007E0000007E00000000",
   "003E:00000000004020100804081020400000",
   "003F:000000003C4242020408080008080000",
   "0040:000000001C224A565252524E201E0000",
   "0041:0000000018242442427E424242420000",
   "0042:000000007C4242427C424242427C0000",
   "0043:000000003C42424040404042423C0000",
   "0044:00000000784442424242424244780000",
   "0045:000000007E4040407C404040407E0000",
   "0046:000000007E4040407C40404040400000",
   "0047:000000003C424240404E4242463A0000",
   "0048:00000000424242427E42424242420000",
   "0049:000000003E08080808080808083E0000",
```

```
    "004A:000000001F0404040404044444380000",
    "004B:00000000424448506060504844420000",
    "004C:0000000040404040404040404407E0000",
    "004D:00000000424266665A5A424242420000",
    "004E:0000000042626252524A4A4646420000",
    "004F:000000003C424242424242423C0000",
    "0050:000000007C4242427C40404040400000",
    "0051:000000003C4242424242425A663C0300",
    "0052:000000007C4242427C48444442420000",
    "0053:000000003C424240300C0242423C0000",
    "0054:000000007F080808080808080808080000",
    "0055:000000004242424242424242423C0000",
    "0056:00000000414141222222141408080000",
    "0057:000000004242425A5A666642420000",
    "0058:00000000424224241818242442420000",
    "0059:00000000414122221408080808080000",
    "005A:000000007E0202040810204040407E0000",
    "005B:0000000E080808080808080808080E00",
    "005C:00000000404020101008080402020000",
    "005D:00000070101010101010101010107000",
    "005E:00001824420000000000000000000000",
    "005F:0000000000000000000000000000007F00",
    "0060:00201008000000000000000000000000",
    "0061:0000000000003C42023E4242463A0000",
    "0062:0000004040405C6242424242625C0000",
    "0063:0000000000003C4240404040423C0000",
    "0064:0000000202023A4642424242463A0000",
    "0065:0000000000003C42427E4040423C0000",
    "0066:0000000C1010107C1010101010100000",
    "0067:0000000000023A44444438203C42423C",
    "0068:0000004040405C624242424242420000",
    "0069:000000080800180808080808083E0000",
    "006A:0000000404000C04040404040404044830",
    "006B:00000040404044485060504844420000",
    "006C:00000018080808080808080808083E0000",
    "006D:00000000000076494949494949490000",
    "006E:0000000000005C624242424242420000",
    "006F:0000000000003C4242424242423C0000",
    "0070:0000000000005C6242424242625C4040",
    "0071:0000000000003A4642424242463A0202",
    "0072:0000000000005C624240404040400000",
    "0073:0000000000003C4240300C02423C0000",
    "0074:000000001010107C10101010100C0000",
    "0075:000000000000424242424242463A0000",
    "0076:00000000000042424224242418180000",
    "0077:00000000000041494949494949360000",
    "0078:00000000000042422418182442420000",
    "0079:0000000000004242424242261A02023C",
    "007A:0000000000007E0204081020407E0000",
    "007B:0000000C10100808102010080810100C",
    "007C:00000808080808080808080808080808",
    "007D:00000030080810100804081010080830",
    "007E:00000031494600000000000000000000",
    "007F:AAAA00018000000180073D1CA104BD1CA1073DF800000018000000180005555"
};


/**
   @brief Array to hold ASCII bitmaps for chart title.

   This array will be created from the strings in ascii_hex[] above.
*/
int ascii_bits[128][16];


/**
   @brief Array of 4x5 hexadecimal digits for legend.

   hexdigit contains 4x5 pixel arrays of tiny digits for the legend.
   See unihexgen.c for a more detailed description in the comments.
*/
char hexdigit[16][5] = {
   {0x6,0x9,0x9,0x9,0x6},  /* 0x0 */
   {0x2,0x6,0x2,0x2,0x7},  /* 0x1 */
   {0xF,0x1,0xF,0x8,0xF},  /* 0x2 */
   {0xE,0x1,0x7,0x1,0xE},  /* 0x3 */
   {0x9,0x9,0xF,0x1,0x1},  /* 0x4 */
   {0xF,0x8,0xF,0x1,0xF},  /* 0x5 */
   {0x6,0x8,0xE,0x9,0x6},  /* 0x6 */
   {0xF,0x1,0x2,0x4,0x4},  /* 0x7 */
   {0x6,0x9,0x6,0x9,0x6},  /* 0x8 */
```

```
    {0x6,0x9,0x7,0x1,0x6}, /* 0x9 */
    {0xF,0x9,0xF,0x9,0x9}, /* 0xA */
    {0xE,0x9,0xE,0x9,0xE}, /* 0xB */
    {0x7,0x8,0x8,0x8,0x7}, /* 0xC */
    {0xE,0x9,0x9,0x9,0xE}, /* 0xD */
    {0xF,0x8,0xE,0x8,0xF}, /* 0xE */
    {0xF,0x8,0xE,0x8,0x8}  /* 0xF */
};

#endif
```

5.27 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.

5.27.1 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

Definition in file unigen-hangul.c.

5.27.2 Function Documentation

5.27.2.1 get_hex_range()

void get_hex_range (
    char * instring,
    unsigned * start,
    unsigned * end)

Scan a hexadecimal range from a character string.

Definition at line 354 of file unigen-hangul.c.

```
00354                                             {
00355
00356    int i;  /* String index variable. */
00357
00358    /* Get first number in range. */
00359    sscanf (instring, "%X", start);
00360    for (i = 0;
00361         instring [i] != '\0' && instring [i] != '-';
00362         i++);
00363    /* Get last number in range. */
00364    if (instring [i] == '-') {
00365       i++;
00366       sscanf (&instring [i], "%X", end);
00367    }
00368    else {
00369       *end = *start;
00370    }
00371
00372    return;
00373 }
```

Here is the caller graph for this function:

5.27.2.2 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.

```
00069                 {
00070
00071    int i;  /* loop variable */
00072    unsigned codept;
00073    unsigned max_codept;
00074    unsigned glyph[MAX_GLYPHS][16];
00075    unsigned tmp_glyph [16];  /* To build one combined glyph at a time. */
00076    int cho, jung, jong;      /* The 3 components in a Hangul syllable. */
00077
00078    /// Default parameters for Hangul syllable generation.
00079    struct PARAMS params = { 0xAC00,  /* Starting output Unicode code point */
00080                            0x1100,  /* First modern choseong              */
00081                            0x1112,  /* Last modern choseong               */
00082                            0x1161,  /* First modern jungseong             */
00083                            0x1175,  /* Last modern jungseong              */
00084                            0x11A7,  /* One before first modern jongseong  */
00085                            0x11C2,  /* Last modern jongseong              */
00086                            stdin,   /* Default input file pointer         */
00087                            stdout   /* Default output file pointer        */
00088                          };
00089
00090    void parse_args (int argc, char *argv[], struct PARAMS *params);
00091
00092    unsigned hangul_read_base16 (FILE *infp, unsigned glyph[][16]);
00093
00094    void print_glyph_hex (FILE *fp, unsigned codept, unsigned *this_glyph);
00095
00096    void combined_jamo (unsigned glyph [MAX_GLYPHS][16],
00097                        unsigned cho, unsigned jung, unsigned jong,
00098                        unsigned *combined_glyph);
00099
00100
00101    if (argc > 1) {
00102       parse_args (argc, argv, &params);
00103
00104 #ifdef DEBUG
00105       fprintf (stderr,
00106                "Range: (U+%04X, U+%04X, U+%04X) to (U+%04X, U+%04X, U+%04X)\n",
00107                params.cho_start, params.jung_start, params.jong_start,
00108                params.cho_end,   params.jung_end,   params.jong_end);
00109 #endif
00110    }
00111
00112    /*
00113       Initialize glyph array to all zeroes.
00114    */
00115    for (codept = 0; codept < MAX_GLYPHS; codept++) {
00116       for (i = 0; i < 16; i++) glyph[codept][i] = 0x0000;
00117    }
00118
00119    /*
00120       Read Hangul base glyph file.
00121    */
00122    max_codept = hangul_read_base16 (params.infp, glyph);
00123    if (max_codept > 0x8FF) {
00124       fprintf (stderr, "\nWARNING: Hangul glyph range exceeds PUA space.\n\n");
00125    }
00126
00127    codept = params.starting_codept;  /* First code point to output */
00128
00129    for (cho = params.cho_start; cho <= params.cho_end; cho++) {
00130       for (jung = params.jung_start; jung <= params.jung_end; jung++) {
00131          for (jong = params.jong_start; jong <= params.jong_end; jong++) {
00132
```

```
#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:

5.27.2.3 parse_args()

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

Parse command line arguments.

Definition at line 163 of file unigen-hangul.c.

```
00163                                                      {
00164    int arg_count;  /* Current index into argv[]. */
00165
00166    void get_hex_range (char *instring, unsigned *start, unsigned *end);
00167
00168    int strncmp (const char *s1, const char *s2, size_t n);
00169
00170
00171    arg_count = 1;
00172
00173    while (arg_count < argc) {
00174       /* If all 600,000+ Hangul syllables are requested. */
00175       if (strncmp (argv [arg_count], "-all", 4) == 0) {
00176          params->starting_codept = 0x0001;
00177          params->cho_start  = CHO_UNICODE_START;       /*          First modern choseong  */
00178          params->cho_end    = CHO_EXTA_UNICODE_END;    /*          Last ancient choseong  */
00179          params->jung_start = JUNG_UNICODE_START;      /*          First modern jungseong */
00180          params->jung_end   = JUNG_EXTB_UNICODE_END;   /*          Last ancient jungseong */
00181          params->jong_start = JONG_UNICODE_START - 1;  /* One before first modern jongseong */
00182          params->jong_end   = JONG_EXTB_UNICODE_END;   /*          Last andient jongseong */
00183       }
00184       /* If starting code point for output Unifont hex file is specified. */
00185       else if (strncmp (argv [arg_count], "-c", 2) == 0) {
00186          arg_count++;
00187          if (arg_count < argc) {
00188             sscanf (argv [arg_count], "%X", &params->starting_codept);
00189          }
00190       }
00191       /* If initial consonant (choseong) range, "jamo 1", get range.  */
00192       else if (strncmp (argv [arg_count], "-j1", 3) == 0) {
00193          arg_count++;
00194          if (arg_count < argc) {
00195             get_hex_range (argv [arg_count],
00196                            &params->cho_start, &params->cho_end);
00197             /*
00198                Allow one initial blank glyph at start of a loop, none at end.
```

```
         */
         if (params->cho_start < CHO_UNICODE_START) {
            params->cho_start = CHO_UNICODE_START - 1;
         }
         else if (params->cho_start > CHO_UNICODE_END &&
                  params->cho_start < CHO_EXTA_UNICODE_START) {
            params->cho_start = CHO_EXTA_UNICODE_START - 1;
         }
         /*
            Do not go past desired Hangul choseong range,
            Hangul Jamo or Hangul Jamo Extended-A choseong.
         */
         if (params->cho_end > CHO_EXTA_UNICODE_END) {
            params->cho_end = CHO_EXTA_UNICODE_END;
         }
         else if (params->cho_end > CHO_UNICODE_END &&
                  params->cho_end < CHO_EXTA_UNICODE_START) {
            params->cho_end = CHO_UNICODE_END;
         }
      }
   }
   /* If medial vowel (jungseong) range, ”jamo 2”, get range. */
   else if (strncmp (argv [arg_count], ”-j2”, 3) == 0) {
      arg_count++;
      if (arg_count < argc) {
         get_hex_range (argv [arg_count],
                     &params->jung_start, &params->jung_end);
         /*
            Allow one initial blank glyph at start of a loop, none at end.
         */
         if (params->jung_start < JUNG_UNICODE_START) {
            params->jung_start = JUNG_UNICODE_START - 1;
         }
         else if (params->jung_start > JUNG_UNICODE_END &&
                  params->jung_start < JUNG_EXTB_UNICODE_START) {
            params->jung_start = JUNG_EXTB_UNICODE_START - 1;
         }
         /*
            Do not go past desired Hangul jungseong range,
            Hangul Jamo or Hangul Jamo Extended-B jungseong.
         */
         if (params->jung_end > JUNG_EXTB_UNICODE_END) {
            params->jung_end = JUNG_EXTB_UNICODE_END;
         }
         else if (params->jung_end > JUNG_UNICODE_END &&
                  params->jung_end < JUNG_EXTB_UNICODE_START) {
            params->jung_end = JUNG_UNICODE_END;
         }
      }
   }
   /* If final consonant (jongseong) range, ”jamo 3”, get range. */
   else if (strncmp (argv [arg_count], ”-j3”, 3) == 0) {
      arg_count++;
      if (arg_count < argc) {
         get_hex_range (argv [arg_count],
                     &params->jong_start, &params->jong_end);
         /*
            Allow one initial blank glyph at start of a loop, none at end.
         */
         if (params->jong_start < JONG_UNICODE_START) {
            params->jong_start = JONG_UNICODE_START - 1;
         }
         else if (params->jong_start > JONG_UNICODE_END &&
                  params->jong_start < JONG_EXTB_UNICODE_START) {
            params->jong_start = JONG_EXTB_UNICODE_START - 1;
         }
         /*
            Do not go past desired Hangul jongseong range,
            Hangul Jamo or Hangul Jamo Extended-B jongseong.
         */
         if (params->jong_end > JONG_EXTB_UNICODE_END) {
            params->jong_end = JONG_EXTB_UNICODE_END;
         }
         else if (params->jong_end > JONG_UNICODE_END &&
                  params->jong_end < JONG_EXTB_UNICODE_START) {
            params->jong_end = JONG_UNICODE_END;
         }
      }
   }
   /* If input file is specified, open it for read access. */
   else if (strncmp (argv [arg_count], ”-i”, 2) == 0) {
```

```c
         arg_count++;
         if (arg_count < argc) {
            params->infp = fopen (argv [arg_count], "r");
            if (params->infp == NULL) {
               fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",
                        argv [arg_count]);
               exit (EXIT_FAILURE);
            }
         }
      }
      /* 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) {
            params->outfp = fopen (argv [arg_count], "w");
            if (params->outfp == NULL) {
               fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
                        argv [arg_count]);
               exit (EXIT_FAILURE);
            }
         }
      }
      /* 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 ("\nunigen-hangul [options]\n\n");
         printf ("    Generates Hangul syllables from an input Unifont .hex file encoded\n");
         printf ("    in Johab 6/3/1 format.  By default, the output is the Unicode Hangul\n");
         printf ("    Syllables range, U+AC00..U+D7A3.  Options allow the user to specify\n");
         printf ("    a starting code point for the output Unifont .hex file, and ranges\n");
         printf ("    in hexadecimal of the starting and ending Hangul Jamo code points:\n\n");

         printf ("        * 1100-115E Initial consonants (choseong)\n");
         printf ("        * 1161-11A7 Medial vowels (jungseong)\n");
         printf ("        * 11A8-11FF Final consonants (jongseong).\n\n");

         printf ("    A single code point or 0 to omit can be specified instead of a range.\n\n");

         printf ("  Option    Parameters    Function\n");
         printf ("  ------    ----------    --------\n");
         printf ("  -h, --help              Print this message and exit.\n\n");
         printf ("  -all                    Generate all Hangul syllables, using all modern and\n");
         printf ("                          ancient Hangul in the Unicode range U+1100..U+11FF,\n");
         printf ("                          U+A960..U+A97C, and U+D7B0..U+D7FB.\n");
         printf ("                          WARNING: this will generate over 1,600,000 syllables\n");
         printf ("                          in a 115 megabyte Unifont .hex format file.  The\n");
         printf ("                          default is to only output modern Hangul syllables.\n\n");
         printf ("  -c      code_point    Starting code point in hexadecimal for output file.\n\n");
         printf ("  -j1     start-end     Choseong (jamo 1) start-end range in hexadecimal.\n\n");
         printf ("  -j2     start-end     Jungseong (jamo 2) start-end range in hexadecimal.\n\n");
         printf ("  -j3     start-end     Jongseong (jamo 3) start-end range in hexadecimal.\n\n");
         printf ("  -i      input_file    Unifont hangul-base.hex formatted input file.\n\n");
         printf ("  -o      output_file   Unifont .hex format output file.\n\n");
         printf ("     Example:\n\n");
         printf ("        unigen-hangul -c 1 -j3 11AB-11AB -i hangul-base.hex -o nieun-only.hex\n\n");
         printf ("     Generates Hangul syllables using all modern choseong and jungseong,\n");
         printf ("     and only the jongseong nieun (Unicode code point U+11AB).  The output\n");
         printf ("     Unifont .hex file will contain code points starting at 1.  Instead of\n");
         printf ("     specifying \"-j3 11AB-11AB\", simply using \"-j3 11AB\" will also suffice.\n\n");

         exit (EXIT_SUCCESS);
      }

      arg_count++;
   }

   return;
}
```

Here is the call graph for this function: Here is the caller graph for this function:

5.28 unigen-hangul.c

Go to the documentation of this file.

```c
/**
   @file unigen-hangul.c

   @brief 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 Copyright © 2023 Paul Hardy
*/
/*
   LICENSE:

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

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

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

#include <stdio.h>
#include <stdlib.h>
#include "hangul.h"

// #define DEBUG


struct PARAMS {
   unsigned starting_codept;      /* First output Unicode code point. */
   unsigned cho_start,  cho_end;  /* Choseong  start and end code points. */
   unsigned jung_start, jung_end; /* Jungseong start and end code points. */
   unsigned jong_start, jong_end; /* Jongseong start and end code points. */
   FILE *infp;
   FILE *outfp;
};


/**
   @brief Program entry point.
*/
int
main (int argc, char *argv[]) {

   int i;  /* loop variable */
   unsigned codept;
   unsigned max_codept;
   unsigned glyph[MAX_GLYPHS][16];
   unsigned tmp_glyph [16];  /* To build one combined glyph at a time. */
   int cho, jung, jong;      /* The 3 components in a Hangul syllable. */

   /// Default parameters for Hangul syllable generation.
   struct PARAMS params = { 0xAC00,  /* Starting output Unicode code point */
                            0x1100,  /* First modern choseong              */
                            0x1112,  /* Last modern choseong               */
```

```
                        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);
}


/**
   @brief Parse command line arguments.

*/
void
```

```
parse_args (int argc, char *argv[], struct PARAMS *params) {
   int arg_count;  /* Current index into argv[]. */

   void get_hex_range (char *instring, unsigned *start, unsigned *end);

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


   arg_count = 1;

   while (arg_count < argc) {
      /* If all 600,000+ Hangul syllables are requested. */
      if (strncmp (argv [arg_count], "-all", 4) == 0) {
         params->starting_codept = 0x0001;
         params->cho_start  = CHO_UNICODE_START;       /*           First modern choseong  */
         params->cho_end    = CHO_EXTA_UNICODE_END;    /*           Last ancient choseong  */
         params->jung_start = JUNG_UNICODE_START;      /*           First modern jungseong */
         params->jung_end   = JUNG_EXTB_UNICODE_END;   /*           Last ancient jungseong */
         params->jong_start = JONG_UNICODE_START - 1;  /* One before first modern jongseong */
         params->jong_end   = JONG_EXTB_UNICODE_END;   /*           Last andient jongseong */
      }
      /* If starting code point for output Unifont hex file is specified. */
      else if (strncmp (argv [arg_count], "-c", 2) == 0) {
         arg_count++;
         if (arg_count < argc) {
            sscanf (argv [arg_count], "%X", &params->starting_codept);
         }
      }
      /* If initial consonant (choseong) range, "jamo 1", get range.  */
      else if (strncmp (argv [arg_count], "-j1", 3) == 0) {
         arg_count++;
         if (arg_count < argc) {
            get_hex_range (argv [arg_count],
                           &params->cho_start, &params->cho_end);
            /*
               Allow one initial blank glyph at start of a loop, none at end.
            */
            if (params->cho_start < CHO_UNICODE_START) {
               params->cho_start = CHO_UNICODE_START - 1;
            }
            else if (params->cho_start > CHO_UNICODE_END &&
                     params->cho_start < CHO_EXTA_UNICODE_START) {
               params->cho_start = CHO_EXTA_UNICODE_START - 1;
            }
            /*
               Do not go past desired Hangul choseong range,
               Hangul Jamo or Hangul Jamo Extended-A choseong.
            */
            if (params->cho_end > CHO_EXTA_UNICODE_END) {
               params->cho_end = CHO_EXTA_UNICODE_END;
            }
            else if (params->cho_end > CHO_UNICODE_END &&
                     params->cho_end < CHO_EXTA_UNICODE_START) {
               params->cho_end = CHO_UNICODE_END;
            }
         }
      }
      /* If medial vowel (jungseong) range, "jamo 2", get range.  */
      else if (strncmp (argv [arg_count], "-j2", 3) == 0) {
         arg_count++;
         if (arg_count < argc) {
            get_hex_range (argv [arg_count],
                           &params->jung_start, &params->jung_end);
            /*
               Allow one initial blank glyph at start of a loop, none at end.
            */
            if (params->jung_start < JUNG_UNICODE_START) {
               params->jung_start = JUNG_UNICODE_START - 1;
            }
            else if (params->jung_start > JUNG_UNICODE_END &&
                     params->jung_start < JUNG_EXTB_UNICODE_START) {
               params->jung_start = JUNG_EXTB_UNICODE_START - 1;
            }
            /*
               Do not go past desired Hangul jungseong range,
               Hangul Jamo or Hangul Jamo Extended-B jungseong.
            */
            if (params->jung_end > JUNG_EXTB_UNICODE_END) {
               params->jung_end = JUNG_EXTB_UNICODE_END;
            }
            else if (params->jung_end > JUNG_UNICODE_END &&
```

```
                     params->jung_end < JUNG_EXTB_UNICODE_START) {
               params->jung_end = JUNG_UNICODE_END;
            }
         }
      }
      /* If final consonant (jongseong) range, "jamo 3", get range.  */
      else if (strncmp (argv [arg_count], "-j3", 3) == 0) {
         arg_count++;
         if (arg_count < argc) {
            get_hex_range (argv [arg_count],
                           &params->jong_start, &params->jong_end);
            /*
               Allow one initial blank glyph at start of a loop, none at end.
            */
            if (params->jong_start < JONG_UNICODE_START) {
               params->jong_start = JONG_UNICODE_START - 1;
            }
            else if (params->jong_start > JONG_UNICODE_END &&
                     params->jong_start < JONG_EXTB_UNICODE_START) {
               params->jong_start = JONG_EXTB_UNICODE_START - 1;
            }
            /*
               Do not go past desired Hangul jongseong range,
               Hangul Jamo or Hangul Jamo Extended-B jongseong.
            */
            if (params->jong_end > JONG_EXTB_UNICODE_END) {
               params->jong_end = JONG_EXTB_UNICODE_END;
            }
            else if (params->jong_end > JONG_UNICODE_END &&
                     params->jong_end < JONG_EXTB_UNICODE_START) {
               params->jong_end = JONG_UNICODE_END;
            }
         }
      }
      /* If input file is specified, open it for read access. */
      else if (strncmp (argv [arg_count], "-i", 2) == 0) {
         arg_count++;
         if (arg_count < argc) {
            params->infp = fopen (argv [arg_count], "r");
            if (params->infp == NULL) {
               fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",
                        argv [arg_count]);
               exit (EXIT_FAILURE);
            }
         }
      }
      /* 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) {
            params->outfp = fopen (argv [arg_count], "w");
            if (params->outfp == NULL) {
               fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
                        argv [arg_count]);
               exit (EXIT_FAILURE);
            }
         }
      }
      /* 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 ("\nunigen-hangul [options]\n\n");
         printf ("     Generates Hangul syllables from an input Unifont .hex file encoded\n");
         printf ("     in Johab 6/3/1 format.  By default, the output is the Unicode Hangul\n");
         printf ("     Syllables range, U+AC00..U+D7A3.  Options allow the user to specify\n");
         printf ("     a starting code point for the output Unifont .hex file, and ranges\n");
         printf ("     in hexadecimal of the starting and ending Hangul Jamo code points:\n\n");

         printf ("          * 1100-115E Initial consonants (choseong)\n");
         printf ("          * 1161-11A7 Medial vowels (jungseong)\n");
         printf ("          * 11A8-11FF Final consonants (jongseong).\n\n");

         printf ("     A single code point or 0 to omit can be specified instead of a range.\n\n");

         printf ("   Option    Parameters    Function\n");
         printf ("   ------    ----------    --------\n");
         printf ("   -h, --help               Print this message and exit.\n\n");
         printf ("   -all                     Generate all Hangul syllables, using all modern and\n");
         printf ("                            ancient Hangul in the Unicode range U+1100..U+11FF,\n");
         printf ("                            U+A960..U+A97C, and U+D7B0..U+D7FB.\n");
         printf ("                            WARNING: this will generate over 1,600,000 syllables\n");
```

```
      printf ("                   in a 115 megabyte Unifont .hex format file.  The\n");
      printf ("                   default is to only output modern Hangul syllables.\n\n");
      printf ("   -c       code_point    Starting code point in hexadecimal for output file.\n\n");
      printf ("   -j1      start-end     Choseong (jamo 1) start-end range in hexadecimal.\n\n");
      printf ("   -j2      start-end     Jungseong (jamo 2) start-end range in hexadecimal.\n\n");
      printf ("   -j3      start-end     Jongseong (jamo 3) start-end range in hexadecimal.\n\n");
      printf ("   -i       input_file    Unifont hangul-base.hex formatted input file.\n\n");
      printf ("   -o       output_file   Unifont .hex format output file.\n\n");
      printf ("      Example:\n\n");
      printf ("         unigen-hangul -c 1 -j3 11AB-11AB -i hangul-base.hex -o nieun-only.hex\n\n");
      printf ("      Generates Hangul syllables using all modern choseong and jungseong,\n");
      printf ("      and only the jongseong nieun (Unicode code point U+11AB).  The output\n");
      printf ("      Unifont .hex file will contain code points starting at 1.  Instead of\n");
      printf ("      specifying \"-j3 11AB-11AB\", simply using \"-j3 11AB\" will also suffice.\n\n");

      exit (EXIT_SUCCESS);
    }

    arg_count++;
  }

  return;
}


/**
  @brief Scan a hexadecimal range from a character string.
*/
void
get_hex_range (char *instring, unsigned *start, unsigned *end) {

  int i;  /* String index variable. */

  /* Get first number in range. */
  sscanf (instring, "%X", start);
  for (i = 0;
       instring [i] != '\0' && instring [i] != '-';
       i++);
  /* Get last number in range. */
  if (instring [i] == '-') {
    i++;
    sscanf (&instring [i], "%X", end);
  }
  else {
    *end = *start;
  }

  return;
}
```

5.29 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.

5.29.1 Detailed Description

unigencircles - Superimpose dashed combining circles on combining glyphs

Author

Paul Hardy

Copyright

Definition in file unigencircles.c.

5.29.2 Macro Definition Documentation

5.29.2.1 MAXSTRING

```
#define MAXSTRING 256
```

Maximum input line length - 1.

Definition at line 62 of file unigencircles.c.

5.29.3 Function Documentation

5.29.3.1 add_double_circle()

```
void add_double_circle (
        char * glyphstring,
        int offset )
```

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

Parameters

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

Definition at line 221 of file unigencircles.c.

```
{

   char newstring[256];
   /* Circle hex string pattern is "00000008000024004200240000000000" */

   /* For double diacritical glyphs (offset = -8) */
   /* Combining circle is left-justified.          */
   char circle08[64]={0x0,0x0,0x0,0x0,  /* row  1 */
                      0x0,0x0,0x0,0x0,  /* row  2 */
                      0x0,0x0,0x0,0x0,  /* row  3 */
                      0x0,0x0,0x0,0x0,  /* row  4 */
                      0x0,0x0,0x0,0x0,  /* row  5 */
                      0x0,0x0,0x0,0x0,  /* row  6 */
                      0x2,0x4,0x0,0x0,  /* row  7 */
                      0x0,0x0,0x0,0x0,  /* row  8 */
                      0x4,0x2,0x0,0x0,  /* row  9 */
                      0x0,0x0,0x0,0x0,  /* row 10 */
                      0x2,0x4,0x0,0x0,  /* row 11 */
                      0x0,0x0,0x0,0x0,  /* row 12 */
                      0x0,0x0,0x0,0x0,  /* row 13 */
                      0x0,0x0,0x0,0x0,  /* row 14 */
                      0x0,0x0,0x0,0x0,  /* row 15 */
                      0x0,0x0,0x0,0x0}; /* row 16 */

   /* For all other combining glyphs (offset = -16) */
   /* Combining circle is centered in 16 columns.    */
   char circle16[64]={0x0,0x0,0x0,0x0,  /* row  1 */
                      0x0,0x0,0x0,0x0,  /* row  2 */
                      0x0,0x0,0x0,0x0,  /* row  3 */
                      0x0,0x0,0x0,0x0,  /* row  4 */
                      0x0,0x0,0x0,0x0,  /* row  5 */
                      0x0,0x0,0x0,0x0,  /* row  6 */
                      0x0,0x2,0x4,0x0,  /* row  7 */
                      0x0,0x0,0x0,0x0,  /* row  8 */
                      0x0,0x4,0x2,0x0,  /* row  9 */
                      0x0,0x0,0x0,0x0,  /* row 10 */
                      0x0,0x2,0x4,0x0,  /* row 11 */
                      0x0,0x0,0x0,0x0,  /* row 12 */
                      0x0,0x0,0x0,0x0,  /* row 13 */
                      0x0,0x0,0x0,0x0,  /* row 14 */
                      0x0,0x0,0x0,0x0,  /* row 15 */
                      0x0,0x0,0x0,0x0}; /* row 16 */

   char *circle; /* points into circle16 or circle08 */

   int digit1, digit2; /* corresponding digits in each string */

   int i; /* index variables */


   /*
      Determine if combining circle is left-justified (offset = -8)
      or centered (offset = -16).
   */
   circle = (offset >= -8) ? circle08 : circle16;

   /* for each character position, OR the corresponding circle glyph value */
   for (i = 0; i < 64; i++) {
      glyphstring[i] = toupper (glyphstring[i]);

      /* Convert ASCII character to a hexadecimal integer */
      digit1 = (glyphstring[i] <= '9') ?
               (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);

      /* Superimpose dashed circle */
      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:

5.29.3.2 add_single_circle()

void add_single_circle (
    char * glyphstring)

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

Parameters

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

Definition at line 163 of file unigencircles.c.

```
00164 {
00165
00166    char newstring[256];
00167    /* Circle hex string pattern is "00000000800002400420024000000000" */
00168    char circle[32]={0x0,0x0,  /* row  1 */
00169                     0x0,0x0,  /* row  2 */
00170                     0x0,0x0,  /* row  3 */
00171                     0x0,0x0,  /* row  4 */
00172                     0x0,0x0,  /* row  5 */
00173                     0x0,0x0,  /* row  6 */
00174                     0x2,0x4,  /* row  7 */
00175                     0x0,0x0,  /* row  8 */
00176                     0x4,0x2,  /* row  9 */
00177                     0x0,0x0,  /* row 10 */
00178                     0x2,0x4,  /* row 11 */
00179                     0x0,0x0,  /* row 12 */
00180                     0x0,0x0,  /* row 13 */
00181                     0x0,0x0,  /* row 14 */
00182                     0x0,0x0,  /* row 15 */
00183                     0x0,0x0}; /* row 16 */
00184
00185    int digit1, digit2; /* corresponding digits in each string */
00186
00187    int i; /* index variables */
00188
00189    /* for each character position, OR the corresponding circle glyph value */
00190    for (i = 0; i < 32; i++) {
00191       glyphstring[i] = toupper (glyphstring[i]);
00192
00193       /* Convert ASCII character to a hexadecimal integer */
00194       digit1 = (glyphstring[i] <= '9') ?
00195                (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);
00196
00197       /* Superimpose dashed circle */
00198       digit2 = digit1 | circle[i];
00199
00200       /* Convert hexadecimal integer to an ASCII character */
00201       newstring[i] = (digit2 <= 9) ?
00202                      ('0' + digit2) : ('A' + digit2 - 0xA);
00203    }
00204
00205    /* Terminate string for output */
00206    newstring[i++] = '\n';
00207    newstring[i++] = '\0';
00208
00209    memcpy (glyphstring, newstring, i);
```

```
   return;
}
```

Here is the caller graph for this function:

5.29.3.3 main()

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

The main function.

Parameters

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

Returns

This program exits with status EXIT_SUCCESS.

Definition at line 73 of file unigencircles.c.

```
{

   char teststring[MAXSTRING];  /* current input line                       */
   int  loc;                    /* Unicode code point of current input line */
   int  offset;                 /* offset value of a combining character    */
   char *gstart;                /* glyph start, pointing into teststring    */

   char combining[0x110000];    /* 1 --> combining glyph; 0 --> non-combining */
   char x_offset [0x110000];    /* second value in *combining.txt files       */

   void add_single_circle(char *);       /* add a single-width dashed circle */
   void add_double_circle(char *, int); /* add a double-width dashed circle */

   FILE *infilefp;

   /*
     if (argc != 3) {
        fprintf (stderr,
                 "\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n");
        exit (EXIT_FAILURE);
     }
   */

   /*
     Read the combining characters list.
   */
   /* Start with no combining code points flagged */
   memset (combining, 0, 0x110000 * sizeof (char));
   memset (x_offset , 0, 0x110000 * sizeof (char));

   if ((infilefp = fopen (argv[1],"r")) == NULL) {
      fprintf (stderr,"ERROR - combining characters file %s not found.\n\n",
              argv[1]);
      exit (EXIT_FAILURE);
   }

   /* Flag list of combining characters to add a dashed circle. */
   while (fscanf (infilefp, "%X:%d", &loc, &offset) != EOF) {
      /*
        U+01107F and U+01D1A0 are not defined as combining characters
```

```
         in Unicode; they were added in a combining.txt file as the
         only way to make them look acceptable in proximity to other
         glyphs in their script.
      */
      if (loc != 0x01107F && loc != 0x01D1A0) {
         combining[loc] = 1;
         x_offset [loc] = offset;
      }
   }
   fclose (infilefp); /* all done reading combining.txt */

   /* Now read the non-printing glyphs; they never have dashed circles */
   if ((infilefp = fopen (argv[2],"r")) == NULL) {
      fprintf (stderr,"ERROR - nonprinting characters file %s not found.\n\n",
              argv[1]);
      exit (EXIT_FAILURE);
   }

   /* Reset list of nonprinting characters to avoid adding a dashed circle. */
   while (fscanf (infilefp, "%X:%*s", &loc) != EOF) combining[loc] = 0;

   fclose (infilefp); /* all done reading nonprinting.hex */

   /*
      Read the hex glyphs.
   */
   teststring[MAXSTRING - 1] = '\0';   /* so there's no chance we leave array  */
   while (fgets (teststring, MAXSTRING-1, stdin) != NULL) {
      sscanf (teststring, "%X", &loc);      /* loc == the Uniocde code point    */
      gstart = strchr (teststring,':') + 1; /* start of glyph bitmap            */
      if (combining[loc]) {                 /* if a combining character         */
         if (strlen (gstart) < 35)
            add_single_circle (gstart);                  /* single-width */
         else
            add_double_circle (gstart, x_offset[loc]);   /* double-width */
      }
      printf ("%s", teststring); /* output the new character .hex string */
   }

   exit (EXIT_SUCCESS);
}
```

Here is the call graph for this function:

5.30 unigencircles.c

Go to the documentation of this file.

```
/**
   @file unigencircles.c

   @brief unigencircles - Superimpose dashed combining circles
                          on combining glyphs

   @author Paul Hardy

   @copyright Copyright (C) 2013, Paul Hardy.
*/
/*
   LICENSE:

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

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

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

```

```
/*
  8 July 2017 [Paul Hardy]:
    - Reads new second field that contains an x-axis offset for
      each combining character in "*combining.txt" files.
    - Uses the above x-axis offset value for a combining character
      to print combining circle in the left half of a double
      diacritic combining character grid, or in the center for
      other combining characters.
    - Adds exceptions for U+01107F (Brahmi number joiner) and
      U+01D1A0 (vertical stroke musical ornament); they are in
      a combining.txt file for positioning, but are not actually
      Unicode combining characters.
    - Typo fix: "single-width"-->"double-width" in comment for
      add_double_circle function.

  12 August 2017 [Paul Hardy]:
    - Hard-code Miao vowels to show combining circles after
      removing them from font/plane01/plane01-combining.txt.

  26 December 2017 [Paul Hardy]:
    - Remove Miao hard-coding; they are back in unibmp2hex.c and
      in font/plane01/plane01-combining.txt.

  11 May 2019 [Paul Hardy]:
    - Changed strncpy calls to memcpy calls to avoid a compiler
      warning.
*/


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

#define MAXSTRING  256  ///< Maximum input line length - 1.


/**
  @brief The main function.

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

  char teststring[MAXSTRING];  /* current input line                       */
  int  loc;                    /* Unicode code point of current input line */
  int  offset;                 /* offset value of a combining character    */
  char *gstart;                /* glyph start, pointing into teststring    */

  char combining[0x110000];    /* 1 --> combining glyph; 0 --> non-combining */
  char x_offset [0x110000];    /* second value in *combining.txt files       */

  void add_single_circle(char *);       /* add a single-width dashed circle */
  void add_double_circle(char *, int); /* add a double-width dashed circle */

  FILE *infilefp;

  /*
    if (argc != 3) {
      fprintf (stderr,
              "\n\nUsage: %s combining.txt nonprinting.hex < unifont.hex > unifontfull.hex\n\n");
      exit (EXIT_FAILURE);
    }
  */

  /*
    Read the combining characters list.
  */
  /* Start with no combining code points flagged */
  memset (combining, 0, 0x110000 * sizeof (char));
  memset (x_offset , 0, 0x110000 * sizeof (char));

  if ((infilefp = fopen (argv[1],"r")) == NULL) {
    fprintf (stderr,"ERROR - combining characters file %s not found.\n\n",
            argv[1]);
    exit (EXIT_FAILURE);
  }
```

```

    /* Flag list of combining characters to add a dashed circle. */
    while (fscanf (infilefp, "%X:%d", &loc, &offset) != EOF) {
       /*
          U+01107F and U+01D1A0 are not defined as combining characters
          in Unicode; they were added in a combining.txt file as the
          only way to make them look acceptable in proximity to other
          glyphs in their script.
       */
       if (loc != 0x01107F && loc != 0x01D1A0) {
          combining[loc] = 1;
          x_offset [loc] = offset;
       }
    }
    fclose (infilefp); /* all done reading combining.txt */

    /* Now read the non-printing glyphs; they never have dashed circles */
    if ((infilefp = fopen (argv[2],"r")) == NULL) {
       fprintf (stderr,"ERROR - nonprinting characters file %s not found.\n\n",
                argv[1]);
       exit (EXIT_FAILURE);
    }

    /* Reset list of nonprinting characters to avoid adding a dashed circle. */
    while (fscanf (infilefp, "%X:%*s", &loc) != EOF) combining[loc] = 0;

    fclose (infilefp); /* all done reading nonprinting.hex */

    /*
       Read the hex glyphs.
    */
    teststring[MAXSTRING - 1] = '\0';   /* so there's no chance we leave array  */
    while (fgets (teststring, MAXSTRING-1, stdin) != NULL) {
       sscanf (teststring, "%X", &loc);      /* loc == the Uniocde code point    */
       gstart = strchr (teststring,':') + 1; /* start of glyph bitmap            */
       if (combining[loc]) {                  /* if a combining character         */
          if (strlen (gstart) < 35)
             add_single_circle (gstart);                  /* single-width */
          else
             add_double_circle (gstart, x_offset[loc]);   /* double-width */
       }
       printf ("%s", teststring); /* output the new character .hex string */
    }

    exit (EXIT_SUCCESS);
 }


 /**
    @brief Superimpose a single-width dashed combining circle on a glyph bitmap.

    @param[in,out] glyphstring A single-width glyph, 8x16 pixels.
 */
 void
 add_single_circle (char *glyphstring)
 {

    char newstring[256];
    /* Circle hex string pattern is "00000008000024004200240000000000" */
    char circle[32]={0x0,0x0,  /* row  1 */
                     0x0,0x0,  /* row  2 */
                     0x0,0x0,  /* row  3 */
                     0x0,0x0,  /* row  4 */
                     0x0,0x0,  /* row  5 */
                     0x0,0x0,  /* row  6 */
                     0x2,0x4,  /* row  7 */
                     0x0,0x0,  /* row  8 */
                     0x4,0x2,  /* row  9 */
                     0x0,0x0,  /* row 10 */
                     0x2,0x4,  /* row 11 */
                     0x0,0x0,  /* row 12 */
                     0x0,0x0,  /* row 13 */
                     0x0,0x0,  /* row 14 */
                     0x0,0x0,  /* row 15 */
                     0x0,0x0}; /* row 16 */

    int digit1, digit2; /* corresponding digits in each string */

    int i; /* index variables */

    /* for each character position, OR the corresponding circle glyph value */
```

```
   for (i = 0; i < 32; i++) {
      glyphstring[i] = toupper (glyphstring[i]);

      /* Convert ASCII character to a hexadecimal integer */
      digit1 = (glyphstring[i] <= '9') ?
               (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);

      /* Superimpose dashed circle */
      digit2 = digit1 | circle[i];

      /* Convert hexadecimal integer to an ASCII character */
      newstring[i] = (digit2 <= 9) ?
                     ('0' + digit2) : ('A' + digit2 - 0xA);
   }

   /* Terminate string for output */
   newstring[i++] = '\n';
   newstring[i++] = '\0';

   memcpy (glyphstring, newstring, i);

   return;
}


/**
   @brief Superimpose a double-width dashed combining circle on a glyph bitmap.

   @param[in,out] glyphstring A double-width glyph, 16x16 pixels.
*/
void
add_double_circle (char *glyphstring, int offset)
{

   char newstring[256];
   /* Circle hex string pattern is "00000008000024004200240000000000" */

   /* For double diacritical glyphs (offset = -8) */
   /* Combining circle is left-justified.          */
   char circle08[64]={0x0,0x0,0x0,0x0,  /* row  1 */
                      0x0,0x0,0x0,0x0,  /* row  2 */
                      0x0,0x0,0x0,0x0,  /* row  3 */
                      0x0,0x0,0x0,0x0,  /* row  4 */
                      0x0,0x0,0x0,0x0,  /* row  5 */
                      0x0,0x0,0x0,0x0,  /* row  6 */
                      0x2,0x4,0x0,0x0,  /* row  7 */
                      0x0,0x0,0x0,0x0,  /* row  8 */
                      0x4,0x2,0x0,0x0,  /* row  9 */
                      0x0,0x0,0x0,0x0,  /* row 10 */
                      0x2,0x4,0x0,0x0,  /* row 11 */
                      0x0,0x0,0x0,0x0,  /* row 12 */
                      0x0,0x0,0x0,0x0,  /* row 13 */
                      0x0,0x0,0x0,0x0,  /* row 14 */
                      0x0,0x0,0x0,0x0,  /* row 15 */
                      0x0,0x0,0x0,0x0}; /* row 16 */

   /* For all other combining glyphs (offset = -16) */
   /* Combining circle is centered in 16 columns.   */
   char circle16[64]={0x0,0x0,0x0,0x0,  /* row  1 */
                      0x0,0x0,0x0,0x0,  /* row  2 */
                      0x0,0x0,0x0,0x0,  /* row  3 */
                      0x0,0x0,0x0,0x0,  /* row  4 */
                      0x0,0x0,0x0,0x0,  /* row  5 */
                      0x0,0x0,0x0,0x0,  /* row  6 */
                      0x0,0x2,0x4,0x0,  /* row  7 */
                      0x0,0x0,0x0,0x0,  /* row  8 */
                      0x0,0x4,0x2,0x0,  /* row  9 */
                      0x0,0x0,0x0,0x0,  /* row 10 */
                      0x0,0x2,0x4,0x0,  /* row 11 */
                      0x0,0x0,0x0,0x0,  /* row 12 */
                      0x0,0x0,0x0,0x0,  /* row 13 */
                      0x0,0x0,0x0,0x0,  /* row 14 */
                      0x0,0x0,0x0,0x0,  /* row 15 */
                      0x0,0x0,0x0,0x0}; /* row 16 */

   char *circle; /* points into circle16 or circle08 */

   int digit1, digit2; /* corresponding digits in each string */

   int i; /* index variables */

```

```
    /*
       Determine if combining circle is left-justified (offset = -8)
       or centered (offset = -16).
    */
    circle = (offset >= -8) ? circle08 : circle16;

    /* for each character position, OR the corresponding circle glyph value */
    for (i = 0; i < 64; i++) {
       glyphstring[i] = toupper (glyphstring[i]);

       /* Convert ASCII character to a hexadecimal integer */
       digit1 = (glyphstring[i] <= '9') ?
                (glyphstring[i] - '0') : (glyphstring[i] - 'A' + 0xA);

       /* Superimpose dashed circle */
       digit2 = digit1 | circle[i];

       /* Convert hexadecimal integer to an ASCII character */
       newstring[i] = (digit2 <= 9) ?
                      ('0' + digit2) : ('A' + digit2 - 0xA);
    }

    /* Terminate string for output */
    newstring[i++] = '\n';
    newstring[i++] = '\0';

    memcpy (glyphstring, newstring, i);

    return;
}

```

5.31 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.

5.31.1 Detailed Description

unigenwidth - IEEE 1003.1-2008 setup to calculate wchar_t string widths

Author

Paul Hardy.

Copyright

All glyphs are treated as 16 pixels high, and can be 8, 16, 24, or 32 pixels wide (resulting in widths of 1, 2, 3, or 4, respectively).

Definition in file unigenwidth.c.

5.31.2 Macro Definition Documentation

5.31.2.1 MAXSTRING

```
#define MAXSTRING 256
```

Maximum input line length - 1.

Definition at line 46 of file unigenwidth.c.

5.31.2.2 PIKTO_END

```
#define PIKTO_END 0x0F11EF
```

End of Pikto code point range.

Definition at line 50 of file unigenwidth.c.

5.31.2.3 PIKTO_SIZE

#define PIKTO_SIZE (PIKTO_END - PIKTO_START + 1)

Number of code points in Pikto range.

Definition at line 52 of file unigenwidth.c.

5.31.2.4 PIKTO_START

#define PIKTO_START 0x0F0E70

Start of Pikto code point range.

Definition at line 49 of file unigenwidth.c.

5.31.3 Function Documentation

5.31.3.1 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.

```
{

   int i; /* loop variable */

   char teststring[MAXSTRING];
   int  loc;
   char *gstart;

```

```
char glyph_width[0x20000];
char pikto_width[PIKTO_SIZE];

FILE *infilep;

if (argc != 3) {
   fprintf (stderr, "\n\nUsage: %s <unifont.hex> <combining.txt>\n\n", argv[0]);
   exit (EXIT_FAILURE);
}

/*
   Read the collection of hex glyphs.
*/
if ((infilep = fopen (argv[1],"r")) == NULL) {
   fprintf (stderr,"ERROR - hex input file %s not found.\n\n", argv[1]);
   exit (EXIT_FAILURE);
}

/* Flag glyph as non-existent until found. */
memset (glyph_width, -1, 0x20000 * sizeof (char));
memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));

teststring[MAXSTRING-1] = '\0';
while (fgets (teststring, MAXSTRING-1, infilep) != NULL) {
   sscanf (teststring, "%X:%*s", &loc);
   if (loc < 0x20000) {
      gstart = strchr (teststring,':') + 1;
      /*
         16 rows per glyph, 2 ASCII hexadecimal digits per byte,
         so divide number of digits by 32 (shift right 5 bits).
      */
      glyph_width[loc] = (strlen (gstart) - 1) >> 5;
   }
   else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
      gstart = strchr (teststring,':') + 1;
      pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
   }
}

fclose (infilep);

/*
   Now read the combining character code points.  These have width of 0.
*/
if ((infilep = fopen (argv[2],"r")) == NULL) {
   fprintf (stderr,"ERROR - combining characters file %s not found.\n\n", argv[2]);
   exit (EXIT_FAILURE);
}

while (fgets (teststring, MAXSTRING-1, infilep) != NULL) {
   sscanf (teststring, "%X:%*s", &loc);
   if (loc < 0x20000) glyph_width[loc] = 0;
}

fclose (infilep);

/*
   Code Points with Unusual Properties (Unicode Standard, Chapter 4).

   As of Unifont 10.0.04, use the widths in the "*-nonprinting.hex"
   files.  If an application is smart enough to know how to handle
   these special cases, it will not render the "nonprinting" glyph
   and will treat the code point as being zero-width.
*/
// glyph_width[0]=0; /* NULL character */
// for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
// for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */

// glyph_width[0x034F]=0; /* combining grapheme joiner               */
// glyph_width[0x180B]=0; /* Mongolian free variation selector one   */
// glyph_width[0x180C]=0; /* Mongolian free variation selector two   */
// glyph_width[0x180D]=0; /* Mongolian free variation selector three */
// glyph_width[0x180E]=0; /* Mongolian vowel separator               */
// glyph_width[0x200B]=0; /* zero width space                        */
// glyph_width[0x200C]=0; /* zero width non-joiner                   */
// glyph_width[0x200D]=0; /* zero width joiner                       */
// glyph_width[0x200E]=0; /* left-to-right mark                      */
// glyph_width[0x200F]=0; /* right-to-left mark                      */
// glyph_width[0x202A]=0; /* left-to-right embedding                 */
// glyph_width[0x202B]=0; /* right-to-left embedding                 */
// glyph_width[0x202C]=0; /* pop directional formatting              */
```

```
// glyph_width[0x202D]=0; /* left-to-right override                */
// glyph_width[0x202E]=0; /* right-to-left override                */
// glyph_width[0x2060]=0; /* word joiner                           */
// glyph_width[0x2061]=0; /* function application                  */
// glyph_width[0x2062]=0; /* invisible times                       */
// glyph_width[0x2063]=0; /* invisible separator                   */
// glyph_width[0x2064]=0; /* invisible plus                        */
// glyph_width[0x206A]=0; /* inhibit symmetric swapping            */
// glyph_width[0x206B]=0; /* activate symmetric swapping           */
// glyph_width[0x206C]=0; /* inhibit arabic form shaping           */
// glyph_width[0x206D]=0; /* activate arabic form shaping          */
// glyph_width[0x206E]=0; /* national digit shapes                 */
// glyph_width[0x206F]=0; /* nominal digit shapes                  */

// /* Variation Selector-1 to Variation Selector-16 */
// for (i = 0xFE00; i <= 0xFE0F; i++) glyph_width[i] = 0;

// glyph_width[0xFEFF]=0; /* zero width no-break space         */
// glyph_width[0xFFF9]=0; /* interlinear annotation anchor     */
// glyph_width[0xFFFA]=0; /* interlinear annotation separator  */
// glyph_width[0xFFFB]=0; /* interlinear annotation terminator */
   /*
      Let glyph widths represent 0xFFFC (object replacement character)
      and 0xFFFD (replacement character).
   */

   /*
      Hangul Jamo:

         Leading Consonant (Choseong): leave spacing as is.

         Hangul Choseong Filler (U+115F): set width to 2.

         Hangul Jungseong Filler, Hangul Vowel (Jungseong), and
         Final Consonant (Jongseong): set width to 0, because these
         combine with the leading consonant as one composite syllabic
         glyph.  As of Unicode 5.2, the Hangul Jamo block (U+1100..U+11FF)
         is completely filled.
   */
   // for (i = 0x1160; i <= 0x11FF; i++) glyph_width[i]=0; /* Vowels & Final Consonants */

   /*
      Private Use Area -- the width is undefined, but likely
      to be 2 charcells wide either from a graphic glyph or
      from a four-digit hexadecimal glyph representing the
      code point.  Therefore if any PUA glyph does not have
      a non-zero width yet, assign it a default width of 2.
      The Unicode Standard allows giving PUA characters
      default property values; see for example The Unicode
      Standard Version 5.0, p. 91.  This same default is
      used for higher plane PUA code points below.
   */
   // for (i = 0xE000; i <= 0xF8FF; i++) {
   //    if (glyph_width[i] == 0) glyph_width[i]=2;
   // }

   /*
      <not a character>
   */
   for (i = 0xFDD0; i <= 0xFDEF; i++) glyph_width[i] = -1;
   glyph_width[0xFFFE] = -1; /* Byte Order Mark */
   glyph_width[0xFFFF] = -1; /* Byte Order Mark */

   /* Surrogate Code Points */
   for (i = 0xD800; i <= 0xDFFF; i++) glyph_width[i]=-1;

   /* CJK Code Points */
   for (i = 0x4E00; i <= 0x9FFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
   for (i = 0x3400; i <= 0x4DBF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
   for (i = 0xF900; i <= 0xFAFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;

   /*
      Now generate the output file.
   */
   printf ("/*\n");
   printf ("   wcwidth and wcswidth functions, as per IEEE 1003.1-2008\n");
   printf ("   System Interfaces, pp. 2241 and 2251.\n\n");
   printf ("   Author: Paul Hardy, 2013\n\n");
   printf ("   Copyright (c) 2013 Paul Hardy\n\n");
   printf ("   LICENSE:\n");
   printf ("\n");
```

```
printf ("      This program is free software: you can redistribute it and/or modify\n");
printf ("      it under the terms of the GNU General Public License as published by\n");
printf ("      the Free Software Foundation, either version 2 of the License, or\n");
printf ("      (at your option) any later version.\n");
printf ("\n");
printf ("      This program is distributed in the hope that it will be useful,\n");
printf ("      but WITHOUT ANY WARRANTY; without even the implied warranty of\n");
printf ("      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the\n");
printf ("      GNU General Public License for more details.\n");
printf ("\n");
printf ("      You should have received a copy of the GNU General Public License\n");
printf ("      along with this program.  If not, see <http://www.gnu.org/licenses/>.\n");
printf ("*/\n\n");

printf ("#include <wchar.h>\n\n");
printf ("/* Definitions for Pikto CSUR Private Use Area glyphs */\n");
printf ("#define PIKTO_START\t0x%06X\n", PIKTO_START);
printf ("#define PIKTO_END\t0x%06X\n", PIKTO_END);
printf ("#define PIKTO_SIZE\t(PIKTO_END - PIKTO_START + 1)\n");
printf ("\n\n");
printf ("/* wcwidth -- return charcell positions of one code point */\n");
printf ("inline int\nwcwidth (wchar_t wc)\n{\n");
printf ("   return (wcswidth (&wc, 1));\n");
printf ("}\n");
printf ("\n\n");
printf ("int\nwcswidth (const wchar_t *pwcs, size_t n)\n{\n\n");
printf ("   int i;                    /* loop variable                                      */\n");
printf ("   unsigned codept;          /* Unicode code point of current character            */\n");
printf ("   unsigned plane;           /* Unicode plane, 0x00..0x10                          */\n");
printf ("   unsigned lower17;         /* lower 17 bits of Unicode code point                */\n");
printf ("   unsigned lower16;         /* lower 16 bits of Unicode code point                */\n");
printf ("   int lowpt, midpt, highpt; /* for binary searching in plane1zeroes[]             */\n");
printf ("   int found;                /* for binary searching in plane1zeroes[]             */\n");
printf ("   int totalwidth;           /* total width of string, in charcells (1 or 2/glyph) */\n");
printf ("   int illegalchar;          /* Whether or not this code point is illegal          */\n");
putchar ('\n');

/*
   Print the glyph_width[] array for glyphs widths in the
   Basic Multilingual Plane (Plane 0).
*/
printf ("   char glyph_width[0x20000] = {");
for (i = 0; i < 0x10000; i++) {
   if ((i & 0x1F) == 0)
      printf ("\n      /* U+%04X */ ", i);
   printf ("%d,", glyph_width[i]);
}
for (i = 0x10000; i < 0x20000; i++) {
   if ((i & 0x1F) == 0)
      printf ("\n      /* U+%06X */ ", i);
   printf ("%d", glyph_width[i]);
   if (i < 0x1FFFF) putchar (',');
}
printf ("\n   };\n\n");

/*
   Print the pikto_width[] array for Pikto glyph widths.
*/
printf ("   char pikto_width[PIKTO_SIZE] = {");
for (i = 0; i < PIKTO_SIZE; i++) {
   if ((i & 0x1F) == 0)
      printf ("\n      /* U+%06X */ ", PIKTO_START + i);
   printf ("%d", pikto_width[i]);
   if ((PIKTO_START + i) < PIKTO_END) putchar (',');
}
printf ("\n   };\n\n");

/*
   Execution part of wcswidth.
*/
printf ("\n");
printf ("   illegalchar = totalwidth = 0;\n");
printf ("   for (i = 0; !illegalchar && i < n; i++) {\n");
printf ("      codept  = pwcs[i];\n");
printf ("      plane   = codept » 16;\n");
printf ("      lower17 = codept & 0x1FFFF;\n");
printf ("      lower16 = codept & 0xFFFF;\n");
printf ("      if (plane < 2) { /* the most common case */\n");
printf ("         if (glyph_width[lower17] < 0) illegalchar = 1;\n");
printf ("         else totalwidth += glyph_width[lower17];\n");
printf ("      }\n");
```

```
   printf ("      else { /* a higher plane or beyond Unicode range */\n");
   printf ("         if  ((lower16 == 0xFFFE) || (lower16 == 0xFFFF)) {\n");
   printf ("            illegalchar = 1;\n");
   printf ("         }\n");
   printf ("         else if (plane < 4) {  /* Ideographic Plane */\n");
   printf ("            totalwidth += 2; /* Default ideographic width */\n");
   printf ("         }\n");
   printf ("         else if (plane == 0x0F) {  /* CSUR Private Use Area */\n");
   printf ("            if (lower16 <= 0x0E6F) { /* Kinya */\n");
   printf ("               totalwidth++; /* all Kinya syllables have width 1 */\n");
   printf ("            }\n");
   printf ("            else if (lower16 <= (PIKTO_END & 0xFFFF)) { /* Pikto */\n");
   printf ("               if (pikto_width[lower16 - (PIKTO_START & 0xFFFF)] < 0) illegalchar = 1;\n");
   printf ("               else totalwidth += pikto_width[lower16 - (PIKTO_START & 0xFFFF)];\n");
   printf ("            }\n");
   printf ("         }\n");
   printf ("         else if (plane > 0x10) {\n");
   printf ("            illegalchar = 1;\n");
   printf ("         }\n");
   printf ("         /* Other non-printing in higher planes; return -1 as per IEEE 1003.1-2008. */\n");
   printf ("         else if (/* language tags */\n");
   printf ("                  codept == 0x0E0001 || (codept >= 0x0E0020 && codept <= 0x0E007F) ||\n");
   printf ("                  /* variation selectors, 0x0E0100..0x0E01EF */\n");
   printf ("                  (codept >= 0x0E0100 && codept <= 0x0E01EF)) {\n");
   printf ("            illegalchar = 1;\n");
   printf ("         }\n");
   printf ("         /*\n");
   printf ("            Unicode plane 0x02..0x10 printing character\n");
   printf ("         */\n");
   printf ("         else {\n");
   printf ("            illegalchar = 1; /* code is not in font */\n");
   printf ("         }\n");
   printf ("\n");
   printf ("      }\n");
   printf ("   }\n");
   printf ("   if (illegalchar) totalwidth = -1;\n");
   printf ("\n");
   printf ("   return (totalwidth);\n");
   printf ("\n");
   printf ("}\n");

   exit (EXIT_SUCCESS);
}
```

5.32 unigenwidth.c

Go to the documentation of this file.

```
/**
   @file unigenwidth.c

   @brief unigenwidth - IEEE 1003.1-2008 setup to calculate
                        wchar_t string widths

   @author Paul Hardy.

   @copyright Copyright (C) 2013, 2017 Paul Hardy.

   All glyphs are treated as 16 pixels high, and can be
   8, 16, 24, or 32 pixels wide (resulting in widths of
   1, 2, 3, or 4, respectively).
*/
/*
   LICENSE:

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

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

      You should have received a copy of the GNU General Public License
```

```
   along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

/*
  20 June 2017 [Paul Hardy]:
   - Now handles glyphs that are 24 or 32 pixels wide.

  8 July 2017 [Paul Hardy]:
   - Modifies sscanf format strings to ignore second field after
     the ":" field separator, newly added to "*combining.txt" files
     and already present in "*.hex" files.
*/

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

#define MAXSTRING   256   ///< Maximum input line length - 1.

/* Definitions for Pikto in Plane 15 */
#define PIKTO_START 0x0F0E70   ///< Start of Pikto code point range.
#define PIKTO_END   0x0F11EF   ///< End of Pikto code point range.
/** Number of code points in Pikto range. */
#define PIKTO_SIZE  (PIKTO_END - PIKTO_START + 1)


/**
  @brief The main function.

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

  int i; /* loop variable */

  char teststring[MAXSTRING];
  int  loc;
  char *gstart;

  char glyph_width[0x20000];
  char pikto_width[PIKTO_SIZE];

  FILE *infilefp;

  if (argc != 3) {
     fprintf (stderr, "\n\nUsage: %s <unifont.hex> <combining.txt>\n\n", argv[0]);
     exit (EXIT_FAILURE);
  }

  /*
     Read the collection of hex glyphs.
  */
  if ((infilefp = fopen (argv[1],"r")) == NULL) {
     fprintf (stderr,"ERROR - hex input file %s not found.\n\n", argv[1]);
     exit (EXIT_FAILURE);
  }

  /* Flag glyph as non-existent until found. */
  memset (glyph_width, -1, 0x20000 * sizeof (char));
  memset (pikto_width, -1, (PIKTO_SIZE) * sizeof (char));

  teststring[MAXSTRING-1] = '\0';
  while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
     sscanf (teststring, "%X:%*s", &loc);
     if (loc < 0x20000) {
        gstart = strchr (teststring,':') + 1;
        /*
           16 rows per glyph, 2 ASCII hexadecimal digits per byte,
           so divide number of digits by 32 (shift right 5 bits).
        */
        glyph_width[loc] = (strlen (gstart) - 1) >> 5;
     }
     else if ((loc >= PIKTO_START) && (loc <= PIKTO_END)) {
        gstart = strchr (teststring,':') + 1;
        pikto_width[loc - PIKTO_START] = strlen (gstart) <= 34 ? 1 : 2;
     }
  }
```

```

  fclose (infilefp);

  /*
     Now read the combining character code points.  These have width of 0.
  */
  if ((infilefp = fopen (argv[2],"r")) == NULL) {
     fprintf (stderr,"ERROR - combining characters file %s not found.\n\n", argv[2]);
     exit (EXIT_FAILURE);
  }

  while (fgets (teststring, MAXSTRING-1, infilefp) != NULL) {
     sscanf (teststring, "%X:%*s", &loc);
     if (loc < 0x20000) glyph_width[loc] = 0;
  }

  fclose (infilefp);

  /*
     Code Points with Unusual Properties (Unicode Standard, Chapter 4).

     As of Unifont 10.0.04, use the widths in the "*-nonprinting.hex"
     files.  If an application is smart enough to know how to handle
     these special cases, it will not render the "nonprinting" glyph
     and will treat the code point as being zero-width.
  */
// glyph_width[0]=0; /* NULL character */
// for (i = 0x0001; i <= 0x001F; i++) glyph_width[i]=-1; /* Control Characters */
// for (i = 0x007F; i <= 0x009F; i++) glyph_width[i]=-1; /* Control Characters */

// glyph_width[0x034F]=0; /* combining grapheme joiner               */
// glyph_width[0x180B]=0; /* Mongolian free variation selector one   */
// glyph_width[0x180C]=0; /* Mongolian free variation selector two   */
// glyph_width[0x180D]=0; /* Mongolian free variation selector three */
// glyph_width[0x180E]=0; /* Mongolian vowel separator               */
// glyph_width[0x200B]=0; /* zero width space                        */
// glyph_width[0x200C]=0; /* zero width non-joiner                   */
// glyph_width[0x200D]=0; /* zero width joiner                       */
// glyph_width[0x200E]=0; /* left-to-right mark                      */
// glyph_width[0x200F]=0; /* right-to-left mark                      */
// glyph_width[0x202A]=0; /* left-to-right embedding                 */
// glyph_width[0x202B]=0; /* right-to-left embedding                 */
// glyph_width[0x202C]=0; /* pop directional formatting              */
// glyph_width[0x202D]=0; /* left-to-right override                  */
// glyph_width[0x202E]=0; /* right-to-left override                  */
// glyph_width[0x2060]=0; /* word joiner                             */
// glyph_width[0x2061]=0; /* function application                    */
// glyph_width[0x2062]=0; /* invisible times                         */
// glyph_width[0x2063]=0; /* invisible separator                     */
// glyph_width[0x2064]=0; /* invisible plus                          */
// glyph_width[0x206A]=0; /* inhibit symmetric swapping              */
// glyph_width[0x206B]=0; /* activate symmetric swapping             */
// glyph_width[0x206C]=0; /* inhibit arabic form shaping             */
// glyph_width[0x206D]=0; /* activate arabic form shaping            */
// glyph_width[0x206E]=0; /* national digit shapes                   */
// glyph_width[0x206F]=0; /* nominal digit shapes                    */

// /* Variation Selector-1 to Variation Selector-16 */
// for (i = 0xFE00; i <= 0xFE0F; i++) glyph_width[i] = 0;

// glyph_width[0xFEFF]=0; /* zero width no-break space         */
// glyph_width[0xFFF9]=0; /* interlinear annotation anchor     */
// glyph_width[0xFFFA]=0; /* interlinear annotation separator  */
// glyph_width[0xFFFB]=0; /* interlinear annotation terminator */
  /*
     Let glyph widths represent 0xFFFC (object replacement character)
     and 0xFFFD (replacement character).
  */

  /*
     Hangul Jamo:

        Leading Consonant (Choseong): leave spacing as is.

        Hangul Choseong Filler (U+115F): set width to 2.

        Hangul Jungseong Filler, Hangul Vowel (Jungseong), and
        Final Consonant (Jongseong): set width to 0, because these
        combine with the leading consonant as one composite syllabic
        glyph.  As of Unicode 5.2, the Hangul Jamo block (U+1100..U+11FF)
        is completely filled.
```

```
*/
// for (i = 0x1160; i <= 0x11FF; i++) glyph_width[i]=0; /* Vowels & Final Consonants */

/*
   Private Use Area -- the width is undefined, but likely
   to be 2 charcells wide either from a graphic glyph or
   from a four-digit hexadecimal glyph representing the
   code point.  Therefore if any PUA glyph does not have
   a non-zero width yet, assign it a default width of 2.
   The Unicode Standard allows giving PUA characters
   default property values; see for example The Unicode
   Standard Version 5.0, p. 91.  This same default is
   used for higher plane PUA code points below.
*/
// for (i = 0xE000; i <= 0xF8FF; i++) {
//    if (glyph_width[i] == 0) glyph_width[i]=2;
// }

/*
   <not a character>
*/
for (i = 0xFDD0; i <= 0xFDEF; i++) glyph_width[i] = -1;
glyph_width[0xFFFE] = -1; /* Byte Order Mark */
glyph_width[0xFFFF] = -1; /* Byte Order Mark */

/* Surrogate Code Points */
for (i = 0xD800; i <= 0xDFFF; i++) glyph_width[i]=-1;

/* CJK Code Points */
for (i = 0x4E00; i <= 0x9FFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
for (i = 0x3400; i <= 0x4DBF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;
for (i = 0xF900; i <= 0xFAFF; i++) if (glyph_width[i] < 0) glyph_width[i] = 2;

/*
   Now generate the output file.
*/
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);
}
```

5.33 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.

5.33.1 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

Definition in file unihangul-support.c.

5.33.2 Function Documentation

5.33.2.1 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.

```
00350                                                    {
00351    int cho_variation;  /* Return value */
00352
00353    /*
00354       The Choseong cho_var is determined by the
00355       21 modern + 50 ancient Jungseong, and whether
00356       or not the syllable contains a final consonant
00357       (Jongseong).
00358    */
00359    static int choseong_var [TOTAL_JUNG + 1] = {
00360          /*
00361             Modern Jungseong in positions 0..20.
00362          */
00363 /* Location Variations  Unicode Range    Vowel #    Vowel Names  */
00364 /* -------- ----------  --------------   -------    -----------  */
00365 /* 0x2FB */ 0, 0, 0, // U+1161..U+1163-->[ 0.. 2]  A,   AE,  YA
00366 /* 0x304 */ 0, 0, 0, // U+1164..U+1166-->[ 3.. 5]  YAE, EO,  E
00367 /* 0x30D */ 0, 0,    // U+1167..U+1168-->[ 6.. 7]  YEO, YE
00368 /* 0x313 */ 1,       // U+1169         -->[ 8]      O
00369 /* 0x316 */ 2, 2, 2, // U+116A..U+116C-->[ 9..11]  WA,  WAE, WE
00370 /* 0x31F */ 1, 4,    // U+116D..U+116E-->[12..13]  YO,  U
00371 /* 0x325 */ 5, 5, 5, // U+116F..U+1171-->[14..16]  WEO, WE,  WI
00372 /* 0x32E */ 4, 1,    // U+1172..U+1173-->[17..18]  YU,  EU
00373 /* 0x334 */ 2,       // U+1174         -->[19]      YI
00374 /* 0x337 */ 0,       // U+1175         -->[20]      I
00375          /*
00376             Ancient Jungseong in positions 21..70.
00377          */
00378 /* Location  Variations  Unicode Range    Vowel #       Vowel Names  */
00379 /* --------  ----------  --------------   -------       -----------  */
00380 /* 0x33A: */ 2, 5, 2, // U+1176..U+1178-->[21..23]   A-O,    A-U,   YA-O
00381 /* 0x343: */ 2, 2, 5, // U+1179..U+117B-->[24..26]  YA-YO,  EO-O,   EU-U
00382 /* 0x34C: */ 2, 2, 5, // U+117C..U+117E-->[27..29]  EO-EU, YEO-O,  YEO-U
00383 /* 0x355: */ 2, 5, 5, // U+117F..U+1181-->[30..32]   O-EO,   O-E,    O-YE,
00384 /* 0x35E: */ 4, 4, 2, // U+1182..U+1184-->[33..35]   O-O,    O-U,   YO-YA,
00385 /* 0x367: */ 2, 2, 5, // U+1185..U+1187-->[36..38]  YO-YAE, YO-YEO, YO-O,
00386 /* 0x370: */ 2, 5, 5, // U+1188..U+118A-->[39..41]  YO-I,    U-A,    U-AE,
00387 /* 0x379: */ 5, 5, 5, // U+118B..U+118D-->[42..44] U-EO-EU,  U-YE,   U-U,
00388 /* 0x382: */ 5, 5, 5, // U+118E..U+1190-->[45..47]  YU-A,   YU-EO,  YU-E,
00389 /* 0x38B: */ 5, 5, 2, // U+1191..U+1193-->[48..50]  YU-YEO, YU-YE,  YU-U,
00390 /* 0x394: */ 5, 2, 2, // U+1194..U+1196-->[51..53]  YU-I,   EU-U,   EU-EU,
00391 /* 0x39D: */ 2, 0, 0, // U+1197..U+1199-->[54..56]  YI-U,    I-A,    I-YA,
00392 /* 0x3A6: */ 2, 5, 2, // U+119A..U+119C-->[57..59]   I-O,    I-U,    I-EU,
00393 /* 0x3AF: */ 0, 1, 2, // U+119D..U+119F-->[60..62] I-ARAEA, ARAEA, ARAEA-EO,
00394 /* 0x3B8: */ 1, 2, 1, // U+11A0..U+11A2-->[63..65] ARAEA-U, ARAEA-I,SSANGARAEA,
00395 /* 0x3C1: */ 2, 5, 0, // U+11A3..U+11A5-->[66..68]   A-EU,  YA-U,  YEO-YA,
00396 /* 0x3CA: */ 2, 2,    // U+11A6..U+11A7-->[69..70]   O-YA,   O-YAE,
00397 #ifdef EXTENDED_HANGUL
00398 /* 0x3D0: */ 2, 4, 5, // U+D7B0..U+D7B2-->[71..73]   O-YEO,  O-O-I, YO-A,
00399 /* 0x3D9: */ 5, 2, 5, // U+D7B3..U+D7B5-->[74..76]  YO-AE,  YO-EO,   U-YEO,
00400 /* 0x3E2: */ 5, 5, 4, // U+D7B6..U+D7B8-->[77..79]   U-I-I, YU-AE,  YU-O,
00401 /* 0x3EB: */ 5, 2, 5, // U+D7B9..U+D7BB-->[80..82]  EU-A,   EU-EO,  EU-E,
00402 /* 0x3F4: */ 4, 2, 3, // U+D7BC..U+D7BE-->[83..85]  EU-O,    I-YA-O, I-YAE,
00403 /* 0x3FD: */ 3, 3, 2, // U+D7BF..U+D7C1-->[86..88]   I-YEO,  I-YE,   I-O-I,
00404 /* 0x406: */ 2, 2, 0, // U+D7C2..U+D7C4-->[89..91]   I-YO,   I-YU,   I-I,
00405 /* 0x40F: */ 2, 2,    // U+D7C5..U+D7C6-->[92..93] ARAEA-A, ARAEA-E,
00406 /* 0x415: */ -1       // Mark end of list of vowels.
00407 #else
00408 /* 0x310: */ -1       // Mark end of list of vowels.
00409 #endif
00410    };
00411
00412
00413    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:

5.33.2.2 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:

5.33.2.3 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:

5.33.2.4 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.

```
00613                                                 {
00614    int overlaps;  /* Return value; 0 if no overlaps, -1 if overlaps. */
00615    int i;
00616 
00617    /* Check for overlaps between the two glyphs. */
00618 
00619    i = 0;
00620    do {
00621       overlaps = (glyph1[i] & glyph2[i]) != 0;
00622       i++;
00623    }  while (i < 16 && overlaps == 0);
00624 
00625    return overlaps;
00626 }
```

Here is the caller graph for this function:

5.33.2.5 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.

```
00201                                  {
00202    unsigned codept;
00203
00204
00205    if (initial >= 0 && initial <= 18 &&
00206        medial  >= 0 && medial  <= 20 &&
00207        final   >= 0 && final   <= 26) {
00208
00209       codept  = 0xAC00;
00210       codept += initial * 21 * 28;
00211       codept += medial  * 28;
00212       codept += final + 1;
00213    }
00214    else {
00215       codept = 0;
00216    }
00217
00218    return codept;
00219 }
```

5.33.2.6 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.

```
00167                                                  {
00168
00169    if (codept < 0xAC00 || codept > 0xD7A3) {
00170       *initial = *medial = *final = -1;
00171    }
00172    else {
00173       codept  -= 0xAC00;
00174       *initial =  codept / (28  * 21);
00175       *medial  = (codept /  28) % 21;
00176       *final   =  codept %  28 - 1;
00177    }
00178
00179    return;
00180 }
```

Here is the caller graph for this function:

5.33.2.7 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.

```
00250                                              {
00251
00252    int cho_variation, jung_variation, jong_variation;  /* Letter variations */
00253
00254    void hangul_variations (int choseong, int jungseong, int jongseong,
00255            int *cho_variation, int *jung_variation, int *jong_variation);
00256
00257
00258    hangul_variations (choseong, jungseong, jongseong,
00259                       &cho_variation, &jung_variation, &jong_variation);
00260
00261     *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:

5.33.2.8 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.

```
00116                                        {
00117    unsigned codept;
00118    unsigned max_codept;
00119    int      i, j;
00120    char     instring[MAXLINE];
00121
00122
00123    max_codept = 0;
00124
00125    while (fgets (instring, MAXLINE, infp) != NULL) {
00126       sscanf (instring, "%X", &codept);
00127       codept -= PUA_START;
00128       /* If code point is within range, add it */
00129       if (codept < MAX_GLYPHS) {
00130          /* Find the start of the glyph bitmap. */
00131          for (i = 1; instring[i] != '\0' && instring[i] != ':'; i++);
00132          if (instring[i] == ':') {
00133             i++;  /* Skip over ':' to get to start of bitmap. */
00134             for (j = 0; j < 16; j++) {
00135                sscanf (&instring[i], "%4X", &base[codept][j]);
00136                i += 4;
00137             }
00138             if (codept > max_codept) max_codept = codept;
00139          }
00140       }
00141    }
00142
```

```
00143     return max_codept;
00144 }
```

Here is the caller graph for this function:

5.33.2.9 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.

```
00063                                                           {
00064     unsigned codept;
00065     unsigned max_codept;
00066     int      i, j;
00067     char     instring[MAXLINE];
00068 
00069 
00070     max_codept = 0;
00071 
00072     while (fgets (instring, MAXLINE, infp) != NULL) {
00073        sscanf (instring, "%X", &codept);
00074        codept -= PUA_START;
00075        /* If code point is within range, add it */
00076        if (codept < MAX_GLYPHS) {
00077           /* Find the start of the glyph bitmap. */
00078           for (i = 1; instring[i] != '\0' && instring[i] != ':'; i++);
00079           if (instring[i] == ':') {
00080              i++;  /* Skip over ':' to get to start of bitmap. */
00081              for (j = 0; j < 32; j++) {
00082                 sscanf (&instring[i], "%2hhX", &base[codept][j]);
00083                 i += 2;
00084              }
00085              if (codept > max_codept) max_codept = codept;
00086           }
00087        }
00088     }
00089 
00090     return max_codept;
00091 }
```

Here is the caller graph for this function:

5.33.2.10 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:

5.33.2.11 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.

```
00299                                                   {
00300
00301    int  cho_variation (int choseong, int jungseong, int jongseong);
00302    int jung_variation (int choseong, int jungseong, int jongseong);
00303    int jong_variation (int choseong, int jungseong, int jongseong);
00304
00305    /*
00306       Find the variation for each letter component.
00307    */
00308     *cho_var =  cho_variation (choseong, jungseong, jongseong);
00309    *jung_var = jung_variation (choseong, jungseong, jongseong);
00310    *jong_var = jong_variation (choseong, jungseong, jongseong);
00311
00312
00313    return;
00314 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.33.2.12 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.

```
00434                         {
00435    int retval;  /* Return value. */
00436
00437    static int wide_vowel [TOTAL_JUNG + 1] = {
00438          /*
```

```
        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:

5.33.2.13 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.

```
00558                                                {
00559 
00560    return 0;  /* There is only one Jongseong variation. */
00561 }
```

Here is the caller graph for this function:

5.33.2.14 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.

```
00524                                                {
00525    int jung_variation;  /* Return value */
00526 
00527    if (jungseong < 0) {
00528       jung_variation = -1;
00529    }
00530    else {
00531       jung_variation = 0;
00532       if (jongseong >= 0) {
00533          if (jongseong == 3)
00534             jung_variation = 2;  /* Vowel for final Nieun. */
00535          else
00536             jung_variation = 1;
00537       }
00538    }
00539 
00540 
00541    return jung_variation;
00542 }
```

Here is the call graph for this function: Here is the caller graph for this function:

5.33.2.15 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.

```
00718                                  {
00719
00720    int i;  /* Loop variable */
00721    int glyph_index;  /* Location of glyph in "hangul-base.hex" array */
00722
00723
00724    /* If jamo is invalid range, use blank glyph, */
00725    if (jamo >= 0x1100 && jamo <= 0x11FF) {
00726       glyph_index = jamo - 0x1100 + JAMO_HEX;
00727    }
00728    else if (jamo >= 0xA960 && jamo <= 0xA97F) {
00729       glyph_index = jamo - 0xA960 + JAMO_EXTA_HEX;
00730    }
00731    else if (jamo >= 0xD7B0 && jamo <= 0xD7FF) {
00732       glyph_index = jamo - 0x1100 + JAMO_EXTB_HEX;
00733    }
00734    else {
00735       glyph_index = 0;
00736    }
00737
00738    for (i = 0; i < 16; i++) {
00739       jamo_glyph [i] = glyph_table [glyph_index] [i];
00740    }
00741
00742    return;
00743 }
```

5.33.2.16 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.

```
00692                                              {
00693
00694    int i;
00695
00696
00697    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:

5.33.2.17 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.

```
00656 {
00657     int i;
00658     unsigned mask;
00659
00660
00661     fprintf (fp, "%04X:", codept);
00662
00663     /* for each this_glyph row */
00664     for (i = 0; i < 16; i++) {
00665         mask = 0x8000;
00666         fputc ('\t', fp);
00667         while (mask != 0x0000) {
00668             if (mask & this_glyph [i]) {
00669                 fputc ('#', fp);
00670             }
00671             else {
00672                 fputc ('-', fp);
00673             }
00674             mask >>= 1;  /* shift to next bit in this_glyph row */
00675         }
00676         fputc ('\n', fp);
00677     }
00678     fputc ('\n', fp);
00679
00680     return;
00681 }
```

Here is the caller graph for this function:

5.34 unihangul-support.c

Go to the documentation of this file.

```
/**
    @file unihangul-support.c

    @brief 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 Copyright © 2023 Paul Hardy
*/
/*
  LICENSE:

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

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

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

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


/**
  @brief 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.

  @param[in] Input file pointer; can be stdin.
  @param[out] Array of bit patterns, with 32 8-bit values per letter.
  @return The maximum code point value read in the file.
*/
unsigned
hangul_read_base8 (FILE *infp, unsigned char base[][32]) {
  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;
}

```

```
/**
   @brief 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.

   @param[in] Input file pointer; can be stdin.
   @param[out] Array of bit patterns, with 16 16-bit values per letter.
   @return The maximum code point value read in the file.
*/
unsigned
hangul_read_base16 (FILE *infp, unsigned base[][16]) {
   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;
}


/**
   @brief 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.

   @param[in] codept The Unicode code point to decode, from 0xAC00 to 0xD7A3.
   @param[out] initial The 1st letter (choseong)  in the syllable.
   @param[out] initial The 2nd letter (jungseong) in the syllable.
   @param[out] initial The 3rd letter (jongseong) in the syllable.
*/
void
hangul_decompose (unsigned codept, int *initial, int *medial, int *final) {

   if (codept < 0xAC00 || codept > 0xD7A3) {
      *initial = *medial = *final = -1;
   }
   else {
      codept  -= 0xAC00;
```

```
        *initial =  codept / (28  * 21);
        *medial  = (codept /  28) % 21;
        *final   =  codept %  28 - 1;
    }

    return;
}


/**
    @brief 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.

    @param[in] initial The first letter (choseong), 0 to 18.
    @param[in] medial  The second letter (jungseong), 0 to 20.
    @param[in] final   The third letter (jongseong), 0 to 26 or -1 if none.
    @return The Unicode Hangul Syllables code point, 0xAC00 to 0xD7A3.
*/
unsigned
hangul_compose (int initial, int medial, int final) {
    unsigned codept;


    if (initial >= 0 && initial <= 18 &&
         medial  >= 0 && medial  <= 20 &&
         final   >= 0 && final   <= 26) {

        codept  = 0xAC00;
        codept += initial * 21 * 28;
        codept += medial  * 28;
        codept += final + 1;
    }
    else {
        codept = 0;
    }

    return codept;
}


/**
    @brief 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.

    @param[in] choseong  The 1st letter in the syllable.
    @param[in] jungseong The 2nd letter in the syllable.
    @param[in] jongseong The 3rd letter in the syllable, or -1 if none.
    @param[out] cho_index Index location to the 1st letter variation from the hangul-base.hex file.
    @param[out] jung_index Index location to the 2nd letter variation from the hangul-base.hex file.
    @param[out] jong_index Index location to the 3rd letter variation from the hangul-base.hex file.
*/
void
hangul_hex_indices (int choseong, int jungseong, int jongseong,
                    int *cho_index, int *jung_index, int *jong_index) {

    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;
}


/**
   @brief 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).

   @param[in] choseong  The 1st letter in the syllable.
   @param[in] jungseong The 2nd letter in the syllable.
   @param[in] jongseong The 3rd letter in the syllable, or -1 if none.
   @param[out] cho_var  Variation of the 1st letter from the hangul-base.hex file.
   @param[out] jung_var Variation of the 2nd letter from the hangul-base.hex file.
   @param[out] jong_var Variation of the 3rd letter from the hangul-base.hex file.
*/
void
hangul_variations (int choseong, int jungseong, int jongseong,
                   int *cho_var, int *jung_var, int *jong_var) {

   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;
}


/**
   @brief 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.

   @param[in] choseong  The 1st letter in the syllable.
   @param[in] jungseong The 2nd letter in the syllable.
   @param[in] jongseong The 3rd letter in the syllable.
   @return The choseong variation, 0 to 5.
*/
int
cho_variation (int choseong, int jungseong, int jongseong) {
   int cho_variation;  /* Return value */

   /*
      The Choseong cho_var is determined by the
      21 modern + 50 ancient Jungseong, and whether
      or not the syllable contains a final consonant
      (Jongseong).
   */
   static int choseong_var [TOTAL_JUNG + 1] = {
         /*
            Modern Jungseong in positions 0..20.
         */
/* Location Variations  Unicode Range    Vowel #    Vowel Names  */
/* -------- ----------  --------------   -------    -----------  */
/* 0x2FB */ 0, 0, 0, // U+1161..U+1163-->[ 0.. 2]  A,   AE,  YA
/* 0x304 */ 0, 0, 0, // U+1164..U+1166-->[ 3.. 5]  YAE, EO,  E
/* 0x30D */ 0, 0,    // U+1167..U+1168-->[ 6.. 7]  YEO, YE
/* 0x313 */ 1,       // U+1169         -->[ 8]     O
/* 0x316 */ 2, 2, 2, // U+116A..U+116C-->[ 9..11]  WA,  WAE, WE
/* 0x31F */ 1, 4,    // U+116D..U+116E-->[12..13]  YO,  U
/* 0x325 */ 5, 5, 5, // U+116F..U+1171-->[14..16]  WEO, WE,  WI
/* 0x32E */ 4, 1,    // U+1172..U+1173-->[17..18]  YU,  EU
/* 0x334 */ 2,       // U+1174         -->[19]     YI
/* 0x337 */ 0,       // U+1175         -->[20]     I
         /*
            Ancient Jungseong in positions 21..70.
         */
/* Location  Variations  Unicode Range    Vowel #       Vowel Names  */
/* --------  ----------  --------------   -------       -----------  */
/* 0x33A: */ 2, 5, 2, // U+1176..U+1178-->[21..23]   A-O,    A-U,   YA-O
/* 0x343: */ 2, 2, 5, // U+1179..U+117B-->[24..26]  YA-YO,  EO-O,   EU-U
/* 0x34C: */ 2, 2, 5, // U+117C..U+117E-->[27..29]  EO-EU, YEO-O,  YEO-U
/* 0x355: */ 2, 5, 5, // U+117F..U+1181-->[30..32]   O-EO,   O-E,    O-YE,
/* 0x35E: */ 4, 4, 2, // U+1182..U+1184-->[33..35]   O-O,    O-U,   YO-YA,
/* 0x367: */ 2, 2, 5, // U+1185..U+1187-->[36..38]  YO-YAE, YO-YEO, YO-O,
/* 0x370: */ 2, 5, 5, // U+1188..U+118A-->[39..41]  YO-I,    U-A,    U-AE,
/* 0x379: */ 5, 5, 5, // U+118B..U+118D-->[42..44] U-EO-EU,  U-YE,   U-U,
/* 0x382: */ 5, 5, 5, // U+118E..U+1190-->[45..47]  YU-A,   YU-EO,  YU-E,
/* 0x38B: */ 5, 5, 2, // U+1191..U+1193-->[48..50]  YU-YEO, YU-YE,  YU-U,
/* 0x394: */ 5, 2, 2, // U+1194..U+1196-->[51..53]  YU-I,   EU-U,   EU-EU,
/* 0x39D: */ 2, 0, 0, // U+1197..U+1199-->[54..56]  YI-U,    I-A,    I-YA,
/* 0x3A6: */ 2, 5, 2, // U+119A..U+119C-->[57..59]   I-O,    I-U,    I-EU,
/* 0x3AF: */ 0, 1, 2, // U+119D..U+119F-->[60..62] I-ARAEA, ARAEA, ARAEA-EO,
/* 0x3B8: */ 1, 2, 1, // U+11A0..U+11A2-->[63..65] ARAEA-U, ARAEA-I,SSANGARAEA,
/* 0x3C1: */ 2, 5, 0, // U+11A3..U+11A5-->[66..68]  A-EU,  YA-U,  YEO-YA,
/* 0x3CA: */ 2, 2,    // U+11A6..U+11A7-->[69..70]   O-YA,   O-YAE,
#ifdef EXTENDED_HANGUL
/* 0x3D0: */ 2, 4, 5, // U+D7B0..U+D7B2-->[71..73]   O-YEO,  O-O-I, YO-A,
/* 0x3D9: */ 5, 2, 5, // U+D7B3..U+D7B5-->[74..76]  YO-AE,  YO-EO,   U-YEO,
/* 0x3E2: */ 5, 5, 4, // U+D7B6..U+D7B8-->[77..79]   U-I-I, YU-AE,  YU-O,
/* 0x3EB: */ 5, 2, 5, // U+D7B9..U+D7BB-->[80..82]  EU-A,   EU-EO,  EU-E,
/* 0x3F4: */ 4, 2, 3, // U+D7BC..U+D7BE-->[83..85]  EU-O,    I-YA-O, I-YAE,
/* 0x3FD: */ 3, 3, 2, // U+D7BF..U+D7C1-->[86..88]   I-YEO,  I-YE,   I-O-I,
/* 0x406: */ 2, 2, 0, // U+D7C2..U+D7C4-->[89..91]   I-YO,   I-YU,   I-I,
/* 0x40F: */ 2, 2,    // 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 (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;
}


/**
   @brief Whether vowel has rightmost vertical stroke to the right.

   @param[in] vowel Vowel number, from 0 to TOTAL_JUNG - 1.
   @return 1 if this vowel's vertical stroke is wide on the right side; else 0.
*/
int
is_wide_vowel (int vowel) {
   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;
}


/**
    @brief 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.

    @param[in] choseong  The 1st letter in the syllable.
    @param[in] jungseong The 2nd letter in the syllable.
    @param[in] jongseong The 3rd letter in the syllable.
    @return The jungseong variation, 0 to 2.
*/
inline int
jung_variation (int choseong, int jungseong, int jongseong) {
    int jung_variation;  /* Return value */

    if (jungseong < 0) {
       jung_variation = -1;
    }
    else {
       jung_variation = 0;
       if (jongseong >= 0) {
          if (jongseong == 3)
             jung_variation = 2;  /* Vowel for final Nieun. */
          else
             jung_variation = 1;
       }
    }


    return jung_variation;
}


/**
    @brief 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.

    @param[in] choseong  The 1st letter in the syllable.
    @param[in] jungseong The 2nd letter in the syllable.
    @param[in] jongseong The 3rd letter in the syllable.
    @return The jongseong variation, always 0.
*/
inline int
jong_variation (int choseong, int jungseong, int jongseong) {

    return 0;  /* There is only one Jongseong variation. */
}


/**
    @brief 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.

    @param[in] choseong  The 1st letter in the composite glyph.
    @param[in] jungseong The 2nd letter in the composite glyph.
    @param[in] jongseong The 3rd letter in the composite glyph.
```

```
   @param[in] hangul_base The glyphs read from the "hangul_base.hex" file.
   @return syllable The composite syllable, as a 16 by 16 pixel bitmap.
*/
void
hangul_syllable (int choseong, int jungseong, int jongseong,
                 unsigned char hangul_base[][32], unsigned char *syllable) {

   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;
}


/**
   @brief See if two glyphs overlap.

   @param[in] glyph1 The first glyph, as a 16-row bitmap.
   @param[in] glyph2 The second glyph, as a 16-row bitmap.
   @return 0 if no overlaps between glyphs, 1 otherwise.
*/
int
glyph_overlap (unsigned *glyph1, unsigned *glyph2) {
   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;
}


/**
   @brief Combine two glyphs into one glyph.

   @param[in] glyph1 The first glyph to overlap.
   @param[in] glyph2 The second glyph to overlap.
   @param[out] combined_glyph The returned combination glyph.
*/
void
combine_glyphs (unsigned *glyph1, unsigned *glyph2,
                unsigned *combined_glyph) {
   int i;

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

   return;
}


/**
   @brief Print one glyph in Unifont hexdraw plain text style.

   @param[in] fp         The file pointer for output.
   @param[in] codept     The Unicode code point to print with the glyph.
   @param[in] this_glyph The 16-row by 16-column glyph to print.
*/
void
print_glyph_txt (FILE *fp, unsigned codept, unsigned *this_glyph) {
   int i;
   unsigned mask;

```

```

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

    /* for each this_glyph row */
    for (i = 0; i < 16; i++) {
       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;
 }


 /**
    @brief Print one glyph in Unifont hexdraw hexadecimal string style.

    @param[in] fp          The file pointer for output.
    @param[in] codept      The Unicode code point to print with the glyph.
    @param[in] this_glyph The 16-row by 16-column glyph to print.
 */
 void
 print_glyph_hex (FILE *fp, unsigned codept, unsigned *this_glyph) {

    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;
 }


 /**
    @brief Convert Hangul Jamo choseong, jungseong, and jongseong into a glyph.

    @param[in]  glyph_table The collection of all jamo glyphs.
    @param[in]  jamo        The Unicode code point, 0 or 0x1100..0x115F.
    @param[out] jamo_glyph  The output glyph, 16 columns in each of 16 rows.
 */
 void
 one_jamo (unsigned glyph_table [MAX_GLYPHS][16],
           unsigned jamo, unsigned *jamo_glyph) {

    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;
}


/**
  @brief 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.

  @param[in] glyph_table The collection of all jamo glyphs.
  @param[in] cho The choseong Unicode code point, 0 or 0x1100..0x115F.
  @param[in] jung The jungseong Unicode code point, 0 or 0x1160..0x11A7.
  @param[in] jong The jongseong Unicode code point, 0 or 0x11A8..0x11FF.
  @param[out] combined_glyph The output glyph, 16 columns in each of 16 rows.
*/
void
combined_jamo (unsigned glyph_table [MAX_GLYPHS][16],
               unsigned cho, unsigned jung, unsigned jong,
               unsigned *combined_glyph) {

  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;
}

```

5.35 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.

5.35.1 Detailed Description

unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points into a bitmap for editing

Author

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

Copyright

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

This program reads in a GNU Unifont .hex file, extracts a range of 256 code points, and converts it a Microsoft Bitmap (.bmp) or Wireless Bitmap file.
Synopsis: unihex2bmp [-iin_file.hex] [-oout_file.bmp] [-f] [-phex_page_num] [-w]
Definition in file unihex2bmp.c.

5.35.2 Macro Definition Documentation

5.35.2.1 MAXBUF

#define MAXBUF 256
Definition at line 50 of file unihex2bmp.c.

5.35.3 Function Documentation

5.35.3.1 hex2bit()

```
int hex2bit (
        char * instring,
        unsigned char character[32][4] )
```

Generate a bitmap for one glyph.
Convert the portion of a hex string after the ':' into a character bitmap.
If string is >= 128 characters, it will fill all 4 bytes per row. If string is >= 64 characters and < 128, it will fill 2 bytes per row. Otherwise, it will fill 1 byte per row.

Parameters

in	instring	The character array containing the glyph bitmap.
out	character	Glyph bitmap, 8, 16, or 32 columns by 16 rows tall.

Returns

Always returns 0.

Definition at line 367 of file unihex2bmp.c.

```
00368 {
00369
00370    int i;  /* current row in bitmap character */
00371    int j;  /* current character in input string */
00372    int k;  /* current byte in bitmap character */
00373    int width;  /* number of output bytes to fill - 1: 0, 1, 2, or 3 */
00374
00375    for (i=0; i<32; i++)  /* erase previous character */
00376       character[i][0] = character[i][1] = character[i][2] = character[i][3] = 0;
00377    j=0;  /* current location is at beginning of instring */
00378
00379    if (strlen (instring) <= 34)  /* 32 + possible '\r', '\n' */
00380       width = 0;
00381    else if (strlen (instring) <= 66)  /* 64 + possible '\r', '\n' */
00382       width = 1;
00383    else if (strlen (instring) <= 98)  /* 96 + possible '\r', '\n' */
00384       width = 3;
00385    else  /* the maximum allowed is quadruple-width */
00386       width = 4;
00387
00388    k = (width > 1) ? 0 : 1; /* if width > double, start at index 1 else at 0 */
00389
00390    for (i=8; i<24; i++) {  /* 16 rows per input character, rows 8..23 */
00391       sscanf (&instring[j], "%2hhx", &character[i][k]);
00392       j += 2;
00393       if (width > 0) { /* add next pair of hex digits to this row */
00394          sscanf (&instring[j], "%2hhx", &character[i][k+1]);
00395          j += 2;
00396          if (width > 1) { /* add next pair of hex digits to this row */
00397             sscanf (&instring[j], "%2hhx", &character[i][k+2]);
00398             j += 2;
00399             if (width > 2) {  /* quadruple-width is maximum width */
00400                sscanf (&instring[j], "%2hhx", &character[i][k+3]);
00401                j += 2;
00402             }
00403          }
00404       }
00405    }
00406
00407    return (0);
00408 }
```

Here is the caller graph for this function:

5.35.3.2 init()

int init (

unsigned char bitmap[17 *32][18 *4])

Initialize the bitmap grid.

Parameters

out	bitmap	The bitmap to generate, with 32x32 pixel glyph areas.

Returns

Always returns 0.

Definition at line 418 of file unihex2bmp.c.

```
00419 {
00420    int i, j;
00421    unsigned char charbits[32][4];  /* bitmap for one character, 4 bytes/row */
00422    unsigned toppixelrow;
00423    unsigned thiscol;
00424    unsigned char pnybble0, pnybble1, pnybble2, pnybble3;
00425
00426    for (i=0; i<18; i++) { /* bitmaps for '0'..'9', 'A'-'F', 'u', '+' */
00427
```

```
   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:

5.35.3.3 main()

int main (
int argc,
char ∗ argv[])

The main function.

Parameters

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

Returns

This program exits with status 0.

Definition at line 99 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 */

    /* initializes bitmap row/col labeling, &c. */
    int init (unsigned char bitmap[17*32][18*4]);

    /* convert hex string --> bitmap */
    int hex2bit (char *instring, unsigned char character[32][4]);

```

```
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:

5.35.4 Variable Documentation

5.35.4.1 flip

int flip =1

Transpose entire matrix as in Unicode book.

Definition at line 88 of file unihex2bmp.c.

5.35.4.2 hex

char∗ hex[18]

Initial value:

```
= {
    "0030:00000000182442424242424224180000",
    "0031:00000000081828080808080808083E0000",
    "0032:000000003C4242020C102040407E0000",
    "0033:000000003C4242021C020242423C0000",
    "0034:00000000040C142444447E0404040000",
    "0035:000000007E4040407C020202423C0000",
    "0036:000000001C2040407C424242423C0000",
    "0037:000000007E020204040408080808080000",
    "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 65 of file unihex2bmp.c.

5.35.4.3 hexbits

unsigned char hexbits[18][32]

The digits converted into bitmaps.

Definition at line 85 of file unihex2bmp.c.

5.35.4.4 unipage

unsigned unipage =0

Unicode page number, 0x00..0xff.

Definition at line 87 of file unihex2bmp.c.

5.36 unihex2bmp.c

Go to the documentation of this file.

```
/**
  @file unihex2bmp.c

  @brief unihex2bmp - Turn a GNU Unifont hex glyph page of 256 code points
                      into a bitmap for editing

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

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

  This program reads in a GNU Unifont .hex file, extracts a range of
  256 code points, and converts it a Microsoft Bitmap (.bmp) or Wireless
  Bitmap file.

  Synopsis: unihex2bmp [-iin_file.hex] [-oout_file.bmp]
            [-f] [-phex_page_num] [-w]
*/
/*
  LICENSE:
```

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

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

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

/*
  20 June 2017 [Paul Hardy]:
  - Adds capability to output triple-width and quadruple-width (31 pixels
    wide, not 32) glyphs.  The 32nd column in a glyph cell is occupied by
    the vertical cell border, so a quadruple-width glyph can only occupy
    the first 31 columns; the 32nd column is ignored.

  21 October 2023 [Paul Hardy]:
  - Added full prototypes in main function for init and hex2bit functions.
*/

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

#define MAXBUF 256


/**
  @brief GNU Unifont bitmaps for hexadecimal digits.

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

  Looking at the final bitmap as a grid of 32*32 bit tiles, the
  first row contains a hexadecimal character string of the first
  3 hex digits in a 4 digit Unicode character name; the top column
  contains a hex character string of the 4th (low-order) hex digit
  of the Unicode character.
*/
char *hex[18]= {
      "0030:00000000182442424242424224180000",  /* Hex digit 0 */
      "0031:000000000818280808080808083E0000",  /* Hex digit 1 */
      "0032:000000003C4242020C102040407E0000",  /* Hex digit 2 */
      "0033:000000003C4242021C020242423C0000",  /* Hex digit 3 */
      "0034:00000000040C142444447E0404040000",  /* Hex digit 4 */
      "0035:000000007E4040407C020202423C0000",  /* Hex digit 5 */
      "0036:000000001C2040407C424242423C0000",  /* Hex digit 6 */
      "0037:000000007E0202040404080808080000",  /* Hex digit 7 */
      "0038:000000003C4242423C424242423C0000",  /* Hex digit 8 */
      "0039:000000003C4242423E02020204380000",  /* Hex digit 9 */
      "0041:0000000018242442427E424242420000",  /* Hex digit A */
      "0042:000000007C4242427C424242427C0000",  /* Hex digit B */
      "0043:000000003C42424040404042423C0000",  /* Hex digit C */
      "0044:00000000784442424242424244780000",  /* Hex digit D */
      "0045:000000007E4040407C404040407E0000",  /* Hex digit E */
      "0046:000000007E4040407C40404040400000",  /* Hex digit F */
      "0055:000000004242424242424242423C0000",  /* Unicode 'U' */
      "002B:0000000000000808087F080808000000"   /* Unicode '+' */
   };
unsigned char hexbits[18][32]; ///< The digits converted into bitmaps.

unsigned unipage=0;   ///< Unicode page number, 0x00..0xff.
int flip=1;           ///< Transpose entire matrix as in Unicode book.


/**
  @brief The main function.

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

```

int i, j;                    /* loop variables                       */
unsigned k0;                 /* temp Unicode char variable         */
unsigned swap;               /* temp variable for swapping values */
char inbuf[256];             /* input buffer                          */
unsigned filesize;           /* size of file in bytes               */
unsigned bitmapsize;         /* size of bitmap image in bytes      */
unsigned thischar;           /* the current character               */
unsigned char thischarbyte; /* unsigned char lowest byte of Unicode char */
int thischarrow;             /* row 0..15 where this character belongs  */
int thiscol;                 /* column 0..15 where this character belongs */
int toppixelrow;             /* pixel row, 0..16*32-1               */
unsigned lastpage=0;         /* the last Unicode page read in font file */
int wbmp=0;                  /* set to 1 if writing .wbmp format file */

unsigned char bitmap[17*32][18*4]; /* final bitmap */
unsigned char charbits[32][4];  /* bitmap for one character, 4 bytes/row */

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

/* initializes bitmap row/col labeling, &c. */
int init (unsigned char bitmap[17*32][18*4]);

/* convert hex string --> bitmap */
int hex2bit (char *instring, unsigned char character[32][4]);

bitmapsize = 17*32*18*4;  /* 17 rows by 18 cols, each 4 bytes */

if (argc > 1) {
   for (i = 1; i < argc; i++) {
      if (argv[i][0] == '-') {  /* this is an option argument */
         switch (argv[i][1]) {
            case 'f':  /* flip (transpose) glyphs in bitmap as in standard */
               flip = !flip;
               break;
            case 'i':  /* name of input file */
               infile = &argv[i][2];
               break;
            case 'o':  /* name of output file */
               outfile = &argv[i][2];
               break;
            case 'p':  /* specify a Unicode page other than default of 0 */
               sscanf (&argv[i][2], "%x", &unipage); /* Get Unicode page */
               break;
            case 'w':  /* write a .wbmp file instead of a .bmp file */
               wbmp = 1;
               break;
            default:   /* if unrecognized option, print list and exit */
               fprintf (stderr, "\nSyntax:\n\n");
               fprintf (stderr, "   %s -p<Unicode_Page> ", argv[0]);
               fprintf (stderr, "-i<Input_File> -o<Output_File> -w\n\n");
               fprintf (stderr, "   -w specifies .wbmp output instead of ");
               fprintf (stderr, "default Windows .bmp output.\n\n");
               fprintf (stderr, "   -p is followed by 1 to 6 ");
               fprintf (stderr, "Unicode page hex digits ");
               fprintf (stderr, "(default is Page 0).\n\n");
               fprintf (stderr, "\nExample:\n\n");
               fprintf (stderr, "   %s -p83 -iunifont.hex -ou83.bmp\n\n\n",
                      argv[0]);
               exit (1);
         }
      }
   }
}
/*
   Make sure we can open any I/O files that were specified before
   doing anything else.
*/
if (strlen (infile) > 0) {
   if ((infp = fopen (infile, "r")) == NULL) {
      fprintf (stderr, "Error: can't open %s for input.\n", infile);
      exit (1);
   }
}
else {
   infp = stdin;
}
if (strlen (outfile) > 0) {
   if ((outfp = fopen (outfile, "w")) == NULL) {
      fprintf (stderr, "Error: can't open %s for output.\n", outfile);
```

```
        exit (1);
    }
}
else {
    outfp = stdout;
}

(void)init(bitmap); /* initialize bitmap with row/column headers, etc. */

/*
    Read in the characters in the page
*/
while (lastpage <= unipage && fgets (inbuf, MAXBUF-1, infp) != NULL) {
    sscanf (inbuf, ”%x”, &thischar);
    lastpage = thischar » 8; /* keep Unicode page to see if we can stop */
    if (lastpage == unipage) {
        thischarbyte = (unsigned char)(thischar & 0xff);
        for (k0=0; inbuf[k0] != ':'; k0++);
        k0++;
        hex2bit (&inbuf[k0], charbits);  /* convert hex string to 32*4 bitmap */

        /*
            Now write character bitmap upside-down in page array, to match
            .bmp file order.  In the .wbmp‘ and .bmp files, white is a ’1’
            bit and black is a ’0’ bit, so complement charbits[][].
        */

        thiscol = (thischarbyte & 0xf) + 2;  /* column number will be 1..16  */
        thischarrow = thischarbyte » 4;      /* charcter row number, 0..15   */
        if (flip) {  /* swap row and column placement */
            swap = thiscol;
            thiscol = thischarrow;
            thischarrow = swap;
            thiscol += 2;        /* column index starts at 1 */
            thischarrow -= 2;   /* row index starts at 0    */
        }
        toppixelrow = 32 * (thischarrow + 1) - 1; /* from bottom to top    */

        /*
            Copy the center of charbits[][] because hex characters only
            occupy rows 8 to 23 and column byte 2 (and for 16 bit wide
            characters, byte 3).  The charbits[][] array was given 32 rows
            and 4 column bytes for completeness in the beginning.
        */
        for (i=8; i<24; i++) {
            bitmap[toppixelrow + i][(thiscol « 2) | 0] =
                ~charbits[i][0] & 0xff;
            bitmap[toppixelrow + i][(thiscol « 2) | 1] =
                ~charbits[i][1] & 0xff;
            bitmap[toppixelrow + i][(thiscol « 2) | 2] =
                ~charbits[i][2] & 0xff;
            /* Only use first 31 bits; leave vertical rule in 32nd column */
            bitmap[toppixelrow + i][(thiscol « 2) | 3] =
                ~charbits[i][3] & 0xfe;
        }
        /*
            Leave white space in 32nd column of rows 8, 14, 15, and 23
            to leave 16 pixel height upper, middle, and lower guides.
        */
        bitmap[toppixelrow +  8][(thiscol « 2) | 3] |= 1;
        bitmap[toppixelrow + 14][(thiscol « 2) | 3] |= 1;
        bitmap[toppixelrow + 15][(thiscol « 2) | 3] |= 1;
        bitmap[toppixelrow + 23][(thiscol « 2) | 3] |= 1;
    }
}
/*
    Now write the appropriate bitmap file format, either
    Wireless Bitmap or Microsoft Windows bitmap.
*/
if (wbmp) {  /* Write a Wireless Bitmap .wbmp format file */
    /*
        Write WBMP header
    */
    fprintf (outfp, ”%c”, 0x00); /* Type of image; always 0 (monochrome) */
    fprintf (outfp, ”%c”, 0x00); /* Reserved; always 0                   */
    fprintf (outfp, ”%c%c”, 0x84, 0x40); /* Width  = 576 pixels          */
    fprintf (outfp, ”%c%c”, 0x84, 0x20); /* Height = 544 pixels          */
    /*
        Write bitmap image
    */
    for (toppixelrow=0; toppixelrow <= 17*32-1; toppixelrow++) {
```

```
      for (j=0; j<18; j++) {
         fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2)    ]);
         fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2) | 1]);
         fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2) | 2]);
         fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2) | 3]);
      }
   }
}
else { /* otherwise, write a Microsoft Windows .bmp format file */
   /*
      Write the .bmp file -- start with the header, then write the bitmap
   */

   /* ’B’, ’M’ appears at start of every .bmp file */
   fprintf (outfp, ”%c%c”, 0x42, 0x4d);

   /* Write file size in bytes */
   filesize   = 0x3E + bitmapsize;
   fprintf (outfp, ”%c”, (unsigned char)((filesize        ) & 0xff));
   fprintf (outfp, ”%c”, (unsigned char)((filesize » 0x08) & 0xff));
   fprintf (outfp, ”%c”, (unsigned char)((filesize » 0x10) & 0xff));
   fprintf (outfp, ”%c”, (unsigned char)((filesize » 0x18) & 0xff));

   /* Reserved - 0’s */
   fprintf (outfp, ”%c%c%c%c”, 0x00, 0x00, 0x00, 0x00);

   /* Offset from start of file to bitmap data */
   fprintf (outfp, ”%c%c%c%c”, 0x3E, 0x00, 0x00, 0x00);

   /* Length of bitmap info header */
   fprintf (outfp, ”%c%c%c%c”, 0x28, 0x00, 0x00, 0x00);

   /* Width of bitmap in pixels */
   fprintf (outfp, ”%c%c%c%c”, 0x40, 0x02, 0x00, 0x00);

   /* Height of bitmap in pixels */
   fprintf (outfp, ”%c%c%c%c”, 0x20, 0x02, 0x00, 0x00);

   /* Planes in bitmap (fixed at 1) */
   fprintf (outfp, ”%c%c”, 0x01, 0x00);

   /* bits per pixel (1 = monochrome) */
   fprintf (outfp, ”%c%c”, 0x01, 0x00);

   /* Compression (0 = none) */
   fprintf (outfp, ”%c%c%c%c”, 0x00, 0x00, 0x00, 0x00);

   /* Size of bitmap data in bytes */
   fprintf (outfp, ”%c”, (unsigned char)((bitmapsize        ) & 0xff));
   fprintf (outfp, ”%c”, (unsigned char)((bitmapsize » 0x08) & 0xff));
   fprintf (outfp, ”%c”, (unsigned char)((bitmapsize » 0x10) & 0xff));
   fprintf (outfp, ”%c”, (unsigned char)((bitmapsize » 0x18) & 0xff));

   /* Horizontal resolution in pixels per meter */
   fprintf (outfp, ”%c%c%c%c”, 0xC4, 0x0E, 0x00, 0x00);

   /* Vertical resolution in pixels per meter */
   fprintf (outfp, ”%c%c%c%c”, 0xC4, 0x0E, 0x00, 0x00);

   /* Number of colors used */
   fprintf (outfp, ”%c%c%c%c”, 0x02, 0x00, 0x00, 0x00);

   /* Number of important colors */
   fprintf (outfp, ”%c%c%c%c”, 0x02, 0x00, 0x00, 0x00);

   /* The color black: B=0x00, G=0x00, R=0x00, Filler=0xFF */
   fprintf (outfp, ”%c%c%c%c”, 0x00, 0x00, 0x00, 0x00);

   /* The color white: B=0xFF, G=0xFF, R=0xFF, Filler=0xFF */
   fprintf (outfp, ”%c%c%c%c”, 0xFF, 0xFF, 0xFF, 0x00);

   /*
      Now write the raw data bits.  Data is written from the lower
      left-hand corner of the image to the upper right-hand corner
      of the image.
   */
   for (toppixelrow=17*32-1; toppixelrow >= 0; toppixelrow--) {
      for (j=0; j<18; j++) {
         fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2)    ]);
         fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2) | 1]);
         fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2) | 2]);
```

```

            fprintf (outfp, ”%c”, bitmap[toppixelrow][(j«2) | 3]);
         }
      }
   }
   exit (0);
}


/**
   @brief Generate a bitmap for one glyph.

   Convert the portion of a hex string after the ’:’ into a character bitmap.

   If string is >= 128 characters, it will fill all 4 bytes per row.
   If string is >= 64 characters and < 128, it will fill 2 bytes per row.
   Otherwise, it will fill 1 byte per row.

   @param[in] instring The character array containing the glyph bitmap.
   @param[out] character Glyph bitmap, 8, 16, or 32 columns by 16 rows tall.
   @return Always returns 0.
*/
int
hex2bit (char *instring, unsigned char character[32][4])
{

   int i;  /* current row in bitmap character */
   int j;  /* current character in input string */
   int k;  /* current byte in bitmap character */
   int width;  /* number of output bytes to fill - 1: 0, 1, 2, or 3 */

   for (i=0; i<32; i++)  /* erase previous character */
      character[i][0] = character[i][1] = character[i][2] = character[i][3] = 0;
   j=0;  /* current location is at beginning of instring */

   if (strlen (instring) <= 34)  /* 32 + possible ’\r’, ’\n’ */
      width = 0;
   else if (strlen (instring) <= 66)  /* 64 + possible ’\r’, ’\n’ */
      width = 1;
   else if (strlen (instring) <= 98)  /* 96 + possible ’\r’, ’\n’ */
      width = 3;
   else  /* the maximum allowed is quadruple-width */
      width = 4;

   k = (width > 1) ? 0 : 1; /* if width > double, start at index 1 else at 0 */

   for (i=8; i<24; i++) {  /* 16 rows per input character, rows 8..23 */
      sscanf (&instring[j], ”%2hhx”, &character[i][k]);
      j += 2;
      if (width > 0) { /* add next pair of hex digits to this row */
         sscanf (&instring[j], ”%2hhx”, &character[i][k+1]);
         j += 2;
         if (width > 1) { /* add next pair of hex digits to this row */
            sscanf (&instring[j], ”%2hhx”, &character[i][k+2]);
            j += 2;
            if (width > 2) {  /* quadruple-width is maximum width */
               sscanf (&instring[j], ”%2hhx”, &character[i][k+3]);
               j += 2;
            }
         }
      }
   }

   return (0);
}


/**
   @brief Initialize the bitmap grid.

   @param[out] bitmap The bitmap to generate, with 32x32 pixel glyph areas.
   @return Always returns 0.
*/
int
init (unsigned char bitmap[17*32][18*4])
{
   int i, j;
   unsigned char charbits[32][4];  /* bitmap for one character, 4 bytes/row */
   unsigned toppixelrow;
   unsigned thiscol;
   unsigned char pnybble0, pnybble1, pnybble2, pnybble3;
```

```

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

   hex2bit (&hex[i][5], charbits);  /* convert hex string to 32*4 bitmap */

   for (j=0; j<32; j++) hexbits[i][j] = ~charbits[j][1];
}

/*
   Initialize bitmap to all white.
*/
for (toppixelrow=0; toppixelrow < 17*32; toppixelrow++) {
   for (thiscol=0; thiscol<18; thiscol++) {
      bitmap[toppixelrow][(thiscol << 2)    ] = 0xff;
      bitmap[toppixelrow][(thiscol << 2) | 1] = 0xff;
      bitmap[toppixelrow][(thiscol << 2) | 2] = 0xff;
      bitmap[toppixelrow][(thiscol << 2) | 3] = 0xff;
   }
}
/*
   Write the "u+nnnn" table header in the upper left-hand corner,
   where nnnn is the upper 16 bits of a 32-bit Unicode assignment.
*/
pnybble3 = (unipage >> 20);
pnybble2 = (unipage >> 16) & 0xf;
pnybble1 = (unipage >> 12) & 0xf;
pnybble0 = (unipage >>  8) & 0xf;
for (i=0; i<32; i++) {
   bitmap[i][1] = hexbits[16][i];  /* copy 'u' */
   bitmap[i][2] = hexbits[17][i];  /* copy '+' */
   bitmap[i][3] = hexbits[pnybble3][i];
   bitmap[i][4] = hexbits[pnybble2][i];
   bitmap[i][5] = hexbits[pnybble1][i];
   bitmap[i][6] = hexbits[pnybble0][i];
}
/*
   Write low-order 2 bytes of Unicode number assignments, as hex labels
*/
pnybble3 = (unipage >> 4) & 0xf;  /* Highest-order hex digit */
pnybble2 = (unipage     ) & 0xf;  /* Next highest-order hex digit */
/*
   Write the column headers in bitmap[][] (row headers if flipped)
*/
toppixelrow = 32 * 17 - 1; /* maximum pixel row number */
/*
   Label the column headers. The hexbits[][] bytes are split across two
   bitmap[][] entries to center a the hex digits in a column of 4 bytes.
   OR highest byte with 0xf0 and lowest byte with 0x0f to make outer
   nybbles white (0=black, 1-white).
*/
for (i=0; i<16; i++) {
   for (j=0; j<32; j++) {
      if (flip) {  /* transpose matrix */
         bitmap[j][((i+2) << 2) | 0]  = (hexbits[pnybble3][j] >> 4) | 0xf0;
         bitmap[j][((i+2) << 2) | 1]  = (hexbits[pnybble3][j] << 4) |
                                        (hexbits[pnybble2][j] >> 4);
         bitmap[j][((i+2) << 2) | 2]  = (hexbits[pnybble2][j] << 4) |
                                        (hexbits[i][j] >> 4);
         bitmap[j][((i+2) << 2) | 3]  = (hexbits[i][j] << 4) | 0x0f;
      }
      else {
         bitmap[j][((i+2) << 2) | 1] = (hexbits[i][j] >> 4) | 0xf0;
         bitmap[j][((i+2) << 2) | 2] = (hexbits[i][j] << 4) | 0x0f;
      }
   }
}
/*
   Now use the single hex digit column graphics to label the row headers.
*/
for (i=0; i<16; i++) {
   toppixelrow = 32 * (i + 1) - 1; /* from bottom to top    */

   for (j=0; j<32; j++) {
      if (!flip) {  /* if not transposing matrix */
         bitmap[toppixelrow + j][4] = hexbits[pnybble3][j];
         bitmap[toppixelrow + j][5] = hexbits[pnybble2][j];
      }
      bitmap[toppixelrow + j][6] = hexbits[i][j];
   }
}
/*
```

```
    Now draw grid lines in bitmap, around characters we just copied.
  */
  /* draw vertical lines 2 pixels wide */
  for (i=1*32; i<17*32; i++) {
    if ((i & 0x1f) == 7)
      i++;
    else if ((i & 0x1f) == 14)
      i += 2;
    else if ((i & 0x1f) == 22)
      i++;
    for (j=1; j<18; j++) {
      bitmap[i][(j « 2) | 3] &= 0xfe;
    }
  }
  /* draw horizontal lines 1 pixel tall */
  for (i=1*32-1; i<18*32-1; i+=32) {
    for (j=2; j<18; j++) {
      bitmap[i][(j « 2)    ] = 0x00;
      bitmap[i][(j « 2) | 1] = 0x81;
      bitmap[i][(j « 2) | 2] = 0x81;
      bitmap[i][(j « 2) | 3] = 0x00;
    }
  }
  /* fill in top left corner pixel of grid */
  bitmap[31][7] = 0xfe;

  return (0);
}
```

5.37 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.

5.37.1 Detailed Description

unihexgen - Generate a series of glyphs containing hexadecimal code points

Author

Paul Hardy

Copyright

Copyright (C) 2013 Paul Hardy

This program generates glyphs in Unifont .hex format that contain four- or six-digit hexadecimal numbers in a 16x16 pixel area. These are rendered as white digits on a black background.
argv[1] is the starting code point (as a hexadecimal string, with no leading "0x".
argv[2] is the ending code point (as a hexadecimal string, with no leading "0x".

For example:

unihexgen e000 f8ff > pua.hex

This generates the Private Use Area glyph file.

This utility program works in Roman Czyborra's unifont.hex file format, the basis of the GNU Unifont package.
Definition in file unihexgen.c.

5.37.2 Function Documentation

5.37.2.1 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.

```
00161 {
00162
00163    int grid[16]; /* the glyph grid we'll build */
00164
00165    int row;      /* row number in current glyph */
00166    int digitrow; /* row number in current hex digit being rendered */
00167    int rowbits;  /* 1 & 0 bits to draw current glyph row */
00168
00169    int d1, d2, d3, d4; /* four hexadecimal digits of each code point */
00170
00171    d1 = (thiscp » 12) & 0xF;
00172    d2 = (thiscp »  8) & 0xF;
00173    d3 = (thiscp »  4) & 0xF;
00174    d4 = (thiscp      ) & 0xF;
00175
00176    /* top and bottom rows are white */
00177    grid[0] = grid[15] = 0x0000;
00178
00179    /* 14 inner rows are 14-pixel wide black lines, centered */
00180    for (row = 1; row < 15; row++) grid[row] = 0x7FFE;
00181
00182    printf ("%04X:", thiscp);
00183
00184    /*
00185       Render the first row of 2 hexadecimal digits
00186    */
00187    digitrow = 0; /* start at top of first row of digits to render */
00188    for (row = 2; row < 7; row++) {
00189       rowbits = (hexdigit[d1][digitrow] « 9) |
00190                 (hexdigit[d2][digitrow] « 3);
00191       grid[row] ^= rowbits; /* digits appear as white on black background */
00192       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:

5.37.2.2 hexprint6()

void hexprint6 (
    int thiscp)

Generate a bitmap containing a 6-digit Unicode code point.

Takes a 6-digit Unicode code point as an argument and prints a unifont.hex string for it to stdout.

Parameters

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

Definition at line 223 of file unihexgen.c.

```
{

    int grid[16]; /* the glyph grid we'll build */

    int row;      /* row number in current glyph */
    int digitrow; /* row number in current hex digit being rendered */
    int rowbits;  /* 1 & 0 bits to draw current glyph row */

    int d1, d2, d3, d4, d5, d6; /* six hexadecimal digits of each code point */

    d1 = (thiscp » 20) & 0xF;
    d2 = (thiscp » 16) & 0xF;
    d3 = (thiscp » 12) & 0xF;
    d4 = (thiscp »  8) & 0xF;
    d5 = (thiscp »  4) & 0xF;
    d6 = (thiscp      ) & 0xF;

    /* top and bottom rows are white */
    grid[0] = grid[15] = 0x0000;

    /* 14 inner rows are 16-pixel wide black lines, centered */
    for (row = 1; row < 15; row++) grid[row] = 0xFFFF;


    printf ("%06X:", thiscp);

    /*
       Render the first row of 3 hexadecimal digits
    */
    digitrow = 0; /* start at top of first row of digits to render */
    for (row = 2; row < 7; row++) {
       rowbits = (hexdigit[d1][digitrow] « 11) |
                 (hexdigit[d2][digitrow] «  6) |
                 (hexdigit[d3][digitrow] «  1);
       grid[row] ^= rowbits; /* digits appear as white on black background */
       digitrow++;
    }

    /*
       Render the second row of 3 hexadecimal digits
    */
```

```
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 9; row < 14; row++) {
      rowbits = (hexdigit[d4][digitrow] << 11) |
                (hexdigit[d5][digitrow] <<  6) |
                (hexdigit[d6][digitrow] <<  1);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);

   putchar ('\n');

   return;
}
```

Here is the caller graph for this function:

5.37.2.3 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:

5.37.3 Variable Documentation

5.37.3.1 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.

5.38 unihexgen.c

Go to the documentation of this file.

```
00001 /**
00002    @file unihexgen.c
00003
00004    @brief unihexgen - Generate a series of glyphs containing
00005                      hexadecimal code points
00006
00007    @author Paul Hardy
00008
00009    @copyright Copyright (C) 2013 Paul Hardy
00010
00011    This program generates glyphs in Unifont .hex format that contain
00012    four- or six-digit hexadecimal numbers in a 16x16 pixel area.  These
00013    are rendered as white digits on a black background.
00014
00015    argv[1] is the starting code point (as a hexadecimal
00016    string, with no leading "0x".
00017
00018    argv[2] is the ending code point (as a hexadecimal
00019    string, with no leading "0x".
00020
00021       For example:
00022
00023          unihexgen e000 f8ff > pua.hex
00024
00025       This generates the Private Use Area glyph file.
```

```

   This utility program works in Roman Czyborra's unifont.hex file
   format, the basis of the GNU Unifont package.
*/
/*
   This program is released under the terms of the GNU General Public
   License version 2, or (at your option) a later version.

   LICENSE:

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

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

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

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


/**
   @brief Bitmap pattern for each hexadecimal digit.

   hexdigit[][] definition: the bitmap pattern for
   each hexadecimal digit.

   Each digit is drawn as a 4 wide by 5 high bitmap,
   so each digit row is one hexadecimal digit, and
   each entry has 5 rows.

   For example, the entry for digit 1 is:

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

   which corresponds graphically to:

      --#-  ==>  0010  ==>  0x2
      -##-  ==>  0110  ==>  0x6
      --#-  ==>  0010  ==>  0x2
      --#-  ==>  0010  ==>  0x2
      -###  ==>  0111  ==>  0x7

   These row values will then be exclusive-ORed with four one bits
   (binary 1111, or 0xF) to form white digits on a black background.


   Functions hexprint4 and hexprint6 share the hexdigit array;
   they print four-digit and six-digit hexadecimal code points
   in a single glyph, respectively.
*/
char hexdigit[16][5] = {
   {0x6,0x9,0x9,0x9,0x6},  /* 0x0 */
   {0x2,0x6,0x2,0x2,0x7},  /* 0x1 */
   {0xF,0x1,0xF,0x8,0xF},  /* 0x2 */
   {0xE,0x1,0x7,0x1,0xE},  /* 0x3 */
   {0x9,0x9,0xF,0x1,0x1},  /* 0x4 */
   {0xF,0x8,0xF,0x1,0xF},  /* 0x5 */
   {0x6,0x8,0xE,0x9,0x6},  /* 0x6 */ // {0x8,0x8,0xF,0x9,0xF} [alternate square form of 6]
   {0xF,0x1,0x2,0x4,0x4},  /* 0x7 */
   {0x6,0x9,0x6,0x9,0x6},  /* 0x8 */
   {0x6,0x9,0x7,0x1,0x6},  /* 0x9 */ // {0xF,0x9,0xF,0x1,0x1} [alternate square form of 9]
   {0xF,0x9,0xF,0x9,0x9},  /* 0xA */
   {0xE,0x9,0xE,0x9,0xE},  /* 0xB */
   {0x7,0x8,0x8,0x8,0x7},  /* 0xC */
   {0xE,0x9,0x9,0x9,0xE},  /* 0xD */
   {0xF,0x8,0xE,0x8,0xF},  /* 0xE */
   {0xF,0x8,0xE,0x8,0x8}   /* 0xF */
};


/**
   @brief The main function.

```

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

  int startcp, endcp, thiscp;
  void hexprint4(int); /* function to print one 4-digit unifont.hex code point */
  void hexprint6(int); /* function to print one 6-digit unifont.hex code point */

  if (argc != 3) {
     fprintf (stderr,"\n%s - generate unifont.hex code points as\n", argv[0]);
     fprintf (stderr,"four-digit hexadecimal numbers in a 2 by 2 grid,\n");
     fprintf (stderr,"or six-digit hexadecimal numbers in a 3 by 2 grid.\n");
     fprintf (stderr,"Syntax:\n\n");
     fprintf (stderr,"     %s first_code_point last_code_point > glyphs.hex\n\n", argv[0]);
     fprintf (stderr,"Example (to generate glyphs for the Private Use Area):\n\n");
     fprintf (stderr,"     %s e000 f8ff > pua.hex\n\n", argv[0]);
     exit (EXIT_FAILURE);
  }

  sscanf (argv[1], "%x", &startcp);
  sscanf (argv[2], "%x", &endcp);

  startcp &= 0xFFFFFF; /* limit to 6 hex digits */
  endcp   &= 0xFFFFFF; /* limit to 6 hex digits */

  /*
     For each code point in the desired range, generate a glyph.
  */
  for (thiscp = startcp; thiscp <= endcp; thiscp++) {
     if (thiscp <= 0xFFFF) {
        hexprint4 (thiscp); /* print digits 2/line, 2 lines */
     }
     else {
        hexprint6 (thiscp); /* print digits 3/line, 2 lines */
     }
  }
  exit (EXIT_SUCCESS);
}


/**
  @brief Generate a bitmap containing a 4-digit Unicode code point.

  Takes a 4-digit Unicode code point as an argument
  and prints a unifont.hex string for it to stdout.

  @param[in] thiscp The current code point for which to generate a glyph.
*/
void
hexprint4 (int thiscp)
{

  int grid[16]; /* the glyph grid we'll build */

  int row;      /* row number in current glyph */
  int digitrow; /* row number in current hex digit being rendered */
  int rowbits;  /* 1 & 0 bits to draw current glyph row */

  int d1, d2, d3, d4; /* four hexadecimal digits of each code point */

  d1 = (thiscp >> 12) & 0xF;
  d2 = (thiscp >>  8) & 0xF;
  d3 = (thiscp >>  4) & 0xF;
  d4 = (thiscp      ) & 0xF;

  /* top and bottom rows are white */
  grid[0] = grid[15] = 0x0000;

  /* 14 inner rows are 14-pixel wide black lines, centered */
  for (row = 1; row < 15; row++) grid[row] = 0x7FFE;

  printf ("%04X:", thiscp);

  /*
     Render the first row of 2 hexadecimal digits
  */
  digitrow = 0; /* start at top of first row of digits to render */
```

```
   for (row = 2; row < 7; row++) {
      rowbits = (hexdigit[d1][digitrow] « 9) |
                (hexdigit[d2][digitrow] « 3);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   /*
      Render the second row of 2 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 9; row < 14; row++) {
      rowbits = (hexdigit[d3][digitrow] « 9) |
                (hexdigit[d4][digitrow] « 3);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);

   putchar ('\n');

   return;
}


/**
   @brief Generate a bitmap containing a 6-digit Unicode code point.

   Takes a 6-digit Unicode code point as an argument
   and prints a unifont.hex string for it to stdout.

   @param[in] thiscp The current code point for which to generate a glyph.
*/
void
hexprint6 (int thiscp)
{

   int grid[16]; /* the glyph grid we'll build */

   int row;      /* row number in current glyph */
   int digitrow; /* row number in current hex digit being rendered */
   int rowbits;  /* 1 & 0 bits to draw current glyph row */

   int d1, d2, d3, d4, d5, d6; /* six hexadecimal digits of each code point */

   d1 = (thiscp » 20) & 0xF;
   d2 = (thiscp » 16) & 0xF;
   d3 = (thiscp » 12) & 0xF;
   d4 = (thiscp »  8) & 0xF;
   d5 = (thiscp »  4) & 0xF;
   d6 = (thiscp      ) & 0xF;

   /* top and bottom rows are white */
   grid[0] = grid[15] = 0x0000;

   /* 14 inner rows are 16-pixel wide black lines, centered */
   for (row = 1; row < 15; row++) grid[row] = 0xFFFF;


   printf ("%06X:", thiscp);

   /*
      Render the first row of 3 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 2; row < 7; row++) {
      rowbits = (hexdigit[d1][digitrow] « 11) |
                (hexdigit[d2][digitrow] «  6) |
                (hexdigit[d3][digitrow] «  1);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   /*
      Render the second row of 3 hexadecimal digits
   */
   digitrow = 0; /* start at top of first row of digits to render */
   for (row = 9; row < 14; row++) {
      rowbits = (hexdigit[d4][digitrow] « 11) |
                (hexdigit[d5][digitrow] «  6) |
```

```
            (hexdigit[d6][digitrow] << 1);
      grid[row] ^= rowbits; /* digits appear as white on black background */
      digitrow++;
   }

   for (row = 0; row < 16; row++) printf ("%04X", grid[row] & 0xFFFF);

   putchar ('\n');

   return;
}

```

5.39 unihexpose.c

```
/**
   @file: unihetranspose.c

   @brief: Transpose Unifont glyph bitmaps.

   This program takes Unifont .hex format glyphs and converts those
   glyphs so that each byte (two hexadecimal digits in the .hex file)
   represents a column of 8 rows.  This simplifies use with graphics
   display controllers that write lines consisting of 8 rows at a time
   to a display.

   The bytes are ordered as first all the columns for the glyph in
   the first 8 rows, then all the columns in the next 8 rows, with
   columns ordered from left to right.

   This file must be linked with functions in unifont-support.c.

   @author Paul Hardy

   @copyright Copyright © 2023 Paul Hardy
*/
/*
   LICENSE:

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

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

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

#define MAXWIDTH 128

int
main (int argc, char *argv[]) {
   unsigned codept;      /* Unicode code point for glyph */
   char instring [MAXWIDTH];    /* input Unifont hex string */
   char outstring [MAXWIDTH];  /* output Unfont hex string */
   int  width;           /* width of current glyph */
   unsigned char glyph      [16][2];
   unsigned char glyphbits [16][16];  /* One glyphbits row, for transposing */
   unsigned char transpose [2][16];  /* Transponsed glyphbits bitmap    */

   void print_syntax ();

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

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

```

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

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

   if (argc > 1) {
      print_syntax ();
      exit (EXIT_FAILURE);
   }

   while (fgets (instring, MAXWIDTH, stdin) != NULL) {
      parse_hex (instring, &width, &codept, glyph);

      glyph2bits (width, glyph, glyphbits);

      hexpose (width, glyphbits, transpose);

      xglyph2string (width, codept, transpose, outstring);

      fprintf (stdout, "%s\n", outstring);
   }

   exit (EXIT_SUCCESS);
}


void
print_syntax () {

   fprintf (stderr, "\nSyntax: unihexpose < input.hex > output.hex\n\n");

   return;
}

```

5.40 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.

5.40.1 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

Definition in file unijohab2html.c.

5.40.2 Macro Definition Documentation

5.40.2.1 BLACK

#define BLACK 0x000000

Color code for HTML black.

Definition at line 62 of file unijohab2html.c.

5.40.2.2 BLUE

#define BLUE 0x0000CC

Color code for slightly unsaturated HTML blue.

Definition at line 61 of file unijohab2html.c.

5.40.2.3 GREEN

#define GREEN 0x00CC00

Color code for slightly unsaturated HTML green.

Definition at line 60 of file unijohab2html.c.

5.40.2.4 MAXFILENAME

#define MAXFILENAME 1024

Definition at line 52 of file unijohab2html.c.

5.40.2.5 RED

#define RED 0xCC0000

Color code for slightly unsaturated HTML red.

Definition at line 59 of file unijohab2html.c.

5.40.2.6 START_JUNG

#define START_JUNG 0

Vowel index of first vowel with which to begin.

Definition at line 54 of file unijohab2html.c.

5.40.2.7 WHITE

#define WHITE 0xFFFFFF

Color code for HTML white.

Definition at line 63 of file unijohab2html.c.

5.40.3 Function Documentation

5.40.3.1 main()

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

The main function.

Definition at line 70 of file unijohab2html.c.

```
00070                                {
00071    int i, j;  /* loop variables */
00072    unsigned codept;
00073    unsigned max_codept;
00074    int      modern_only = 0;  /* To just use modern Hangul */
00075    int      group, consonant1, vowel, consonant2;
00076    int      vowel_variation;
00077    unsigned glyph[MAX_GLYPHS][16];
00078    unsigned tmp_glyph [16];  /* To build one combined glyph at a time. */
00079    unsigned mask;
00080    unsigned overlapped;        /* To find overlaps */
00081    int      ancient_choseong; /* Flag when within ancient choseong range. */
00082 
00083    /*
00084       16x16 pixel grid for each Choseong group, for:
00085 
00086          Group 0 to Group 5 with no Jongseong
00087          Group 3 to Group 5 with Jongseong except Nieun
00088          Group 3 to Group 5 with Jongseong Nieun
00089 
00090       12 grids total.
00091 
00092       Each grid cell will hold a 32-bit HTML RGB color.
00093    */
00094    unsigned grid[12][16][16];
00095 
00096    /*
00097       Matrices to detect and report overlaps.  Identify vowel
00098       variations where an overlap occurred.  For most vowel
```

```
   variations, there will be no overlap.  Then go through
   choseong, and then jongseong to find the overlapping
   combinations.  This saves storage space as an alternative
   to storing large 2- or 3-dimensional overlap matrices.
*/
// jungcho: Jungseong overlap with Choseong
unsigned jungcho [TOTAL_JUNG * JUNG_VARIATIONS];
// jongjung: Jongseong overlap with Jungseong -- for future expansion
// unsigned jongjung [TOTAL_JUNG * JUNG_VARIATIONS];

int glyphs_overlap;     /* If glyph pair being considered overlap. */
int cho_overlaps  = 0;  /* Number of choseong+vowel overlaps.      */
// int jongjung_overlaps = 0;  /* Number of vowel+jongseong overlaps. */

int  inindex = 0;
int  outindex = 0;
FILE *infp, *outfp;     /* Input and output file pointers. */

void     parse_args (int argc, char *argv[], int *inindex, int *outindex,
                     int *modern_only);
int      cho_variation      (int cho, int jung, int jong);
unsigned hangul_read_base16 (FILE *infp, unsigned glyph[][16]);
int  glyph_overlap (unsigned *glyph1, unsigned *glyph2);

void combine_glyphs (unsigned *glyph1, unsigned *glyph2,
                     unsigned *combined_glyph);
void print_glyph_txt (FILE *fp, unsigned codept, unsigned *this_glyph);


/*
   Parse command line arguments to open input & output files, if given.
*/
if (argc > 1) {
   parse_args (argc, argv, &inindex, &outindex, &modern_only);
}

if (inindex == 0) {
   infp = stdin;
}
else {
   infp = fopen (argv[inindex], "r");
   if (infp == NULL) {
      fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",
               argv[inindex]);
      exit (EXIT_FAILURE);
   }
}
if (outindex == 0) {
   outfp = stdout;
}
else {
   outfp = fopen (argv[outindex], "w");
   if (outfp == NULL) {
      fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
               argv[outindex]);
      exit (EXIT_FAILURE);
   }
}

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

/*
   Initialize overlap matrices to all zeroes.
*/
for (i = 0; i < TOTAL_JUNG * JUNG_VARIATIONS; i++) {
   jungcho  [i] = 0;
}
// jongjung is reserved for expansion.
// for (i = 0; i < TOTAL_JONG * JONG_VARIATIONS; i++) {
//    jongjung [i] = 0;
// }

/*
   Read Hangul base glyph file.
*/
max_codept = hangul_read_base16 (infp, glyph);
```

```
if (max_codept > 0x8FF) {
   fprintf (stderr, ”\nWARNING: Hangul glyph range exceeds PUA space.\n\n”);
}

/*
   If only examining modern Hangul, fill the ancient glyphs
   with blanks to guarantee they won't overlap.  This is
   not as efficient as ending loops sooner, but is easier
   to verify for correctness.
*/
if (modern_only) {
   for (i = 0x0073; i < JUNG_HEX; i++) {
      for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
   }
   for (i = 0x027A; i < JONG_HEX; i++) {
      for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
   }
   for (i = 0x032B; i < 0x0400; i++) {
      for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
   }
}

/*
   Initialize grids to all black (no color) for each of
   the 12 Choseong groups.
*/
for (group = 0; group < 12; group++) {
   for (i = 0; i < 16; i++) {
      for (j = 0; j < 16; j++) {
         grid[group][i][j] = BLACK;  /* No color at first */
      }
   }
}

/*
   Superimpose all Choseong glyphs according to group.
   Each grid spot with choseong will be blue.
*/
for (group = 0; group < 6; group++) {
   for (consonant1 = CHO_HEX + group;
        consonant1 < CHO_HEX +
                     CHO_VARIATIONS * TOTAL_CHO;
        consonant1 += CHO_VARIATIONS) {
      for (i = 0; i < 16; i++) { /* For each glyph row */
         mask = 0x8000;
         for (j = 0; j < 16; j++) {
            if (glyph[consonant1][i] & mask) grid[group][i][j] |= BLUE;
            mask »= 1;  /* Get next bit in glyph row */
         }
      }
   }
}

/*
   Fill with Choseong (initial consonant) to prepare
   for groups 3-5 with jongseong except niuen (group+3),
   then for groups 3-5 with jongseong nieun (group+6).
*/
for (group = 3; group < 6; group++) {
   for (i = 0; i < 16; i++) {
      for (j = 0; j < 16; j++) {
         grid[group + 6][i][j] = grid[group + 3][i][j]
                               = grid[group][i][j];
      }
   }
}

/*
   For each Jungseong, superimpose first variation on
   appropriate Choseong group for grids 0 to 5.
*/
for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
   group = cho_variation (-1, vowel, -1);
   glyphs_overlap = 0;  /* Assume the 2 glyphs do not overlap. */

   for (i = 0; i < 16; i++) { /* For each glyph row */
      mask = 0x8000;
      for (j = 0; j < 16; j++) {
         if (glyph[JUNG_HEX + JUNG_VARIATIONS * vowel][i] & mask) {
            /*
               If there was already blue in this grid cell,
```

```
                  mark this vowel variation as having overlap
                  with choseong (initial consonant) letter(s).
               */
               if (grid[group][i][j] & BLUE) glyphs_overlap = 1;

               /* Add green to grid cell color. */
               grid[group][i][j] |= GREEN;
            }
            mask »= 1;  /* Mask for next bit in glyph row */
         }  /* for j */
      }  /* for i */
      if (glyphs_overlap) {
         jungcho [JUNG_VARIATIONS * vowel] = 1;
         cho_overlaps++;
      }
   }  /* for each vowel */

   /*
      For each Jungseong, superimpose second variation on
      appropriate Choseong group for grids 6 to 8.
   */
   for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
      /*
         The second vowel variation is for combination with
         a final consonant (Jongseong), with initial consonant
         (Choseong) variations (or ”groups”) 3 to 5.  Thus,
         if the vowel type returns an initial Choseong group
         of 0 to 2, add 3 to it.
      */
      group = cho_variation (-1, vowel, -1);
      /*
         Groups 0 to 2 don’t use second vowel variation,
         so increment if group is below 2.
      */
      if (group < 3) group += 3;
      glyphs_overlap = 0;  /* Assume the 2 glyphs do not overlap. */

      for (i = 0; i < 16; i++) { /* For each glyph row */
         mask = 0x8000;  /* Start mask at leftmost glyph bit */
         for (j = 0; j < 16; j++) {  /* For each column in this row */
            /* ”+ 1” is to get each vowel’s second variation */
            if (glyph [JUNG_HEX +
                        JUNG_VARIATIONS * vowel + 1][i] & mask) {
               /* If this cell has blue already, mark as overlapped. */
               if (grid [group + 3][i][j] & BLUE) glyphs_overlap = 1;

               /* Superimpose green on current cell color. */
               grid [group + 3][i][j] |= GREEN;
            }
            mask »= 1;  /* Get next bit in glyph row */
         }  /* for j */
      }  /* for i */
      if (glyphs_overlap) {
         jungcho [JUNG_VARIATIONS * vowel + 1] = 1;
         cho_overlaps++;
      }
   }  /* for each vowel */

   /*
      For each Jungseong, superimpose third variation on
      appropriate Choseong group for grids 9 to 11 for
      final consonant (Jongseong) of Nieun.
   */
   for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
      group = cho_variation (-1, vowel, -1);
      if (group < 3) group += 3;
      glyphs_overlap = 0;  /* Assume the 2 glyphs do not overlap. */

      for (i = 0; i < 16; i++) { /* For each glyph row */
         mask = 0x8000;
         for (j = 0; j < 16; j++) {
            if (glyph[JUNG_HEX +
                       JUNG_VARIATIONS * vowel + 2][i] & mask) {
               /* If this cell has blue already, mark as overlapped. */
               if (grid[group + 6][i][j] & BLUE) glyphs_overlap = 1;

               grid[group + 6][i][j] |= GREEN;
            }
            mask »= 1;  /* Get next bit in glyph row */
         }  /* for j */
      }  /* for i */
```

```
      if (glyphs_overlap) {
         jungcho [JUNG_VARIATIONS * vowel + 2] = 1;
         cho_overlaps++;
      }
   }  /* for each vowel */


   /*
      Superimpose all final consonants except nieun for grids 6 to 8.
   */
   for (consonant2 = 0; consonant2 < TOTAL_JONG; consonant2++) {
      /*
         Skip over Jongseong Nieun, because it is covered in
         grids 9 to 11 after this loop.
      */
      if (consonant2 == 3) consonant2++;

      glyphs_overlap = 0;  /* Assume the 2 glyphs do not overlap. */
      for (i = 0; i < 16; i++) { /* For each glyph row */
         mask = 0x8000;
         for (j = 0; j < 16; j++) {
            if (glyph [JONG_HEX +
                       JONG_VARIATIONS * consonant2][i] & mask) {
               if (grid[6][i][j] & GREEN ||
                   grid[7][i][j] & GREEN ||
                   grid[8][i][j] & GREEN) glyphs_overlap = 1;

               grid[6][i][j] |= RED;
               grid[7][i][j] |= RED;
               grid[8][i][j] |= RED;
            }
            mask »= 1;  /* Get next bit in glyph row */
         }  /* for j */
      }  /* for i */
      // jongjung is for expansion
      // if (glyphs_overlap) {
      //    jongjung [JONG_VARIATIONS * consonant2] = 1;
      //    jongjung_overlaps++;
      // }
   }  /* for each final consonant except nieun */

   /*
      Superimpose final consonant 3 (Jongseong Nieun) on
      groups 9 to 11.
   */
   codept = JONG_HEX + 3 * JONG_VARIATIONS;

   for (i = 0; i < 16; i++) { /* For each glyph row */
      mask = 0x8000;
      for (j = 0; j < 16; j++) {
         if (glyph[codept][i] & mask) {
            grid[ 9][i][j] |= RED;
            grid[10][i][j] |= RED;
            grid[11][i][j] |= RED;
         }
         mask »= 1;  /* Get next bit in glyph row */
      }
   }


   /*
      Turn the black (uncolored) cells into white for better
      visibility of grid when displayed.
   */
   for (group = 0; group < 12; group++) {
      for (i = 0; i < 16; i++) {
         for (j = 0; j < 16; j++) {
            if (grid[group][i][j] == BLACK) grid[group][i][j] = WHITE;
         }
      }
   }


   /*
      Generate HTML output.
   */
   fprintf (outfp, "<html>\n");
   fprintf (outfp, "<head>\n");
   fprintf (outfp, "  <title>Johab 6/3/1 Overlaps</title>\n");
   fprintf (outfp, "</head>\n");
   fprintf (outfp, "<body bgcolor=\"#FFFFCC\">\n");
```

```
fprintf (outfp, "<center>\n");
fprintf (outfp, "  <h1>Unifont Hangul Jamo Syllable Components</h1>\n");
fprintf (outfp, "  <h2>Johab 6/3/1 Overlap</h2><br><br>\n");

/* Print the color code key for the table. */
fprintf (outfp, "  <table border=\"1\" cellpadding=\"10\">\n");
fprintf (outfp, "    <tr><th colspan=\"2\" align=\"center\" bgcolor=\"#FFCC80\">");
fprintf (outfp, "<font size=\"+1\">Key</font></th></tr>\n");
fprintf (outfp, "    <tr>\n");
fprintf (outfp, "      <th align=\"center\" bgcolor=\"#FFFF80\">Color</th>\n");
fprintf (outfp, "      <th align=\"center\" bgcolor=\"#FFFF80\">Letter(s)</th>\n");
fprintf (outfp, "    </tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", BLUE);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Choseong (Initial Consonant)</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", GREEN);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Jungseong (Medial Vowel/Diphthong)</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", RED);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Jongseong (Final Consonant)</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", BLUE | GREEN);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Choseong + Jungseong Overlap</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", GREEN | RED);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Jungseong + Jongseong Overlap</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", RED | BLUE);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Choseong + Jongseong Overlap</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", RED | GREEN | BLUE);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Choseong + Jungseong + Jongseong Overlap</td></tr>\n");

fprintf (outfp, "  </table>\n");
fprintf (outfp, "  <br><br>\n");


for (group = 0; group < 12; group++) {
   /* Arrange tables 3 across, 3 down. */
   if ((group % 3) == 0) {
      fprintf (outfp, "  <table border=\"0\" cellpadding=\"10\">\n");
      fprintf (outfp, "    <tr>\n");
   }

   fprintf (outfp, "      <td>\n");
   fprintf (outfp, "        <table border=\"3\" cellpadding=\"2\">\n");
   fprintf (outfp, "          <tr><th colspan=\"16\" bgcolor=\"#FFFF80\">");
   fprintf (outfp, "Choseong Group %d, %s %s</th></tr>\n",
            group < 6 ? group : (group > 8 ? group - 6 : group - 3),
            group < 6 ? (group < 3 ? "No" : "Without") : "With",
            group < 9 ? "Jongseong" : "Nieun");

   for (i = 0; i < 16; i++) {
      fprintf (outfp, "          <tr>\n");
      for (j = 0; j < 16; j++) {
         fprintf (outfp, "            <td bgcolor=\"#%06X\">",
                  grid[group][i][j]);
         fprintf (outfp, "    </td>\n");
      }
      fprintf (outfp, "          </tr>\n");
   }

   fprintf (outfp, "            </td>\n");
   fprintf (outfp, "          </tr>\n");
   fprintf (outfp, "        </table>\n");
   fprintf (outfp, "      </td>\n");

   if ((group % 3) == 2) {
      fprintf (outfp, "    </tr>\n");
      fprintf (outfp, "  </table>\n  </br>\n");
   }
}
```

```
/* Wrap up HTML table output. */
fprintf (outfp, "</center>\n");

/*
   Print overlapping initial consonant + vowel combinations.
*/
fprintf (outfp, "<h2>%d Vowel Overlaps with Initial Consonants Found</h2>",
         cho_overlaps);
fprintf (outfp, "<font size=\"+1\"><pre>\n");

for (i = JUNG_HEX;
     i < JUNG_HEX + TOTAL_JUNG * JUNG_VARIATIONS;
     i++) {
   /*
      If this vowel variation (Jungseong) had overlaps
      with one or more initial consonants (Choseong),
      find and print them.
   */
   if (jungcho [i - JUNG_HEX]) {
      ancient_choseong = 0;  /* Not within ancient choseong range yet. */
      fprintf (outfp, "<font color=\"#0000FF\"><b>");
      if (i >= JUNG_ANCIENT_HEX) {
         if (i >= JUNG_EXTB_HEX) fprintf (outfp, "Extended-B ");
         fprintf (outfp, "Ancient ");
      }
      fprintf (outfp, "Vowel at 0x%04X and&hellip;</b>", i + PUA_START);
      fprintf (outfp, "</font>\n\n");

      /*
         Get current vowel number, 0 to (TOTAL_JUNG - 1), and
         current vowel variation, 0 or 1, or 2 for final nieun.
      */
      vowel = (i - JUNG_HEX) / JUNG_VARIATIONS;
      vowel_variation = (i - JUNG_HEX) % JUNG_VARIATIONS;

      /* Get first Choseong group for this vowel, 0 to 5. */
      group = cho_variation (-1, vowel, -1);

      /*
         If this vowel variation is used with a final consonant
         (Jongseong) and the default initial consonant (Choseong)
         group for this vowel is < 3, add 3 to current Chosenong
         group.
      */
      if (vowel_variation > 0 && group < 3) group += 3;

      for (consonant1 = 0; consonant1 < TOTAL_CHO; consonant1++) {
         overlapped = glyph_overlap (glyph [i],
                          glyph [consonant1 * CHO_VARIATIONS
                               + CHO_HEX + group]);

         /*
            If we just entered ancient choseong range, flag it.
         */
         if (overlapped && consonant1 >= 19 && ancient_choseong == 0) {
            fprintf (outfp, "<font color=\"#0000FF\"><b>");
            fprintf (outfp, "&hellip;Ancient Choseong&hellip;</b></font>\n");
            ancient_choseong = 1;
         }
         /*
            If overlapping choseong found, print combined glyph.
         */
         if (overlapped != 0) {

            combine_glyphs (glyph [i],
                            glyph [consonant1 * CHO_VARIATIONS
                                 + CHO_HEX + group],
                            tmp_glyph);

            print_glyph_txt (outfp,
                             PUA_START +
                             consonant1 * CHO_VARIATIONS +
                             CHO_HEX + group,
                             tmp_glyph);

         }  /* If overlapping pixels found. */
      }  /* For each initial consonant (Choseong) */
   }  /* Find the initial consonant that overlapped this vowel variation. */
}  /* For each variation of each vowel (Jungseong) */
```

```
00585 fputc ('\n', outfp);
00586
00587 fprintf (outfp, "</pre></font>\n");
00588 fprintf (outfp, "</body>\n");
00589 fprintf (outfp, "</html>\n");
00590
00591 fclose (infp);
00592 fclose (outfp);
00593
00594
00595 exit (EXIT_SUCCESS);
00596 }
```

Here is the call graph for this function:

5.40.3.2 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.

```
00609                                 {
00610 int arg_count;  /* Current index into argv[]. */
00611
00612 int strncmp (const char *s1, const char *s2, size_t n);
00613
00614
00615 arg_count = 1;
00616
00617 while (arg_count < argc) {
00618    /* If input file is specified, open it for read access. */
00619    if (strncmp (argv [arg_count], "-i", 2) == 0) {
00620       arg_count++;
00621       if (arg_count < argc) {
00622          *inindex = arg_count;
00623       }
00624    }
00625    /* If only modern Hangul is desired, set modern_only flag. */
00626    else if (strncmp (argv [arg_count], "-m", 2) == 0 ||
00627             strncmp (argv [arg_count], "--modern", 8) == 0) {
00628       *modern_only = 1;
00629    }
00630    /* If output file is specified, open it for write access. */
00631    else if (strncmp (argv [arg_count], "-o", 2) == 0) {
00632       arg_count++;
00633       if (arg_count < argc) {
00634          *outindex = arg_count;
00635       }
00636    }
00637    /* If help is requested, print help message and exit. */
00638    else if (strncmp (argv [arg_count], "-h",     2) == 0 ||
00639             strncmp (argv [arg_count], "--help", 6) == 0) {
00640       printf ("\nunijohab2html [options]\n\n");
00641       printf ("     Generates an HTML page of overlapping Hangul letters from an input\n");
00642       printf ("     Unifont .hex file encoded in Johab 6/3/1 format.\n\n");
00643
00644       printf ("     Option         Parameters    Function\n");
00645       printf ("     ------         ----------    --------\n");
00646       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;
}
```

Here is the caller graph for this function:

5.41 unijohab2html.c

Go to the documentation of this file.

```
/**
   @file unijohab2html.c

   @brief 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
*/
/*
   LICENSE:

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

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

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

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

#define MAXFILENAME 1024

#define START_JUNG 0    ///< Vowel index of first vowel with which to begin.
// #define START_JUNG 21  /* Use this #define for just ancient vowels */


/* (Red, Green, Blue) HTML color coordinates. */
#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.


/**
   @brief The main function.
*/
int
main (int argc, char *argv[]) {
   int i, j;  /* loop variables */
   unsigned codept;
   unsigned max_codept;
   int      modern_only = 0;  /* To just use modern Hangul */
   int      group, consonant1, vowel, consonant2;
   int      vowel_variation;
   unsigned glyph[MAX_GLYPHS][16];
   unsigned tmp_glyph [16];  /* To build one combined glyph at a time. */
   unsigned mask;
   unsigned overlapped;       /* To find overlaps */
   int      ancient_choseong; /* Flag when within ancient choseong range. */

   /*
      16x16 pixel grid for each Choseong group, for:

         Group 0 to Group 5 with no Jongseong
         Group 3 to Group 5 with Jongseong except Nieun
         Group 3 to Group 5 with Jongseong Nieun

      12 grids total.

      Each grid cell will hold a 32-bit HTML RGB color.
   */
   unsigned grid[12][16][16];

   /*
      Matrices to detect and report overlaps.  Identify vowel
      variations where an overlap occurred.  For most vowel
      variations, there will be no overlap.  Then go through
      choseong, and then jongseong to find the overlapping
      combinations.  This saves storage space as an alternative
      to storing large 2- or 3-dimensional overlap matrices.
   */
   // jungcho: Jungseong overlap with Choseong
   unsigned jungcho [TOTAL_JUNG * JUNG_VARIATIONS];
   // jongjung: Jongseong overlap with Jungseong -- for future expansion
   // unsigned jongjung [TOTAL_JUNG * JUNG_VARIATIONS];

   int glyphs_overlap;     /* If glyph pair being considered overlap. */
   int cho_overlaps  = 0;  /* Number of choseong+vowel overlaps.      */
   // int jongjung_overlaps = 0;  /* Number of vowel+jongseong overlaps. */

   int  inindex = 0;
   int  outindex = 0;
   FILE *infp, *outfp;     /* Input and output file pointers. */

   void     parse_args (int argc, char *argv[], int *inindex, int *outindex,
                        int *modern_only);
   int      cho_variation      (int cho, int jung, int jong);
   unsigned hangul_read_base16 (FILE *infp, unsigned glyph[][16]);
   int  glyph_overlap (unsigned *glyph1, unsigned *glyph2);

   void combine_glyphs (unsigned *glyph1, unsigned *glyph2,
                        unsigned *combined_glyph);
   void print_glyph_txt (FILE *fp, unsigned codept, unsigned *this_glyph);


   /*
      Parse command line arguments to open input & output files, if given.
   */
   if (argc > 1) {
      parse_args (argc, argv, &inindex, &outindex, &modern_only);
   }

   if (inindex == 0) {
      infp = stdin;
   }
   else {
      infp = fopen (argv[inindex], "r");
      if (infp == NULL) {
```

```
            fprintf (stderr, "\n*** ERROR: Cannot open %s for input.\n\n",
                     argv[inindex]);
            exit (EXIT_FAILURE);
         }
      }
      if (outindex == 0) {
         outfp = stdout;
      }
      else {
         outfp = fopen (argv[outindex], "w");
         if (outfp == NULL) {
            fprintf (stderr, "\n*** ERROR: Cannot open %s for output.\n\n",
                     argv[outindex]);
            exit (EXIT_FAILURE);
         }
      }

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

      /*
         Initialize overlap matrices to all zeroes.
      */
      for (i = 0; i < TOTAL_JUNG * JUNG_VARIATIONS; i++) {
         jungcho  [i] = 0;
      }
      // jongjung is reserved for expansion.
      // for (i = 0; i < TOTAL_JONG * JONG_VARIATIONS; i++) {
      //    jongjung [i] = 0;
      // }

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

      /*
         If only examining modern Hangul, fill the ancient glyphs
         with blanks to guarantee they won't overlap.  This is
         not as efficient as ending loops sooner, but is easier
         to verify for correctness.
      */
      if (modern_only) {
         for (i = 0x0073; i < JUNG_HEX; i++) {
            for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
         }
         for (i = 0x027A; i < JONG_HEX; i++) {
            for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
         }
         for (i = 0x032B; i < 0x0400; i++) {
            for (j = 0; j < 16; j++) glyph[i][j] = 0x0000;
         }
      }

      /*
         Initialize grids to all black (no color) for each of
         the 12 Choseong groups.
      */
      for (group = 0; group < 12; group++) {
         for (i = 0; i < 16; i++) {
            for (j = 0; j < 16; j++) {
               grid[group][i][j] = BLACK;  /* No color at first */
            }
         }
      }

      /*
         Superimpose all Choseong glyphs according to group.
         Each grid spot with choseong will be blue.
      */
      for (group = 0; group < 6; group++) {
         for (consonant1 = CHO_HEX + group;
              consonant1 < CHO_HEX +
                           CHO_VARIATIONS * TOTAL_CHO;
```

```
         consonant1 += CHO_VARIATIONS) {
      for (i = 0; i < 16; i++) { /* For each glyph row */
         mask = 0x8000;
         for (j = 0; j < 16; j++) {
            if (glyph[consonant1][i] & mask) grid[group][i][j] |= BLUE;
            mask »= 1;  /* Get next bit in glyph row */
         }
      }
   }
}

/*
   Fill with Choseong (initial consonant) to prepare
   for groups 3-5 with jongseong except niuen (group+3),
   then for groups 3-5 with jongseong nieun (group+6).
*/
for (group = 3; group < 6; group++) {
   for (i = 0; i < 16; i++) {
      for (j = 0; j < 16; j++) {
         grid[group + 6][i][j] = grid[group + 3][i][j]
                               = grid[group][i][j];
      }
   }
}

/*
   For each Jungseong, superimpose first variation on
   appropriate Choseong group for grids 0 to 5.
*/
for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
   group = cho_variation (-1, vowel, -1);
   glyphs_overlap = 0;  /* Assume the 2 glyphs do not overlap. */

   for (i = 0; i < 16; i++) { /* For each glyph row */
      mask = 0x8000;
      for (j = 0; j < 16; j++) {
         if (glyph[JUNG_HEX + JUNG_VARIATIONS * vowel][i] & mask) {
            /*
               If there was already blue in this grid cell,
               mark this vowel variation as having overlap
               with choseong (initial consonant) letter(s).
            */
            if (grid[group][i][j] & BLUE) glyphs_overlap = 1;

            /* Add green to grid cell color. */
            grid[group][i][j] |= GREEN;
         }
         mask »= 1;  /* Mask for next bit in glyph row */
      }  /* for j */
   }  /* for i */
   if (glyphs_overlap) {
      jungcho [JUNG_VARIATIONS * vowel] = 1;
      cho_overlaps++;
   }
}  /* for each vowel */

/*
   For each Jungseong, superimpose second variation on
   appropriate Choseong group for grids 6 to 8.
*/
for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
   /*
      The second vowel variation is for combination with
      a final consonant (Jongseong), with initial consonant
      (Choseong) variations (or ”groups”) 3 to 5.  Thus,
      if the vowel type returns an initial Choseong group
      of 0 to 2, add 3 to it.
   */
   group = cho_variation (-1, vowel, -1);
   /*
      Groups 0 to 2 don’t use second vowel variation,
      so increment if group is below 2.
   */
   if (group < 3) group += 3;
   glyphs_overlap = 0;  /* Assume the 2 glyphs do not overlap. */

   for (i = 0; i < 16; i++) { /* For each glyph row */
      mask = 0x8000;  /* Start mask at leftmost glyph bit */
      for (j = 0; j < 16; j++) {  /* For each column in this row */
         /* ”+ 1” is to get each vowel’s second variation */
         if (glyph [JUNG_HEX +
```

```
                JUNG_VARIATIONS * vowel + 1][i] & mask) {
            /* If this cell has blue already, mark as overlapped. */
            if (grid [group + 3][i][j] & BLUE) glyphs_overlap = 1;

            /* Superimpose green on current cell color. */
            grid [group + 3][i][j] |= GREEN;
         }
         mask »= 1;  /* Get next bit in glyph row */
      }  /* for j */
   }  /* for i */
   if (glyphs_overlap) {
      jungcho [JUNG_VARIATIONS * vowel + 1] = 1;
      cho_overlaps++;
   }
}  /* for each vowel */

/*
   For each Jungseong, superimpose third variation on
   appropriate Choseong group for grids 9 to 11 for
   final consonant (Jongseong) of Nieun.
*/
for (vowel = START_JUNG; vowel < TOTAL_JUNG; vowel++) {
   group = cho_variation (-1, vowel, -1);
   if (group < 3) group += 3;
   glyphs_overlap = 0;  /* Assume the 2 glyphs do not overlap. */

   for (i = 0; i < 16; i++) { /* For each glyph row */
      mask = 0x8000;
      for (j = 0; j < 16; j++) {
         if (glyph[JUNG_HEX +
               JUNG_VARIATIONS * vowel + 2][i] & mask) {
            /* If this cell has blue already, mark as overlapped. */
            if (grid[group + 6][i][j] & BLUE) glyphs_overlap = 1;

            grid[group + 6][i][j] |= GREEN;
         }
         mask »= 1;  /* Get next bit in glyph row */
      }  /* for j */
   }  /* for i */
   if (glyphs_overlap) {
      jungcho [JUNG_VARIATIONS * vowel + 2] = 1;
      cho_overlaps++;
   }
}  /* for each vowel */


/*
   Superimpose all final consonants except nieun for grids 6 to 8.
*/
for (consonant2 = 0; consonant2 < TOTAL_JONG; consonant2++) {
   /*
      Skip over Jongseong Nieun, because it is covered in
      grids 9 to 11 after this loop.
   */
   if (consonant2 == 3) consonant2++;

   glyphs_overlap = 0;  /* Assume the 2 glyphs do not overlap. */
   for (i = 0; i < 16; i++) { /* For each glyph row */
      mask = 0x8000;
      for (j = 0; j < 16; j++) {
         if (glyph [JONG_HEX +
                JONG_VARIATIONS * consonant2][i] & mask) {
            if (grid[6][i][j] & GREEN ||
                grid[7][i][j] & GREEN ||
                grid[8][i][j] & GREEN) glyphs_overlap = 1;

            grid[6][i][j] |= RED;
            grid[7][i][j] |= RED;
            grid[8][i][j] |= RED;
         }
         mask »= 1;  /* Get next bit in glyph row */
      }  /* for j */
   }  /* for i */
   // jongjung is for expansion
   // if (glyphs_overlap) {
   //    jongjung [JONG_VARIATIONS * consonant2] = 1;
   //    jongjung_overlaps++;
   // }
}  /* for each final consonant except nieun */

/*
```

```
   Superimpose final consonant 3 (Jongseong Nieun) on
   groups 9 to 11.
*/
codept = JONG_HEX + 3 * JONG_VARIATIONS;

for (i = 0; i < 16; i++) { /* For each glyph row */
   mask = 0x8000;
   for (j = 0; j < 16; j++) {
      if (glyph[codept][i] & mask) {
         grid[ 9][i][j] |= RED;
         grid[10][i][j] |= RED;
         grid[11][i][j] |= RED;
      }
      mask >>= 1;  /* Get next bit in glyph row */
   }
}


/*
   Turn the black (uncolored) cells into white for better
   visibility of grid when displayed.
*/
for (group = 0; group < 12; group++) {
   for (i = 0; i < 16; i++) {
      for (j = 0; j < 16; j++) {
         if (grid[group][i][j] == BLACK) grid[group][i][j] = WHITE;
      }
   }
}


/*
   Generate HTML output.
*/
fprintf (outfp, "<html>\n");
fprintf (outfp, "<head>\n");
fprintf (outfp, "  <title>Johab 6/3/1 Overlaps</title>\n");
fprintf (outfp, "</head>\n");
fprintf (outfp, "<body bgcolor=\"#FFFFCC\">\n");

fprintf (outfp, "<center>\n");
fprintf (outfp, "  <h1>Unifont Hangul Jamo Syllable Components</h1>\n");
fprintf (outfp, "  <h2>Johab 6/3/1 Overlap</h2><br><br>\n");

/* Print the color code key for the table. */
fprintf (outfp, "  <table border=\"1\" cellpadding=\"10\">\n");
fprintf (outfp, "    <tr><th colspan=\"2\" align=\"center\" bgcolor=\"#FFCC80\">");
fprintf (outfp, "<font size=\"+1\">Key</font></th></tr>\n");
fprintf (outfp, "    <tr>\n");
fprintf (outfp, "      <th align=\"center\" bgcolor=\"#FFFF80\">Color</th>\n");
fprintf (outfp, "      <th align=\"center\" bgcolor=\"#FFFF80\">Letter(s)</th>\n");
fprintf (outfp, "    </tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", BLUE);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Choseong (Initial Consonant)</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", GREEN);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Jungseong (Medial Vowel/Diphthong)</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", RED);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Jongseong (Final Consonant)</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", BLUE | GREEN);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Choseong + Jungseong Overlap</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", GREEN | RED);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Jungseong + Jongseong Overlap</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", RED | BLUE);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Choseong + Jongseong Overlap</td></tr>\n");

fprintf (outfp, "    <tr><td bgcolor=\"#%06X\">", RED | GREEN | BLUE);
fprintf (outfp, "    </td>");
fprintf (outfp, "<td>Choseong + Jungseong + Jongseong Overlap</td></tr>\n");

```

```
fprintf (outfp, "  </table>\n");
fprintf (outfp, "  <br><br>\n");


for (group = 0; group < 12; group++) {
   /* Arrange tables 3 across, 3 down. */
   if ((group % 3) == 0) {
      fprintf (outfp, "  <table border=\"0\" cellpadding=\"10\">\n");
      fprintf (outfp, "    <tr>\n");
   }

   fprintf (outfp, "      <td>\n");
   fprintf (outfp, "        <table border=\"3\" cellpadding=\"2\">\n");
   fprintf (outfp, "          <tr><th colspan=\"16\" bgcolor=\"#FFFF80\">");
   fprintf (outfp, "Choseong Group %d, %s %s</th></tr>\n",
            group < 6 ? group : (group > 8 ? group - 6 : group - 3),
            group < 6 ? (group < 3 ? "No" : "Without") : "With",
            group < 9 ? "Jongseong" : "Nieun");

   for (i = 0; i < 16; i++) {
      fprintf (outfp, "          <tr>\n");
      for (j = 0; j < 16; j++) {
         fprintf (outfp, "            <td bgcolor=\"#%06X\">",
                  grid[group][i][j]);
         fprintf (outfp, "    </td>\n");
      }
      fprintf (outfp, "          </tr>\n");
   }

   fprintf (outfp, "            </td>\n");
   fprintf (outfp, "          </tr>\n");
   fprintf (outfp, "        </table>\n");
   fprintf (outfp, "      </td>\n");

   if ((group % 3) == 2) {
      fprintf (outfp, "    </tr>\n");
      fprintf (outfp, "  </table>\n  </br>\n");
   }
}

/* Wrap up HTML table output. */
fprintf (outfp, "</center>\n");

/*
   Print overlapping initial consonant + vowel combinations.
*/
fprintf (outfp, "<h2>%d Vowel Overlaps with Initial Consonants Found</h2>",
         cho_overlaps);
fprintf (outfp, "<font size=\"+1\"><pre>\n");

for (i = JUNG_HEX;
     i < JUNG_HEX + TOTAL_JUNG * JUNG_VARIATIONS;
     i++) {
   /*
      If this vowel variation (Jungseong) had overlaps
      with one or more initial consonants (Choseong),
      find and print them.
   */
   if (jungcho [i - JUNG_HEX]) {
      ancient_choseong = 0;  /* Not within ancient choseong range yet. */
      fprintf (outfp, "<font color=\"#0000FF\"><b>");
      if (i >= JUNG_ANCIENT_HEX) {
         if (i >= JUNG_EXTB_HEX) fprintf (outfp, "Extended-B ");
         fprintf (outfp, "Ancient ");
      }
      fprintf (outfp, "Vowel at 0x%04X and&hellip;</b>", i + PUA_START);
      fprintf (outfp, "</font>\n\n");

      /*
         Get current vowel number, 0 to (TOTAL_JUNG - 1), and
         current vowel variation, 0 or 1, or 2 for final nieun.
      */
      vowel = (i - JUNG_HEX) / JUNG_VARIATIONS;
      vowel_variation = (i - JUNG_HEX) % JUNG_VARIATIONS;

      /* Get first Choseong group for this vowel, 0 to 5. */
      group = cho_variation (-1, vowel, -1);

      /*
         If this vowel variation is used with a final consonant
         (Jongseong) and the default initial consonant (Choseong)
```

```
         group for this vowel is < 3, add 3 to current Chosenong
         group.
      */
      if (vowel_variation > 0 && group < 3) group += 3;

      for (consonant1 = 0; consonant1 < TOTAL_CHO; consonant1++) {
         overlapped = glyph_overlap (glyph [i],
                                     glyph [consonant1 * CHO_VARIATIONS
                                          + CHO_HEX + group]);

         /*
            If we just entered ancient choseong range, flag it.
         */
         if (overlapped && consonant1 >= 19 && ancient_choseong == 0) {
            fprintf (outfp, "<font color=\"#0000FF\"><b>");
            fprintf (outfp, "&hellip;Ancient Choseong&hellip;</b></font>\n");
            ancient_choseong = 1;
         }
         /*
            If overlapping choseong found, print combined glyph.
         */
         if (overlapped != 0) {

            combine_glyphs (glyph [i],
                            glyph [consonant1 * CHO_VARIATIONS
                                 + CHO_HEX + group],
                            tmp_glyph);

            print_glyph_txt (outfp,
                             PUA_START +
                             consonant1 * CHO_VARIATIONS +
                             CHO_HEX + group,
                             tmp_glyph);

         } /* If overlapping pixels found. */
      } /* For each initial consonant (Choseong) */
   } /* Find the initial consonant that overlapped this vowel variation. */
} /* For each variation of each vowel (Jungseong) */

fputc ('\n', outfp);

fprintf (outfp, "</pre></font>\n");
fprintf (outfp, "</body>\n");
fprintf (outfp, "</html>\n");

fclose (infp);
fclose (outfp);


exit (EXIT_SUCCESS);
}


/**
   @brief Parse command line arguments.

   @param[in] argc The argc parameter to the main function.
   @param[in] argv The argv command line arguments to the main function.
   @param[in,out] infile The input filename; defaults to NULL.
   @param[in,out] outfile The output filename; defaults to NULL.
*/
void
parse_args (int argc, char *argv[], int *inindex, int *outindex,
            int *modern_only) {
   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;
}

```

5.42 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.

5.42.1 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

This program counts the number of glyphs that are defined in each "page" of 256 code points, and prints the counts in an 8 x 8 grid. Input is from stdin. Output is to stdout.

The background color of each cell in a 16-by-16 grid of 256 code points is shaded to indicate percentage coverage. Red indicates 0% coverage, green represents 100% coverage, and colors in between pure red and pure green indicate partial coverage on a scale.

Each code point range number can be a hyperlink to a PNG file for that 256-code point range's corresponding bitmap glyph image.

Synopsis:

```
unipagecount < font_file.hex > count.txt
unipagecount -phex_page_num < font_file.hex  -- just 256 points
unipagecount -h < font_file.hex              -- HTML table
unipagecount -P1 -h < font.hex > count.html  -- Plane 1, HTML out
unipagecount -l < font_file.hex              -- linked HTML table
```

Definition in file unipagecount.c.

5.42.2 Macro Definition Documentation

5.42.2.1 MAXBUF

`#define MAXBUF 256`

Maximum input line size - 1.

Definition at line 59 of file unipagecount.c.

5.42.3 Function Documentation

5.42.3.1 main()

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

The main function.

Parameters

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

Returns

This program exits with status 0.

Definition at line 70 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 */

/* make (print) flipped HTML table */
void mkftable (unsigned plane, int pagecount[256], int links);

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) {
```

```
00160             if (pagecount[i*16+j])
00161                fprintf (stdout,” *  ”);
00162             else
00163                fprintf (stdout,” .  ”);
00164          }
00165          else {
00166             fprintf (stdout, ”%3X ”, pagecount[i*16+j]);
00167          }
00168       }
00169       fprintf (stdout,”\n”);
00170    }
00171
00172  }
00173  exit (0);
00174 }
```

Here is the call graph for this function:

5.42.3.2 mkftable()

void mkftable (
unsigned plane,
int pagecount[256],
int links)

Create an HTML table linked to PNG images.

This function creates an HTML table to show PNG files in a 16 by 16 grid. The background color of each ”page” of 256 code points is shaded from red (for 0% coverage) to green (for 100% coverage).

Parameters

in	plane	The Unicode plane, 0..17.
in	pagecount	Array with count of glyphs in each 256 code point range.
in	links	1 = generate hyperlinks, 0 = do not generate hyperlinks.

Definition at line 190 of file unipagecount.c.

```
00191 {
00192    int i, j;
00193    int count;
00194    unsigned bgcolor;
00195
00196    printf (”<html>\n”);
00197    printf (”<body>\n”);
00198    printf (”<table border=\”3\” align=\”center\”>\n”);
00199    printf (”  <tr><th colspan=\”16\” bgcolor=\”#ffcc80\”>”);
00200    printf (”GNU Unifont Glyphs<br>with Page Coverage for Plane %d<br>(Green=100%%, Red=0%%)</th></tr>\n”,
      plane);
00201    for (i = 0x0; i <= 0xF; i++) {
00202       printf (”  <tr>\n”);
00203       for (j = 0x0; j <= 0xF; j++) {
00204          count = pagecount[ (i « 4) | j ];
00205
00206          /* print link in cell if links == 1 */
00207          if (plane != 0 || (i < 0xd || (i == 0xd && j < 0x8) || (i == 0xf && j > 0x8))) {
00208             /* background color is light green if completely done */
00209             if (count == 0x100) bgcolor = 0xccffcc;
00210             /* otherwise background is a shade of yellow to orange to red */
00211             else bgcolor = 0xff0000 | (count « 8) | (count » 1);
00212             printf (”    <td bgcolor=\”#%06X\”>”, bgcolor);
00213             if (plane == 0)
00214                printf (”<a href=\”png/plane%02X/uni%02X%X%X.png\”>%X%X</a>”, plane, plane, i, j, i, j);
00215             else
00216                printf (”<a href=\”png/plane%02X/uni%02X%X%X.png\”>%02X%X%X</a>”, plane, plane, i, j, plane, i, j);
00217             printf (”</td>\n”);
00218          }
00219          else if (i == 0xd) {
00220             if (j == 0x8) {
00221                printf (”    <td align=\”center\” colspan=\”8\” bgcolor=\”#cccccc\”>”);
00222                printf (”<b>Surrogate Pairs</b>”);
00223                printf (”</td>\n”);
00224             }  /* otherwise don’t print anything more columns in this row */
00225          }
00226          else if (i == 0xe) {
```

```
            if (j == 0x0) {
               printf ("    <td align=\"center\" colspan=\"16\" bgcolor=\"#cccccc\">");
               printf ("<b>Private Use Area</b>");
               printf ("</td>\n");
            }  /* otherwise don't print any more columns in this row */
         }
         else if (i == 0xf) {
            if (j == 0x0) {
               printf ("    <td align=\"center\" colspan=\"9\" bgcolor=\"#cccccc\">");
               printf ("<b>Private Use Area</b>");
               printf ("</td>\n");
            }
         }
      }
      printf ("  </tr>\n");
   }
   printf ("</table>\n");
   printf ("</body>\n");
   printf ("</html>\n");

   return;
}
```

Here is the caller graph for this function:

5.43 unipagecount.c

Go to the documentation of this file.

```
/**
   @file unipagecount.c

   @brief unipagecount - Count the number of glyphs defined in each page
                         of 256 code points

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

   @copyright Copyright (C) 2007, 2008, 2013, 2014 Paul Hardy

   This program counts the number of glyphs that are defined in each
   "page" of 256 code points, and prints the counts in an 8 x 8 grid.
   Input is from stdin.  Output is to stdout.

   The background color of each cell in a 16-by-16 grid of 256 code points
   is shaded to indicate percentage coverage.  Red indicates 0% coverage,
   green represents 100% coverage, and colors in between pure red and pure
   green indicate partial coverage on a scale.

   Each code point range number can be a hyperlink to a PNG file for
   that 256-code point range's corresponding bitmap glyph image.

   Synopsis:

            unipagecount < font_file.hex > count.txt
            unipagecount -phex_page_num < font_file.hex  -- just 256 points
            unipagecount -h < font_file.hex              -- HTML table
            unipagecount -P1 -h < font.hex > count.html  -- Plane 1, HTML out
            unipagecount -l < font_file.hex              -- linked HTML table
*/
/*
   LICENSE:

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

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

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

/*
   2018, Paul Hardy: Changed "Private Use" to "Private Use Area" in
   output HTML file.

```

```
  21 October 2023 [Paul Hardy]:
  - Added full prototype for mkftable function in main function.
*/

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

#define MAXBUF 256 ///< Maximum input line size - 1.


/**
  @brief The main function.

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

  char inbuf[MAXBUF]; /* Max 256 characters in an input line */
  int i, j;  /* loop variables */
  unsigned plane=0; /* Unicode plane number, 0 to 0x16 */
  unsigned page;  /* unicode page (256 bytes wide) */
  unsigned unichar; /* unicode character */
  int pagecount[256] = {256 * 0};
  int onepage=0; /* set to one if printing character grid for one page */
  int pageno=0; /* page number selected if only examining one page */
  int html=0;   /* =0: print plain text; =1: print HTML */
  int links=0;  /* =1: print HTML links; =0: don’t print links */

  /* make (print) flipped HTML table */
  void mkftable (unsigned plane, int pagecount[256], int links);

  size_t strlen();

  if (argc > 1 && argv[1][0] == ’-’) {  /* Parse option */
    plane = 0;
    for (i = 1; i < argc; i++) {
      switch (argv[i][1]) {
        case ’p’:  /* specified -p<hexpage> -- use given page number */
          sscanf (&argv[1][2], ”%x”, &pageno);
          if (pageno >= 0 && pageno <= 255) onepage = 1;
          break;
        case ’h’:  /* print HTML table instead of text table */
          html = 1;
          break;
        case ’l’:  /* print hyperlinks in HTML table */
          links = 1;
          html = 1;
          break;
        case ’P’:  /* Plane number specified */
          plane = atoi(&argv[1][2]);
          break;
      }
    }
  }
  /*
    Initialize pagecount to account for noncharacters.
  */
  if (!onepage && plane==0) {
    pagecount[0xfd] = 32;  /* for U+FDD0..U+FDEF */
  }
  pagecount[0xff] = 2;   /* for U+nnFFFE, U+nnFFFF */
  /*
    Read one line at a time from input.  The format is:

      <hexpos>:<hexbitmap>

    where <hexpos> is the hexadecimal Unicode character position
    in the range 00..FF and <hexbitmap> is the sequence of hexadecimal
    digits of the character, laid out in a grid from left to right,
    top to bottom.  The character is assumed to be 16 rows of variable
    width.
  */
  while (fgets (inbuf, MAXBUF-1, stdin) != NULL) {
    sscanf (inbuf, ”%X”, &unichar);
    page = unichar » 8;
    if (onepage) { /* only increment counter if this is page we want */
      if (page == pageno) { /* character is in the page we want */
```

```
            pagecount[unichar & 0xff]++; /* mark character as covered */
         }
      }
      else { /* counting all characters in all pages */
         if (plane == 0) {
            /* Don't add in noncharacters (U+FDD0..U+FDEF, U+FFFE, U+FFFF) */
            if (unichar < 0xfdd0 || (unichar > 0xfdef && unichar < 0xfffe))
               pagecount[page]++;
         }
         else {
            if ((page » 8) == plane) { /* code point is in desired plane */
               pagecount[page & 0xFF]++;
            }
         }
      }
   }
   if (html) {
      mkftable (plane, pagecount, links);
   }
   else {  /* Otherwise, print plain text table */
      if (plane > 0) fprintf (stdout, "  ");
      fprintf (stdout,
         "    0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F\n");
      for (i=0; i<0x10; i++) {
         fprintf (stdout,"%02X%X ", plane, i); /* row header */
         for (j=0; j<0x10; j++) {
            if (onepage) {
               if (pagecount[i*16+j])
                  fprintf (stdout," *  ");
               else
                  fprintf (stdout," .  ");
            }
            else {
               fprintf (stdout, "%3X ", pagecount[i*16+j]);
            }
         }
         fprintf (stdout,"\n");
      }

   }
   exit (0);
}


/**
   @brief Create an HTML table linked to PNG images.

   This function creates an HTML table to show PNG files
   in a 16 by 16 grid.  The background color of each "page"
   of 256 code points is shaded from red (for 0% coverage)
   to green (for 100% coverage).

   @param[in] plane The Unicode plane, 0..17.
   @param[in] pagecount Array with count of glyphs in each 256 code point range.
   @param[in] links 1 = generate hyperlinks, 0 = do not generate hyperlinks.
*/
void
mkftable (unsigned plane, int pagecount[256], int links)
{
   int i, j;
   int count;
   unsigned bgcolor;

   printf ("<html>\n");
   printf ("<body>\n");
   printf ("<table border=\"3\" align=\"center\">\n");
   printf ("  <tr><th colspan=\"16\" bgcolor=\"#ffcc80\">");
   printf ("GNU Unifont Glyphs<br>with Page Coverage for Plane %d<br>(Green=100%%, Red=0%%)</th></tr>\n", plane);
   for (i = 0x0; i <= 0xF; i++) {
      printf ("  <tr>\n");
      for (j = 0x0; j <= 0xF; j++) {
         count = pagecount[ (i « 4) | j ];

         /* print link in cell if links == 1 */
         if (plane != 0 || (i < 0xd || (i == 0xd && j < 0x8) || (i == 0xf && j > 0x8))) {
            /* background color is light green if completely done */
            if (count == 0x100) bgcolor = 0xccffcc;
            /* otherwise background is a shade of yellow to orange to red */
            else bgcolor = 0xff0000 | (count « 8) | (count » 1);
            printf ("    <td bgcolor=\"#%06X\">", bgcolor);
```

```
                if (plane == 0)
                    printf ("<a href=\"png/plane%02X/uni%02X%X%X.png\">%X%X</a>", plane, plane, i, j, i, j);
                else
                    printf ("<a href=\"png/plane%02X/uni%02X%X%X.png\">%02X%X%X</a>", plane, plane, i, j, plane, i, j);
                printf ("</td>\n");
            }
            else if (i == 0xd) {
                if (j == 0x8) {
                    printf ("    <td align=\"center\" colspan=\"8\" bgcolor=\"#cccccc\">");
                    printf ("<b>Surrogate Pairs</b>");
                    printf ("</td>\n");
                }  /* otherwise don't print anything more columns in this row */
            }
            else if (i == 0xe) {
                if (j == 0x0) {
                    printf ("    <td align=\"center\" colspan=\"16\" bgcolor=\"#cccccc\">");
                    printf ("<b>Private Use Area</b>");
                    printf ("</td>\n");
                }  /* otherwise don't print any more columns in this row */
            }
            else if (i == 0xf) {
                if (j == 0x0) {
                    printf ("    <td align=\"center\" colspan=\"9\" bgcolor=\"#cccccc\">");
                    printf ("<b>Private Use Area</b>");
                    printf ("</td>\n");
                }
            }
        }
        printf ("  </tr>\n");
    }
    printf ("</table>\n");
    printf ("</body>\n");
    printf ("</html>\n");

    return;
}
```